NORTHWESTERN UNIVERSITY

Black Box Methods for Inferring Parallel Applications' Properties in Virtual Environments

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Field of Computer Science

By

Ashish Gupta

EVANSTON, ILLINOIS

June 2008

Thesis Committee

Peter A. Dinda, Northwestern University, Committee Chair
Fabian Bustamante, Northwestern University, Committee Member
Yan Chen, Northwestern University, Committee Member
Dongyan Xu, Purdue University, Committee Member

Abstract

Black Box Methods for Inferring Parallel Applications' Properties in Virtual Environments

Ashish Gupta

Virtual machine based distributed computing greatly simplifies and enhances adaptive/autonomic computing by lowering the level of abstraction, benefiting both resource providers and users. We are developing Virtuoso, a middleware system for virtual machine shared-resource computing (e.g. grids) that provides numerous advantages and overcomes many obstacles a user faces in using shared resources for deploying distributed applications. A major hurdle for distributed applications to function in such an environment is locating, reserving/scheduling and dynamically adapting to the appropriate communication and computational resources so as to meet the applications' demands, limited by cost constraints. Resources can be very heterogeneous, especially in wide area or shared infrastructures, and their availability is also highly dynamic.

To achieve such automated adaptation, one must first learn about the various demands and properties of the distributed application running inside the VMs. My thesis is that it is feasible to infer the applications' demands and behavior, to a significant degree, without actually knowing much about the application or its operating system; I have investigated and demonstrated numerous novel techniques to infer many useful properties of parallel applications, in an automated fashion. Throughout my work I have used a black box approach to inference. Thus the results are applicable to *existing, unmodified* applications and operating systems and are widely applicable.

I show how to infer the communication behavior and the runtime topology of a parallel

application. I also show how to infer very useful runtime properties of a parallel application such as its runtime performance. I have developed the algorithms to understand quantitatively the slowdown of an application under external load to find global bottlenecks at the resource and process level using time decomposition of the application's execution.

In addition, my dissertation also gives limited evidence and points to other work that shows that this inferred information can actually be used to benefit the application, without having any knowledge of the application/OS itself.

Dedications

To my parents, Anuradha Gupta and J.B. Gupta, for boring me, raising me, educating me, and most importantly giving me strong roots and aspirations in life that have led me this far.

Acknowledgments

It is a pleasure to thank the many people who made this thesis possible.

It is difficult to overstate my gratitude to my Ph.D. advisor, Prof. Peter Dinda. Mainly because of his enthusiasm, his enormous patience and his great efforts to guide me through the Ph.D. maze, I have been able to complete this dissertation. Throughout my thesis-writing period, he provided encouragement, sound advice and lots of good ideas. He has been a fantastic mentor as well as a great friend in times of need, when I needed the push to continue and strive towards completion. His strong mark on my education and career will be felt through out my life.

I would further like to thank the many wonderful faculty members with whom I had the good fortune to work and associate with: Prof. Fabian Bustamante for giving me a sound understanding of operating systems and distributed systems concepts, being a great mentor, supporter and friend throughout my Ph.D; Prof. Yan Chen for his contribution to my networking knowledge, for my long lasting association with him for an exciting network anomaly detection project and his invaluable mentorship and guidance during my Ph.D; Prof. Ming Kao for teaching me about algorithms and being a great mentor to me as well. He always believed in me and his belief gave me great confidence in all my academic related endeavors.

Apart from faculty members at Northwestern, I would also like to thank Prof. Dongyan Xu, my external committee member from Purdue University. His work in the area of distributed virtual computing has provided great support to my arguments and has greatly contributed to the importance and growth of this area in general. He has also given invaluable comments during my thesis proposal and later on, to enhance the presentation and value of my contribution.

I would also like to thank many teachers who have taught me Computer Science and given me a solid foundation without which this dissertation would have been an impossible task. Specifically Prof. B.N. Jain, Prof. Anshul Kumar, Prof. Naveen Garg, Prof. K.K. Biswas, Prof.

S.N. Maheshwari, Prof. S. Arun Kumar and Prof. Prem Kalra taught me fundamental courses in Computer Science at the Indian Institute of Technology, New Delhi, India and I owe a strong foundation to their teaching and tests.

I am indebted to my many student colleagues for providing a stimulating and fun environment in which to learn and grow. Most of them have been very dear friends as well. I am especially grateful to Abhinav Dayal, Manan Sanghi, Ankit Mohan, Robbie Schweller, Ananth Sundararaj, Bin Lin, Jack Lange, Amit Mondal, Anup Goyal, Sanjay Sood, Sara Owsley, Holger Winnemoeller, Jason Skicewicz, Praveen Paritosh, Dong Lu, Stefan Birrer and Gergely Biczk. Ankit has been a great friend, roommate and a PhD mate with whom I have had several inspiring discussions and received great support on my journey towards Ph.D. completion.

I am indebted to my many of my fabulous friends at Northwestern and for all the emotional support, friendship, entertainment, and caring they provided, and for helping me get through the difficult times. I am especially thankful to Ankit Mohan, Vageesh Kumar, Padmanav, Robbie Schweller, Bin Lin, Smita Desai, Manish Agarwal, Roopam Khare, Angela Manuel, Simran Sudan, Swati Gupta, Amit Mondal, Dibyendu Dey, Anjali Singhal, Aparna Ramachandran, Nidhi Parashar, Sunanda Prabhu-Gaunkar, Chetan Kumar, Sankar Narayan, Dhwanil Damania, Anup Goyal, Aniket Kaloti, Robin Koshy, Ravi Agarwal, Sagar Vemuri and Monika Patel. Many of these friends have provided enormous support and comradeship. Swati Gupta, who I came to know during my writing phase only, has been a great companion and a dear friend. She has always been there during my writing phase and we've also had many stimulating conversations about work, life, career and her frequent reminders about "How's the work going ?" on instant messaging during late hours often alerted me to my real task at hand and speeded up the writing.

Apart from my friends at Northwestern, a lot of my friends provided great encouragement for my Ph.D. to whom I am especially thankful to. Angad Chowdhry, one of my best childhood buddies, encouraged me greatly to leap towards the finish line, stimulated many amazing discussions and also introduced me to the great trance band Infected Mushroom, under whose

influence, I have spent many hours writing and coding. I am very thankful to Shipra Jindal, perhaps my oldest friend in terms of time we know each other (25 years), for being especially pushing and persuasive to help me finish my Ph.D. She supported and encouraged me a lot in many low times, and that definitely gave me a lot of impetus to carry on forward. My buddy Ashish Gupta (who shares my name too) and fellow batchmate from undergraduate, has been a great source of inspiration and an awesome friend as well.

I wish to thank my entire extended family for providing a loving environment for me.

Lastly, and most importantly, I wish to thank my parents, Anuradha Gupta and J.B. Gupta. They bore me, raised me, supported me, taught me, and loved me. To them I dedicate this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Virtual machines have the potential to simplify the use of distributed resources in a way unlike any other technology available today, making it possible to run diverse applications with high performance after only minimal or no programmer and administrator effort. Network and host bottlenecks, difficult placement decisions, and firewall obstacles are routinely encountered, making effective use of distributed resources an obstacle to innovative science. Such problems, and the human effort needed to work around them, limit the development, deployment, and scalability of distributed parallel applications.

A detailed case for virtual machine-based distributed and parallel computing was earlier presented in [43], and I have been part of the collaborative effort that is developing a system, Virtuoso, which has the following model:

- The user receives what appears to be a new computer or computers on his network at very low cost. The user can install, use, and customize the operating system, environment, and applications with full administrative control.
- The user chooses where to execute the virtual machines. Checkpointing and migration is handled efficiently through Virtuoso. The user can delegate these decisions to the system.
- A service provider need only install the VM management software to support a diverse set of users and applications.

- Monitoring, adaptation, resource reservation, and other services are retrofitted to existing applications at the VM level with no modification of the application code, resulting in broad application of these technologies with minimal application programmer involvement.

A major impediment for resource providers in supplying resources as well as for users in utilizing these resources for distributed computing is the heterogeneity of the underlying hardware resources, operating systems and middleware that may be different for each resource provider. Dealing with portability issues in such an environment is very difficult. A virtual machine image that includes pre-installed versions of the correct operating system, libraries, middleware and applications can make the deployment of new software far simpler. The goal here is that the user can then use and configure a VM as he likes and just make multiple copies to provide a distributed computing environment that fits his requirements.

Along with using virtual machines, an important concept in Virtuoso is VM-level virtual overlay networking (VNET) [134] that can project a VM running on a remote network on to the user's local LAN. Thus the end result is that user feels as if he has access to a complete raw machine attached to his local network. The virtual machine monitor deals with any computational resource related hurdles which users face whereas the VM-level virtual networking alleviates communication issues that are common in wide area distributed applications. For example, different policies, administration, proxies and firewall rules at different sites can make matters complicated for the application programmer. Virtuoso hides these details, presenting a simple abstraction of purchasing a new wholly owned machine connected to the user's network. Simplifying distributed computation over the wide area to such a level can make autonomic distributed computing over shared infrastructure very attractive for a wide range of users, ranging from scientific computational apps like CFD applications (NAS benchmarks [17, 148]) , enterprise IT applications to even deploying virtual web services over a Grid [20]. For a classification of certain applications that can be leveraged, I refer the reader to my colleague Ananth

Sundararaj's dissertation [138].

1.1 Introducing Runtime Adaptation

An application running in some distributed computing environment, must adapt to the (possibly changing) available computational and networking resources. Despite many efforts [15, 24, 34, 55, 56, 68, 82, 95, 96, 111, 127, 140, 150, 159], adaptation mechanisms and control have remained very application-specific or required rewriting applications. Our group has shown that that adaptation using the low-level, application-independent adaptation mechanisms made possible by virtual machines interconnected with a virtual network is effective [135, 139].

Custom adaptation by either the user or the resource provider is exceedingly complex as the application requirements, computational and network resources can vary over time. VNET is in an ideal position to

1. measure the traffic load and application topology of the virtual machines,
2. monitor the underlying network and its topology,
3. adapt the application as measured in step 1 to the network as measured in step 2, and
4. adapt the network to the application by taking advantage of resource reservation mechanisms.

These services can be done on behalf of *existing, unmodified applications and operating systems* running in the virtual machines. One previous paper [134] laid out the argument and formalized the adaptation problem, while a second paper [135] gave very preliminary results on automatic adaptation using one mechanism. In the following discussion we also highlight how to control three adaptation mechanisms provided by Virtuoso in response to the inferred communication behavior of the application running in a collection of virtual machines, and provide extensive evaluation.

To give a structured overview of the system in the context of adaptation, I briefly discuss some important components of Virtuoso and related systems that have been developed. I also show where my work fits in within these areas.

Infrastructure: Infrastructure refers to the underlying services that are needed to support the idea of distributed virtual computing over a wide area. These include:

VNET [134]: VNET is a data link layer virtual network tool. Using VNET, virtual machines have no network presence at all on a remote site. Instead, VNET provides a mechanism to project their virtual network cards onto another network, which also moves the network management problem from one network to another. For example, all of a user's virtual machines can be made to appear to be connected to the user's own network, where the user can use his existing mechanisms to assure that they have appropriate network presence. Because the virtual network is at the data link layer, a machine can be migrated from site to site without changing its presence- it always keeps the same IP address, routes, etc. Vnet supports arbitrary overlay network topology and routing. This can assist in adapting distributed/parallel applications if we can infer their communication topology.

WREN(developed by Zangrilli et al [155]): Watching Resources from the Edge of the Network (Wren) is designed to passively monitor applications network traffic and use those observations to determine the available bandwidth along the network paths used by the application. The key observation behind Wren is that even when the application is not saturating the network it is sending bursts of traffic that can be used to measure the available bandwidth of the network.

Adaptation Mechanisms: As discussed above, adaptation mechanisms are the means by which we can change the application's resource configuration to better suit its demands and underlying resource availability, for the purpose of improving its performance. Some of these mechanisms are:

- **Virtual Machine Migration:** Virtuoso allows us to migrate a VM from one physical host to another. Much work exists that demonstrates that fast migration of VMs running commodity applications and operating systems is possible [85, 112, 121]. Migration times down to 5 seconds have been reported [85]. Migration of entire OS instances on a commodity cluster with service downtimes as low as 60ms have been demonstrated [31]. As migration times decrease, the rate of adaptation we can support and our work's relevance increases. Note that while process migration and remote execution has a long history [40, 108, 131, 141, 154], to use these facilities, we must modify or relink the application and/or use a particular OS. Neither is the case with VM migration.

- **Overlay Topology Modification:** VNET allows us to modify the overlay topology among a user's VMs at will. A key difference between it and overlay work in the application layer multicast community [19, 23, 73] is that the VNET provides global control of the topology, which our adaptation algorithms currently (but not necessarily) assume.

- **Overlay Forwarding:** VNET allows us to modify how messages are routed on the overlay. Forwarding tables are globally controlled, and topology and routing are completely separated, unlike in multicast systems.

- **VRESERVE** [88]: VRESERVE automatically and dynamically creates network reservation requests based on the inferred network demands of running distributed and/or parallel applications with no modification to the application or operating system, and no input from the user or developer.

- **VSCHED** [91]: Virtuoso must be able to mix batch and interactive VMs on the same physical hardware, while satisfying constraints on responsiveness and compute rates for each workload. VSched is the component of Virtuoso that provides this capability. VSched is an entirely user-level tool that interacts with the stock Linux kernel running below any type-II virtual machine monitor to schedule all VMs using a periodic real-time schedul-

ing model. This abstraction allows compute rate and responsiveness constraints to be straightforwardly described using a period and a slice within the period, and it allows for fast and simple admission control.

Adaptation Algorithms: VADAPT [136]: The adaptation control algorithms are implemented in the VADAPT component of Virtuoso. For a formalization of the adaptation control problem, please see the dissertation of my colleague Ananth Sundararaj [138]. The full control problem, informally stated in English, is "Given the network traffic load matrix of the application and its computational intensity in each VM, the topology of the network and the load on its links, routers, and hosts, what is the mapping of VMs to hosts, the overlay topology connecting the hosts, and the forwarding rules on that topology that maximizes the application throughput?" This component greatly overlaps with my thesis and dissertation and forms an important part of it.

User Feedback-based Adaptation [59, 92]: The optimization problems associated with adaptive and autonomic computing systems are often difficult to pose well and solve efficiently. A key challenge is that for many applications, particularly interactive applications, the user or developer is unlikely or unable to provide either the objective function f, or constraints. It is a key problem encountered broadly in adaptive and autonomic computing. This part uses Virtuoso context to explore two core ideas. In human-driven specification, it explores how to use direct human input from users to pose specific optimization problems, namely to determine the objective function f and expose hidden constraints. Once there is a well-specified problem, there is a need to search for a solution in a very large solution space. In human -driven search, it explore how to use direct human input to guide the search for a good solution, a valid configuration x that optimizes f(x). My colleague Bin Lin explores this topic in detail in his dissertation [93].

Inference Techniques: Inference is a critical part of the adaptation process and the Virtuoso system as it is the information driver in the system that provides us insight into the demands and behavior of parallel applications at run time. Runtime Black-box Inference is the main con-

tribution that I make to the Virtuoso system and most of this dissertation provides contributions in this area. An early work of mine called **VTTIF** [58](Virtual Topology and Traffic Inference Framework) integrates with VNET to automatically infer the dynamic topology and traffic load of applications running inside the VMs in the Virtuoso system. In our earlier work, I demonstrated that it is possible to successfully infer the behavior of a BSP application by observing the low level traffic sent and received by each VM in which it is running. Further in [136] I showed how to smooth VTTIF's reactions so that adaptation decisions made on its output cannot lead to oscillation. This component is an essential and initial part of my dissertation's theme and problem statement.

1.2 The Inference Problem and the Thesis

A major hurdle for distributed applications is locating, reserving/scheduling and dynamically adapting to the appropriate communication and computational resources so as to meet the applications' demands, limited by cost constraints. Resources can be heterogeneous over a wide area and if shared, their availability is also highly dynamic. Thus, proper automatic placement and scheduling of application's computation and communication satisfying performance and cost constraints, is an important challenge. If distributed computing is to become popular over shared resources spread over the wide area, these difficult tasks must not be an operation which the user himself has to deal with. At the same time, performance provided by Virtuoso must be decent, so that users are motivated to use the wide area resources for their applications. Their goals are all geared towards making distributed computing an autonomic experience for the end users.

To achieve this, there must be a understanding of what the distributed application wants, in order to adapt it and improve its performance. Apart from its wants, its also important to be able to measure certain fundamental properties of an application like its performance or to figure out if an application is under distress in the first place.

My thesis is that it is feasible to infer various useful demands and behavior of an application running inside a collection of VMs to a significant degree using a black box model. To evaluate this thesis, I enumerate and define various demands and types of behavior that can be inferred, and also design, implement and evaluate ideas and approaches towards inferring these. One of the demands I infer is the communication behavior and the runtime topology of a parallel application. I also show how to infer some very useful runtime properties of a parallel application like its runtime performance, its slowdown under external load, its global bottlenecks. Significantly all of this is done using black-box assumptions and without specific assumptions about the application or the operating system. I also give evidence of how automatic black box inference can assist in adapting the application and its resource usage resulting in improved performance.

I have demonstrated that it is possible to create techniques and algorithms that automatically understand an application's needs and bottlenecks without any external input from the user or application. I have found evidence and describe how we can automatically meet these demands by employing various mechanisms provided by a Virtuoso-like infrastructure, such as VM migration, modifications to the VNET overlay topology and forwarding rules, and use of resource reservation mechanisms in the underlying network.

1.3 Major Threads of my Work

Application Inference: Here the objective is to understand the various demands of the application like computational load, communication behavior, application topology (e.g. in BSP-style parallel applications). Additionally we infer its current collective performance, its current performance bottlenecks, etc. One of our main ambitions for performance adaptation in Virtuoso is that it should be fully automated, i.e., without any intervention from the user or the developer. This can be achieved if Virtuoso can automatically infer these requirements.

For performance adaptation, we also need to do system inference i.e. infer the configuration

and availability of the underlying physical resources that include computational and network resources. This is described earlier in section 1.1.

Benefit for autonomic adaptation from inference: An interesting challenge in adaptation control algorithms is dynamic adaptation. Adaptation only begins with the initial placement of the application in Virtuoso. With time, the application's demands may change and the resource availability is also dynamic. Therefore, it is also important to support dynamic adaptation of application to prevent and performance degradation and boost it when possible. This involves dynamic inference and adaptation i.e. keeping an updated view of the application and underlying resources and adapting it midway if the performance requirements specified by the user are threatened in the existing scenario. We have shown some initial examples of this dynamic inference and some interesting challenges that come up, like oscillations [134, 136]. To demonstrate this, I will refer to my work on dynamic adaptation based on inference, which can boost on-the-fly performance of these distributed applications. However it is important to note that the adaptation being an endless subject, the main goal of my thesis is application inference, with adaptation as a user to give an idea of how such inference can be used to benefit the application in a dynamic fashion without any knowledge of application itself. Apart from adaptation, other forms of management and problem detection can also benefit from inference.

1.3.1 Difference between Black Box and Gray Box Techniques

To clarify here I discuss the working definitions of black box vs gray box techniques. Black box techniques do not assume any inside knowledge of the system or the object they attempt to understand. Gray box techniques can use information internal to the system. For example, in a multi-tier web application, using the application logs or request information available inside the VM would qualify as gray box techniques. A further subtle distinction is that even if inside signals are not used, but knowledge of how the system or the object functions is used to assist inference, this also qualifies as a gray box system. This is well elaborated in the work by

Arpaci-Dusseau. et al [16]. For example the TCP congestion control algorithm uses timeouts to infer network congestion - this assumes this relationship of congestion to timeouts, which may not be true in other domains such as wireless networks, where other causes may contribute to timeouts. Therefore, the TCP congestion control mechanisms are more aptly gray box than black box.

To make our techniques to be generally useful and applicable to a wide range of applications without modification our goal is to assume no knowledge of the application or anything inside the guest VM itself i.e. a black box approach.

There is an obvious tradeoff in assuming more knowledge about the application and the guest VM. On one hand, having access to application specific information like the HTTP request log for a web server, or the input parameters to a BSP application could assist in inference - on the other hand it will make these techniques tougher to deploy. For wider adoption its an important goal to keep our knowledge of the application to a minimum. Also ideally we do not want to avail ourselves of any knowledge inside the guest VM like OS logs, individual process information and other statistics available in *procfs* or *sysfs*. Without this goal, it again requires extra effort from the developer's or deployer's side to equip the guest OS with special monitoring daemons to monitor and report these statistics outside the guest VM. We want to avoid this additional barrier to entry.

1.3.2 Formalization for Adaptation and Role of Inference

The following formalization is for the Virtuoso Adaptation problem and has been defined by my colleague Ananth Sundararaj. This formalization defines the various input and output parameters for the adaptation problem. To indicate how inference fits integrally with the adaptation problem, I point out portions of the formalization that refer to the inference part that I is relevant to the application part and I cover in my dissertation.

Note that this formalization only covers some relevant properties that can be inferred in the

INPUT:

- A directed graph $G = (H, E)$ (Remos)
- A function bw : $E \rightarrow \mathbb{R}$ (Remos, NWS, RPS)
- A function lat : $E \rightarrow \mathbb{R}$ (Remos, NWS, RPS)
- A function compute : $H \rightarrow \mathbb{R}$ (NWS)
- A function size : $H \rightarrow \mathbb{R}$ (Remos)
- A set, VM $= (\text{vm}_1, \text{vm}_2 \ldots \text{vm}_n)$, $n \in \mathbb{N}$
- **A function** vm_compute **:** VM $\rightarrow \mathbb{R} \rightarrow$ The CPU utilization demanded by the VM
- **A function** vm_size **:** VM $\rightarrow \mathbb{R} \rightarrow$ The size of the VM
- **A set of ordered 4-tuples** $\mathcal{A} = \{(s_i, d_i, b_i, l_i) \mid s_i, d_i \in \text{VM};\ b_i, l_i \in \mathbb{R};\ i = 1, \ldots, m\} \rightarrow$ Here the input is the current inferred traffic and latency between each pair of VMs.
- A set of ordered pairs $\mathcal{M} = \{(\text{vm}_i, h_i) \mid \text{vm}_i \in \text{VM}, h_i \in H;\ i = 1, 2 \ldots r, r \leq n\}$

Figure 1.1: **Problem 1** (Input for the Generic Adaptation Problem In Virtual Execution Environments (GAPVEE)). The inference parts that are relevant to my dissertation are highlighted using a box.

context of adaptation. There are other useful properties also that I describe in later chapters that are not covered in this formalization. However this gives a good understanding of how inference plays a critical role towards adaptation.

Figure 1.1 shows the problem formalization. All of the input described in the figure can be inferred. However I focus only on inference that is connected to the application itself. Other aspects of the input have also been covered by parts of the Virtuoso project by my colleagues and other projects. These parts are mentioned in the figure (Remos [38], NWS [151] and RPS [12]) For a complete definition of the terminology, I refer the reader to the problem formulation in Chapter 4 of Ananth Sundararaj's dissertation [138].

The main categories are resource availability, VM/application demands and user-imposed

OUTPUT: vmap : $VM \rightarrow H$ and $R : \mathcal{A} \rightarrow \mathcal{P}$ such that $\rightarrow$ *Denotes the final adaptation output which consists of the the VM to host mapping and the routing specification*

- $\sum_{\text{vmap(vm)}=h} \left(\text{vm\_compute(vm)}\right) \leq \text{compute(h)}, \forall\, h \in H \rightarrow$ *To make sure computation needed is less than available*
- $\sum_{\text{vmap(vm)}=h} \left(\text{vm\_size(vm)}\right) \leq \text{size}(h), \forall\, h \in H \rightarrow$ *To make sure VMs fit space-wise*
- $h_i = \text{vmap}(\text{vm}_i), \forall M_i = (\text{vm}_i, h_i) \in \mathcal{M} \rightarrow$ *Checks if the particular VM to host mapping is allowed*
- $\left(bw_e - \sum_{e \in R(A_i)} b_i\right) \geq 0, \forall e \in E \rightarrow$ *Checks if bandwidth capacity is remaining on all edges after adaptation*
- $\left(\sum_{e \in R(A_i)} \text{lat}_e\right) \leq l_i, \forall e \in E \rightarrow$ *Checks that the latency demands are met.*
- $\sum_{i=1}^{m} \left(\min_{e \in R(A_i)} \left\{\text{rc}_e\right\}\right)$, where $\text{rc}_e = (\text{bw}_e - \sum_{e \in R(A_i)} b_i)$, is maximized $\rightarrow$ *The objective function that we need to optimize for the output*

Figure 1.2: Output for the **Problem 1** (Generic Adaptation Problem In Virtual Execution Environments (GAPVEE)). Each line is annotated with italic text explaining what it denotes

constraints like VM to host mapping etc. Some of the inference aspects in this problem statement are highlighted above. These correspond to the compute demands, the size of VMs, time of execution remaining and the bandwidth/latency demands for all communicating VMs. The user or the application developer does not know these input values in advance. They can depend on the particular execution environment, resource availability, input data etc. Therefore these must be inferred dynamically at runtime to serve as input to the adaptation problem

The goal of the adaptation problem is to output the final configuration of the VM to host mapping and the routing specification for each VM-to-VM communication. The formalization for the output is shown in Figure 1.2.

The output is also annotated to give an idea of what requirement or goal is being met with each line of the output statement. It contains inference relevant objectives like: VM must fit space-wise on each physical host. There must be bandwidth capacity remaining on each edge, after the application's demands are met. Moreover there is an unspecified objective function

whose goal is to maximize certain parameters like residual capacity remaining, application execution time etc. However to reach this objective one must first know the details about the VMs and the application running inside it, apart from resource availability themselves. My goal is to recover as much information as possible from the VMs and the application using black box techniques.

1.4 Application Model

In my dissertation I mainly focus on parallel applications, specifically Bulk-Synchronous Parallel [50] (BSP) style applications. The BSP model is originally introduced in an article from Leslie Valiant [143]. It has become a very popular model for implementing a large variety of algorithms and scientific applications [17, 27, 49, 52, 53, 103, 115, 133].

We consider parallel programs whose execution alternates between one or more computing phases and one or more communication phases, including metaphases. The next phase can begin only after all operations from the current phase are finished. In the computation phase, each node does local computation, with all the data required for it available in the local memory. In the communication phase, there is either inter-process communication, synchronization or aggregation (e.g. reduction and scan) operation [133].

BSP algorithms consist of a sequence of super-steps. In each superstep the processors operate independently performing local operations and global communications by send/receive operations. The sent messages can be assumed to be received in the next superstep. At the end of a superstep a barrier synchronization is realized. The time for the BSP algorithm is the sum of times for the supersteps [51]. Figure 1.3 illustrates the superstep structure of a very popular BSP benchmark FT from the NAS set of benchmarks from NASA [17, 133].

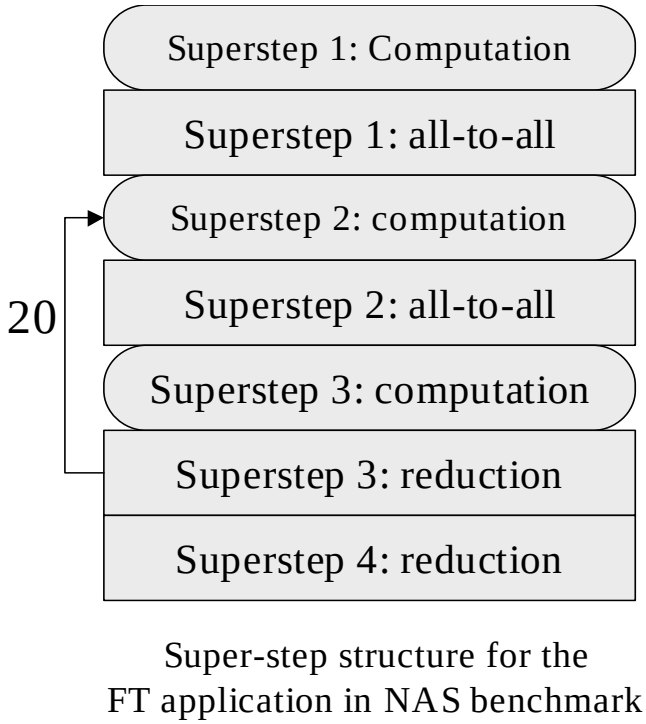


Figure 1.3: An example of superstep structure for a very popular benchmark Fourier Transform from NASA

Some Applications that I Consider

In my dissertation I focus on a set of BSP benchmarks for experimentation and testing. I introduce two benchmarks that I use widely in my work.

Patterns: This is a synthetic workload generator that I developed in C. It can execute many different kinds of topologies common in BSP parallel programs. We use this extensively to test our framework. Patterns does message exchanges according to a topology provided at the command line. Patterns emulates a BSP program with alternating dummy compute phases and communication phases according to the chosen topology. It takes the following arguments:

- pattern: The particular topology to be used for communication.
- numprocs: The number of processors to use. The processors are determined by a special

hostfile.

- messagesize: The size of the message to exchange.
- numiters: The number of compute and communicate phases
- flopsperelement: The number of multiply-add steps
- readsperelement: The number of main memory reads
- writesperelement: The number of main memory writes

Patterns generates a deadlock free and efficient communication schedule at startup time for the given topology and number of processors to be used. The following topologies are supported:

- *n*-dimensional mesh, neighbor communication pattern
- *n*-dimensional torus, neighbor communication pattern
- *n*-dimensional hypercube, neighbor communication pattern
- Binary reduction tree
- All-to-all communication

Scheduling Mechanism: The patterns application uses a flexible schedule mechanism, wherein each host can store its sending and receiving schedule. The schedule contains the action (SEND or RECEIVE) and the host id. This schedule decides the topology of communication. For example, a simple linear topology schedule for node 1 can be:

```
RECEIVE 0
SEND 0
SEND 2
RECEIVE 2
```

For each topology, a separate schedule needs to be written down for each participating host. There are functions for each topology, to generate the schedule automatically for a specified process number. For some generalized versions of topologies, finding the schedule may be non-trivial e.g. mesh, toroid and hypercube. A new algorithm was devised to schedule a n-dimensional toroid or mesh for any configuration.

The ordering of actions needs to be correct to avoid deadlock. For example, if the schedule for process 1 is (and similarly defined for other processes):

```
SEND 0
RECEIVE 0
SEND 2
RECEIVE 2
```

then all hosts will be trying to send messages in the first step but no host is waiting to receive, thus resulting in a deadlock. Thus the ordering of events is important here.

Table1.1 lists the topologies which have been implemented along with the parameters for the patterns program for each particular topology.

NAS Parallel Benchmarks [133, 148]: The NAS Parallel Benchmarks (or NPB) were developed by the Numerical Aerodynamic Simulation (NAS) Program at NASA Ames Research Center and has been widely used to evaluate the performance of various parallel computers [74]. NPB were carefully derived from real codes and mimic the computation and data movement characteristics of large-scale computational fluid dynamics (CFD) applications required by NASA. NPB consists of 8 benchmark programs. The first five kernel benchmarks (EP, MG, FT, IS, CG) represent the computational core of five frequently used numeric methods. The remaining three simuluted application benchmarks (LU, SP, BT) are representative of full-scale applications. In my work, I use a C implementation of NAS benchmarks for the PVM parallel execution environment [129].

Topology	pattern parameter	process number param.	other params
1D Mesh (a.k.a. neighbor or linear)	mesh_neighbor-1d	number of nodes	as desired
2D Mesh	mesh_neighbor-2d	number of nodes the number is split up into two factors for x and y direction	as desired
n-D Mesh	meshxxyyzz... where xx is depth of 1st dimension, yy is depth of 2nd dim, and so on... Example: mesh030203 for 3x2x3 mesh	=xx*yy*zz... 12	as desired
1D Torus	toroid_neighbor-1d	number of nodes	as desired
2D Torus	toroid_neighbor-2d	number of nodes the number is split up into two factors for x and y direction	as desired
n-D Toroid	toroidxxyyzz... where xx is depth of 1st dimension, yy is depth of 2nd dim, and so on... Example: toroid030203 for 3x2x3 toroid	=xx*yy*zz... 12	as desired
Reduction Tree	reduction_tree	number of nodes Example: 7 for a binary tree of depth 3	as desired
n-D Hypercube	hypercube_neighbor	number of nodes Example: 8 for a 3D hypercube	as desired
All to All	all-to-all	number of nodes	as desired

Table 1.1: List of topologies implemented in patterns along with required parameters

1.5 Infrastructure and Virtual Machine Model

Infrastructure Used: In our evaluations we use the nodes of our Virtuoso cluster, which is an IBM e1350 with 32 compute nodes, each of which is a dual 2.2 GHz Intel HT Xeon Processors (except the management node which is faster), 1.5 GB RAM, and 40 GB of disk.

For the evaluation in Chapter 2, each node runs Red Hat Linux 9. The communication measured is is via a 100 mbit switched network, specifically a Cisco 3550 48 port switch.

The machines also have 1 Gbit interfaces and in some evaluation scenarios in later chapters(Chapters 3, 4, 5), I use 4 of these machines which form a mini-cluster of their own. These machines are then connected via 100 Mbit switch. The machines run Fedora Core release 6 (Zod) in the evaluation for these chapters.

Virtual Machine Monitor Used: Since most of inference work is focused on virtual distributed environments, I work extensively with popular VMMs to execute parallel applications. I use both VMWare and the popular open source Xen VMM in my evaluations.

I use VMWare GSX Server 2.5 in my evaluations in Chapter 2. VMware software [109, 145, 146] provides a completely virtualized set of hardware to the guest operating system. VMware software virtualizes the hardware for a video adapter, a network adapter, and hard disk adapters. The host provides pass-through drivers for guest USB, serial, and parallel devices [9].

For my evaluation scenarious in Chapters 3, 4, 5 I use the Xen VMM. Xen [20, 33, 65] is a free software virtual machine monitor for IA-32, x86-64, IA-64 and PowerPC 970 architectures. I use Xen version 3.0.3-rc3-1.2798.f in my evaluations in Linux Kernel release 2.6.18-1.2798.fc6xen. It allows several guest operating systems to be executed on the same computer hardware at the same time.On most CPUs, Xen uses a form of virtualization known as paravirtualization, meaning that the guest operating system must be modified to use a special hypercall ABI instead of certain architectural features. Through paravirtualization, Xen can achieve high performance even on its host architecture (x86) which is notoriously uncooperative with traditional virtualization techniques [10].

Xen also offers better flexibility in extracting useful information when required. Access to source code enables a better understanding of some of its mechanisms that may be useful in creating black box approaches. Possible modification to the hypervisor code may also be possible to assist in giving us some further fine-grained information for black box inference, apart from the information readily available from a off the shelf Xen installation.

1.6 Summary of the Work Ahead and its Layout

The formalization mentioned in Section 1.3.2 is specific to the problem statement described in Ananth Sundararaj's dissertation [138]. It points out some areas where inference is an important requirement in the context of adaptation.

However the inference aspect goes far beyond that. In my dissertation I propose many interesting and useful inference problems and develop novel techniques to solve them. In the following chapters I look at the following interesting aspects of inference for BSP applications:

- **Dynamic Topology and Traffic Matrix inference:** In this chapter I investigate how can we infer the communication behavior of a parallel application, especially its topology and traffic matrix. This could then be used to adapt the underlying overlay networks to the topology for faster communication or to map the VMs encapsulating the parallel application on hosts such that the connecting links have enough residual bandwidth to accommodate the traffic matrix. [58, 61].

- **Absolute Measures of Performance:** In this chapter I propose and demonstrate black box techniques that can provide accurate proxies for the performance of typical BSP applications. I propose a new metric called the Round-trip Iteration Rate (RIR) and based on this metric I also propose some more involved representations of of dynamic performance of an application including the cumulative distribution function (CDF), Dominant

Frequencies etc that could be used for other applications as well such as dynamic performance mapping under stress, or application fingerprinting.

- **Ball in the Court Methods for Understanding Load Situations:** In this chapter I propose some novel methods using which we can understand the existing performance scenario of a single BSP application and even figure out how distressed the current application is because of local external load, in quantitative terms. These methods help improve application performance by simulating loads and predicting the change in performance thus greatly enhancing the utility in performance adaptation. I also propose some metrics that give quantitative estimates of how imbalanced different processes of a single BSP application are, which could further hint at how to improve the balance by shifting/adjusting the external loads appropriately.

- **Non-BIC metrics to compute overall time decomposition of an application:** In this chapter I further explore techniques that give a clear understanding of where the BSP application is spending its time, including time spent in the network part of the application execution, which I term as the non-BIC (Ball In the Court) delay. Along with the BIC delays computed in the previous chapter, this gives a fairly good understanding of any bottlenecks in the application that point to components and resources that are slowing down the whole application. This can be used to further diagnose the root cause and improve application performance.

In Appendices A and B, I also describe a portion of my work that demonstrate how some of these inference techniques can actually be used for runtime application performance adaptation. I discuss the various mechanisms and algorithms that I have developed in collaboration with my colleagues that can detect application demands and adapt the application using the various adaptation mechanisms to improve performance. I have published some of this work in various reputable conferences as well.

1.7 Impact

Automated understanding of a distributed application's behavior, properties and demands can further the goal of autonomic computing significantly. If we can figure out the needs without modifying the application or operating system, then a huge set of applications can be transferred to the autonomic framework of Virtuoso and thus adaptation methods can be leveraged to boost performance or adapt to resources. Overall, my work can help in drastically lowering the entry costs for distributed and parallel computing. It will allow those who are not willing or able to pay the price to write distributed applications in new shared resource or dynamic environments to deploy these applications with confidence and convenience.

1.7.1 Impact Outside Virtuoso

Most of my techniques are not tied to a particular implementation of a virtual machine, applications or operating systems. Hence this work could be used in any other virtual environment (e.g. softUDC [80] and XenoServer [46]) towards the goal of learning more about the distributed application, and adapting the application to the resources or vice versa. These techniques and the need for them is not just applicable to Virtuoso. It's equally applicable to other adaptive/autonomic systems that strive to adapt applications automatically. For example, the SODA [77] and the VIOLIN system [78], developed at Purdue University create virtual environments for creating and executing on demand applications. They allow custom configuration and adaptation of applications and resources. An understanding of the application's resource requirements can aid in this process and also help in resource mapping and allocation for future instances of the same application. Similarly in the context of Grid Computing, the In-VIGO [13] system, developed at University of Florida provides a distributed environment where multiple application instances can coexist in virtual or physical resources, such that clients are unaware of the complexities inherent to grid computing. The resource management functionality in In-VIGO [13] is responsible for creating/reserving resources to run the job based on current

available resources. For this it needs to determine the resource specification for the job(s). Application inference can automate this process instead of relying on intimate knowledge of the application or input from the user. Moreover, this process is dynamic, i.e. the resource requirements will be updated as the application demands change, which is more flexible than taking up static requirements upfront.

The VioCluster [119] project at Purdue University creates a computational resource sharing platform based on borrowing and lending policies amongst different physical domains. These policies are greatly affected by the nature of the application itself. A tightly coupled application may not be worthwhile to be spread across multiple physical domains. Thus application inference forms an integral part of making any decision towards creating autonomic resource sharing platforms.

Apart from the motivation of autonomic adaptation above, black box inference techniques can also be used for application and resource management, dynamic problem detection at runtime and intrusion detection. For example, detecting blocked states of some processes in a distributed application can lead to discovery of some serious problems in the infrastructure or the application itself, that can aid in debugging. Unusual network activity or demands could be tied to intrusion detection if they deviate from the expectations from the particular distributed application.

Chapter 2

Dynamic Runtime Inference of Traffic Matrix and Topology

An important element of the Virtuoso system is a layer 2 virtual network, VNET [134], which we initially developed to create the "networking illusion" needed for the first element of the model. It can "move" a set of virtual machines in a WAN environment to the user's local layer 2 domain. VNET also supports arbitrary overlay topologies and routing rules, passive application and network monitoring, adaptation (based on VM migration and topology/routing changes), and resource reservation. VNET is described in detail in a previous paper [134].

In Chapter 1 I introduced runtime adaptation for parallel applications. This chapter proposes and describes on one of our first steps toward achieving the last element of the model i.e. monitoring the application for the purpose of adapting it. The question I address in particular is: can we monitor, with low overhead and no application or operating system modifications, the communication traffic of a parallel application running in a set of virtual machines interconnected with a virtual network, and compute from it the traffic load matrix and application communication topology? Results in this chapter demonstrate that this is possible. We have also integrated the online implementation of our ideas, VTTIF (Virtual Topology and Traffic Inference Framework), into the VNET system.

I consider here Bulk-Synchronous Parallel [50] (BSP) style applications as introduced in

Chapter 1. The ultimate motivation behind recovering the spatial and temporal properties of a parallel application running in a virtual environment is to be able to maximize the parallel efficiency of the running application by migrating its VMs, changing the topology and routing rules of the communication network, and taking advantage of underlying network reservations on the application's behalf.

A parallel program may employ various communication patterns for its execution. A communication pattern consists of a list of all the message exchanges of a representative processor during a communication phase. The result of each processor executing its communication pattern gives us the application topology, such as a mesh, toroid, hypercube, tree, etc, which is in turn mapped to the underlying network topology [89]. In this chapter, I attempt to infer the application topology and the costs of its edges, the traffic load matrix, by observing the low-level traffic entering and leaving each node of the parallel application, which is running inside of a virtual machine.

It is important to note that application topologies may be arbitrarily complex. Although our initial results are for BSP-style applications, our techniques can be used with arbitrary applications, indeed, any application or OS that the virtual machine monitor (I use VMWare GSX server in this work) can support. However, I do not yet know the effectiveness of our load matrix and topology inference algorithms for arbitrary applications.

In general, it is difficult for an application developer, or, for that matter, the user of a "dusty deck" application, to analyze and describe his application at the level of detail needed in order for a virtual machine distributed computing system to make adaptation decisions on its behalf. Furthermore, the description may well be time or data dependent or react to the conditions of the underlying network.

The goal of VTTIF is to provide these descriptions automatically, as the unmodified application runs on an unmodified operating system. In conjunction with information from other monitoring tools, and on the policy constraints, VTTIF information will then be used to sched-

ule the VMs, migrate them to appropriate hosts, and change the virtual network connecting them. The adaptation control mechanisms will query VTTIF to understand what, from a communication perspective, the parallel application is attempting to accomplish.

I begin by offline analysis, using traffic logs of parallel applications to develop our three step monitoring and analysis process. Although this initial work was carried out without the use of VMs, using PVM applications whose traffic was captured using tcpdump techniques, it is directly applicable for two reasons. First, VNET interacts with the virtual interfaces of virtual machines in a manner identical (packet filter on the virtual interface) to how tcpdump interacts with physical interfaces (packet filter on a physical interface). Second, the physical machines generate considerably more "noise" than the virtual machines, thus making the problem harder. In Section 2.1, I describe our three step process and how it is implemented for physical monitoring. In Section 2.2 I describe a set of synthetic applications and benchmarks I will use to evaluate VTTIF. In Section A.4, I show the performance results of applying the process to a wide variety of application topologies and parallel benchmarks.

The results for the offline, physical machine-based were extremely positive, so I designed and implemented an online process that is integrated with our VNET virtual networking tool. Section 2.4 describes the design of the online VTTIF tool and provides an initial evaluation of it. I am able to recover application topologies online for a NAS benchmark running in VMs and communicating via VNET. The performance overhead of the VTTIF implementation in VNET is negligible.

In Section 2.7, I conclude by describing our plans for using the VTTIF and other monitoring information for heuristic adaptive control of the VMs and VNET to maximize application performance.

2.1 VTTIF and its Offline Implementation

The inference of parallel application communication is based on the analysis of low level traffic. I first wanted to test whether this approach was practical at all, and, if so, to develop an initial framework for traffic monitoring, analysis and inference, enabling us to test our ideas and algorithms. This initial step resulted in an offline process that focused on parallel programs running on physical hosts. In Section 2.4, I describe how these results have been extended to an online process that focuses on parallel programs running in virtual machines.

In both our online and offline work, I study PVM [48] applications. Note that the techniques described here are general and are also applicable to other parallel applications such as MPI programs. I run programs on the nodes of our Virtuoso cluster, which is an IBM e1350 with 32 compute nodes, each of which is a dual 2.2 GHz Intel HT Xeon Processors, 1.5 GB RAM, and 40 GB of disk. Each node runs Red Hat Linux 9, PVM 3.4.4, and VMWare GSX Server 2.5. Each VM runs Red Hat Linux 7.3 and PVM 3.4.4. The communication measured here is via a 100 mbit switched network, specifically a Cisco 3550 48 port switch. The nodes speak NFS and NIS back to a separate management machine via a separate network.

The VTTIF framework has three stages as shown in Figure 2.1. In the first stage, I monitor the traffic being sourced and sinked by each process in the parallel program. In the offline analysis, this is accomplished by using tcpdump on the physical interface with a packet filter that rejects all but PVM traffic. In the online analysis, I integrate monitoring into our virtual network tool VNET. VNET does the equivalent of running tcpdump on the virtual interface of the virtual machine, capturing all traffic. The Virtuoso cluster uses a switched LAN, so the interface of each node must be monitored separately and the data aggregated. A challenge in the online system is that it must decide when to start and stop this monitoring.

The second stage of the framework eliminates irrelevant traffic from the aggregated traffic and integrates the packet header traces captured by tcpdump to produce a traffic matrix, T. Element $T_{i,j}$ represents the amount of traffic sent from node i to node j. A challenge in the

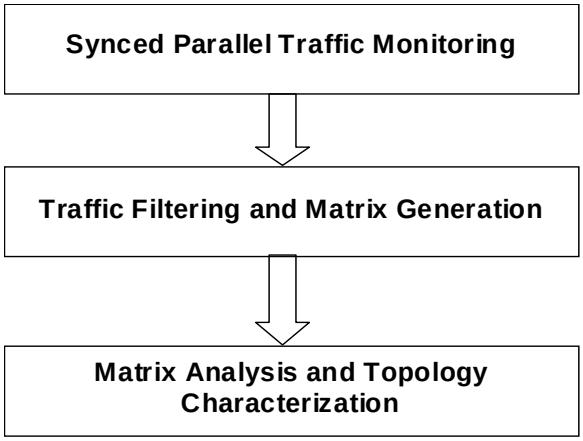


Figure 2.1: The three stages involved in inferring the topology and traffic load matrix of a parallel application

online system is to decide when to recompute this matrix.

The final stage of the framework applies inference algorithms to eliminate noise from the traffic matrix to infer from it the likely application topology. Both the original matrix and the inferred topology are then returned. The topology is displayed graphically. A challenge in the online system is to decide when to recompute the topology.

The current offline framework is designed to automate all of the above steps, allowing the user to run a single command, `infer [parallel PVM program]` This runs the PVM program mentioned in the argument, monitors it for its entire execution, completes the remaining steps of the framework, and prints the matrix and displays the topology. The framework is implemented as a set of Perl scripts, as described below.

Monitor This script is responsible for synchronized traffic monitoring on all the physical hosts, running the parallel program, and storing the packet header traces to files. The script also reads a configuration file that describes the set of hosts on which monitoring is to be done. It runs tcpdump on each of the hosts. It then executes the parallel program and waits for it to

finish. Each tcpdump stores its packet header trace output into a file named by the hostname, the date, and the time of execution. Hence, each execution produces a group of related packet header trace files.

Generate This script parses, filters and analyzes the packet header traces to generate a traffic matrix for the given hosts. It sums the packets sizes between each pair of hosts, filtering out irrelevant packets. Filtering is done according to the following criteria:

- Type of packet. Packets which are known not to be a part of the parallel program communication, like ARP, X11, ssh, etc, are discarded. This filtering has only a modest effect in the Virtuoso cluster because there is little extra traffic. However, in the future, we may want to run parallel programs in a wide area environment or one shared with many other network applications, where filtering and extracting the relevant traffic may pose extra challenges.
- The source and destination hosts involved in the packet transmission. We are only interested traffic among a specific group of hosts.

The matrix is emitted in a single file.

Infer This script infers the application topology from the traffic matrix file. In effect, topology inference amounts to taking the potentially "noisy" graph described by the traffic matrix and eliminating edges that are unlikely to be significant. The script also outputs a version of the topology that is designed to be viewed by the algorithm animation system Samba [130]. For inferring the topology, various algorithms are possible. One method is to prune all matrix entries below a certain threshold. More complex algorithms could employ pattern detection techniques to choose an archetype topology that the traffic matrix is most similar to. For the results I show, topology inference is done using a matrix normalization and simple edge pruning technique. The pseudo-code description of the algorithm is:

```
InferTopology(traffic_matrix T,pruning_threshold b_min)
{
    b_max ← max(T_i,j) ∀_i,j
    G ← ∅
    foreach(T_i,j)
    {
        r_i,j ← T_i,j / b_max
        if (r_i,j ≥ b_min)
        {
            add edge(i,j) to G
        }
    }
    return G
}
```

In effect, if the maximum bandwidth entry in T is b_{max}, then if ratio of any edge value ($T_{i,j}$) to b_{max} is below a certain threshold b_{min}, then the edge is pruned. The value of b_{min} determines the sensitivity of topology inference.

Visualization makes it very convenient to quickly understand the topology used by the parallel program. By default, we have Samba draw each topology with the nodes laid out in a circle, as this is versatile for a variety of different topologies. However, there is an option to pass a custom graph layout. An automated layout tool such as Dot could also be used.

Figure 2.2 shows an example of the final output for the program PVM POV, a parallel ray tracer, running on four hosts. The thickness of an edge indicates the amount of traffic for that particular run. Each host is represented by a different color and color of the edge represents the

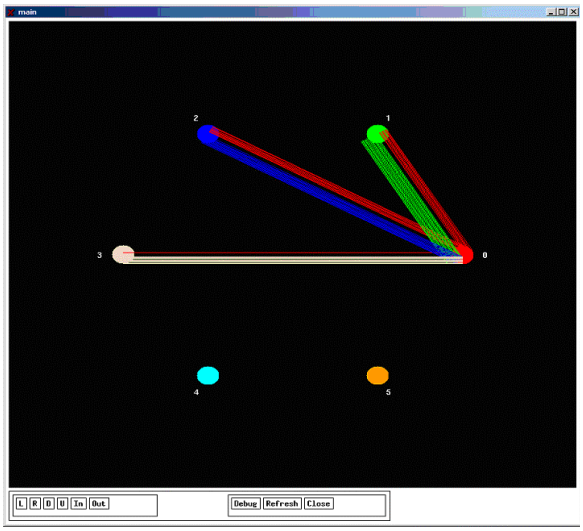


Figure 2.2: An example of the final output of the Topology Inference Framework for the PVM-POV application. The PVM-POV application runs on four hosts.

source host for the edge traffic.

2.2 Workloads for VTTIF

To test our ideas, I first needed some actual parallel applications to measure. I created and collected the following applications.

- Patterns: This is a synthetic workload generator, which I described earlier in Chapter 1. It can execute many different kinds of topologies common in BSP parallel programs. I use this extensively to test our framework.
- NAS Parallel Benchmarks: I use the PVM implementation of the NAS benchmarks [18] IS, MG, FT, and EP as developed by Sundaram, et al [149].
- PVM POV: PVM version of the popular ray tracer POVRAY. The PVM version gives it the ability to distribute a rendering across multiple heterogeneous systems. [36].

Except for patterns, these are all well known benchmark programs.

2.3 Evaluation of Offline VTTIF

I evaluated our offline inference framework with the various parallel benchmarks described in the previous section. Figure 2.3 shows the inferred application topologies of various patterns benchmark runs, as detected by our offline framework. These results suggest that there is indeed considerable promise in traffic-based topology inference: parallel program communication behavior can be inferred without any knowledge of the parallel application itself. Of course, more complex filtering processes may need to be used for more complex applications and complex network environments where parallel application traffic is just a part of the network traffic.

I also ran the application benchmarks described earlier in Chapter 1. These results are also promising. Figure 2.4 shows a representative, the traffic matrix for an execution of the Integer Sort (IS) NAS kernel benchmark on 8 physical hosts, with the corresponding topology shown in Figure 2.5. The topology resembles an all-to-all communication, but the thickness of the edges vary indicating that the bandwidth requirements vary depending on the host pairs. A closer look at the traffic matrix reveals that $host_1$ receives data in the range of 20 MB from each of the other hosts, indicating that this is a communication intensive benchmark. Other hosts $host_2$ to $host_8$ transfer data of $\simeq 10-11$ MB with each other, almost half of that exchanged with $host_1$.

Notice that this information could be used to boost the performance of the IS benchmark if it were running in our VM computing model. Ideally, we would move the VM $host_1$ to a host with relatively high bandwidth links and reconfigure the virtual network with appropriate virtual routes over the physical network [123]. Such decisions need to be dynamic, as the properties of a physical network vary [158]. Without any intervention by the application developer or knowledge of the parallel application itself, it is feasible to infer the spatial and temporal [37] properties of the parallel application. Equipped with this knowledge, we can use VM checkpointing and migration along with VNET's virtual networking capabilities to create a efficient

network and host environment for the application.

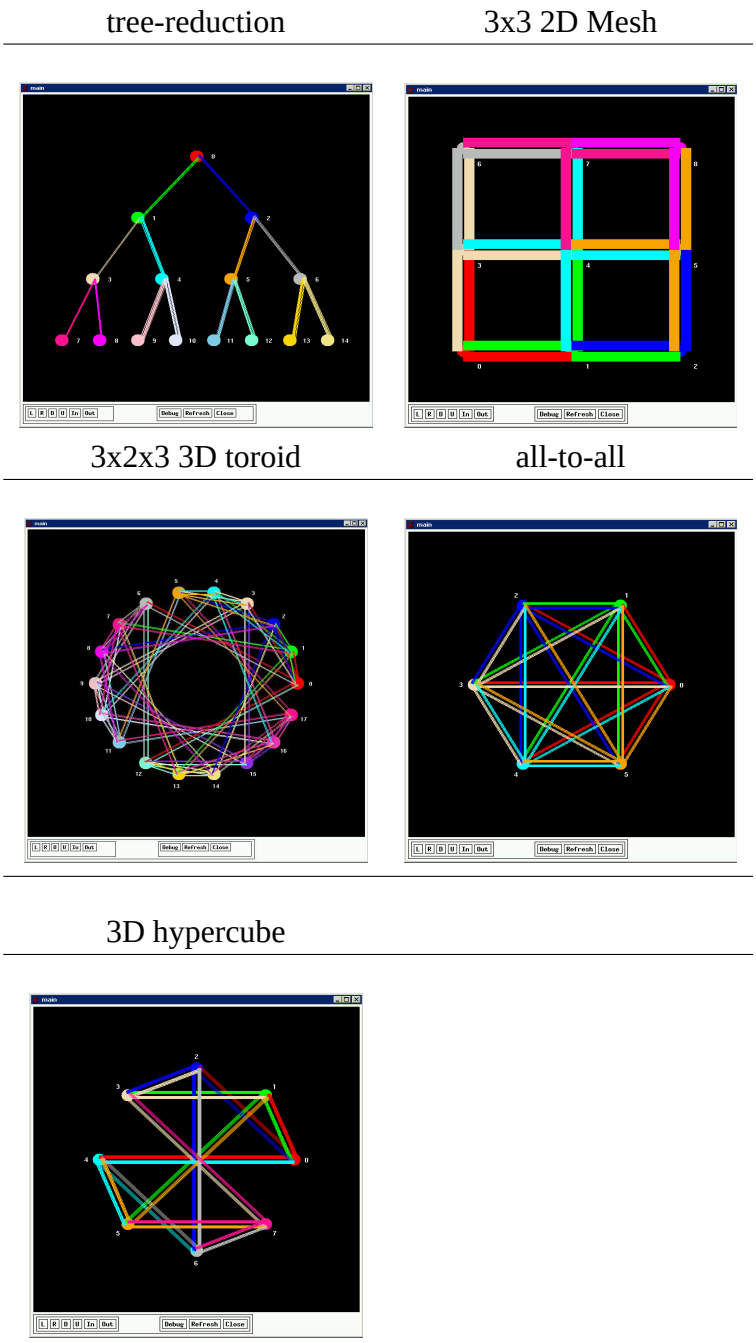


Figure 2.3: The communication topologies inferred by the framework from the patterns benchmark. It shows the inferred tree-reduction, 3x3 2D Mesh, 3x2x3 3D toroid, all-to-all for 6 hosts and 3D hypercube topologies.

	h1	h2	h3	h4	h5	h6	h7	h8
h1		19.0	19.6	19.2	19.6	18.8	13.7	19.3
h2	22.6		10.7	10.8	10.7	10.9	9.7	10.5
h3	22.2	8.78		11.2	10.4	10.1	10.5	10.5
h4	22.4	8.9	9.5		11.1	10.8	10.6	10.2
h5	22.3	10.0	9.51	9.72		11.7	10.9	11.9
h6	24.0	8.9	10.7	9.9	10.8		12.2	12.1
h7	23.2	10.0	9.7	9.5	10.3	10.2		12.0
h8	24.9	11.2	11.0	11.8	11.5	11.2	10.7	

*numbers indicate MB of data transferred.

Figure 2.4: The traffic matrix for the NAS IS kernel benchmark on 8 hosts.

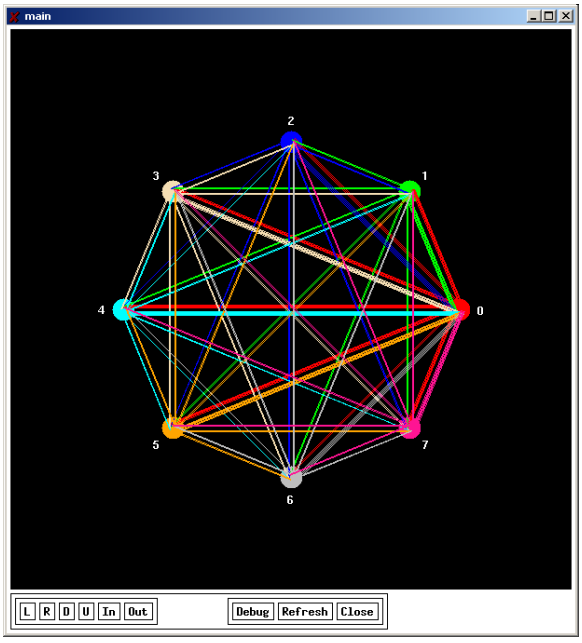


Figure 2.5: The inferred topology for the NAS IS kernel benchmark

2.4 Online VTTIF

After working with offline parallel program topology inference on the physical hosts, the next step was to develop an online framework for a virtual machine environment. I extended VNET [134], our virtual networking tool, to include support for traffic analysis and topology inference. VNET allows the creation of layer 2 virtual networks interconnecting VMs distributed over an underlying TCP/IP networking infrastructure. A VNET daemon manages all the network traffic of the VMs running on its host, and thus is an excellent place to observe the application's network traffic. All traffic monitoring is done at layer 2, providing flexibility in analyzing and filtering the traffic at many layers.

Due to a networking issue with the Virtuoso cluster, the work in the section was done on a slower cluster consisting of dual 1 GHz Pentium III processors with 1 GB of RAM and 30 GB hard disks. I used a switched 100 mbit network connecting the machines. As before VMWare GSX Server 2.5 was used, except here it was run on Red Hat Linux 7.3. The VMs were identical. A Dell PowerEdge 4400 (dual 1 GHz Xeon, 2 GB, 240 GB RAID) running Red Hat 7.1 was used as the VNET proxy machine.

2.4.1 Observing Traffic Phenomena of Interest: Reactive and Proactive Mechanisms

VMs can run for long periods of time, but their traffic may change dramatically over time as they run multiple applications in parallel or serially. In the offline VTTIF, monitoring and aggregation are triggered manually while running the parallel application. This is not possible in an online design. The online VTTIF needs a mechanism to detect and capture traffic patterns of interest, reacting automatically to interesting changes in the communication behavior of the VMs. It must switch between active states, when it is accumulating data and computing topologies, and passive states, when it is waiting for traffic to intensify or otherwise become relevant. Ideally, VTTIF would have appropriate information available whenever a scheduling

agent requests it.

I have implemented two mechanisms for detecting interesting dynamic changes in communication behavior: reactive and proactive. In the reactive mechanism, VTTIF itself alerts the scheduling agent when it detects certain pre-specified changes in communication. For example, in the current implementation, VTTIF monitors the rate of traffic for all flows passing through it and starts aggregating traffic information whenever the rate crosses a threshold. If this rate is sustained, then VTTIF can alert the scheduling agent about this interesting behavior along with conveying its local traffic matrix.

In the proactive mechanism, VTTIF allows an external agent to make traffic-related queries such as: *what is traffic matrix for the last 512 seconds?* VTTIF stores sufficient history to answer various queries of interest, but it does not alert the scheduling agent, unlike the reactive mechanism. The agent querying traffic information can determine its own policy, for example polling periodically to detect any traffic phenomena of interest and thus making appropriate scheduling, migration and network routing decisions to boost parallel application performance. Figure 2.6 shows the high level view of the VNET-VTTIF architecture. The VM and overlay network scheduling agent may be located outside the VM-side VNET daemon, and all relevant information can conveyed to it so that it can make appropriate scheduling decisions.

2.4.2 Implementation

I extended VNET so that each incoming and outgoing Ethernet packet passes through a packet analyzer module. This function parses the packet into protocol headers (Ethernet, IP, TCP) and can filter it if it is irrelevant. Currently all non-IP packets are filtered out—additional filtering mechanisms can be installed here. Packets that are accepted are aggregated into a local traffic matrix. Specifically, for each flow, a row and column of the matrix are determined in this way. The matrix is stored in a specialized module TrafficMatrix. TrafficMatrix is invoked on every packet arrival.

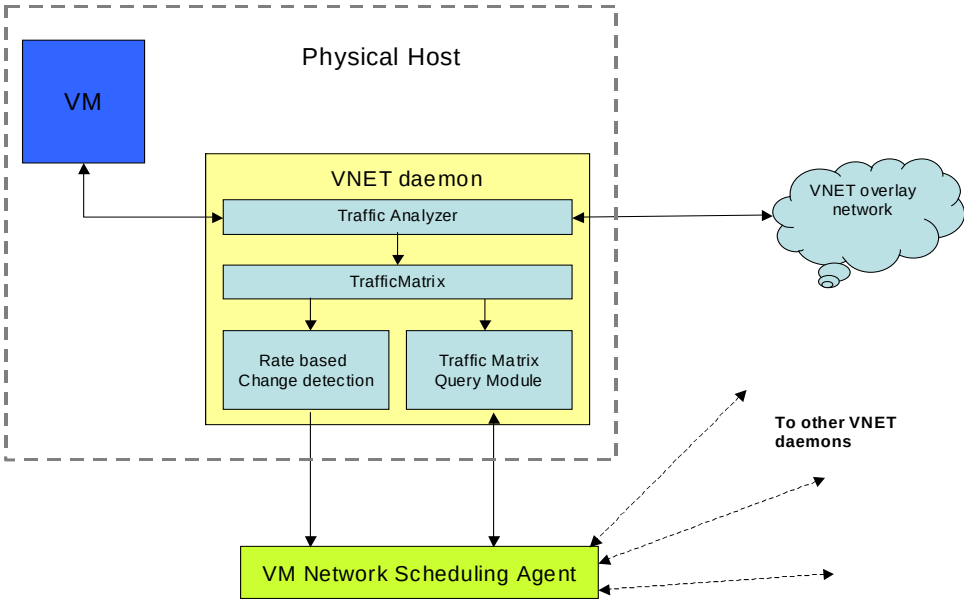


Figure 2.6: The VNET-VTTIF topology inference architecture. VTTIF provides both reactive and proactive services for the scheduling agent.

Reactive mechanism The TrafficMatrix module does non-uniform discrete event sampling for each source/destination VM pair to infer the traffic rate between the pair. The functioning of `rate_threshold` mechanism is illustrated in Figure 2.7. It takes two parameters: `byte_threshold` and `time_bound`. Traffic is said to cross the `rate_threshold`, if for a particular VM pair, `byte_threshold` bytes of traffic is transmitted in a time less than `time_bound`. This is detected by time-stamping the packet arrival event whenever the number of transmitted bytes for a pair exceeds a integral multiple of `byte_threshold`. If two successive time-stamps are less than `time_bound`, this indicates our `rate_threshold` requirement has been met. Once a pair crosses the `rate_threshold`, TrafficMatrix starts accumulating traffic information for all the pairs. Before the `rate_threshold` is crossed, TrafficMatrix doesn't accumulate any information, i.e. it is a memoryless system. After the `rate_threshold` is crossed, TrafficMatrix alerts the scheduling agent in two situations. First, if the high traffic rate is sustained up to

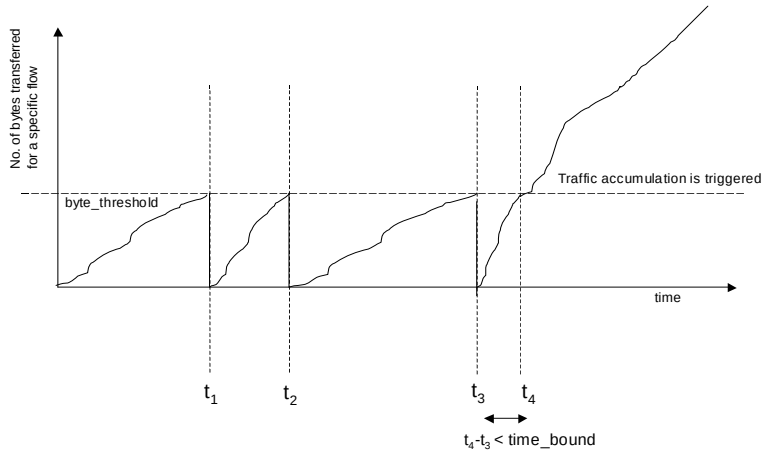


Figure 2.7: The rate-threshold detection based reactive mechanism in VNET-VTTIF. Whenever two successive byte thresholds are exceeded within a time bound, the accumulation of traffic is triggered.

time t_{max}, then it sends all its traffic matrix information to the scheduling agent. In other words, TrafficMatrix informs the scheduling agent if an interesting communication behavior persists for a long enough period of time. The second situation is if the rate falls below the threshold and remains there for more than t_{wait} seconds, in which case TrafficMatrix alerts the scheduling agent that the application has gone quiet.

Figure 2.8 illustrates the operation of the reactive mechanism in flowchart form.

Proactive mechanism The proactive mechanism allows an external agent to pose queries to VTTIF and then take decisions based on its own policy. VTTIF is responsible solely for providing the answers to useful queries. TrafficMatrix maintains a history for all pairs it is aware in order to answer queries of the following form: What is the traffic matrix over the last n seconds? To do so, it maintains a circular buffer for all pairs in which each entry corresponds to the number of bytes transferred in a particular second. As every packet transmission is reported to TrafficMatrix, it updates the circular buffer for the particular pair. To answer the query, the

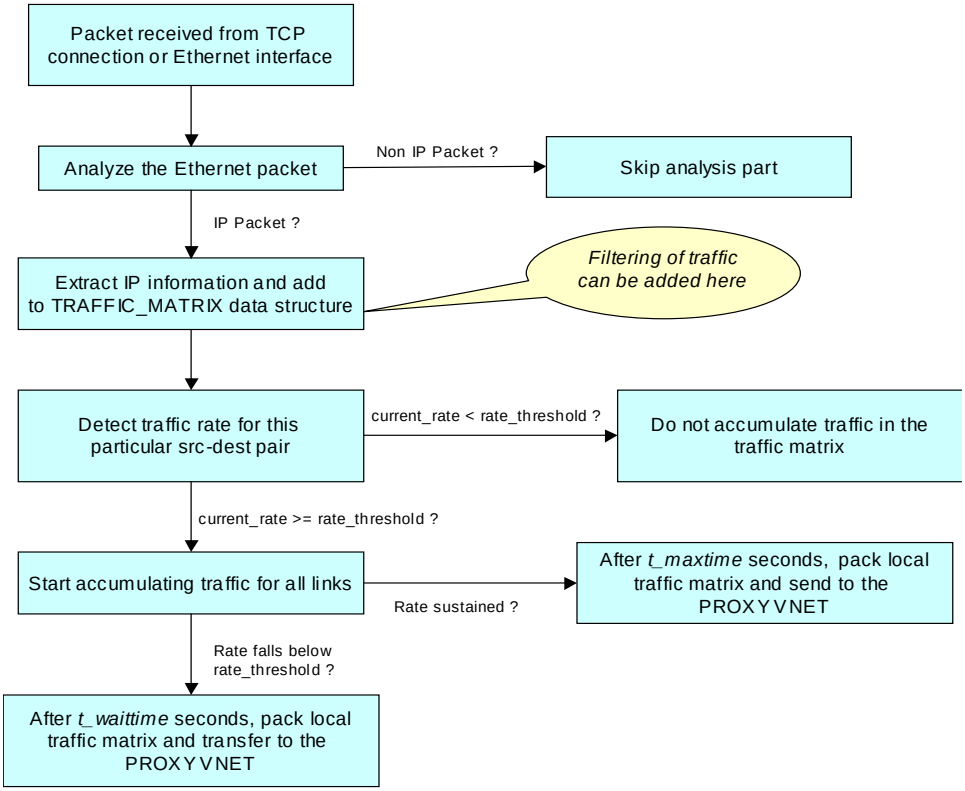


Figure 2.8: The steps taken in the VM-side VNET daemon for the reactive mechanism.

last n entries are summed up, up to the size of the buffer.

The space requirements for storing the state history needs some consideration. The space requirements depends on the maximum value of n. For each pair, $4n$ bytes are needed for the circular buffer. If there are m VMs, then the total space allocation is $4nm^2$. For $n = 3600$ (1 hour) and $m = 16$ VMs, the worst case total space requirement is 3.7 Mbytes. A sparse matrix representation could considerably reduce this cost and thus the communication cost in answering the queries.

2.4.3 Aggregation

Aggregation of traffic matrices from the various VNET daemons provides a global view of the communication behavior exhibited by the VMs. Currently, I aggregate the locally collected traffic matrices to a global, centralized matrix that is stored on the VNET proxy daemon, which is responsible for managing the virtual overlay network in VNET. I use a push mechanism—the VNET daemons decide when to send their traffic matrix based on their reactive mechanism. A pull mechanism could also be provided, in which the proxy would request traffic matrices when they are needed based on queries.

The storage analysis of the previous sections assumes that we will collect a complete copy of the global traffic matrix on each VNET daemon—in other words, that we will follow the reduction to the proxy VNET daemon with a broadcast to the other VNET daemons. This is desirable so that the daemons can make independent decisions. However, if we desire only a single global copy of the whole matrix, or a distributed matrix, the storage and computation costs will scale with the number of VMs hosted on the VNET daemon.

Scalability is an issue in larger instances of the VM-based distributed environment. Many possibilities exist for decreasing the computation, communication and storage costs of VTTIF. One optimization would be to maintain a distributed traffic matrix. Another would be to implement reduction and broadcast using a hierarchical structure, tuned to the performance of the underlying network as in ECO [96]. Fault tolerance is also a concern that needs to be addressed.

2.4.4 Performance Overhead

Based on our measurements, VTTIF has minimal impact on bandwidth and latency. I considered communication between two VMs in our cluster, measuring round-trip latency with ping and bandwidth with ttcp. Figure 2.9 compares the latency between the VMs for three cases:

- Direct communication. Here VNET is not involved. The machines communicate locally using VMWare's bridged networking. This measures the maximum performance achiev-

Method	Average	STDEV	Min	Max
Direct	0.529 ms	0.026 ms	0.483 ms	0.666 ms
VNET	1.563 ms	0.222 ms	1.277 ms	2.177 ms
VNET-VTTIF	1.492 ms	0.198 ms	1.269 ms	2.218 ms

Figure 2.9: Latency comparison between VTTIF and other cases

Method	Throughput
Direct	11485.75 KB/sec
VNET	8231.82 KB/sec
VNET-VTTIF	7895.06 KB/sec

Figure 2.10: Throughput comparison between VTTIF and other cases

able between the hosts, without any network virtualization.

- VNET. Here I use VNET to proxy the VMs to a different network through the PowerEdge 4400. This shows the overhead of network virtualization. Note that I am using an initial version of VNET here without any performance enhancements running on a stock kernel. I continue to work to make VNET itself faster.

- VNET-VTTIF. This case is identical to VNET except that I am monitoring the traffic using VTTIF.

There is no significant difference between the latency of VNET and VNET-VTTIF.

Figure 2.10 shows the effect on throughput for the three cases enumerated above. These tests were run using ttcp with a 200K socket buffer, and 8K writes. The overhead of VNET-VTTIF compared to VNET is a mere 4.1%.

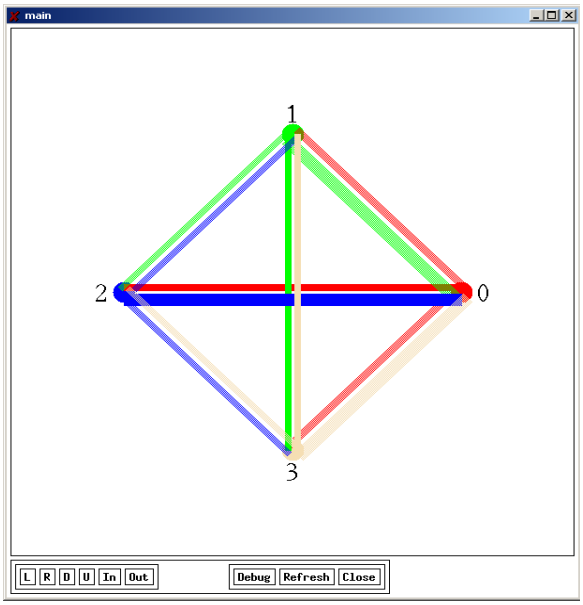


Figure 2.11: The PVM IS benchmark running on 4 VM hosts as inferred by VNET-VTTIF

2.4.5 Online VTTIF in Action

Here I show results of running a parallel program in the online VNET-VTTIF system. I use the NAS Integer Sort (IS) benchmark for illustration because of its interesting communication pattern and traffic matrices. I executed NAS IS on 4 VMs interconnected with VNET-VTTIF. Here, the Virtuoso cluster, as used in the offline work, was employed. The rate-based reactive mechanism was used to intelligently trigger aggregation mechanisms on detecting traffic flow from the benchmark. When the benchmark finished executing, the traffic matrix was automatically aggregated at the VNET proxy. For comparison, I also executed the same benchmark with on 4 physical hosts and analyzed the traffic using the offline method.

Figures A.4 and 2.12 show the topology and traffic matrix as inferred by the online system. Figure 2.13 shows the matrix inferred from the physical hosts using the offline method. The topology for the offline method is identical to that for the offline method and is not shown. There

	h1	h2	h3	h4
h1		7.7	7.6	7.8
h2	13.1		6.6	6.5
h3	13.5	6.4		6.6
h4	13.2	6.5	6.5	
*numbers indicate MB of data transferred.				

Figure 2.12: The PVM IS benchmark traffic matrix as inferred by VNET-VTTIF

	h1	h2	h3	h4
h1		5.1	5.0	5.0
h2	4.5		4.3	3.8
h3	4.7	3.9		3.8
h4	4.5	3.9	3.9	
*numbers indicate MB of data transferred.				

Figure 2.13: The PVM IS benchmark traffic matrix running on physical hosts and inferred using the offline method.

are some differences between the online and offline traffic matrices. This can be attributed to two factors. First, the byte count in VNET-VTTIF includes the size of the entire ethernet packet whereas in the offline method, only the TCP payload size is taken into account. Second, tcpdump, as used in the offline method, is configured to allow packet drops by the kernel packet filter. In the online method, VNET's packet filter is configured not to allow this. Hence, the offline method is seeing a random sampling of packets while the online method is seeing all of the packets.

The main point here is that the online method (VNET-VTTIF) can effectively infer the application topology and traffic matrix for a BSP parallel program running in a collection of

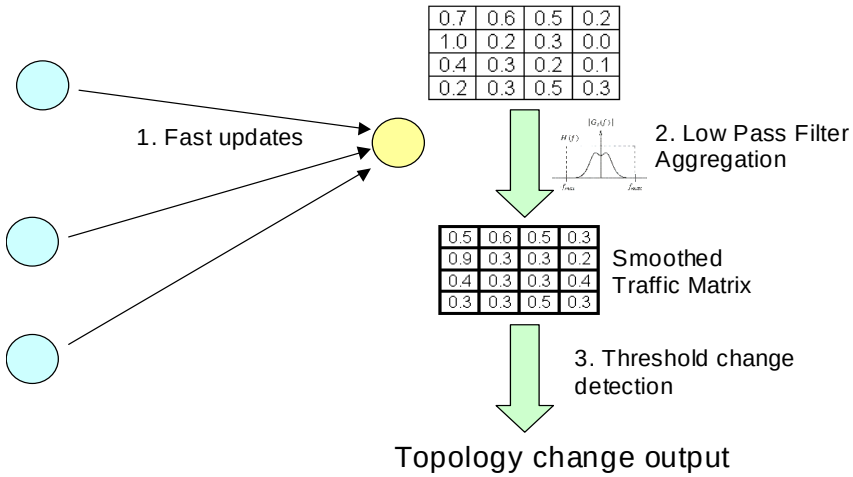


Figure 2.14: An overview of the dynamic topology inference mechanism in VTTIF.

VMs.

2.5 Dynamic Traffic Conditions

In previous sections I discussed how VTTIF can be used to recover topologies under largely static conditions where the topology behavior of the application may not change abruptly with time. However in practice, the communication behavior does change, especially for more complex BSP applications which may consist of multiple phases during the entire execution of the application. Thus decisions made at one stage may become invalid and a new adaptation strategy based on the new topology may be needed as per the dynamic run time behavior. Figure 2.14 illustrates the overall operation of VTTIF under dynamic conditions.

Operation of VTTIF under highly dynamic conditions VTTIF runs continuously, updating its view of the topology and traffic load matrix among a collection of Ethernet addresses being supported by VNET. However, in the face of dynamic changes, natural questions arise: How fast can VTTIF react to topology change? If the topology is changing faster than VTTIF can react, will it oscillate or provide a damped view of the different topologies? VTTIF also depends

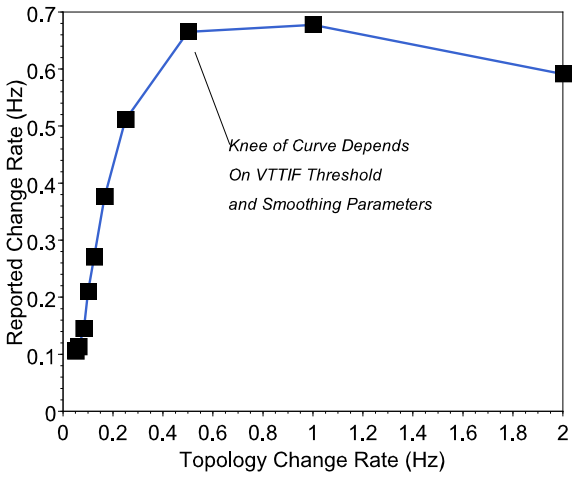


Figure 2.15: VTTIF is well damped.

on certain configuration parameters which affect its decision whether the topology has changed. How sensitive is VTTIF to the choice of configuration parameters in its inference algorithm?

The reaction time of VTTIF depends on the rate of updates from the individual VNET daemons. A fast *update rate* imposes network overhead but allows a finer time granularity over which topology changes can be detected. In our current implementation, at the fastest, these updates arrive at a rate of 20 Hz. At the central VNET deamon, VTTIF then aggregates the updates into a global traffic matrix. To provide a stable view of dynamic changes, it applies a low pass filter to the updates, aggregating the updates over a sliding window and basing its decisions upon this aggregated view.

Whether VTTIF reacts to an update by declaring that the topology has changed depends on the *smoothing interval* and the *detection threshold*. The smoothing interval is the sliding window duration over which the updates are aggregated. This parameter depends on the adaptation time of VNET, which is measured at startup, and determines how long a change must persist before VTTIF notices. The detection threshold determines if the change in the aggregated global traffic matrix is large enough to declare a change in topology. After VTTIF determines that a topology has changed, it will take some time for it to settle, showing no further topology

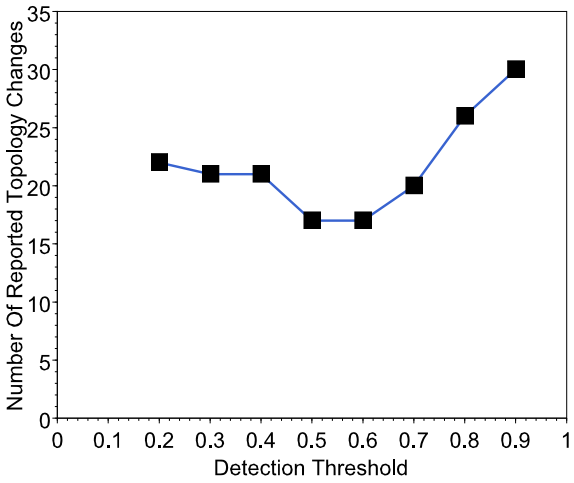


Figure 2.16: VTTIF is largely insensitive to the detection threshold.

changes. The best case settle time that I have measured is one second, on par with the adaptation mechanisms.

Given an update rate, smoothing interval, and detection threshold, there is a maximum rate of topology change that VTTIF can keep up with. Beyond this rate, I have designed VTTIF to stop reacting, settling into a topology that is a union of all the topologies that are unfolding in the network. Figure 2.15 shows the reaction rate of VTTIF as a function of the topology change rate and shows that it is indeed well damped. Here, I are using two separate topologies and switching rapidly between them. When this topology change rate exceeds VTTIF's configured rate, the reported change rate settles and declines. The knee of the curve depends on the choice of smoothing interval and update rate, with the best case being $\sim$1 second. Up to this limit, the rate and interval set the knee according to the Nyquist criterion.

VTTIF is largely insensitive to the choice of detection threshold, as shown in Figure 2.16. However, this parameter does determine the extent to which similar topologies can be distinguished. Note that appropriate settings of the VTTIF parameters are determined by the *adaptation mechanisms*, not the *application*.

2.6 Using VTTIF-based Inference for Performance Adaptation

Inference work from this chapter has been widely used in some of my work in the area of automated runtime performance adaptation, that I have done in the past with my colleagues. This work is described in Appendices A and B. The work described in Appendix A(Increasing Application Performance In Virtual Environments through Run-time Inference and Adaptation [135, 139]) shows how to use the continually inferred application topology and traffic to dynamically control three mechanisms of adaptation, VM migration, overlay topology, and forwarding to significantly increase the performance of two classes of applications, bulk synchronous parallel applications and transactional web ecommerce applications.

The work in Appendix B (Free Network Measurement For Adaptive Virtualized Distributed Computing [61]) demonstrates the feasibility of free automatic network measurement by fusing the Wren passive monitoring [155] and analysis system with Virtuoso's virtual networking system. We explain how Wren has been extended to support online analysis, and we explain how Virtuoso's adaptation algorithms have been enhanced to use Wren's physical network level information to choose VM-to-host mappings, overlay topology, and forwarding rules.

These projects serve as great evidence of the utility of black box inference and how properties derived from such inference can guide the application of runtime adaptation for improved performance of applications without any manual intervention or specific knowledge of the application.

2.7 Conclusions and Future Work

In this chapter I demonstrated that it is feasible to infer the topology and traffic matrix of a bulk synchronous parallel application running in a virtual machine-based distributed computing environment by observing the network traffic each VM sends and receives. I have also designed

and implemented an online framework (VTTIF) for automated inference in such an environment. This monitoring can be piggy-backed, with very low overhead, on existing, necessary infrastructure that establishes and optimizes network connectivity for the VMs.

Chapter 3

Black Box Measures of Absolute Performance

The performance of a BSP application is a major concern in my scenarios. Much research has been conducted to identify ways of improving BSP application performance by both modification of the application itself via algorithms, profiling etc and the external environment. For example our earlier work of virtual runtime adaptation of a BSP application has the performance of the application specifically as the objective parameter to optimize.

However this goal can be very difficult to achieve in general if there is no way to measure the performance of the application itself. Some questions that arise are: What is the metric of performance? How do we measure it? Is it possible estimate performance without having any specific knowledge or modification of the application? Affirmative solutions to these questions would be powerful tools for adaptation, because now the adaptation mechanisms and algorithms could actually ascertain the results of their decisions and quantify the effectiveness of any adaptation decisions taken. Apart from predicting run-times and judging the effectiveness of the algorithms. Thus it could be used to further improve the adaptation mechanisms and algorithms themselves by understanding over time what decisions result in increased performance and slowly incorporating them in a learning system.

For example, my colleagues Bin Lin and Ananth Sundararaj have developed a system [93]

that takes as input a target execution rate for each application, and automatically and continuously adjusts the applications' realtime schedules to achieve those rates with proportional CPU utilization. However currently the execution rate is explicitly provided by the application. Being able to derive the execution rate without application modification would boost the utility of the system.

This chapter shows that these questions can indeed be answered in the affirmative-thus resulting in a black box understanding of BSP application performance.

The specific contributions in this chapter include the following:

1. An understanding of the correlation between network traffic and inter-process interactions in a BSP application/
2. A definition of an absolute measure of application performance and a set of ways of measuring it.
3. A sampling theory based approach to dealing with dynamic situations where application performance is variable.
4. The application of frequency domain analysis to get an even greater insight into the application including macro phase-structure of the BSP application.
5. A potentially useful way to fingerprint BSP applications and match similar processes in a network without any internal knowledge. This knowledge can be leveraged in scheduling decisions or classifying applications.

3.1 Measures of BSP Application Performance

BSP application performance has been always been a major concern while designing, implementing and deploying parallel applications. The primary purpose of a BSP application itself is

to split the work to speed up the execution, hence it's natural that speed is a major concern. In this section we examine how the performance of BSP applications is usually quantified.

A BSP application typically consists of a sequence of supersteps. As described in [72], each super-step can decomposed into three phases. The computation phase where computation is done on the data held locally. Secondly the communication phase where the processors communicate with each other. Thirdly, the synchronization phase where all processors sync with each other signalling the end of the superstep.

3.1.1 Cost model for BSP Applications

There have been various ways to talk about BSP application performance. One popular way is to break each application super-step into its constituent parts:

1. The computational cost of the superstep
2. Communication cost of the global exchange of the data
3. Synchronization cost

The total execution cost then be computed by combing the costs of these individual constituent parts. From [72], the cost can be expressed as the expression:

$$ExecutionTime = (\alpha + \beta g + \gamma l)/s \tag{3.1}$$

Here γ is the number of super-steps, α is the total cost of computation over all super-steps and β is the total communication cost. There are the application dependent costs. The other three parameters depend on the environment and architecture. Here s is the speed of computation in flops, l is the sync latency cost in units of s and g indicates the communication cost for all processors to do so simultaneously.

This is a brief summary of the cost model, and more detail is discussed in calculating more intricate parameters like g in the cited literature. However the gist is the breakdown of the

application performance into its constituent parts.

3.1.2 Super-step or Iteration Rate

One important property of the BSP application is its breakdown into a sequence of super-steps. For example, Figure 3.1 shows how the Multi-Grid (MG) application from the NAS benchmarks consists of several phases consisting of communication and computation. Typically for a given application, the super-step structure is an invariant, though the actual number of steps may vary depending on input data and parameters. So one possible way to talk about BSP application performance is the number of super-steps executed per second for example. This can sometimes be also called the iteration rate - if the BSP application is a simple loop consisting of a computation and communication phase, then the number of iterations executed per second can serve an indicator of performance. If the total number of iterations is known in advance, then the total execution time of the application can also be determined. This is a simpler approach than the cost model of BSP outlined previously.

In practice, however for complex, BSP applications like MG, owing to various phases of the application, the number of super-steps executed per second is highly variable, thus making it difficult to output a single measure of performance. This dynamic nature needs to be incorporated into the performance metric somehow to enable an accurate comparison between different executions of the same application.

3.2 Determining Performance in a Black Box Scenario

Now the question that arises is that in what way can we quantify the performance of a BSP application, without actually having any knowledge of the application itself? The goal is to be able to measure some rate of execution of the application, that can then be used to determine the total execution time of the application or to compare the performance of the application running under different instances.

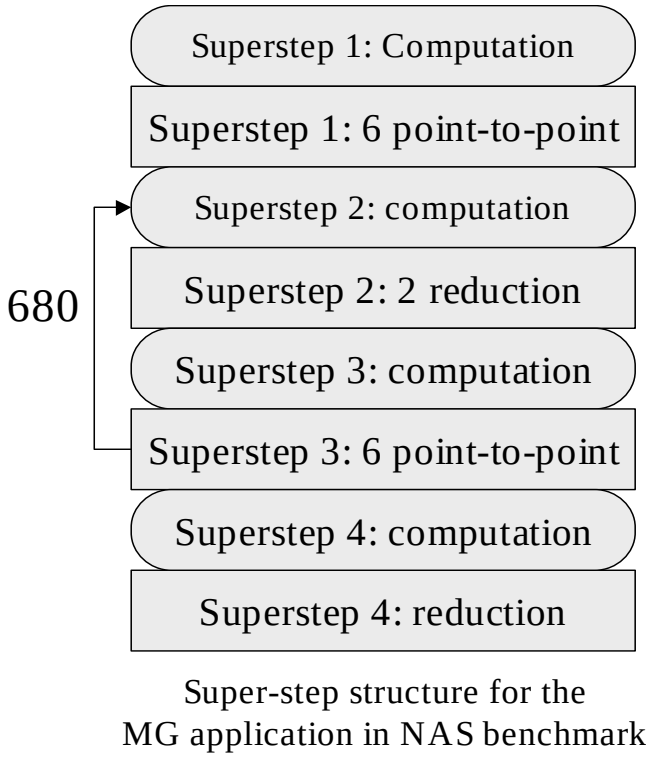


Figure 3.1: The super-step structure of the MG application

One possible way is to determine the parameters presented in Equation 3.1 and then compute the total execution cost. However Equation 3.1 is a fairly intricate model and it's not clear if its even possible to determine these parameters. Determining the total number of super-steps of an unknown application, the synchronization cost and the abstract communication cost per super-step seems very difficult without actually profiling the application and having some knowledge about it. Then there are the architecture dependent parameters that must be determined separately. These complications make a black box approach difficult.

In my experiments with the BSP applications, I have discovered and investigated a novel approach to provide some metrics for application performance. My methodology for determining black box performance takes a super-step or iteration based approach. However in a black box mode, its extremely difficult to identify the super-steps of a BSP application. Moreover, for more complex BSP applications like the NAS benchmarks, the super-step structure can be complex as in Figure 3.1. Hence the number of super-steps executed per second itself can be highly dynamic. Thus a distribution or a more intricate approach is needed to quantify the performance of these applications in usable terms.

Instead of directly working with the super-steps inside the application, I define a new metric called *RTT iteration rate* (abbreviated as *RIR*) that is based on the communication behavior of the BSP application. RIR is directly correlated to the iteration rate of the application or the number of super-steps executed per second, so it acts as a good proxy for performance. This metric is based on the fact that multiple processes interact with each other in a BSP application to carry out a task co-operatively. It is correlated to the number of interactions the processes have each other. To better explain the background behind this metric I provide some insight into some application behavior in the next section and the intuition behind this metric. Then I define this metric in Section 3.4.

An Evaluation Note: As mentioned in Chapter 1, I have used both VMware and Xen VMMs for evaluation. From this chapter onwards, I will be using Xen as the VMM for reasons

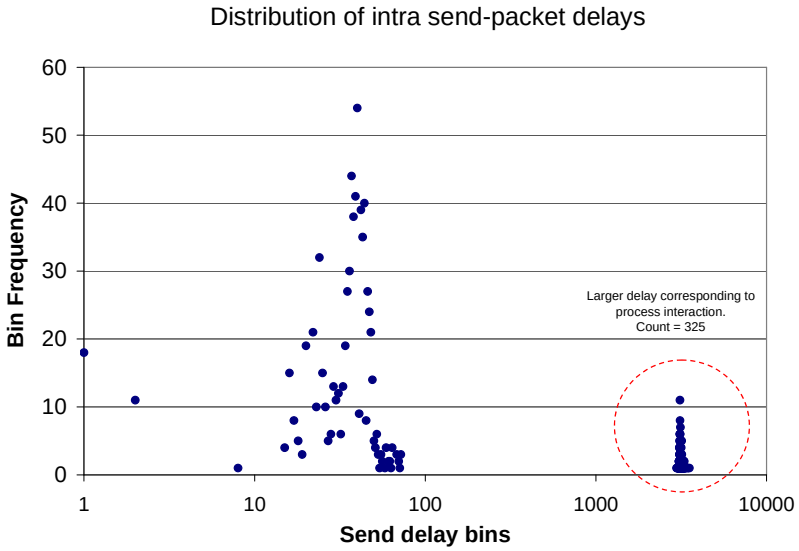
mentioned earlier.

3.3 A Look at the Traffic of a Simple BSP Application

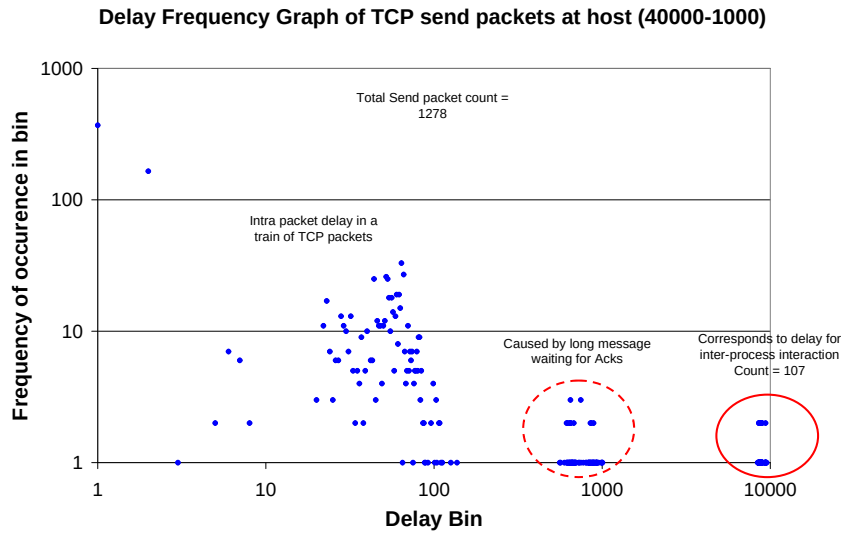
In my initial investigation into identifying ways of measuring BSP application performance, I tried looking at the various traces available to me from various tools. These tools include XenMon [62, 63], XenTop [11], tcpdump [8] etc. When examining the tcpdump traces especially from the communication between the BSP processes, I examined various properties of the trace. One particular property, the *inter-send packet delay* exhibits interesting properties.

I plotted the distribution of the delays between successive sends from one of the processes to others. Figure 3.2(a) plots the distribution of inter send-packet delays for a simple instance of the Patterns benchmark. The Pattern benchmark was run inside a Xen VM on 2 separate hosts, with a per iteration message size of 4000 bytes and computation per iteration of 100 MFlops. As mentioned earlier, Patterns runs a simple iteration loop with a computation phase followed the communication phase. As the figure shows, the inter send-packet delay exhibits strong clustering behavior around different delays. After examination of the trace, the explanation of the clusters is intuitive. The clusters actually belong to two phenomenon: the first one being the inter send-packet delay for the packets belonging to a single message being pumped out of the host and the second cluster corresponds to the delay involved when inter-process interaction is involved. At the end of each iteration, there is computation and a receive phase or vice versa depending on the process. So it atleast involves a round-trip plus other dynamics including computation time, which is exhibited by the second cluster. This behavior gives a hint that we can derive some useful properties about the application by observing its inter send-packet delay properties.

The second and the more interesting part of this observation is counting the number of members of the second cluster. The Patterns program is configured to output a report at the end of execution where it outputs various statistics that are profiled during its execution. When



(a) Patterns run with message size = 4000, compute flops = 1000 per iteration



(b) Patterns run with message size = 40000, compute flops = 1000 per iteration

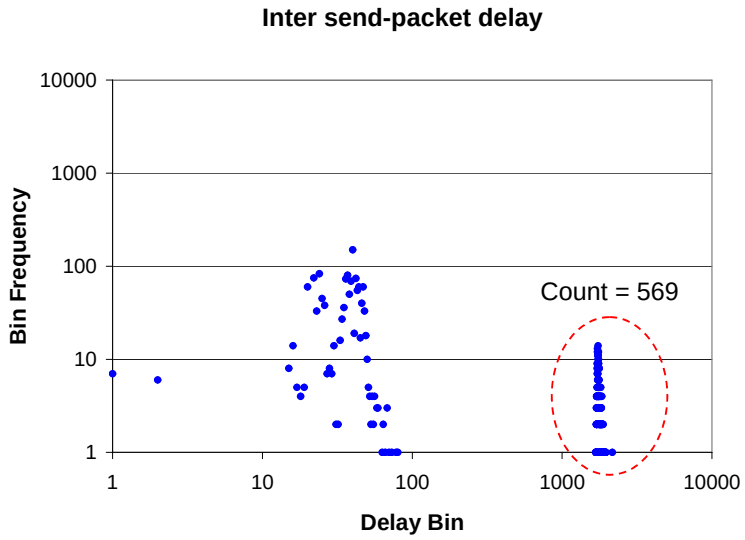
Figure 3.2: This figure shows the inter send-packet delay varies for the simple Patterns benchmark. For the relatively simplistic benchmark, the delay clusters into two groups. The duration of time is 1 second.

we look at the iteration rate for this instance, the number of iterations reported per second on the average are 325.493. The total time of the run was 9.2 seconds, with 4.55 seconds going to computation and rest going to doing communication or blocking. Now when we count the number of items in second cluster, they come out be exactly 325, which is actually the true iteration rate of the application. For this simple case, we were actually able to determine the internal iteration rate using black box means! And it gives us some interesting insight into how the inter send-packet behavior can be leveraged to measure and indicate application performance.

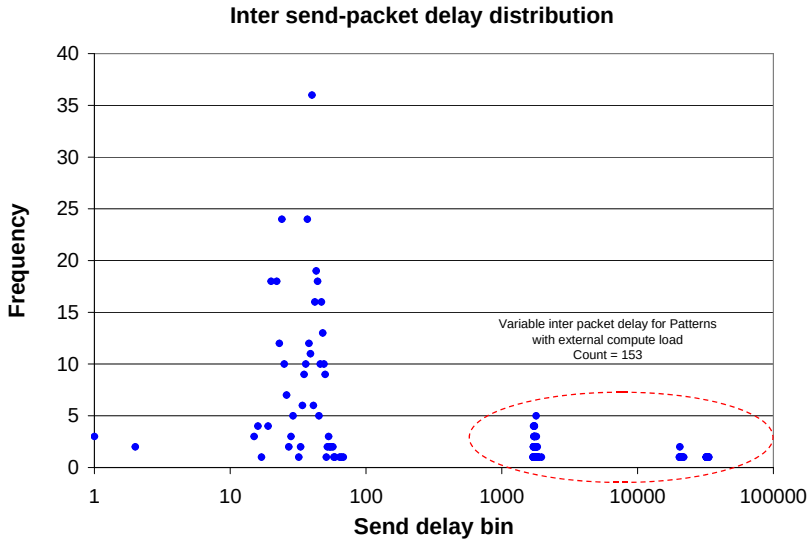
Figure 3.2(b) shows the same graph but for a much larger message size of 40,000 bytes per iteration. Here we see that the delay actually splits into more than 2 clusters. On examining the trace, the middle cluster is found to correspond to the round trip delay involved when the sender is waiting for the acknowledgement before sending more packets belonging to the current message. This may be due to either congestion or flow control mechanisms of TCP. The last cluster is found to correspond to the inter-process interaction phase where receiving, computation and other dynamics are involved. Again, the average iteration rate per second reported by the program itself is 106.57. The number of items in the last cluster are 107. Thus again, with a bit more diligence (because of the extra clusters involved), we can infer the true iteration rate of this relatively simple BSP benchmark.

What the above discussion shows is the promise of this approach to glean some performance behavior of the application with significant accuracy. Intuitively, the fact that processes need to interact with each other in a BSP application is leveraged quite nicely and cleanly when the inter send-packet delay is examined.

Figure 3.3 shows what happens to this behavior when the same application is run under different conditions. Figure 3.3(a) shows the delay distribution when the Patterns benchmark with the parameters shown, is run without any load. The last cluster accurately infers the iteration rate to be 569 iterations per second, where the actual reported number is 567.75. The



(a) Patterns run with message size = 4000, compute flops = 100 per iteration, with no load



(b) Patterns run with message size = 4000, compute flops = 100 per iteration, with full computational load

Figure 3.3: This figure shows the effect of external load on the inter send-packet delay. The clusters break up into more than one in the loaded case, showing more variable delay effects for inter-process interaction. The duration of time is 1 second.

total runtime of the application was 17.5947 seconds, with 2.15 seconds going to computation. Figure 3.3(b) shows the same graph, however one of the hosts is stressed by another Xen Guest VM that is running a very compute intensive process that is basically doing infinite computation. Under normal circumstances it soaks up 100% of the CPU. Here we see three distinct clusters. Counting the last two clusters, we get a count of 153. The actual reported iteration rate per second by the application is 142.27. The actual runtime of the application was around 69 seconds. The ratio of the execution time for the application under and without external load is thus $69/17.59 = 3.922$. The ratio of the reported iteration rate is 3.72. Thus the performance indicator derived from the send behavior is actually quite close to the total execution time of the application, indicating the great usefulness of this approach.

3.3.1 Examining Inter Send-Packet Delay for a Complex BSP Application

The previous results gave great intuition and results for a simple BSP benchmark that implements a basic BSP model. How does the distribution of these delays look for a more complex application? Figure 3.4 answers this question. It shows the send behavior delay distribution for the MG (Multi-grid) benchmark from the NAS package, under different load conditions. First of all we see (as somewhat expected) that the distribution of the delays is less separated and with a larger spread compared to Patterns. Since MG consists of multiple complex phases, a lot of different kinds of communication is going on behind the scenes along with other dynamics. Secondly we see the effect of load on the delay distribution. The distribution shifts towards the right indicating higher inter packet delays with the external load as expected. What do these results indicate for black box estimation of performance? First of all it's difficult to expect a single iteration rate per second out of MG as the actual number of super-steps happening per second are quite different as earlier described in Figure 3.1. Secondly the unclear separation of clusters may not lend itself to the clean counting approach mentioned previously.

In the next section, I propose a new metric for application performance called the RTT

Iteration Rate (RIR) that does actually works well for more complex applications. Later on, I further describe some more derivative metrics from the base RIR metric, that are needed to describe performance for more dynamic BSP applications whose performance may not remain constant as they exhibit more complex superstep structures.

3.4 Defining RTT Iteration Rate (RIR)

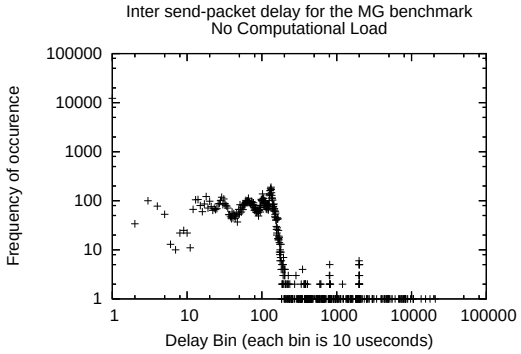
In the previous section, we saw how we could infer the application performance behavior for simple cases with a great level of accuracy, by looking its traffic, specifically at its inter send-packet delay behavior. For simpler cases like Patterns, the approach was quite accurate. However, for more complex applications like MG, the behavior is not so clean. There seems to be no obvious similar way to infer the performance of the MG application. To discover more avenues for black box performance inference, I further investigated into other potential options for indicating application performance and have discovered a useful black box metric that serves as a good proxy of the application performance.

3.4.1 RTT Iteration Rate (RIR)

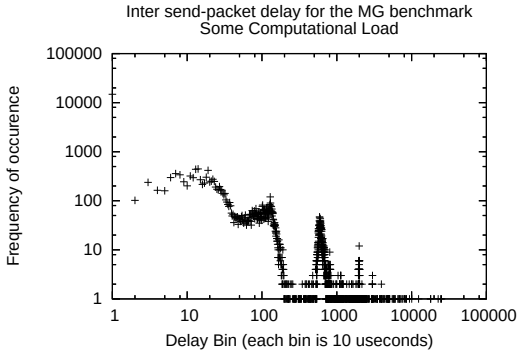
The idea is to simply measure and count the inter-send packet delays that satisfy two conditions:

1. The delay is greater than the Round Trip Time between the two guest VMs by some factor.
2. There is some traffic from the other receiver towards the sender, between the two sends whose inter packet delay satisfies the previous condition.

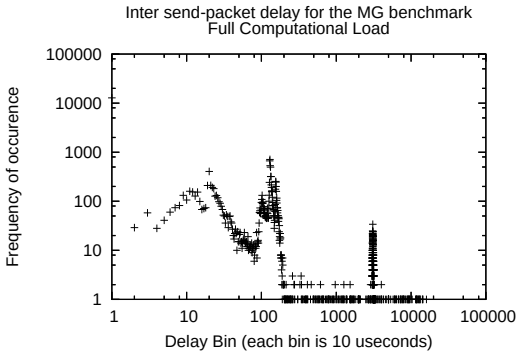
The intuition behind these conditions is to model the performance of the application based on some known BSP principles and use the inter-process interactions as a metric to indicate performance. The first condition intuitively says that that there a inter process interaction might have taken place since the second send was sent after a great delay exceeding the RTT. Normally the sends belonging to a single train of packets are sent at very low inter packet delays. The



(a) MG application inter send-packet delay with no load



(b) MG application inter send-packet delay with some load



(c) MG application inter send-packet delay with full load

Figure 3.4: The inter send-packet delay distribution for a more sophisticated benchmark MG from the NAS package. The three graphs show the distribution for different load conditions. The frequency shifts towards right (more delay) and the clusters are not well defined but spread because of the more complex nature of the application. The duration is the entire packet trace from a single run of the application.

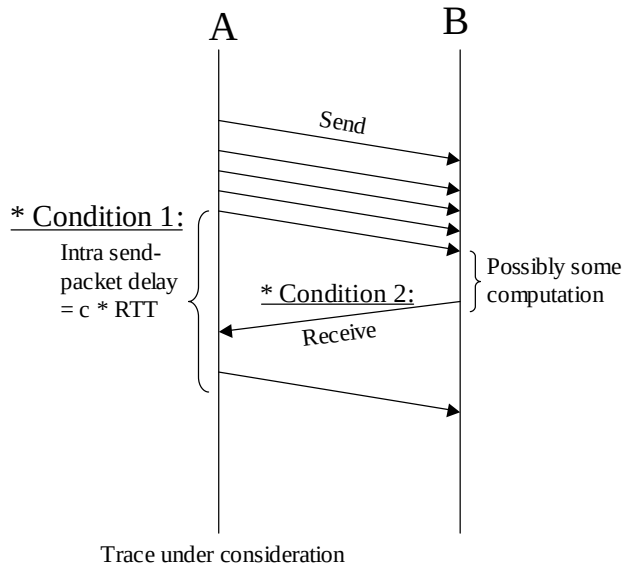


Figure 3.5: A temporal diagram for the RTT - iteration conditions that need to be satisfied

second condition validates the process interaction, to ensure that a *two-way interaction* has taken place. This is because receiving a packet from the partner process indicates that both processes are communicating with each other and hence the interaction is two-way. The second condition also takes care of some cases where the inter send-packet delay may indeed be greater than RTT but does not correspond to an iteration. One such example is as shown in Figure 3.2(b), where the large message size causes TCP congestion/flow control to kick in and results in large inter send-packet delays. The second condition would not be true in that case. Thus in those simple cases, this method would automatically pick the right clusters to count. We denote this count over a given period of time as the RTT-iteration rate or in short, RIR.

3.4.2 Computation of RIR: Sliding Windows and Sampling Rate

To capture the dynamic nature of the iteration rate over a certain portion of the traffic trace, we use a sliding window approach to capture various different iteration rates across the spectrum. We have three different parameters for this approach:

1. Unit window size (*ws*) - this is the window over which the iteration rate is captured per unit time. We fix this unit window as 1 second. Thus all numbers reported are *iterations per second*.

2. Sample rate ($\frac{1}{\Delta t}$)- the sampling rate for capturing iterations over the trace where Δt is the interval of capturing iterations. The unit window is advanced by the sample rate to capture a new data point

3. Sampling duration (T) - this is the total time over which the unit window is slided.

Figure 3.7 illustrates the sliding window mechanism used to capture the iteration rate time series according to the above three parameters.

The sample rate helps to capture the dynamic nature of the iteration rate. Higher sample rate helps us capture finer changes. Increasing the total sliding size helps us get a better picture of the application - the longer the period, more representative and confident we can be about our average or other derived (like the iteration rate distribution for example) measures.

However we obviously cannot sample at infinitely high rates or capture very long duration of applications - both for conciseness and resource reasons. Capturing for longer periods than necessary also introduces redundant data. So we want to find a reasonable *sampling rate* and *sampling duration* such that we still capture representative behavior of the application, from which useful measures of its performance like the average RTT-iteration rate can be further derived.

3.4.3 Process of Deriving Iteration Rate from the Traffic Trace

Figure 3.6 shows the step by step process of deriving the RIR metric and other derivative metric starting from the traffic trace. As shown in the workflow, the steps involved in computing the RIR time series (from which everything else then can be derived) are:

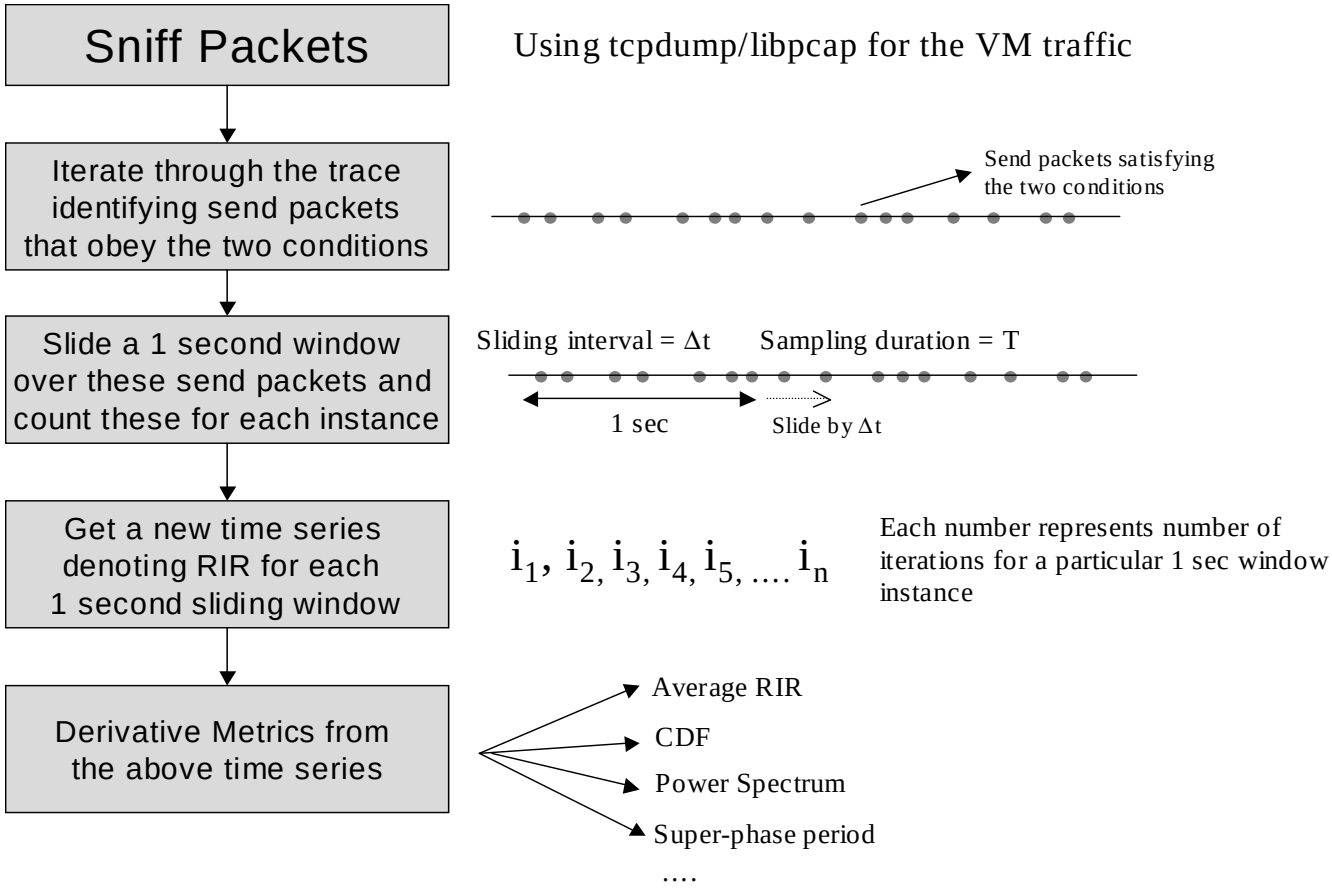


Figure 3.6: The process of deriving RIR and other derivative metrics from the traffic trace

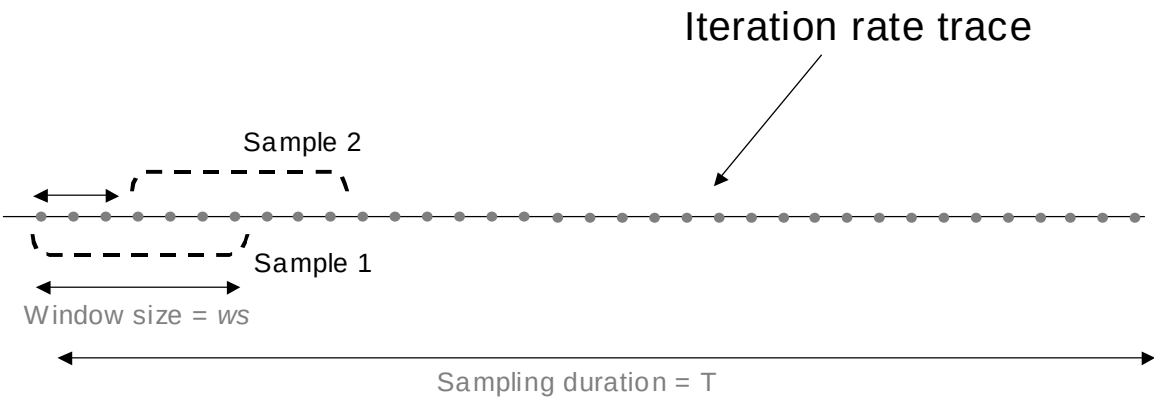


Figure 3.7: Sliding window mechanism used to capture iteration rate at the given sample rate over a sampling duration (T)

1. I first make a pass through the traffic trace for each VM and look at the send packet inter-departure times. Using the two conditions identified in Section 3.4.1 I locate the send packets that indicate an inter-process interaction.

2. As described above, I count the instances of this send packets over a 1 second window. I think form a time series of these counts by sliding this 1 second window by Δt over the sampling period T.

3. At the end I get a time series that reflects the dynamic performance of the application. The parameters Δt and T determine the nature of dynamism captured. This time series can then be used to calculate other derived performance-related metrics as summarized previously.

A very important point to make here is that this metric is a proxy for the actual iterations executed by the application. This metric may not actually correspond one to one with the actual number of super steps or iterations that may have happened in the application. However it can act as a reliable proxy for them. What is important is that whether this metric can serve as a good indicator of application performance and also be used to compare the application's performance when run under different scenarios? An answer in the affirmative would lead to accurate measurement of various scheduling and adaptation decisions for the application and can even be used to predict the total execution time of the application if the input to the application does not change. Given the black box nature of this approach, this can indeed be a powerful inference methodology since performance is the major concern in any scheduling and adaptation system.

3.5 Evaluation on a Static Application: Patterns

The above approach was applied in an automated fashion to the Patterns benchmark under various conditions and input parameters. The Patterns benchmark was describe in detail in

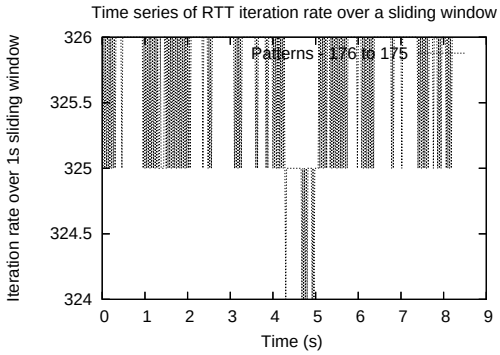
Chapter 1. The Patterns application was run for different computation/communication load scenarios with the following parameters:

- Message size = 4000, Computer Parameter = 1000
- Message size = 4000, Compute Parameter = 0 (no computation)
- Message size = 50, Compute Parameter = 50

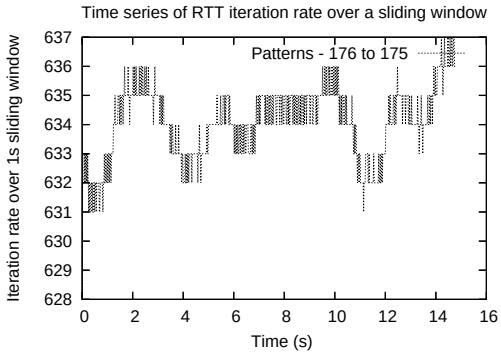
For example, for the first case the command line entered in the shell was: `pvmpattern all-to-all 4 1000 500 100 4000 50 50`. The number of processes were 4, each process being encapsulated inside a separate Xen Guest VM with each VM on a seperate physical host. All the 4 machines have the same configuration as the cluster mentioned in Chapter 1. While the application was running, tcpdump collected the traffic traces for the VMs in the background filtering out other ports like 22 that correspond to SSH traffic for example.

After the trace was collected, the algorithm described in Figure 3.6 was run over the trace to output the RIR time series, which is plotted in Figure 3.8 for the three different cases. As shown in the figures the RIR computation consistently gave accurate measures of performance as validated by the internal iteration output from the application. This in turn also led to accurate comparison of performance and execution time across the different cases. The reported average iteration rate for each run is also shown in the captions. We see that the time-series for the RTT-iteration rate is almost equivalent to the reported iteration rate, hovering around the average by tiny amounts. Thus for the simple case of patterns that has a very simple phase structure, the RTT-iteration rate (RIR) actually is equal to the actual iteration rate reported by the application itself.

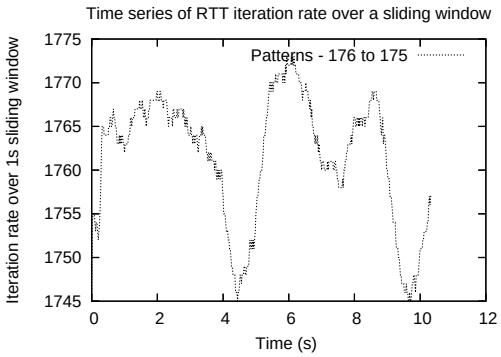
One advantage of this technique is the lack of any manual intervention. There is no counting or identification of clusters involved. Given a traffic trace for one of the processes belonging to the BSP application, the approach automatically gave a time series outputting the iteration rate that remained mostly constant throughout.



(a) Time series of the iteration rate for Patterns, message size = 4000, compute parameter = 1000. Reported Average = 325.49



(b) Time series of the iteration rate for Patterns, message size = 4000, compute parameter = 0. Reported Average = 633.924



(c) Time series of the iteration rate for Patterns, message size = 50, compute parameter = 0. Reported Average = 1760.31

Figure 3.8: Time-series of the iteration rate reported using RTT iteration rate metric for 3 different patterns run under zero load.

3.5.1 Predicted Execution Time under Different Load Scenarios

Ultimately it is crucial that we are able to predict the total running time of the application using our metric of RIR. Table 3.5.1 shows the average RIR inferred for the Patterns applications with the following command-line parameters: `pvmpattern all-to-all 4 1000 500 100 4000 50 50` i.e with a message size of 500 and compute paramter of 4000. Patterns was run under three different external load conditions. These cases correspond to different amount of computation load being imposed on the host of one of the guest Xen VMs. Note that the loaded VM was seperate from the VM on which Patterns was running. No load corresponds to 0 computational load. Partial loads corresponds to a cpu spinner that keeps the CPU busy 60% of the time if run alone. Full load corresponds to a constant cpu spinner that computes all the time. This corresponds to 100% CPU utilization when run alone

Imposing CPU load: I wrote a simple cpu spinner that has a computation and a sleep cycle within an infinite loop. Within each iteration of the loop, there is an internal loop that does a simple calculation for specified number of iterations as specified on the command line and then sleeps for another specified number of milliseconds. Adjusting these two parameters can impose the desired amount of CPU load. These values were adjusted to find out the right cases for 100% and 60% CPU load.

Table 3.5.1 also shows the predicted execution times for Patterns for different load cases, by taking the execution time from the first case as the base case. Wwe see that the Average RIR is used to compute the predicted run time works extremely well to predict performance with a low error rate. Note that we cannot predict the execution time for the first case as we have no idea how long the application will run. This is generally impossible to predict for a particular instance unless we know the number of iterations that will be executed from the source code.

Application	Load	Sampling rate	Advisable sampling rate	Average iteration rate	Execution predicted(s)	Actual(s)	Error%
Patterns(4000,1000)	100%	50 Hz	1.5625	113.710	NA	4.389	NA
Patterns(4000,1000)	60%	50 Hz	0.5859	65.726	7.593	7.572	0.2%
Patterns(4000,1000)	100%	50 Hz	0.0976	11.4243	43.685	43.244	1%

Table 3.1: This table shows the average RTT-iteration rate reported for the Patterns application under three different load conditions and the effectiveness of the metric as a performance predictor

3.6 Derivative Performance Metrics for Dynamic Applications

In the previous section we talked about the static case of the patterns application, which has a fairly static phase structure following the parameters given on the command line. However in practice we often see much more complex phase structure of BSP applications whose performance may be more difficult to represent or summarize. For example the MG application from the NAS benchmarks [17] follows the phase structure as shown in Figure 3.1. The MG application uses hierarchical algorithms such as the multi-grid method that lie at the core of many large-scale scientific computations [25, 26, 29, 71]. These algorithms accelerate the solution of large discrete problems by solving a coarse approximation of the original problem and then refining that solution until it forms a sufficiently precise answer to the original problem. The bulk of the work in MG is done in four routines, each of which is implemented using one or more 27-point stencils. Two of these stencils *resid* and *psinv* operate at a single level of the hierarchy, whereas the other two *interp* and *rprj3* interpolate and project between adjacent levels of the hierarchy, respectively. Parallel implementations of these stencils require point-to-point communication to update every processor's boundary values for each dimension that is distributed. Note that this communication may be with more distant processors at coarser levels of the hierarchy. In addition, the benchmark requires periodic boundary conditions to be maintained, and for global reductions to be performed over the finest grid.

For such a dynamic application, we actually take a multi-dimensional approach to output

various useful inferred properties about the application. There properties are:

1. The Average RTT-iteration rate of the application over a considerable period. (RIR_{avg})
2. Time series of the RTT-iteration rate. (G-RIR)
3. CDF of the RTT-iteration rate indicating what parts of the applications (low or high RTT-iteration rate) are more dominant in time. (G-RIR-CDF)
4. The Power Spectrum of the application indicating the dominant frequencies in the RTT-iteration time series. This gives some idea about the phase structure of the application and also serves as a fingerprint of the application. (G-RIR-PS)
5. Power Spectrum Estimation: The dominant frequency points extracted from the above Power Spectrum graph thus summarizing the behavior in the frequency domain. (PSE)

Table 3.2 enumerates all the metrics that I talk about in this chapter along with a brief summary and a pointer to the section where the metric is described and discussed in more detail.

3.6.1 A Note on Application of Performance Metrics

In black box performance measures for dynamic applications, I define and infer various metrics. All of these metrics can be computed using completely automated means on the application traces. However not all the metrics may serve the same purpose. Some metrics can be used in automated algorithms like adaptation and scheduling to result in better performance for applications. Example of these metrics include the average RTT-iteration rate, the CDFs, the lowest dominant frequency from the power spectrum, etc.

Some statistics like the RTT-iteration time series, the power spectrum, etc, have less uses for automated purposes but are more useful for visual inspection by the developer or the expert user who can glean useful information from them about a particular application. These statistics

Metric	Denoted by	Explanation	Section
RTT-iteration rate	RIR	The RTT-iteration rate is a blackbox measure of the execution rate of the BSP application	Section 3.4
Average RTT-iteration rate	RIR_{avg}	The long term average of the RTT-iteration rate	Section 3.7
RIR Time Series Graph	G-RIR	The time series of RIR over an emperically stationary time period	Section 3.8.2
RIR Cummulative Distribution Graph	G-RIR-CDF	The CDF of the RIR over an emperically stationary time period	Section 3.9
RIR Spread	$RIRS_{5\%-95\%}$	Indicates the spread in RIR of the application or the variation in inter-process interaction	Section 3.9
RIR Median	$RIR_{50\%}$	The half way point for the RIR in the CDF to indiate the median rate	Section 3.9
Power Spectrum of RIR Time series	G-RIR-PS	Gives the frequency domain view of the time series indicating dominant frequencies	Section 3.10
Period of Super Phase	T_{SP}	Infers the time period for the macro repetitive phase of the application corresponding to the lowest dominant frequency from the spectrum	Section 3.10.2
Number of Super Phases	n_{SP}	Infers the total number of super-phases in the BSP application based on Super Phase length and execution period	Section 3.10.2
Predicted Execution Time	PET	The execution time predicted for the current application from a previous basis case	Section 3.10.2
Power Spectrum Estimation	PSE	A concise summary or fingerprint of the dominant frequencies by identifying the local maxima in the Power Spectrum	Section 3.10.4

Table 3.2: A table summarizing the main performance related metrics and graphs described in this chapter

and graphs could be interpreted in useful semantic ways and used to discover any anomalies or issues with application execution in different environments.

3.6.2 Notes on Evaluation

For evaluation of our techniques we used two BSP applications: Patterns (as described earlier) and the MultiGrid application from the NAS benchmarks. Both are PVM applications.

The BSP applications were run inside Xen VMs (build xen-3.0.3-rc3-1.2798.f-x86_32p) that used credit scheduling. Each VM used a Redhat Fedora Core 6 installation including dom0. There were no utilization caps on any VM. The VMs themselves were hosted on a single IBM e1350 cluster that were connected by 100Mbps ethernet links. The configuration of the cluster is described in Chapter 1.

3.7 Average Iteration Rate - Sampling Rate and Determining Time Window (RIR-avg)

A simple scalar metric is the long-term average RTT-iteration rate, averaged over multiple overlapping 1-second windows. The metric can then compared across different instances to evaluate difference in performance. The RIR computation process was described earlier in Section 3.4.2. There are two issues involved in capturing the long-term average: 1) sliding window mechanism and sampling issues, 2) capturing the right time duration.

An assumption that I make for this type of analysis is that the iteration dynamics of the application are indeed *empirically stationary* for the long run. A stationary process is a stochastic process whose probability distribution is the same for all times or positions. As a result, parameters such as the mean and variance, if they exist, also do not change over time or position. For a dynamic application that consist of repetitive phases, it means capturing enough of its performance behavior to capture this repetitive element. For example, even for a complex application like MG, there is a repetition element, the super-phase, that consists of multiple nested super-steps.

However theoretically if the application changes its behavior in a non-periodic fashion, then the performance statistics can be less accurate and less comparable across instances. The reason being that if we try to compare performance numbers across two different time periods of the application that display different frequency properties, it may not make sense to compare the statistics derived out of these different signals. More advanced approaches that take into account the time dynamics of the frequency response as well, may be needed to develop a more sophisticated model in those cases.

3.7.1 Deciding the Sampling Rate

Currently we capture at a high enough sample rate to capture the high frequency dynamics of the RTT iteration rate. For example a typical sampling rate of 0.02 seconds captures a wide

frequency response of the application (25 Hz according to the Nyquist Theorem. The highest frequency captured = $\frac{1}{2*\Delta t}$). We capture at a high sampling rate because given a traffic trace it is not an issue to derive the iteration rate at any desired sampling rate. This was discussed in the RIR workflow description previously. However, its important to ensure that this rate is high enough so that we are not missing any significant high frequency. We use a power spectrum based approach to decide if our sampling rate is reasonable enough.

The power spectrum approach enables us to calculate the energy in the signal at a given sampling rate. The energy is calculated by integrating the power spectrum. We want to capture most of the energy. To find out if this is the case, we first compute the power spectrum of the signal at a very "high" sample rate, a rate which is we think is reasonable as mentioned above, i.e. sufficient to capture most of the energy in the signal. Then, to decide if it indeed captured most of the energy, we re-sample the signal at a lower sampling rate and recompute the power spectrum and the resulting energy in the signal. We then figure out the difference in energy. We keep repeating this until we hit upon a 5% energy loss at a given lower sampling rate. At this sampling rate, we begin to lose significant energy information from the signal. We want to ensure that our actual sampling frequency is actually much higher than this 95% cutoff sampling frequency. As part of our analysis, we output this difference and if the difference is high enough (a factor of 10 at-least), we can deduce that our sampling rate does indeed capture the signal and its frequency response well. Equation 3.2 summarizes the sampling condition that must be met.

For sampling rates s_1 , s_2, find lowest s_2 such that

$$\sum_i PS_{s_2}(\omega_i) > 0.95 * \sum_i PS_{s_1}(\omega_i) \tag{3.2}$$

3.7.2 Capturing the Right Time Duration

Another issue in capturing the signal is the duration of the signal that is used to compute performance statistics. If we capture for a short duration, we will not be able to capture the full dynamicity of the signal and miss various frequencies, thus resulting in inaccurate statistics like the RTT-iteration average which in turn will depend on when the signal was measured. In essence, we need to capture enough of the signal so that the signal is *empirically stationary* as explained earlier and we capture most of the energy in the signal.

To evaluate the stationarity of the signal we use a energy-based technique similar to the one mentioned above. We shorten the sample size by a certain threshold T_s and then recompute the power spectrum. We then compute the error compared to the original power spectrum and check the amount of the loss of energy. The idea is that after shortening the sample to some extent, we should not lose a significant part of the energy if we did indeed over-capture the signal by a comfortable amount. This means that the significant frequencies have been captured in the signal that do not get lost even on shortening the trace a bit. For long running applications for which we cannot capture the entire execution trace (for e.g. order of hours/days), we can capture for an estimated amount of time which we think might be enough to capture the stationarity in the signal. After capturing, then the above test can be applied to ascertain if the stationarity part was captured in the signal. If not, we need to recapture for a longer time period. This can be a iterative process and depending on the performance, the environment etc this time period itself can change over time. Ideally the goal is be to capture as much as possible as we comfortably can and then to validate the stationarity assumption by this method.

For some applications that do not last very long, the best approach is to capture the whole signal if possible. In this case the stationarity assumption may indeed be violated to some extent, however we do get an RTT-iteration average and other performance statistics that last the entire run of the application - the best we can do for a dynamic application.

Computing the Average RTT-iteration Rate

As discussed earlier, using the above two techniques we want to make sure that we have a representative trace from the application. Once we are satisfied with the sampling rate and the stationarity of the signal, we can then compute the average RTT-iteration rate which we denote by RIR_{avg}. This is done by computing the RTT-iteration rate, time series over a sliding window of 1 second that is advanced by the sample rate over the entire sampling period as described previously in section 3.4.3. Then an average is taken of the entire time-series. This represents a scalar performance summary of the entire application that can be used to compare run-times of the application under different conditions. Evaluation of this in the case of MG and its validity to compare performance across different loads is discussed in the Section 3.8.

Sidenote: The Gotcha of Extra Packets Interfering with the Average

A subtle problem that creeps up in automated RTT-iteration rate average computation is that of stray packets contaminating the average. Even after the application is finished with its primary computation and communication part, it may send some packets for management/logging/display purposes at a different rate (or the guest machines or the hosts may exchange some packets that may not be easily distinguished from application packets because of the nature of the black box approach). While capturing, efforts are made to filter out packets that may not belong to the application. In the ideal case we capture packets only belonging to the application, based on the IP and port numbers. However as mentioned, even in this case the application may send out packets after it is finished with the main iterative loop. These packets will be accounted in computing the iteration rate time series as a valid part of the trace and thus may result in the computation of a very low average iteration rate.

The key realization here is that those stray packets are not really representative of the application. To deal with this in an automated fashion, we follow an approach similar to the one used in identifying stationarity. We try to truncate the total traffic trace captured on both sides

Application	Load	Sampling rate	Sugg. sampling rate	Avg. iteration rate	Predicted(s)	Actual(s)	Error%
MG	No load	50 Hz	3.42	283.61	19.44	19.44	NA
MG	Partial	50 Hz	1.66	169.42	32.55	28.78	13%
MG	Full	50 Hz	2.44	125.72	43.85	46.68	7.1%

Table 3.3: This table shows the average RTT-iteration rate reported for the MG application under three different conditions and the effectiveness of the metric as a performance predictor

of the traffic trace (the beginning and the ending part) up to the point where we don't lose significant energy (say 1% of the total energy). This simple approach easily eliminates the effects of these stray packets in practice and helps us focus on the part of the trace that seems to be truly representative of the application. However, this does assume that the extra traffic will be small compared to that of the main computation/communication phase of the application.

In cases where this may not be true, ideally we would capture the traffic in the middle of the execution with the hope of missing the management packets that are sent in the end or beginning.

3.8 Evaluation on a Dynamic BSP Application: MG NAS Benchmark

In this section we discuss the validity of the concept of average RTT-iteration rate and how it can be a useful indicator and predictor of performance even for more complex BSP applications like MG. Table 3.8 shows the summary of the RTT-iteration rate averages computed for the entire trace of the application runtime for three different cases. These cases correspond to different amount of computation load being imposed on the host by a sibling VM. *No load* corresponds to 0 computational load. *Partial load* corresponds to a CPU spinner that keeps the CPU busy 60% of the time if run alone. It achieves this by sleeping and computing by pre-determined amounts. *Full load* corresponds to a constant cpu spinner that computes all the time. This corresponds to 100% CPU utilization when run alone.

The table illustrates different values computed according to the discussion above. The primary value of note is the *Average Iteration rate*, that is computed according to methods specified

above. We treat the *no load* case as the baseline for prediction and then compute the predicted execution time for the other loaded cases. The actual execution time is also shown. From the prediction we see that the predicted computation time is a good indicator of the actual execution time from a baseline case, computed using completely black-box techniques. This is a very useful result and validates the usefulness of the average RTT-iteration metric as a reliable performance indicator. Note that the first row of the table is the base case and we cannot know the execution time for that.

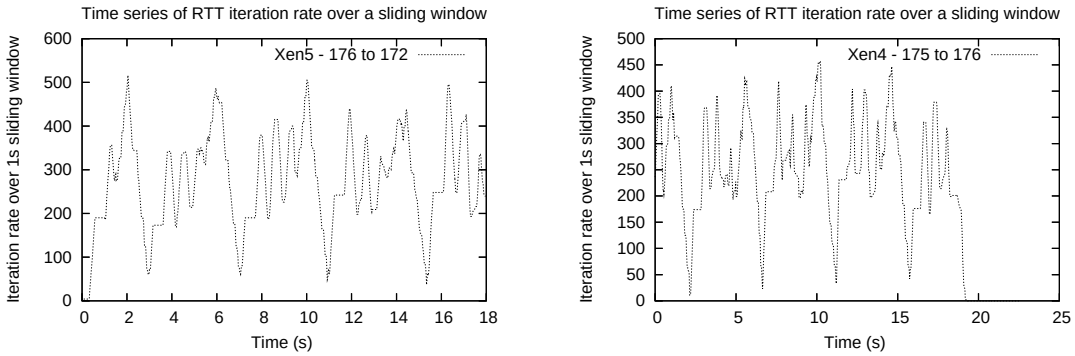
The table also shows the advisable sampling rate as computed by the energy-cutoff method described above. The advisable sampling rate is much lower then the rate used (50 Hz), by more than an order of magnitude. The energy cut-off used here is 95%. This means that if we had sampled the signal at the rate suggested in the table, we would still have captured 95% of the energy in the signal. Thus our choice of sampling rate (50 Hz) seems to be high enough since the 95% cutoff point comes at a much lower sampling rate.

3.8.1 Multiple Process Interaction

For every traffic trace, each process actually interacts with multiple other processes that are part of the BSP application. Therefore we partition the traffic based on the receiver and keeping the sender constant for each process. Thus for k receivers we will receive k sets of results, each corresponding to a different receiver. Similarly for each different process corresponding to the BSP application, we can have traces and hence different results.

For a BSP application with the same kernel for all processes, the results should correspond with each other. Our evaluation with the MG application from the NAS benchmarks confirms this. For other applications where processes may behave differently on different hosts or may interact differently with different receivers, monitoring may be needed to be done at multiple places to capture a coherent picture of the application.

Our results here are for MG and highlight the consistency expected for these processes since



(a) Iteration rate time series with no load (b) Iteration rate time series on a partner host

Figure 3.9: This figure shows the RTT iteration rate time series indicating the dynamic nature of the MG application. Various phases are visible in this graph with 4 super phases. And it shows how the iteration rates correspond with each other on separate hosts

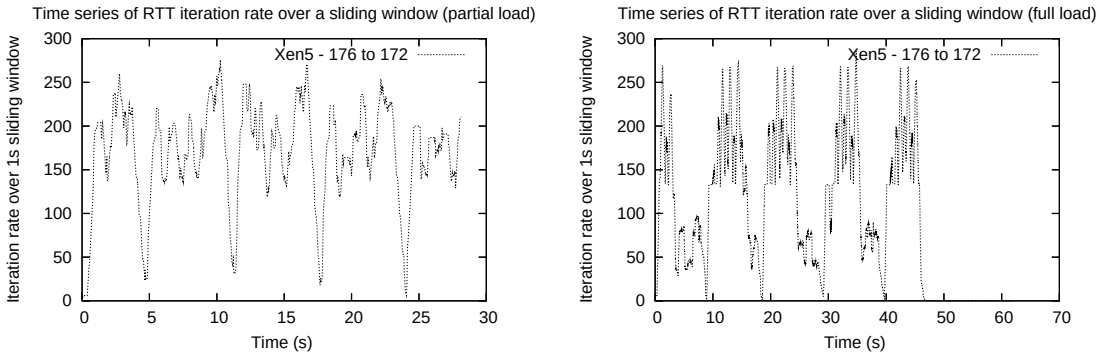
they are all executing the same kernel.

3.8.2 Outputting the Iteration Rate Time-series

Along with the average RTT-iteration rate the time series for the iteration rate computed is also output as a graph that we denote by G-RIR[1]. This gives a visual view of the runtime behavior of the application and often provides more insight about the application to the developer or the user. For example, Figure 3.9(a) shows the RTT-iteration time series for one of the hosts with no load for the MG application. A super structure for the application is immediately visible from the graph. It is clear that there are 4 major super-phases within which there is further varied super-steps that correspond to different sections of RTT-iteration rates. For a given application, these graphs can give useful hints to the developer or the user, without any extra instrumentation in the application, to discover any anomalies or clues for performance problems.

Figure 3.9 shows two time-series for the RTT-iteration rate corresponding to 2 of the 4 hosts involved in the MG application. What is important to note here is the correspondence between

[1]We denote metrics in the form of graphs as starting with a captial G



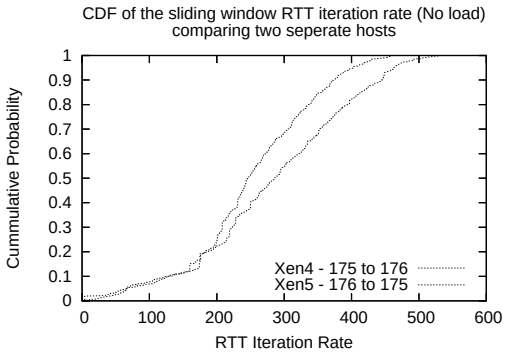
(a) Iteration rate time series with partial load (b) Iteration rate time series with full load

Figure 3.10: The RTT iteration rate for one of the hosts under different load. As we note, the iteration scale falls on the y-axis, and the total time period and per-phase time period expands.

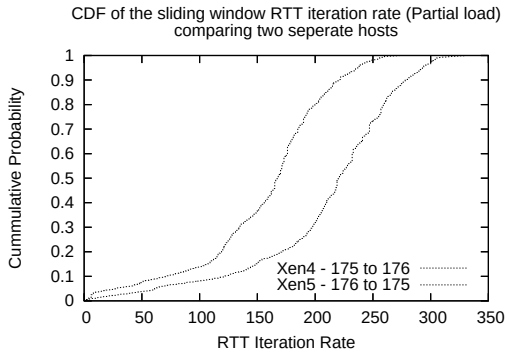
the time series for the two separate processes indicating the consistency of the RTT-iteration metric for a BSP application across different processes. If the different processes are executing the same kernel, the RTT-iteration behavior is expected to be the same for these processes as they execute in synchrony iff the RTT-iteration rate is indeed a more fundamental metric that captures the inter-process behavior. The consistency of the time series for the RTT-iteration rate validates this fact.

Figure 3.9 shows the time-series for the MG application under two different conditions of load (Partial and Full computational load, as defined earlier) for one of the hosts (the host, xen5, is the one on which the Xen VM for the extra load was imposed). We can clearly see the expansion effect of the time series because of the load. The basic structure of the time series is similar, except that the super-phase is now more expanded and each phase is more irregular compared to the no-load case. This further shows how the RTT-iteration rate captures the dynamics of the BSP application.

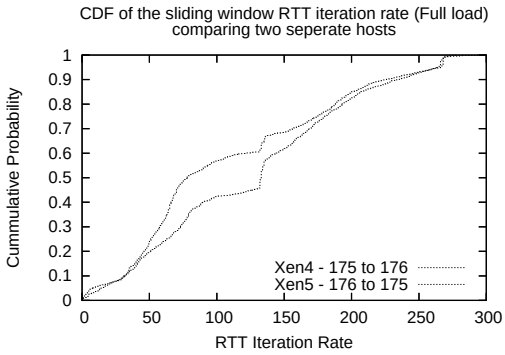
Using the time series we can visually infer that this application seems to have 4 super phases. In Section 3.10.2 we shall further see how this could be computed automatically by inferring the dominant frequencies in the power spectrum using a fourier transform of the above signal.



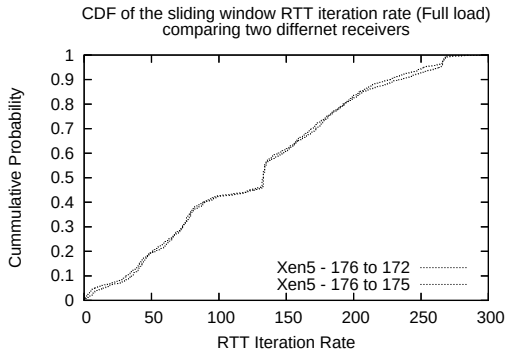
(a) CDF of the iteration rate for no load, for two different hosts



(b) CDF of the iteration rate for partial load, for two different hosts



(c) CDF of the iteration rate for full load, for two different hosts



(d) CDF of the iteration rate for no load, for one host and two different receivers

Figure 3.11: CDF of the iteration rate under different load conditions. The first three show different load conditions for two different hosts. The last graph shows the CDF under no load for just one host, but for two separate receivers.

3.9 A New Dynamic Metric: CDF of the Iteration Rate

The computation/communication ratio is a useful metric for BSP applications that can be used to make predictions and decisions about the application. This metric has been used before in our work for adaptation in numerous contexts [61, 135, 139]. For example, communication-intensive applications can be placed closer to each other in the network to avoid the penalty of high latency or low bandwidth depending on the frequency and the nature of each inter-process communication.

Similar in spirit, we also compute the Cumulative Distribution Function (CDF) for the RTT-iteration rate. We denote this graph by G-RIR-CDF. RIR-CDF gives an idea of the how much time the application spends in various RTT-iteration rate regions. As we discuss later, this can be extremely helpful in scheduling these applications and mixing them with other workload. It could also serve as a fingerprint. CDF comparison techniques could be used to compare performance of different runs of the same application at a more detailed level than the average RTT-iteration rate [21, 110, 128].

Figure 3.11 shows the CDFs for the MG application under different scenarios of load and different combination of hosts and processes. Figure 3.11(a) shows the CDF of the application RIR for the two different processes (for the same application execution instance) under no computational load. The approximate overlap of the CDF for the two processes (which are plotted in the same graph) illustrates the level of consistency in the iteration rate behavior and the performance dynamism amongst different processes for the same execution instance. The purpose of this is to demonstrate that different processes, when executing the same BSP kernel, do follow a global performance trend and thus it is sufficient to sample the traffic trace from a single process. In other words, an RIR-CDF for a process are an application property, not just a process property.

Figure 3.11(b) and 3.11(c) show the same CDFs but for partial and full computational load respectively. Figure 3.11(d) shows the CDF for a single host (xen5) but for different receivers to indicate that the performance dynamism matches for communication trace that differs by the recipient of the packet. We see that the CDFs overlap quite closely for different receivers.

3.9.1 Dynamism and the Peak RTT-iteration Rate

The x-axis on the CDF shows the range of the iteration rates that the application exhibits throughout its lifetime. This spread gives us an estimate of the dynamism of the application, i.e.,

Load Condition	$RIRS_{5\%-95\%}$	$RIR_{50\%}$
No Load	397	286
Partial	209	168
Full	256	133

Table 3.4: Computing the metrics defined above for the MG benchmark under different load conditions to show the effect on these metrics.

how much does its RIR change during execution? To formalize this, we compute the iteration rate difference between the 5% and 95% quantile of the CDF. We term this as $RIRS_{5\%-95\%}$ or the *RTT-iteration rate spread*. Additionally, the median RTT-iteration rate or the 50% quantile of the CDF can be used as another metric to compare application performance across difference instances. This gives an idea of how the application is affected performance-wise across the median case, not just the average. We denote this metric by $RIR_{50\%}$.

Table 3.4 shows the $RIRS_{5\%-95\%}$ and $RIR_{50\%}$ values for the MG benchmark under the three different conditions. We see that $RIRS_{5\%-95\%}$ actually increases from the partial load to the full load case, but the $RIR_{50\%}$ indicates that the RIR climbs up more rapidly in the less loaded cases relative to higher ones. The increase can be attributed to the varying effect of different loads on the same RIR region of an application. For example, a higher RIR region is affected much more by full computational load than a partial load. This can lead to much more spread in the extreme RIR regions for an application under full load.

3.9.2 Guidance from the RIR CDF

Another important aspect of the CDF is the RIR portion where the application spends most of its time. This is important because the effect of external load depends on RIR of the application. The same computational load affects an application with a higher RIR more adversely than an application with a lower RIR. Conversely, network proximity can benefit higher RIR applications more than those with lower RIR. The exact differences depend on each particular scenario. The reason for the variable effect of external load on a BSP applications with different

RIRs is because of the way scheduling is handled for CPU-bound and IO-bound processes. As Govindan et.al. [54] notes, for applications with communicating components, providing enough CPU alone is not enough; an equally important consideration is to provide the CPU at the right time. The paper also describes communication-agnostic CPU schedulers and the adverse affects they can have on communication bound applications. They suggest new communication-aware CPU scheduling algorithms.

The CDF can give an excellent idea of the RIR profile of the application and hence aid these scheduling decisions. Higher RIR indicates more inter-process interaction that demand more efficient communication. Lower RIR indicates more compute-intensive processing or low interaction with large messages. Both of the latter conditions can be distinguished by looking at the CPU utilization and bandwidth at those instances. In fact a combination of the RIR CDF and the bandwidth/CPU utilization distribution could be used to make better scheduling decisions.

To give an idea how applications with different RIRs suffer from external load, we noted the difference in performance of the Patterns benchmark using different command-line parameters to get different iteration rates under different load conditions. Figure 3.12 shows some interesting trends. I ran patterns for different iteration rates and then noted the change in performance when an external (Partial and Full) load was imposed on it. Now the iteration rate can change because of two factors: either the message size per iteration is increasing or the computation per iteration increases.

Figures 3.12(a) and 3.12(b) show the change in iteration rate for both cases under different load conditions. Figures 3.12(c) and 3.12(d) show the slowdown as a fraction of the performance when the application is run without external load. We term this fraction as the *fractional slowdown* of the application. We can see in both cases that higher iteration rate scenarios are affected much more than lower ones. For the increasing message size case (Figure 3.12(c)), the difference in performance is more than 100% for small (50 bytes per iteration) and big message sizes(over 20,000 bytes per iteration). For the increasing computation case, the difference is

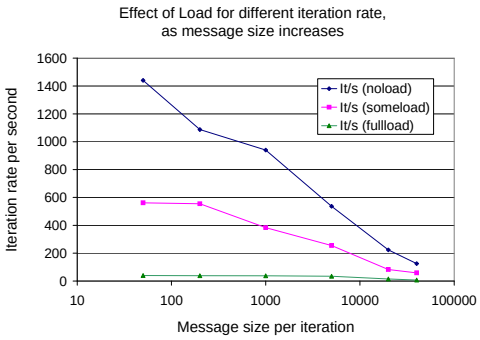
even more pronounced, with a difference in performance of over 12.46 times for low and high computation cases. Thus, when the iteration rate is lower because of higher computation, the effect on performance is less drastic compared to scenarios where a lower iteration rate is due to large message size. Irrespective of the cause, applications with higher iteration rates always seem to be more drastically affected by external load then those with lower iteration rates.

It is clear that the RIR CDF can be useful in making scheduling decisions. For example, when making a decision to move load to one host over the other, the CDF profile of the already running BSP applications can be considered to understand which application might be more affected due to extra computational load. The next section illustrates a simple proof of concept.

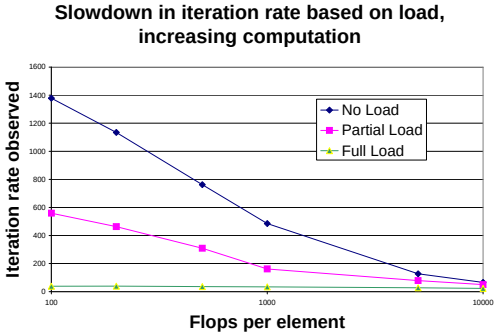
3.9.3 Using CDF Analysis to Make Scheduling Decisions

Earlier we introduced the notion of the RIR CDF that gives a profile of the application's RIR behavior. We discussed how the CDF profile could be used to make scheduling decisions for applications, especially motivated by the observation that the effect on performance of an application from external load depends on its RIR. An application that is more communication-intensive with high RIR can be affected more drastically than a compute-intensive application from the same computational load imposed on the physical host. Earlier in this section, we illustrated this concept for the a simple BSP application like Patterns. We now explore that avenue and give a proof of concept of how the CDF profile can be used to determine what we term as the *Slowdown CDF* or the *Slowdown Profile* and the average slowdown.

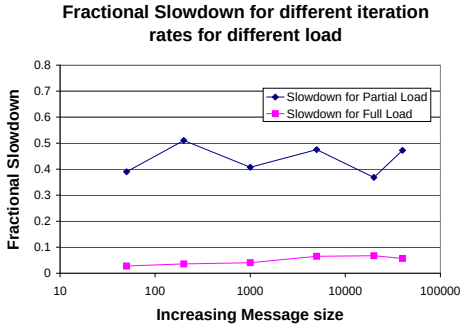
Slowdown CDF is simply the predicted RIR-CDF of a BSP application under some external load. The *Slowdown CDF* is derived from the current RIR CDF and a mapping function that maps each RIR value in the RIR CDF to a potential slowdown value that indicates how much the application could be slowed down if executing at this rate. Mapping each value in the RIR CDF to a slowdown value thus gives us a new CDF that indicates how the different parts of the application (indicated by their RIR values) will be affected in terms of performance.



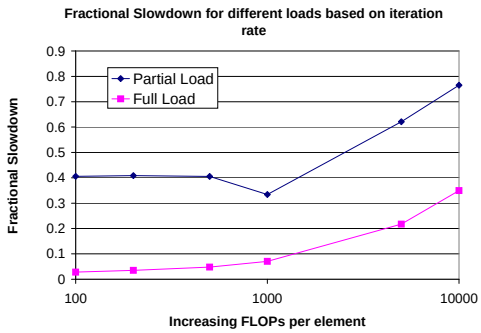
(a) Effect of Load on Patterns for different iteration rates as message size increases



(b) Effect of Load on Patterns for different iteration rates as computation per iteration increases



(c) Fractional performance degradation due to external Load on Patterns for different iteration rates as message size increases



(d) Fractional performance degradation due to external Load on Patterns for different iteration rates as computation per iteration increases

Figure 3.12: This figure shows how external computational load can affect the performance differently depending on its iteration Rate. It shows the change in performance for changing iteration rates caused by two cases: increasing message size and increasing computation per iteration. As shown, higher iteration cases get affected much more drastically than lower iteration cases for both, especially for the case of increasing computation. The last two figures show the change in performance as a fraction - lower means worse.

The mapping can be complicated. As we saw in the simple case of Patterns in Section 3.9.2, the slowdown can depend not only on the RIR rate but also on whether that RIR is influenced by large message sizes or increased computation. Hence considerable more work is required to empirically determine accurate mapping functions that are robust. However we can use some intuition from the Patterns example. From its RIR-CDFs we can get an idea of which application will be affected more in performance depending on which applications spends more time in higher RIR regions. In this section, I use a simple mapping derived from imposing load on different instance of the Patterns application.

To illustrate the concept, we pose the question: *For the IS and MG applications we have seen till now, which application may be hurt more if one of the processes from each application shares the physical host with an external computational load?* This is a practical question faced frequently in adaptive runtime systems. We have evaluated both the MG and IS applications using black-box techniques for performance and here we illustrate a proof of concept of how such a decision can be made. This decision making power can be quite illuminating for an adaptive runtime system, as we can now determine in *advance*, the impact of external load if we must choose one of these applications to be influenced by the load. And all this power comes from using completely black box means.

For our example, we first define a simple mapping function $Slowdown_{100\%} : \Re \longrightarrow \Re$, that takes in an RIR value as input and outputs the fractional slowdown as output. The external imposed computational load is 100%. Note that the *Slowdown* function in general will depend on the external load. Here we assume the external load to be 100% for simplicity. This function is empirically derived from benchmarking the different instances of the application and a database of observations that could grow over time as the instances of mixing external load and BSP applications grows with time. In this case I derive the function by running different instances of the Patterns application with and without load (100% computational load) and then noting the effect on its RIR values. Note that the actual and more realistic mapping function may take

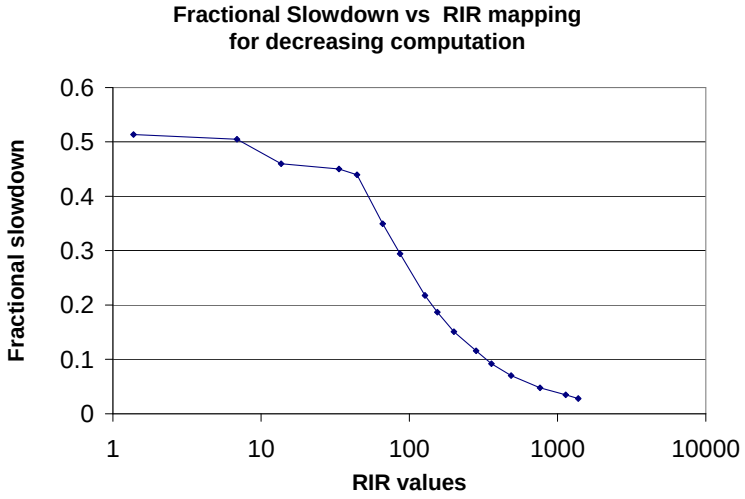
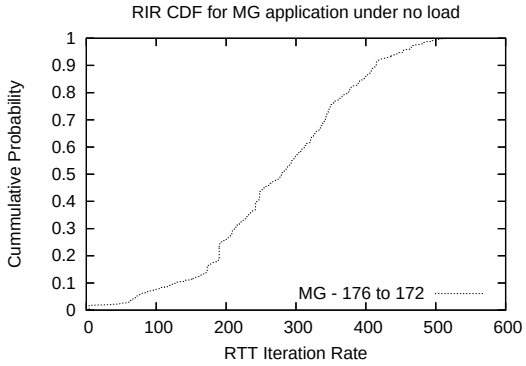


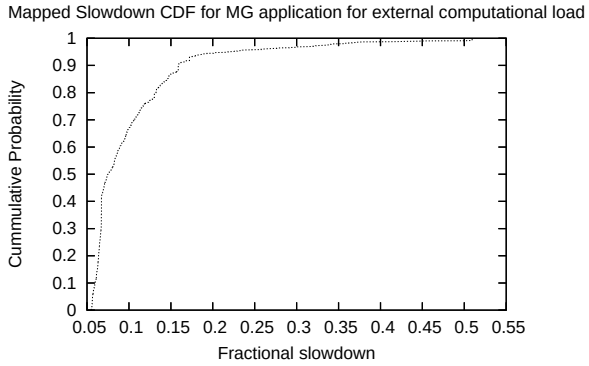
Figure 3.13: A refined mapping from RIR values to fractional slowdown (0 to 1) for an external computational load of 100%. The RIR values are changed by changing the amount of computation done in each iteration.

in more inputs, such as bandwidth or computational utilization, etc. In this example, we use a more refined version of Figure 3.12(b) as a mapping function that maps iteration rates to its fractional slowdown, shown in Figure 3.13. We obtain a discrete mapping that maps certain RIR values to their slowdown and we use linear interpolation to obtain slowdown values for other RIR values.

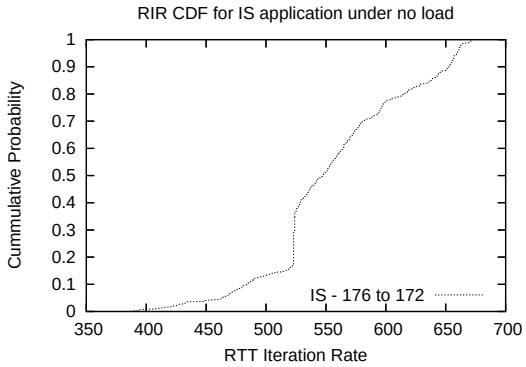
Obtaining the Slowdown CDF: Using this defined mapping function, $Slowdown(x)$, we transform the RIR CDF to a slowdown CDF. This transformation is done by multiplying each RIR value in the original CDF to its slowdown RIR obtained from the slowdown mapping function $Slowdown_{100\%} : \Re \longrightarrow \Re$ described above. The transformed CDFs for both IS and MG applications are shown in Figure 3.14. As we see from the *slowdown CDFs* on the right hand side, we see that IS spends more time in lower regions of slowdown (a range 0.05 - 0.09) than MG which goes all the way to 0.5. We also compute an average slowdown from the slowdown CDFs. The averages in this case are 0.103 and 0.065 respectively for MG and IS. The slowdown CDFs and the averages indicate that the IS will be more drastically affected



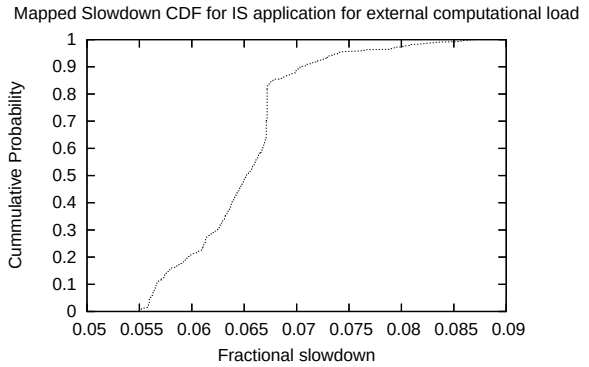
(a) RIR CDF for the MG application under no load



(b) The mapped slowdown CDF for the MG application under external computational load



(c) RIR CDF for the IS application under no load



(d) The mapped slowdown CDF for the IS application under external computational load

Figure 3.14: This figure shows the concept of mapping a RIR CDF for an application to its "slowdown" CDF that shows how the different parts of the application may be affected differently under external load.

by external computational load. If we see the actual execution times after and before load for these applications, this is indeed true. For MG application, the execution times bloats from 19.44 seconds to 46.68 seconds, a slowdown of 2.4. For IS, the change is from 23.34 seconds to 362.477, a factor of 15.529. IS is slowed down much more than the MG application, a prediction that is also implied by the slowdown mapping CDF. Our slowdown averages don't actually correspond to the experienced degradation in performance because of the greatly simplified mapping function taken from the patterns application and ignoring other factors like bandwidth, etc. However this proof of concept shows a powerful result that can enable an adaptive system to be prescient in making decisions about migration and load sharing for different BSP applications without actually knowing any detail about them. Thus this approach could work for other BSP applications without any changes.

3.9.4 Other Possible Scheduling Implications

Another possible avenue for utilizing the CDF profile is statistical scheduling. When multiplexing different applications amongst a limited number of physical hosts, the decision to mix and match applications must be made. It is possible that some applications go well with some but not others. For example, a communication intensive application (with a dominant high RIR profile) may be better suited to share the same host with another communication intensive application rather than a computation intensive (lower RIR profile with low bandwidth consumption) since the compute intensive application can really pull down the performance of the communication intensive application. Hence it could be possible to derive a metric for mixing applications from the CDFs and their bandwidth profile for proper scheduling.

This is a promising area for potential research for black box scheduling of BSP applications. We leave this as a potential future topic of study.

3.10 Power Spectrum of the Iteration Rate

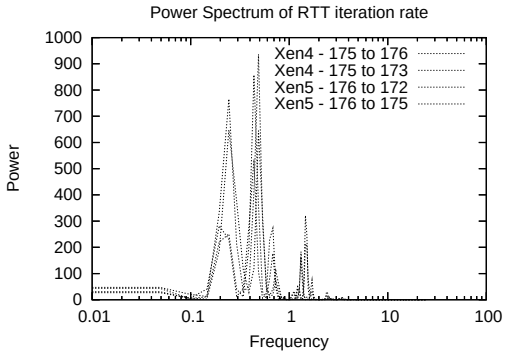
We have previously discussed time domain-related performance statistics like the average RIR, RIR time series and the RIR CDF. The frequency domain transformation of the RIR time series can be another useful tool in getting to discover more about the application. For this performance metric, we compute the Power Spectrum of the RIR time series after applying windowing, zero padding and the DC bias elimination methods discussed previously. We denote this graph as G-RIR-PS. The Power spectrum can help us in three ways:

1. It can give us an idea about the length of the super-phase of the application that can itself be used as a reliable performance metric.

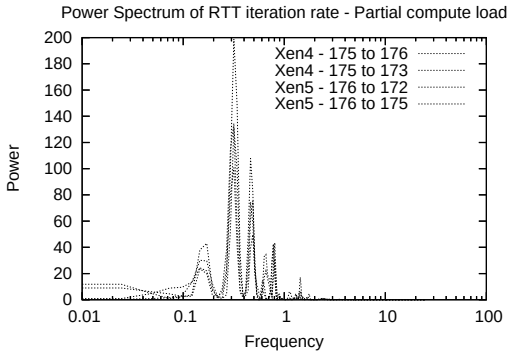
2. It can give the developer or the user a black box idea about the dominant phase frequencies in the application. To a trained eye, this can serve as a valuable tool for discovering anomalies or changes in the super-step structure performance under different conditions.

3. A summary of the power spectrum itself can serve as a compact fingerprint of the process that highlights the dominant frequencies and gives a snap-shot of its super-step structure. This fingerprint can be used for other applications besides performance.

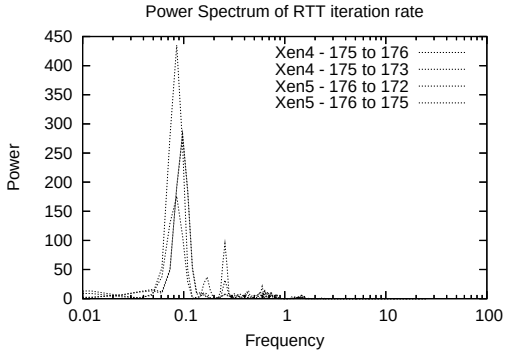
Figure 3.15 shows the power Spectrum of the MG application for different processes under different load conditions. The first thing to note is that in each figure, 4 super-imposing power spectra are shown that originate from 4 different processes - 2 from one physical host and the rest from another. For each load scenario, they overlap quite well, indicating the consistency in their RIR time series and its resulting spectrum. This also gives rise to the notion of a frequency fingerprint that could be used to identify and group processes based on if they are synchronized.



(a) Power Spectrum for two separate hosts and two receivers each with no load



(b) Power spectrum of the iteration rate for partial load, for two different hosts



(c) Power spectrum of the iteration rate for full load, for two different hosts

Figure 3.15: Discrete Power Spectrum of the iteration rate time series showing the principal frequencies under different conditions. We see that the primary frequencies shift and decrease as load increases. And moreover we see that the power spectrum matches for different processes on different hosts for the same application.

3.10.1 Issues in Doing the Fourier Transform for the Discrete RTT-iteration Time Series

Applying windowing to the time series: Before we proceed onto the actual properties, I first discuss the process of taking the Fourier transform [75] for a discrete truncated time series like ours. The role of Fourier transform (FT) is to give an idea of the dominant frequencies in the signal. However an unavoidable problem when processing discrete signals is that of spectral leakage. Leakage amounts to spectral information from an FT showing up at the wrong frequencies. If we could perform a Fourier analysis on a signal that goes on forever, the results would represent perfectly the frequencies and the amplitudes of the frequencies present in the signal. However in practice we have limited space for data storage and the series gets truncated on both ends, so we acquire only a few hundred or a few thousand values. This results in polluting the spectral data because in the time domain, truncating is equivalent to multiplying the time series with a signal of value 1 (unit function) of the truncation duration. In the frequency domain however this is equivalent to convolving the frequency response of the original signal with the frequency response of the truncated unit function. The frequency response for such a signal is the sinc function or $sin(x)/x$. The FT for the truncated unit function or the resulting sinc function is shown in Figure 3.16 [1]. This has the effect of producing spectral values *riding* on a curve. To reduce the effects of this spectral leakage, one often employed strategy is to reduce the side lobes of the sinc function. The side lobes arise from the effect on the FFT of the abrupt rising and falling edges of the rectangular window. Smoothing the sharp rising and falling edges of the rectangular window reduced the side lobes in the $sin(t)/t$ curve and thus greatly reduced spectral leakage. But we can't smooth the data just any way. We need to use special windowing functions to get the samples ready for an FT. We use is Hanning Window [70, 100, 101]. The following equation defines the Hanning window shown in Figure 3.17:

$$w(n)_{Hanning} = 0.5 - 0.5 \times cos(2\pi n/N) \tag{3.3}$$

Performing an FT on the windowed sine-wave data, the resulting sin(x)/x curve shows reduced side lobes, and thus the FT data exhibit less leakage. In effect, by windowing the data before we run them through the FT routine, we improve the sensitivity of our spectral measurements. One side effect of applying the window function is that it may reduce frequency resolution. This can be alleviated by increasing the sampling rate. This is a practical side effect that must be taken into account when sampling the RTT-iteration rate from the traffic trace. More details about spectral leakage and windowing are available in [39, 70, 100].

Zero padding: Most available Fourier transform libraries expect a power of two length time series as the FFT algorithm is optimized for such lengths. However in most cases, we cannot control the length of the input data sequence, and the length of the data might not be an integer power of two. In that case we should not simply discard data samples to shorten the length of the data sequence so it is a power of two. Instead, appending zero-valued samples (zero padding) to match the number of points of the size of the next largest radix-2 FFT is a better approach. For example, if we have 1000 time-domain samples to transform, we didnt discard 488 data samples and use a 512-point FFT. Rather, we append 24 trailing zero-valued samples to the original sequence and use a 1024-point FFT. This also has a side effect of increasing the spectral resolution as we have more data points now at the same sampling rate. Applying zero padding must be done after applying the windowing function to the time-series not before.

Eliminating the DC bias: Even when we use a windowing function, very high amplitude spectral components can obscure nearby low-amplitude spectral components of the signal we want to test. This obscuring is especially pronounced when the original time data has a non-zero average that is, when it is *riding* on a DC bias. When we perform an FFT on the data, the high-amplitude DC spectral component at 0 Hz will overwhelm its low-frequency spectral neighbors. This effect is particularly true for large FFTs. Note that the 0 Hz component actually represents the long term average of the time series. However since it can be easily calculated otherwise, we do not need its component in the frequency domain.

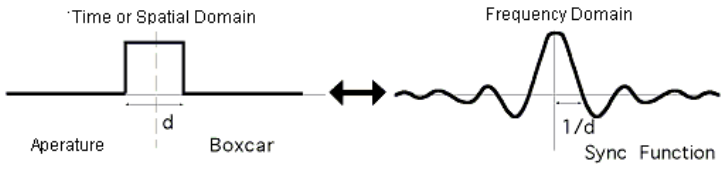


Figure 3.16: The truncated 1 signal in time domain and the corresponding frequency signal after a FFT

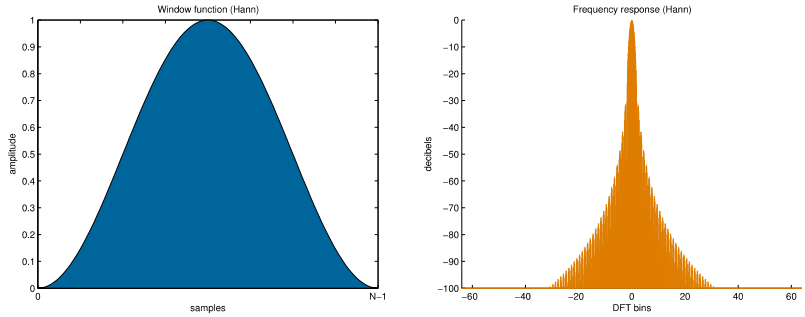


Figure 3.17: The Hanning window is to smooth the discrete time series before conducting the FFT

To eliminate this problem, we calculate the average of the original signal and then subtract it from each sample in the original signal. This removes the DC bias by making the new signal's average (mean) value equal to zero. This eliminates any high-level 0 Hz component in the FFT results and improves the response of other frequency components in the original signal. In practice, when DC bias elimination was applied to the RTT-iteration power spectrum, there was a big improvement in the clarity of the different frequencies.

3.10.2 Measuring the Super Phase Length and Predicting Execution Time

In Figure 3.15 we see that the dominant frequencies shift to the left, decreasing as the load increases. This matches our expectation that the phases in the BSP application execute more slowly. Of special note is the left-most dominant frequency. This frequency actually corresponds to the super-phase of the MG application, which is also shown in Figure 3.1. On close inspection these frequencies are 0.244 Hz, 0.17 Hz and 0.09 Hz respectively for no load, partial

Application	Load	Sampling rate	Inferred Super Phase Length(s)	Execution predicted(s)	Actual(s)	Error%
MG	No load	50 Hz	4.09	NA	19.44	NA
MG	Partial	50 Hz	5.882	27.96	28.78	2.8%
MG	Full	50 Hz	11.11	52.80	46.68	13%

Table 3.5: Using the super-phase length derived from the power spectrum of the RIR time series, we can also predict application execution time from a base case

load and full load conditions. This amounts to a super-phase length of 4.09s, 5.882s, 11.11 seconds. When we compare these numbers to the visually intuitive super-phase length from the RIR time series in Figure 3.9, we see that the lengths indeed correspond well. Table 3.5 also shows how the phase lengths can be used as application performance indicators and used to predict execution time from a base case. The predicted times are quite accurate.

Also from these phase lengths and the program execution times (19.44 seconds for the no load case), it appears that there are 4 super-phases in the this MG application instance. The part of the source code shown in Figure 3.18 confirms this. Thus we see that we could use the power Spectrum in the following ways:

1. We can figure out the length of the super-phase of the MG application in different cases in an automated way. We denote this by T_{SP}.

2. These lengths also act as good performance indicators and predictors and another reliable method to compute PET (Predicted Execution Time).

3. We can also infer the total number of super-phases using completely automated means by dividing the total execution time by the phase length. We denote this by n_{SP}.

3.10.3 The Power Spectrum as a Visual Aid

Apart from the lowest significant frequency, other frequencies also indicate phase behavior. We see the power spectrum as a visual aid to the developer and/or the user to diagnose or get a visual black-box picture of the application behavior that could be used to provide more detailed analysis of the application under different scenarios.

No load		Partial Load		Full Load	
Hz	Power	Hz	Power	Hz	Power
0.244	765	0.1709	43	0.0977	176
0.488	935	0.3174	134	0.25	95
0.732	74	0.4639	75		
1.465	321	0.781	41	0.598	21.2
1.7	75	1.416			

Table 3.6: Power spectrum summary of the dominant frequencies

```
  for (iter = 1; iter <= 4; iter++) {
    fprintf(out, "Iteration %d\n", iter);
    myflush;
    multigrid(u, r, v, chunk, dim);
    resid(r, v, u, chunk, dim);
#ifdef DEBUG
norm = L2norm(r, chunk, dim, &absmax);
fprintf(out, "[%d] L2norm = %.9e,   max elt = %.9e\n",
iter, norm, absmax);
  myflush;
#endif
  }
```

Figure 3.18: Code from the MG source code indicating the number of super-phases hard coded.

3.10.4 Power Spectrum Estimation of Principal Frequencies

The power spectrum can be further condensed by outputting only the dominant frequencies and their power as a vector. This amounts to a dominant frequency fingerprint of the application. For example, the dominant frequencies for the MG application under three different loads are shown in Table 3.6.

3.10.5 Process Fingerprinting and Scheduling Decisions

The above power spectrum summary can also serve as a compact fingerprint for the BSP application. As we saw in Figure 3.15, the frequencies for processes involved in the same BSP application that are synchronous with each other match quite well. This leads to a simple way of categorizing and partitioning processes in a cluster based on their synchronization. This could also be used to make scheduling decisions as described next.

Task and Data Parallel or Component Based BSP Applications

There are some BSP applications that consist of multiple task and data parallel components integrated into the application. For example, in multi-disciplinary simulations a complex physical phenomenon may leverage different unrelated simulation methods and co-ordinate their execution to develop a higher level model [87]. Scheduling and managing such applications can be complex. For scheduling, it can especially help to keep a single data parallel component corresponding to a particular type of simulation together and tightly coupled as the amount of interaction is highly synchronized within a component. The power spectrum fingerprint for processes that belong to a single component should be similar and this can help identify these processes using black box means. This can then help partition the application into its components and schedule them accordingly.

Algorithm 1 presents a simple greedy algorithm that separates an application into its components. It takes in a set of power spectrums corresponding to the different processes of an application and then divides them into similar components. Note that this algorithm assumes that there is a common power spectrum for all its receivers. However if a process has different interaction behavior with different receivers (which is a possibility in a multi-component application), then a more sophisticated approach may be required that takes into account multiple power spectrums per process and partitions them taking the receiver into account.

Statistical Scheduling

Some other interesting ideas could be applied to multiplex different BSP processes on the same hardware. For example, its not clear how processes with different power spectra interact and affect each other in terms of performance when scheduled together. It could be that statistical multiplexing may yield better performance results when processes with the least overlapping power spectra are scheduled together. This is solely a hypothesis but the idea has some merit since the power spectrum gives an idea of dominant periods of its super-steps and mixing ap-

input : PSSet: A set containing all power spectrums (count = n) for different processes
input : PS-Distance (PS_i, PS_j) : A distance metric that returns low distance for two Power Spectrums are similar.
output: $Component_i$: Components containing similar processes according to their Power Spectrum similarity. Total number of components is unknown in advance

```
1  cp ← 0;        // cp is the current Power spectrum being processed;
2  m ← 0; // m is the current component set for collecting similar
   processes;
3  PS_cp ← random element from PSSet ;
4  while cp < n do
5  |  P_min ← min_{P_i ∈ PSSet} (PS-Distance (PS_cp, PS_i));
6  |  if PS-Distance (PS_cp, PS_min) > DistanceThreshold then
7  |  |  m ← m + 1;
8  |  end
9  |  Component_m ← Component_m ⋃ PS_min;
10 |  cp ← cp + 1;
11 |  PS_cp ← PS_min;
12 end
```

Algorithm 1: A simple greedy algorithm for partitioning a multi-component application into its separate component processes

plications with least overlap could help these phases not interfere with each other in a statistical sense.

Application Categorization

The fingerprint could be even used to identify an application under different conditions of load. From the power spectrum estimation, we actually see that the dominant frequencies for higher load cases are actually scaled versions of the frequencies for no load case. This indicates that the frequency transformation is a scaling operation when the application is affected by external load. Using scale-invariant pattern matching algorithms it could be used to classify applications in different environments. Such a categorization system coupled with a learning component could be a powerful black-box application categorization tool that could help in management and scheduling of applications.

3.11 Evaluation with Another NAS BSP application: Integer Sort (IS)

I also use another application, the NAS benchmark, Integer Sort (IS), to further validate and test the black box measures proposed in this chapter to study its performance. IS is a large integer sort operation testing both integer computation speed and inter-processor communication. This kernel stresses the integer performance of the underlying node [64]. There is a class of programs called particle pusher codes that accumulate data into target arrays using indirect index vectors. These codes often appear in application programs of high performance computing, and therefore ate very important. The NAS Parallel Benchmarks (NPB) include integer sorting (IS) as an example of particle pusher codes [132].

We ran the IS application to sort 1024 * 1024 numbers per node with each number having an 18 bit range. We then captured the trace for a part of the execution of the program, not its full execution trace. Note that this is important as it reflects how trace collection would also happen

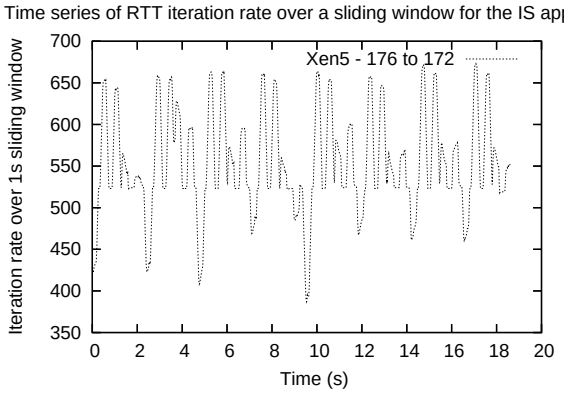
Application	Load	Sampling rate	Sugg. sampling rate	RIR_{avg}	**PET(s)**	Actual(s)	Comm. time	Error%
IS (1024,18)	No load	50 Hz	4.785	556.255	NA	23.342	16.570	NA
IS (1024,18)	Partial	50 Hz	1.2695	264.345	49.118	50.423	39.054	2.6%
IS (1024,18)	Full	50 Hz	1.13	36.843	352.417	362.477	313.550	2.77%

Table 3.7: Using RIR_{avg} to predict the performance if the IS application from the NAS benchmarks under different load conditions.

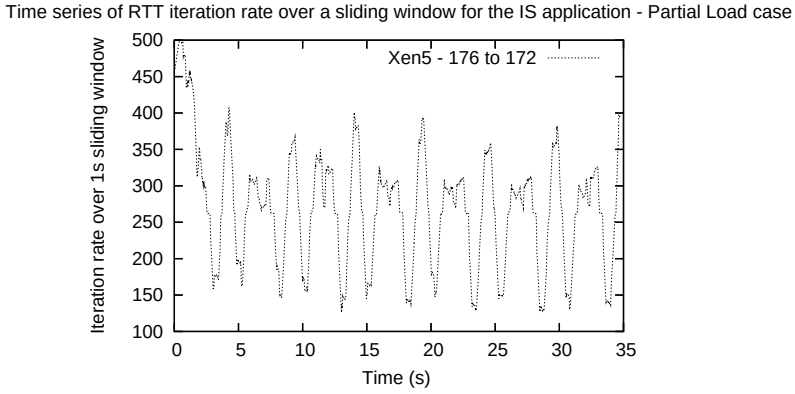
in reality - collecting a part of the trace should be able to help in performance prediction and characterization using our black-box measures.

Figure 3.19 shows the RIR time series under different load conditions. Note that the RIR is extremely high for the no load case: the RIR almost always above 400. However, the change in performance can be drastic as seen in the full load case where the RIR touches 0 for some periods. We are able to accurately predict the total execution time for this drastic slowdown based on the average RIR measurement from a part of the application's runtime. Table 3.7 summarizes the predicted execution times based on the RIR_{avg} metric. Three different load conditions (No Load, Partial Load, and Full Load) were imposed as before. As we can see the predictions based on RIR_{avg} are quite accurate and close to the actual running time. The table also shows the time spent communicating for different load cases. As we can see, the application is slowed down by more than an order of magnitude because of the full external computation load imposed on it.

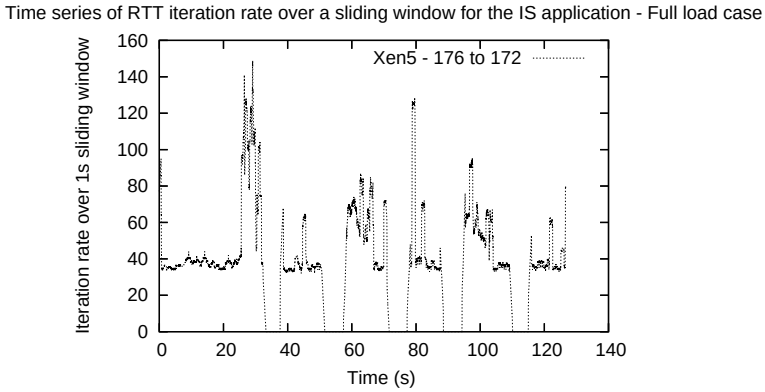
Figure 3.20 shows the power spectrum for the IS application for different receivers and different load conditions. As observed before, it shows the periodicity values corresponding to different phases inside the IS application and helps us understand the periodic behavior of the application. The power spectrum is quite consistent across different receivers showing that the inter-process behavior is quite similar for different sender-receiver pairs. Table 3.8 shows PET determination using only the super phase length as inferred from the lowest dominant frequency from the power spectrum of the RIR time series. The results are quite close the actual runtimes observed for the external load cases of IS. It also interesting to note that the trace was captured only for a partial duration of the application's execution time. For example the fully loaded



(a) Iteration rate time series with no load for the IS application



(b) Iteration rate time series with Partial load for the IS application



(c) Iteration rate time series with Full external load for the IS application

Figure 3.19: The RTT iteration rate for one of the hosts under different loads for the IS application. As we note, the iteration scale falls on the y-axis, and the total time period and per-phase time period expands as the load increases.

Application	Load	Sampling rate	T_{SP}	**PET(s)**	Actual(s)	Comm. time	Error%
IS (1024,18)	No load	50 Hz	2.2758	NA	23.342	16.570	NA
IS (1024,18)	Partial	50 Hz	5.1411	52.7303	50.423	39.054	4.6%
IS (1024,18)	Full	50 Hz	32.7761	336.172	362.477	313.550	7.26%

Table 3.8: Determining PET using the Super Phase length derived from the Power Spectrum. PET is quite close to the actual run time observed for the slower instances, even though partial data was captured.

Message size per iteration	Iter Rate	Predicted Exec time	Actual	Error %
0	456.43	NA	27.8	NA
400	411.27	30.8	31.4	1.94%
800	415.57	30.5	30.8	0.98%
2000	367.33	34.5	36.1	4.6%
8000	283.41	44.8	49	9.37%

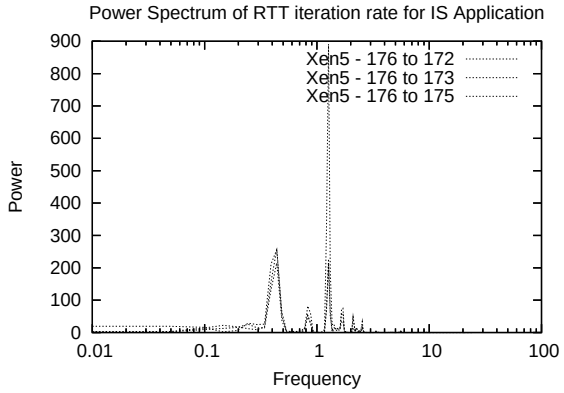
Table 3.9: Predicted Iteration rate for the IS benchmark under different Patterns runs

version of IS takes 313 seconds to execute. We captured the trace for a duration of 127 seconds for that application.

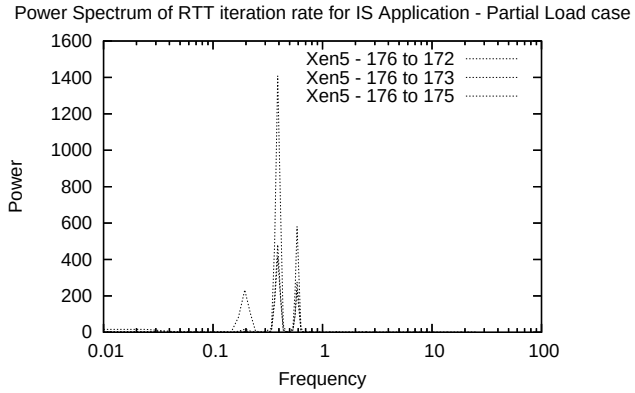
3.12 Study of Cross Application Effects

This section studies another possible measure of accuracy of the methods described in this section for measuring performance of BSP applications. In a real scenario, multiple applications will be running on the same shared hardware and thus their execution and cross traffic might interfere with each other. To evaluate such effects, I ran the NAS IS benchmark and studied the effect of running the Patterns benchmark with different parameters on the accuracy of the RIR algorithms described in this chapter. The goal is to understand if the cross traffic from another BSP application can significantly affect the accuracy of the RIR methods since they depend so closely on the traffic trace of the application under examination.

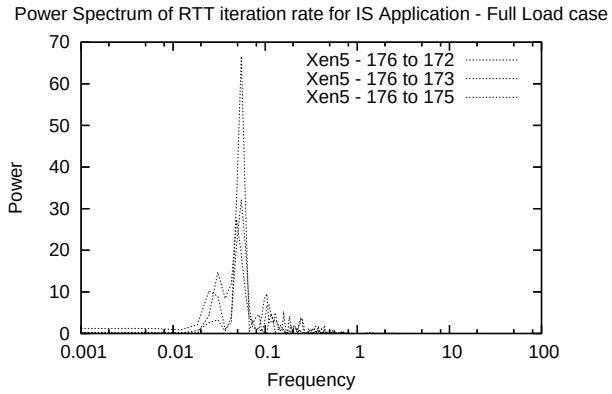
Table 3.9 shows the result of measuring the Average iteration rate for the IS benchmark under different conditions of cross traffic, achieved by changing the message size of the Patterns benchmark. We see that the application slows down slightly, as the message size increases. This is somewhat expected as the IS benchmark is more communication intensive. The iteration rate



(a) Power Spectrum for a single host and three receivers, No load case



(b) Power Spectrum for a single host and three receivers, Partial Load case



(c) Power Spectrum for a single host and three receivers, Full Load case

Figure 3.20: Discrete Power Spectrum of the iteration rate time series showing the principal frequencies under different conditions for the IS application. The Power Spectrum overlap for different receivers is quite consistent showing stable behavior across receivers. We see that the primary frequencies shift and decrease as load increases.

measured also falls. The predicted execution time based on the measured iteration rate is quite close to the actual iteration rate, with accuracy within 1% to 10% of the actual predicted time. The accuracy does fall to a certain extent as the message size increases, indicating some level of interference.

Overall, the effect of cross traffic in this experiment does not seem to affect the RIR iteration rate methods significantly in their ability to measure and predict the absolute performance of the IS benchmark. This is a good sign towards the use of these methods in shared environments where the hardware resources are shared amongst multiple BSP applications.

3.13 Implementation

All of the above techniques have been encapsulated in a single Perl script (currently 550 lines). The Perl script takes in 5 input parameters:

1. The tcpdump trace for the particular BSP process VM in text format.
2. The source IP identifier for the VM.
3. The sampling rate or the time by which the sliding window is advanced for every new value.
4. The RTT time amongst the VMs.
5. A short description string used to output the result files from the script - we shall denote this by *dstring*.

The script then analyzes the trace and outputs all the performance metrics and any other statistics that have been discussed previously namely, separately for each receiver:

1. The RIR_{avg}

2. The RIR_{ts} in a separate file named *dstring-sourceip-destip.ts*. Each line has two values: time and the number of RTT-iterations inferred during the window starting at that time.

3. The RIR-CDF in a file named *dstring-sourceip-destip.cdf*.

4. The RIR-PS in a file named *dstring-sourceip-destip.cdf*.

5. A concise summary of dominant frequencies in the Power Spectrum by identifying the various local maxima. These are output in a file named *dstring-sourceip-destip.pse*.

6. The suggested sampling rate that would result in an energy cutoff of 95%. It validates that the current sampling rate is reasonably higher than this cut-off sampling rate.

7. Stationary assumptions about the trace by computing the energy drop off as the trace is truncated from the end.

The BSP applications themselves can use UDP or TCP protocols to communicate. The script can handle both traces. A sample command line execution for the trace is:

```
perl ../sendDist.pl xen5-mg4-someload-4its.txt 165.124.184.176 0.02 400
check > check.report
```

The stdout output from the script has lot of extra information about each receiver including the time series of the RIR-iteration rate, its power spectrum etc, along with summary results for each receiver.

The script uses a third-party Perl Module [84] to compute the FFT and the Power spectrum.

3.13.1 Notes on Packet Capturing and RTT Determination

For high performance packet capture, I used tcpdump as: `tcpdump -n -nn -q -tt -i vif12.0 net 165.124.184 -w dummy` where dummy is the file where the raw packet trace is output. Here `vif12.0` is the virtual ethernet interface corresponding to the guest Xen VM. Later a text

dump from this raw dump can be recovered using: `tcpdump -n -nn -q -tt -i vif12.0 tcp and not port 22 -r dummy > dummy.txt`

The RTT is an important number for inferring RTT-iteration rate. In our evaluation we assume that the RTT is similar amongst all VMs since they were on the same cluster. The RTT value is measured with *ping* as a composite of the minimum RTT value reported and the stddev reported.

3.14 Making the System Work Online

In this chapter we have primarily explored the offline computation of the various metrics. However in a runtime adaptation scenario, this information needs to be deduced online, while the application is running, so that any adaptation decisions can be taken in a timely fashion. There are two aspects to computation of metrics discussed in this chapter:

1. Computing the RIR time series from the traffic trace, and
2. Deriving other metrics from the RIR time series like the RIR-CDF or the power spectrum, that require additional computation over the time series.

Computing the RIR time series requires sampling the traffic from one of the processes. Since only one side of the trace needs to be examined (the sender side), the RIR time series can be derived in a very straightforward manner once the round trip time is estimated. Only the send packet inter-departure delays are required for the packets that belong to the VM/application. This only requires looking at the timestamp of the packets and can be easily integrated into VNET, in a similar fashion to VTTIF. This a constant stream of RIR values can be computed from live traffic.

The 2nd order metrics like RIR_{avg}, RIR-CDF and RIR-PS require computing a Fourier Transform of the time series. However this operation need not be real time. Computing the Fourier Transform is required to understand how long the RIR series needs to be in order to

capture the empirical stationarity behavior for dynamic applications. However this is not in the critical path of capturing the packets and hence can be computed whenever the desired metric is finally computed.

In an online environment, periodic probing might be done for dynamic applications to update these metrics and record their performance. Note that the RIR time series does not need to be computed throughout, but only for the period of the probing, whose length is ultimately decided by the stationarity criteria.

3.15 Conclusion

In this chapter, I have looked at a very important problem for an adaptation system whose aim is to improve the performance of the application: How can we ascertain its performance at runtime without specific knowledge of the parallel application or its environment? I have developed a novel technique based on the send packet analysis of the traffic emitted by processes in the application, one that gives an accurate proxy for the performance of BSP applications. I proposed a new black box metric called RTT Iteration Rate (RIR) based on this technique.

Building on the RIR metric, I further looked at much more complex applications like the NAS benchmarks, whose performance behavior is dynamic. I then proposed a list of many derivative metrics like RIR_{avg}, RIR-CDF and RIR-PS that capture the dynamic performance behavior of the application. These metrics are summarized in Section 3.6. I evaluated their application to performance prediction of applications under different loads and show that these metrics indeed are quite accurate for different applications and load conditions.

I also showed how some of these metrics could be used for other novel purposes like application fingerprinting and for complex scheduling decisions. For example, I showed how the application performance degradation under load actually depends on how computation or communication intensive the application is and how this fact can affect scheduling decisions for complex applications that display much a dynamic profile of computation/communication be-

havior. I also talked about isolating different components of a task and data parallel application in order to automatically cluster and schedule them in proximity.

Chapter 4

Ball in the Court Principles for Performance Imbalance

In this chapter we explore how to predict performance of a BSP application under different load conditions using only black box techniques. The problem is stated as:
Given a BSP application that has its processes subject to some external load in a shared environment, can we infer what its performance would be if we removed one or more of the external loads (without actually removing them), using passive black box means and without using new probes?

A BSP application usually consists of multiple processes executing a common BSP kernel that implements the self-same algorithm. These processes compute, communicate and synchronize with each other according to some schedule and different phases. In such an orchestrated application, even if one process is affected, this can lead to a dramatic impact on performance for the application. This is because this process will be unable to respond and work in tandem with other processes effectively thus slowing the entire application down. Even for a highly parallel BSP application, a single load can drastically alter the execution time.

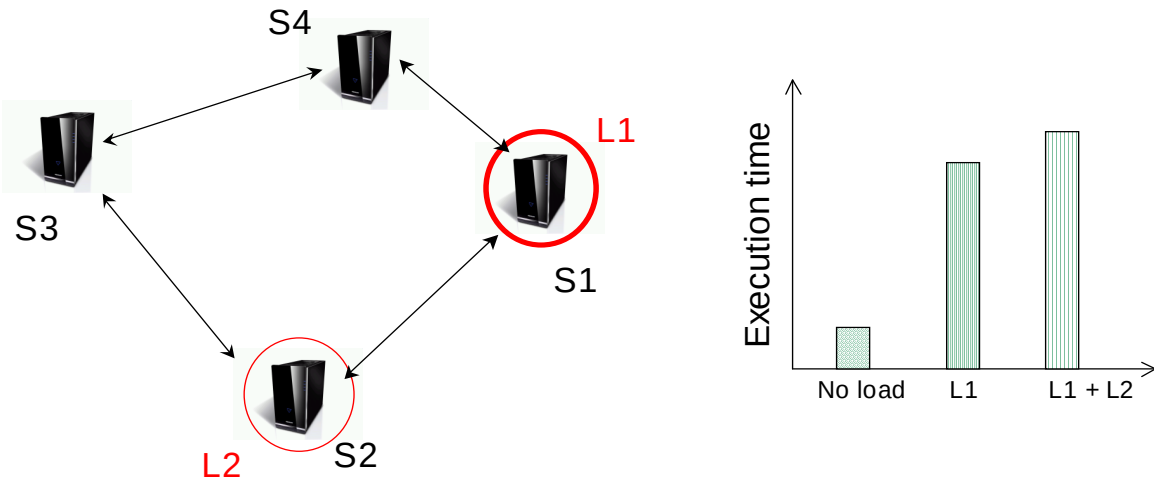


Figure 4.1: The figure shows how one or more processes affected by local external load can drastically affect the performance of a BSP application. It shows the possible effect of applying two different loads L1 and L2 on two different hosts and their impact on execution time

4.1 Motivation and Benefits

Figure 4.1 shows an example scenario where the impact of external load on a 4 process BSP application is illustrated. As shown on the right, even a single load can multiply the execution time manifold. This was actually experienced in the last chapter, Black Box Measures of Absolute Performance (Chapter 3), where we studied the ability to measure the performance of an application under different load conditions. For example, for the IS application, the execution time increased from 23 seconds to over 320 seconds, almost a 15 fold increase, when just one of its processes was stressed with an external computational load. We also saw that the *same external load can affect different applications differently*, thus ruling out a uniform decision making process for improving the performance of a BSP application under external load. This observation is quite important: Since different applications are impacted to different degrees for the same external load, it is important to figure out that impact before making any decisions to improve such performance, especially when many such applications are involved in the scenario.

This leads us to the motivation and benefits for solving this problem. When such applications run in shared load environments where different processes can be under different stresses, it can be unclear what the actual effect on application performance is. For a runtime adaptation system like Virtuoso [38], that has to make decisions for adapting the resources and re-arranging workloads so as to improve the individual and overall utilization and efficiency of the system, this insight can make all the difference. Here we look at exactly this question: *For example, in the scenario of Figure 4.1, if we removed load L1 and L2, or migrated the processes to different hosts, how will the application benefit?* If we can figure out the answer without actually activating the potentially costly adaptation mechanisms themselves, we have achieved a huge win for such a system. *If the system knows how badly the application is actually affected by external load on one of the processes, it can act accordingly to alleviate the situation, especially when there are limited resources and multiple BSP applications are involved.*

Note that we have two constraints to solving this problem that further increase the value of the system:

1. Passive black box measurements: No new instrumentation of the application or individual knowledge about the BSP application should be needed, thus making the techniques generic and useful to a wide range of applications.

2. No extra loading scenario needed: This constraint is very powerful. We want to derive the no-load performance of the application using measurements completely self contained in the loaded scenario, without using measurements from other scenarios. If available, these extra measurements may be useful in improving the accuracy and robustness of the model, but this should not be a core requirement.

Operating with these constraints we would then figure out the performance impact of load on an application using passive measurements completely self-contained in the running scenario. Thus this capability could be readily included in any resource or adaptation system without alteration of the applications or extra measurements.

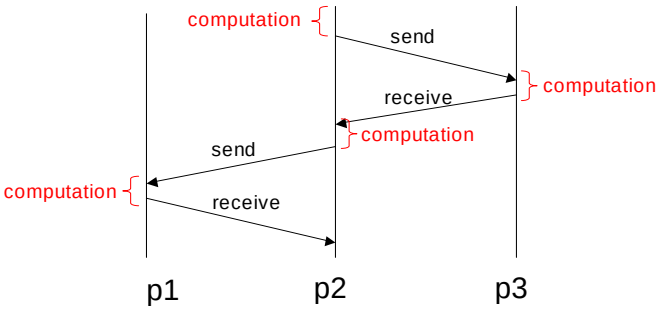


Figure 4.2: An example of a BSP step that consists of p2 communicating with p1 and p3 and also computing before sending out each processed message

4.2 Ball in the Court Principle

To approach this problem of finding possible application performance without the load, without actually removing the load, we need to take a closer look at the BSP application model and how we can capture it indirectly using black box measurements.

A BSP application can have a multi-step super-phase where each step can consist of computation, communication and synchronization phases. Within each step, each process could communicate with various other processes according to a certain schedule and perform computation asynchronously. The computation amongst different processes can overlap. In fact parallel computation is what makes BSP an attractive model for computation. Figure 4.2 shows an example BSP step where p2 does some computation in between a fixed communication pattern with p1 and p3. The fixed communication pattern or the schedule can itself change multiple times for a complex BSP application.

Intuitively, from the BSP model, each application does some local processing before sending out every message. Whenever it receives a message, it acts on the message and then sends another message out, after some local processing. If we were to decompose the time for a BSP application, this would consist of the local computation/data processing time for each process (these can overlap amongst multiple processes) plus the communication time (which also includes synchronization primitives). For a typical BSP application, the nature of the local

processing/computation is usually quite similar for each process resulting in a balanced workload for each process, maximizing the parallelization.

Our approach towards solving this problem is to actually look at how a process gets affected by external load and how this stretches the global execution time. Currently, we focus on the situation where only one process is subject to external load and all others are running without load on identical hosts. The network links and conditions are assumed to be symmetrical across all processes. Theoretically, if just one process is subject to external load, the other processes are still unaffected and their behavior should not change. The only way the entire application can be slowed down is because of the *slower processing and response times* of the loaded process towards other processes. Since all processes operate in synchrony, the iterations can proceed ahead only after the loaded process has completed its duties for that iteration or step. For example, the computation times shown in Figure 4.2 can get stretched. This would result in a larger application running time.

The essence of the idea is to get a measure of the processing/computation time on the side of the loaded process for which it's solely responsible and other processes depend on to proceed. If we can compare this with another unloaded process belong to the same application, we could get an idea of the imbalance in the behavior of these loaded and unloaded processes and possibly determine how much extra time the loaded process is taking and thus causing the entire application to slow down.

We term these delays for tasks/responses that a process is responsible for as **Ball In The Court** delays. The ball here is the responsibility to interact with the other processes after its share of the computation for that iteration is complete. The court here is the local host. It signifies the fact that the ball of returning the next message after the current iteration's computation is in process's court and until it is done with its responsibility of sending the next message to its partner processes (according to the BSP topology), it will slow down other processes and thus the entire application.

4.2.1 Measuring BIC Delays

Having given a high level idea of the Ball In the Court principle, the essence is thus measuring the BIC (Ball In the Court) delays of loaded processes and seeing them in comparison with BIC delays of other processes, thus giving a clear picture of any performance imbalance and also its exact magnitude. This is also based on the symmetry principle - the assumption that different processes of a BSP application are workload balanced and thus these BIC delays are comparable to figure out if a process is the source of extra delay.

What is the nature of these BIC delays? BIC delay could come from computation or communication-related delays. As we have seen communication intensive applications can in fact be more drastically affected than compute-intensive applications for the same external computational load. Thus both of these sources can contribute to the net BIC delay.

The next step is actually being able to measure these BIC delays using black box means. This is a challenging task, since without actually instrumenting the application, its difficult to determine the exact BIC delay in each step. To approach this problem, lets take a look at the packet traces corresponding to one of the processes. First of all we make the basic observation that the packet trace of the process does capture the behavior of the process for the entire runtime of the application. If we load a process externally, the extra delay and elongation of application's execution time will also reflect in the packet traces for *all* processes, not just the loaded processes. How does this extra delay actually show up in a packet trace?

Lets examine a packet trace from a sample Patterns run. This Patterns application consisted of 4 processes executing the all-to-all message topology. A sample excerpt of the packet trace for an unloaded process is shown below:

```
1189054440.763595 IP 165.124.184.173.45824 > 165.124.184.176.54313: P 7433:8433(1000) ack 9629 win 411 <nop,nop,timestamp 280954206 281110627>
1189054440.763659 IP 165.124.184.176.54313 > 165.124.184.173.45824: . ack 8433 win 342 <nop,nop,timestamp 281110630 280954206>
1189054440.769566 IP 165.124.184.176.38608 > 165.124.184.172.59979: P 8581:8629(48) ack 7385 win 310 <nop,nop,timestamp 281110630 281086251>
1189054440.769567 IP 165.124.184.176.38608 > 165.124.184.172.59979: P 8629:9629(1000) ack 7385 win 310 <nop,nop,timestamp 281110630 281086251>
1189054440.769964 IP 165.124.184.172.59979 > 165.124.184.176.38608: . ack 9629 win 379 <nop,nop,timestamp 281086260 281110630>
1189054440.775789 IP 165.124.184.172.59979 > 165.124.184.176.38608: P 7385:7433(48) ack 9629 win 379 <nop,nop,timestamp 281086261 281110630>
1189054440.776001 IP 165.124.184.172.59979 > 165.124.184.176.38608: P 7433:8433(1000) ack 9629 win 379 <nop,nop,timestamp 281086261 281110630>
1189054440.776073 IP 165.124.184.176.38608 > 165.124.184.172.59979: . ack 8433 win 342 <nop,nop,timestamp 281110633 281086261>
1189054440.781934 IP 165.124.184.176.43412 > 165.124.184.175.35181: P 9629:9677(48) ack 8433 win 342 <nop,nop,timestamp 281110633 281017098>
1189054440.781955 IP 165.124.184.176.43412 > 165.124.184.175.35181: P 9677:10677(1000) ack 8433 win 342 <nop,nop,timestamp 281110633 281017098>
1189054440.782320 IP 165.124.184.175.35181 > 165.124.184.176.43412: . ack 10677 win 443 <nop,nop,timestamp 281017106 281110633>
```

Event sequence	Type of Communication	Delay (us)
1	Receive Packet	0
2	Send Ack	**64**
3	Send Packet	5907
4	Send Packet	1
5	Receive Ack	497
6	Receive Packet	5825
7	Receive Packet	212
8	Send Ack	**72**
9	Send Packet	5861
10	Send Packet	21
11	Receive Ack	365
12	Receive Packet	5883

Table 4.1: Table showing the delays between consecutive events from the above trace, for the unloaded case

```
1189054440.788203 IP 165.124.184.175.35181 > 165.124.184.176.43412: P 8433:8481(48) ack 10677 win 443 <nop,nop,timestamp 281017106 281110633>
```

In the above trace we see IP *.173 sending a packet to *.176. After that *.176 sends an ack and then sends packets to *.172. *.172 then returns some packets, after which *.176 repeats the behavior with *.175. If we analyze the timestamps on the left, we can compute the time differential for each event from the previous event as shown in Table 4.1. We note that the ack is sent by *.176 immediately after a delay of 64 us, followed by a larger delay of around 6 ms for sending the next packet. This intuitively corresponds to some processing and computation before sending the next message.

Keeping these inter-packet event delays in the mind for one of the processes (no external load), now let's examine the corresponding trace for the same application, where the process being monitored is subject to a full 100% computational load in another sibling Xen Guest machine.

```
1188377325.385764 IP 165.124.184.173.46574 > 165.124.184.176.33567: P 4289:5289(1000) ack 6485 win 315 <nop,nop,timestamp 111673138 111824712>
1188377325.409586 IP 165.124.184.176.33567 > 165.124.184.173.46574: . ack 5289 win 248 <nop,nop,timestamp 111824720 111673138>
1188377325.415445 IP 165.124.184.176.36308 > 165.124.184.172.55489: P 5437:5485(48) ack 4241 win 217 <nop,nop,timestamp 111824721 111805985>
1188377325.415467 IP 165.124.184.176.36308 > 165.124.184.172.55489: P 5485:6485(1000) ack 4241 win 217 <nop,nop,timestamp 111824721 111805985>
1188377325.415845 IP 165.124.184.172.55489 > 165.124.184.176.36308: . ack 6485 win 315 <nop,nop,timestamp 111806011 111824721>
1188377325.421760 IP 165.124.184.172.55489 > 165.124.184.176.36308: P 4241:4289(48) ack 6485 win 315 <nop,nop,timestamp 111806012 111824721>
1188377325.421944 IP 165.124.184.172.55489 > 165.124.184.176.36308: P 4289:5289(1000) ack 6485 win 315 <nop,nop,timestamp 111806012 111824721>
```

Sequence No	Type of Communication	Delay (us)
1	Receive Packet	0
2	Send Ack	**23822**
3	Send Packet	5859
4	Send Packet	22
5	Receive Ack	378
6	Receive Packet	5915
7	Receive Packet	184
8	Send Ack	**23621**
9	Send Packet	5965
10	Send Packet	28
11	Receive Ack	374
12	Receive Packet	5487

Table 4.2: The inter-packet event delays for the above loaded process case

```
1188377325.445565 IP 165.124.184.176.36308 > 165.124.184.172.55489: . ack 5289 win 248 <nop,nop,timestamp 111824729 111806012>
1188377325.451530 IP 165.124.184.176.39408 > 165.124.184.175.47575: P 6485:6533(48) ack 5289 win 248 <nop,nop,timestamp 111824730 111737671>
1188377325.451558 IP 165.124.184.176.39408 > 165.124.184.175.47575: P 6533:7533(1000) ack 5289 win 248 <nop,nop,timestamp 111824730 111737671>
1188377325.451932 IP 165.124.184.175.47575 > 165.124.184.176.39408: . ack 7533 win 347 <nop,nop,timestamp 111737698 111824730>
1188377325.457419 IP 165.124.184.175.47575 > 165.124.184.176.39408: P 5289:5337(48) ack 7533 win 347 <nop,nop,timestamp 111737698 111824730>
```

We notice something peculiar here. Table 4.2 shows the time differentials for the corresponding events. We note that the delay for sending the acks back from *.176 is huge (23822 us) compared to the previous unloaded case (64 us). In effect, the ball was in the court of *.176 to send the ack back so that sending for the partner process is confirmed. The time required for this responsibility was greatly inflated due to external load. Besides the differential corresponding to sending ACKs, the other differentials are comparable. Especially note that the delays that actually fall in the BIC delay for other processes are similar to the unloaded case, thus indicating that the other processes are not affected by this load. Examining the actual packet traces for other processes confirms this observation: their sending acknowledgement or sending packet differentials are unchanged.

The elongation in BIC delay for *.176 results in a dramatically slowed down application. When one of the processes is loaded, the application actually takes 52.32 seconds to execute compared to 18.37 s in the unloaded case.

4.2.2 Developing a Formal Strategy

The above observations lead us to an interesting question: is it possible to estimate the BIC delays for a process purely from the packet trace corresponding to a process? I have investigated this question and many different trace-based approaches to come up with a satisfactory answer. After some experiments I have formed a hypothesis and tested it successfully with various applications under different circumstances.

To get an estimate of the BIC delay of the process using only the packet trace (which is a black box requirement), I observed that every pair of consecutive events in a packet trace can be classified as either a BIC delay or a non-BIC delay. Each record in the trace can be either of the following:

1. Send Packet (SP)
2. Send Ack (SA)
3. Receive Packet (RP)
4. Receive Ack (RA)

As TCP packets have an ack piggy backed on them, such a packet can be actually viewed as two packets: The ack and the packet itself.

Now we can start pairing up consecutive events to form event pairs. For example, SA followed by SP would result in a SA-SP pair. From the 4 event possibilities, we can have 16 types of event pairs for every record in the packet trace. Intuitively and also supported by empirical experimentation, we can classify every * - S* combination as a BIC delay - the intuition is that the process has either received a packet and its upto the process now to send the next appropriate packet, or that it has sent a packet and has to send the next packet in the message. For example, after receiving a message from a partner process, its up to the local

BIC event pairs	Non-BIC event pairs
RA-SA	SA-RA
RA-SP	SA-RP
RP-SA	SP-RA
RP-SP	SP-RP
SA-SA	RA-RA
SA-SP	RA-RP
SP-SA	RP-RA
SP-SP	RP-RP

Table 4.3: A table showing the event pairs classified as Ball in the Court (BIC) delays and non-BIC delays.

process to send the ack. This event pair is a RP-SA pair and is classified as a BIC event pair. Table 4.3 shows all the possible event pairs and the classification into BIC and non-BIC delays.

To actually estimate the BIC delay profile of a process, we take the following inputs:

1. The trace corresponding to that process for its entire execution, or a part of it. It consists of n records, $record_1$ to $record_n$.

2. The IP address corresponding to the guest VM or the host containing the local process. (IP_{local})

For every record R_i in the trace, we create an event pair tuple $< E_{i_1}, E_i >$ where E_{i-1} corresponds to $event(R_{i-1})$ and E_i corresponds to $event(R_i)$. We also associate two properties with every event pair: The time differential for the event pair and the partner process IP involved in the current event. The time differential is simply the difference in arrival timestamps, $T_i - T_{i-1}$. The partner process IP is the other IP involved in the record with the local process IP. Thus for each record R_i in the packet trace, we extract 4 pieces of information:

1. The timestamp corresponding to the record = T_i.

2. The event pair tuple $< E_{i-1}, E_i >$.

3. The time differential for these event pairs.

4. The partner process IP communicating with the local process IP.

All of these are output in a separate file also, to facilitate manual debugging and investigation.

After extracting this information from the trace, we output the following after further processing of the above derived information:

1. The starting and the ending times of the trace (T_{start} and T_{end}).

2. The cummulative BIC delay: We sum up the time differentials corresponding to the BIC event pairs and this is termed as the *cumulative BIC delay*. This is the total estimate for the BIC delay for the process.

3. The BIC delay partitioned by partner IPs: Since different BIC event pairs correspond to different partner IPs, different BIC delays fall in different IP buckets. We output this paritioned BIC delay as well. The need for this will be further explained later.

4. The breakdown of BIC delays by different event pairs. This gives a finer picture of the BIC delays.

5. The breakdown of BIC delays by event pair-partner IP tuples.

The last two items are actually quite useful for debugging, when comparing the BIC delays for a process under different circumstances.

Some Concerns Regarding this Method

The above method seems fairly simple and may seem to disregard some complex situations especially scenarios that involve overlapping communications with two or more partner processes. It is also not clear how the computation is properly accounted for in the above BIC delay counting method.

For example, suppose the following sequence of event transpires:

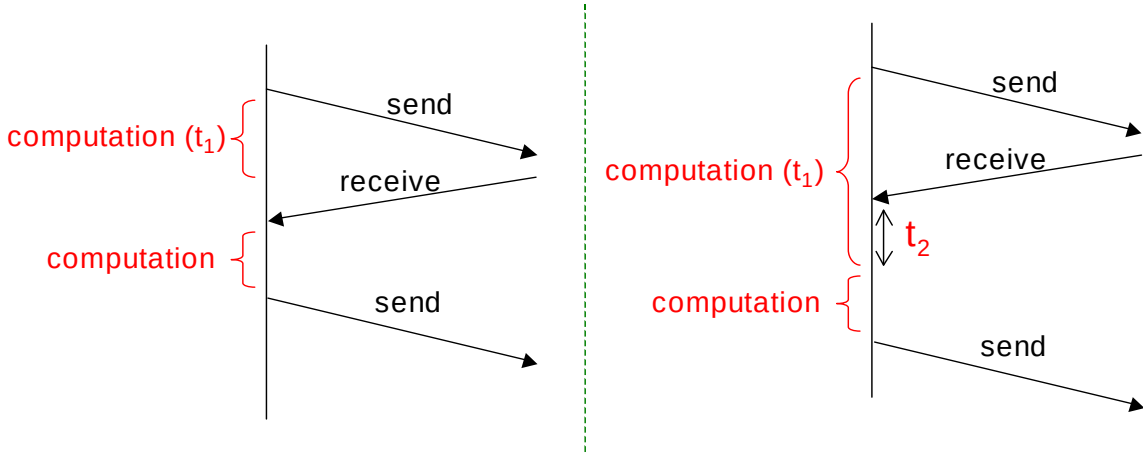


Figure 4.3: Two scenarios for computation illustrating if it also happens in the non-BIC region.

- A $\Leftarrow$ B,
- A $\Leftarrow$ C,
- A $\Rightarrow$ B

. In this case, A is sending a response back to B, but the BIC delay that will be counted will be the time difference in the last two events. It may be argued that the actual BIC delay is from the 1st event to the 3rd, not the 2nd to the 3rd. *This is actually a problem of finding the right dependencies between receives and sends*, which is quite difficult to do simply by looking at the trace. However in our experiments it turns out that, statistically, these random packet overlap effects cancel and balance out amongst multiple processes. Also, the communication schedules for most BSP applications are quite clean - the sequence of receives and sends is quite deterministic, thus reducing the cases of such overlap. Thirdly, it can be argued that the delay between the 1st and the 2nd event doesn't really need to be counted as a BIC delay since its not really blocking the progress of other processes, since A is actually waiting for a message to arrive from C.

One more concern is the proper accounting of local computation into BIC delays. If most

of the computation happens before sending a message, then it will be captured properly in the BIC event pairings since these capture the delays between receiving a message, doing some computation and then sending out the next message. However what about the computation that may happen in a non-BIC region? For example, the process may be computing after sending a message and until the next message arrives. Figure 4.3 shows two possible cases for such a scenario. In both the cases, time t_1 corresponds to the computation done when the process has sent a message and is waiting for a response. In the first case, t_1 is shorter than the time between sending and receiving the next message. This computation actually does not impede the overall progress of the application since the process is waiting for a message from the other host anyway. So this is not really a BIC delay. However, in the 2nd case, where the computation may actually last longer then the time period between sending and reception of a message, that time will be counted as part of the BIC delay in the above scheme, since it will be part of the RP-SP event pairing for example. Thus, any effects on slowdown will be captured properly with a high probability using the BIC delay approach. One of the claims here is that other random effects will be balanced out amongst different processes and will not bias the BIC delay negatively and positively for just one process and not for others.

4.2.3 Evaluation: BIC Times for a Balanced Application

Having discussed the idea, detailed principles and methods to capture the BIC delay for processes of a BSP application, lets examine the BIC delay for a sample BSP application. We run Patterns using the following command-line arguments:

```
cd ~/pvm3/bin/LINUX
pvmpattern all-to-all 4 1000 500 100 12000 50 50
```

s

Traffic capture: We then capture the traffic for all 4 processes using tcpdump using the following command line arguments:

```
tcpdump -n -nn -q -tt -v -i  vif1.0 tcp and not port 22 -w xen1-unloaded-1
```

The above arguments allow quick capture with lesser overhead. The `-q` switch captures directly in binary packet format without parsing and writing a text description. We also do filtering in this context to filter out packets that we know do not belong to the application. The `not port 22` argument for example, filters out the SSH traffic. The file generated can then be converted into text format for processing by a script.

In an online setting, the overhead of traffic capture is likely to be similar to the overhead in VTTIF's case described in Chapter 2. We mainly need to count the inter-packet timing differences for a set of selective events which is quite simple in principle and does not require any sophisticated trace processing.

The application is run without any external load on any of the processes. Under these circumstances we expect the BIC delay to be similar for all processes. The total runtime of the application observed is 18.4 seconds. The BIC delays reported for the 4 processes are:

1. 176 : 8.85 seconds
2. 173 : 8.45 seconds
3. 172 : 5.845 seconds
4. 175 : 8.50 seconds

From the above numbers, we see that the BIC delays are quite balanced for an application without any load. Almost all processes except one have the same BIC delays. One of the processes (172) has some difference in its BIC delay. However what is important here is the relative differences and here the imbalance is not high. That is our major concern here.

The report also outputs the number of BIC delays which were approximately 4500 *for all processes*. This indicates we are actually taking into account the same type of delays for all processes.

Now let us see what happens if we load one of the processes with an external load.

4.3 Figuring No-Load Run Time Using BIC Principles (Loaded Scenario)

Going back to our original problem statement, we want to find out if we can determine or predict the runtime of a stressed BSP application if it ran without the external stress. In this section we investigate an approach to solve this problem. We discuss various approaches, that vary in their complexity and accuracy and demonstrate the techniques with actual BSP applications.

To formalize the problem, we have a set of n BSP processes $P_1 ... P_n$, that may be under different load conditions. What we want to find out is that how would the overall application behave in terms of performance if a loaded process P_i were to perform without that external load. We do so by comparing to one or more other presumably unloaded or less stressed processes. The way we compare the loaded process to others, and try to predict the run time as if P_i was not loaded, can result in different approaches towards determining the improvement in performance of the application. We discuss 3 different methods to compare these processes and come up with a predicted no-load time for the application.

4.3.1 Global BIC Balance Algorithm

We first start with a relatively simple approach that works reasonably well in predicting the no-load runtime of a BSP application which has one of its processes externally loaded. We call this the Global BIC Balance Algorithm.

The way it works is by comparing the overall BIC delay of a loaded process with another presumably unloaded or less loaded target process and determining the imbalance in performance. As discussed before, the BIC or Ball in the Court Delay is supposed to give the time the process spends locally on the host. By comparing the BIC delays of different processes belonging to the same BSP Application instance, we get a quantitative estimate of how unbalanced different processes are.

In the end, what we want to find out is how might the overall application perform if a loaded

process P_i were to behave like another (presumably unloaded or less stressed) process P_j. It is assumed that all these processes belong to the same BSP application and are thus executing the same BSP kernel. We term P_i as the "loaded process" and P_j as the "partner process". To figure out the imbalance in the application, we have three steps:

1. Select out the "loaded process" that we think is delaying the overall execution of the application. To determine this there are various methods: We can monitor the local load on the physical hosts and select the most loaded machine assuming it has the most loaded process. Another approach is to take the process with the highest BIC delay as the biggest contributor in application slowdown. A combination of these two may also be used to validate each other.

2. Select a "partner" process and use it to measure the imbalance of the "loaded" process. The goal behind choosing a partner process is to ask the question "What if the loaded process was put in the conditions of the partner process ? How would the application behave then?" Depending on the circumstance and opportunities for performance adaptation, this partner process can be manually picked, we can pick the process on the least loaded machine automatically to get the most optimistic answers. Another approach is to use the BIC delays for several other processes to arrive at a range of possible performance improvement instead of a single value. This also takes into account any inherent imbalances in the BSP application regardless of external load, and thus could give a more realistic idea of the possible range of execution time improvement.

3. Compute the imbalances in the BIC delay between the "loaded" process and the other "partner" process(es). This imbalance gives an idea of the slowdown being caused by the loaded process.

As Figure 4.4 shows, we compute the global BIC delays for all processes belonging to the application. We call it the global BIC delay because a process's BIC delay actually contains

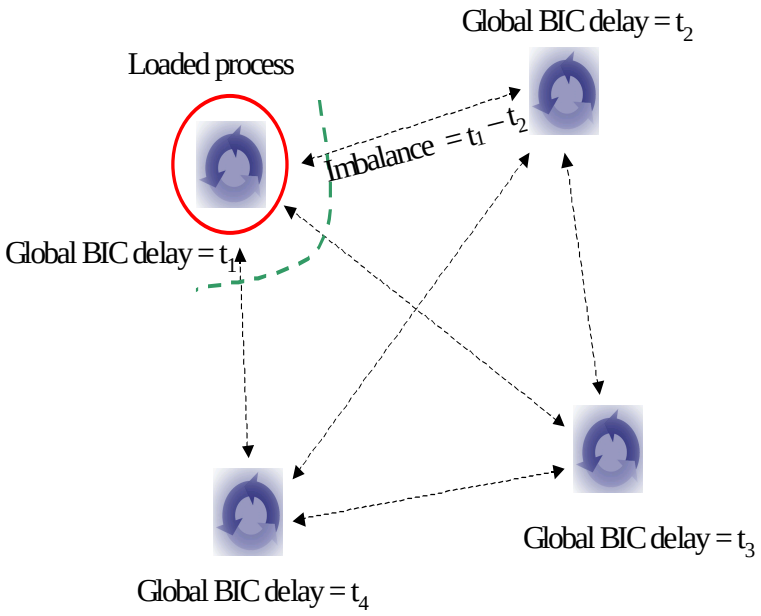


Figure 4.4: The Global BIC Balance Algorithm illustrated

components of interaction with different processes and the total BIC delay is the sum of all local processing. To get the extra BIC delay for the loaded process we simply subtract the BIC delay of the "partner" process.

Different estimates of slowdown: based on how we choose the "partner" process we can get the most optimistic and pessimistic estimates of performance slowdown.

For a total of k processes running for a BSP application, An upper estimate of slowdown can be obtained from the following equation:

$$Slowdown_{max} = Max_i\{BIC_l - BIC_i\} \tag{4.1}$$

where BIC_i denotes the BIC delay of the i^{th} process and and BIC_l denotes the BIC delay of the loaded process. Here i ranges from 1 to k except l.

A lower estimate of the slowdown is obtained by:

$$Slowdown_{min} = Min_i\{BIC_l - BIC_i\} \tag{4.2}$$

where i ranges from 1 to k except l.

The range of slowdown is then thus $[Slowdown_{min}, Slowdown_{max}]$.

As discussed earlier, sometimes we want to actually find out how the application might speed up if the "loaded" process behaved like a particular "partner" process. In that case we just use the BIC delay of that particular process and compute the imbalance.

4.3.2 Evaluation with Patterns Benchmark

We evaluate the above algorithm with the Patterns benchmark. We run the all-to-all topology version of Patterns with 4 processes in guest Xen VMs, one of which is externally loaded with another sibling guest VM. Obviously the entire BSP application is slowed down dramatically because of the affected process. We want to find out how drastically the application has actually been affected.

We run the previous Patterns application using the following parameters:

```
cd ~/pvm3/bin/LINUX
pvmpattern all-to-all 4 1000 500 100 12000 50 50
```

We impose full computational load on one of the physical hosts in a sibling Xen guest VM (Host 176). The total runtime for the same patterns application in this scenario is 52.34 seconds, which is a substantial increase over the 18.4 second runtime in the unloaded case. Now let us look at the BIC delays for the loaded vs the unloaded processes. On processing the TCP packet traces for the loaded guest VM process vs an unloaded one, the BIC delays were as follows:

1. 176 : 43.098 seconds (loaded)

2. 173 : 8.59 seconds (unloaded)

The above numbers show something really striking. We see that for 173 the BIC delay is practically unchanged! However for the loaded host, where an extra loaded Xen VM is running,

the BIC delay is greatly increased to 43.098 seconds. The extra imbalance caused by this stressed process can be calculated by the difference between its BIC delay and another relatively ”unloaded” process. The intuition here is that we want to find out what would happen if the loaded process was instead running in the same condition as the other process that has a lesser BIC delay. In this case the difference is 34.508 seconds. If we subtract this from the total run time (52.34 seconds), we get a run time of 17.83 seconds. This is almost the same as the original runtime of 18.4 seconds in the unloaded case.

This indicates that measuring the BIC delays as discussed above can be a very powerful method to detect imbalance amongst processes in a BSP application, which can then be used to infer the runtime of the application once these BIC imbalances are taken into account. In this case we were able to infer almost exactly the runtime of the patterns application.

4.3.3 Applying the Algorithm on the IS Application

In this section we examine the IS application using the simple Global BIC algorithm to compute the imbalance.

Balanced case: In this case the total runtime is 14.36 seconds. BIC delays for all processes in the unloaded case:

1. 176 : 4.92 seconds
2. 175 : 4.59 seconds
3. 173 : 5.22 seconds
4. 172 : 3.66 seconds

The BIC delays are quite balanced in the unloaded case. The sum is not equal to the total runtime of course, since the BIC delays actually overlap amongst different processes.

Now let us look at the loaded case for the IS application. The total runtime in this case is 152.67 seconds which is 10 times larger than than the unloaded case (14.36 seconds). Being

able to tell this just by looking at the loaded case would be very useful to help migrate or adapt the application and speed it up.

Now let us look at the BIC delays for each process:

1. 176 (the loaded host) : 133.58 seconds
2. 175 : 9.09 seconds
3. 172 : 10.75 seconds
4. 173 : 10.16 seconds

We see that the BIC delay of the loaded host is severely bloated compared to other hosts. If we compare the BIC delays of 176 to 175, *we forecast a difference of 124.49 seconds for the unloaded case*. The actual difference in runtime between the loaded and the unloaded case is 138.41 seconds. The BIC imbalance is actually pretty close to indicating how much extra time the loaded process is taking. Here it is off by around 10% using the simple BIC difference technique. Getting such a good measure of the extra delay in the runtime of the process just using passive measurements is quite promising and convenient.

4.4 Process-level BIC Imbalance Algorithm: Balancing Out Biases

In the previous section, we saw a relatively simple algorithm that used the concept of BIC delay to determine the quantitative slowdown of the BSP application. Though the algorithm gives a rough estimate and worked well for the relatively simple Patterns application, in this section we introduce a more refined and sophisticated algorithm that takes different process level BIC components into account to compute a more accurate estimate of the slowdown.

Before we discuss the algorithm, I summarize some of the observations in the course of my investigation with more complex BSP applications, that eventually led to this algorithm.

- It often turns out that even though all the processes have the same BSP kernel, the amount of work they end up doing is not identical. This makes using the Global BIC delay approach less accurate as it assumes that the BIC delay as a global metric regardless of the individual process-level interaction.

- Another phenomenon that I call the "load bias" comes into play if one of the process is heavily loaded compared to other processes. The loaded process can negatively affect the performance or the BIC delay of a non-loaded process too. Thus the BIC delay of a process that remains unloaded gets inflated because of its interaction with the loaded process. This inflation should actually be accounted in the BIC delay of the "loaded" process, not the "partner" process.

- The Global BIC delay algorithm uses the BIC delay of one of the processes to compute the slowdown for that partner process or the bounds of the range of slowdown. However taking all processes into account to arrive at the slowdown can result in a more accurate estimate of the slowdown. This is especially applicable if the different "partner" processes are loaded differently and not running under the identical conditions of zero load. Under such circumstances, if we just use one partner process to compute the imbalance, it will result in a more inaccurate estimate of slowdown, since the other processes may not be in conditions similar to the partner process used to compute the imbalance.

The above observations have led to a more refined approach to computing the BIC imbalance and then computing the ultimate slowdown. Instead of using the global BIC delay of the process, it actually breaks down the BIC delay based on process interaction or the IP address of the BIC delay event pair. For example, for the RP-SP BIC delay event pair, the BIC delay gets classified according to the IP address of the receiver in the SP (Send Packet) event. Thus, for all BIC delay event pairs combined, we get the BIC delay according to the recipient IP addresses. Once we know the inter-process BIC delay, not just the global BIC delay, we can compute the

imbalance at the process level. This is the first refinement in the algorithm.

The second major refinement is the computation of the "load" bias that the loaded process might have caused in the partner process. As we discussed above, the loaded process can cause a slowdown in the partner process. We try to compute this "load" bias and account this in the BIC delay of the "loaded" process.

4.4.1 The Algorithm

Input:

- Inter-process BIC delays of all processes. We denote by BIC_{ij} the cumulative BIC delay of process i for all BIC delay event pairs whose recipient is process j. Thus for k processes, we will have $k(k-1)$ inter-process BIC delays, $k-1$ for each process.
- The loaded process that we believe is causing the slowdown of the BSP application and if migrated towards better load situations could result in improvement in performance. We denote this process by P_l.

Output:

- A quantitative estimate of slowdown for the BSP application caused by the suspected "loaded" process. This could be a range denoting the bounds of slowdown for the given instance of the application.

The Algorithm:

1. **Bias Computation Step**: For each non-loaded process P_i where $i \neq l$, compute a *pessimistic* and *optimistic* estimate of the load bias caused by the "loaded" process.
   The most pessimistic estimate of bias is given by:

$$\begin{aligned} BIC_{i,min} &= Min_j \{BIC_{ij}\} \; \forall \; j \\ BIAS_{i,pes} &= \sum_{j=1}^{k} BIC_{ij} - k \times BIC_{min} \end{aligned} \tag{4.3}$$

The above equations assumes that the best BIC delay of process *i* towards other processes is the minimum BIC delay we find towards any other process. If we subtract that minimum BIC delay from the other BIC delays (which are obviously more than the minimum BIC delay, thus indicating that the process *i* is acting slower towards other processes), then the residue is the bias, or the slowdown towards other processes that is *assumed* to be caused by the loaded process. This is the pessimistic estimate because this is the maximum bias that we can compute since we take the minimum BIC delay as the baseline.

The most optimistic estimate of bias is given by:

$$\begin{aligned} BIC_{i,avg} &= Average\{BIC_{ij}\} \ \forall \ j \\ BIAS_{i,opt} &= \sum_{j=1}^{k} BIAS_{ij}^{avg} \quad \text{where} \\ BIAS_{ij}^{avg} &= \begin{cases} 0 & \text{if } BIC_{ij} \leq BIC_{avg}, \\ BIC_{ij} \ - \ BIC_{avg} & \text{if } BIC_{ij} > BIC_{i,avg} \end{cases} \end{aligned} \tag{4.4}$$

The above expression computes a more optimistic number for the slowdown bias that is assumed to be caused by the loaded process *l*. Instead of taking the minimum, we take the average of the BIC delays of process *i* towards other processes. Then we subtract that average from BIC delays for process *i* that *exceed* this average. This gives a more conservative estimate of the bias.

The above optimistic and pessimistic estimates of the load bias are made using the assumption that any differences of Process P_i towards other processes are caused by a slowdown bias caused by the loaded process P_l. We attribute the reason for this difference to the "loaded" process.

Reason for choosing Minimum and Average for pessimistic and optimistic bias: The slowdown bias caused in Process P_i due to the loaded process P_j can be computed by assuming an ideal BIC delay of P_i towards other processes if this bias had not been present. The way we compute this ideal bias affects our estimate of the slowdown bias.

Choosing the minimum of all BIC delays exhibited by process P_i towards other processes is the case that assumes the most optimistic case of ideal BIC delay and thus the most pessimistic case of the slowdown bias, since we subtract the ideal BIC delay from the actual BIC delays experienced by process P_i towards other processes to compute the bias.

Choosing the average for the optimistic case of the slowdown bias requires some more explanation. First of all when we choose the minimum BIC delay of process P_i towards others, we make the assumption that the BIC delay is actually the same towards all processes. In practice this is not the case and there is usually some intrinsic imbalance in the BIC delays of a process towards other processes, as seen in the balanced case for the MG application (Table 4.4). Thus taking the minimum BIC delay is overly optimistic. The other end of the spectrum is to compute a higher value of the ideal BIC delay that could somehow take into account this intrinsic imbalance in the BIC delays. We compute this ideal BIC delay by re-distributing the total global BIC delay experienced by P_i equally towards rest of processes. *This ensures that we do not exceed the global BIC delay for P_i - we do not make the process slower overall.* From this higher value of the ideal BIC delay, we compute what may be the extra bias for process P_i from the average BIC delay. This gives a much more conservative value of the bias and is thus the optimistic version of the slowdown.

2. **Imbalance computation step**: Once we have computed the estimates of the load biases, we now compute the un-biased BIC imbalance for the loaded process at the inter-process level and then sum them up. To compute this BIC imbalance for the loaded process, we compute for each non-loaded process i, the following quantities:

$$\begin{aligned} Imbalance_{li}^{opt} &= BIC_{li} + BIAS_{i,opt} \quad - (BIC_{il} - BIAS_{i,opt}) = BIC_{li} + 2.BIAS_{i,opt} - BIC_{il} \\ Imbalance_{li}^{pes} &= BIC_{li} + BIAS_{i,pes} \quad - (BIC_{il} - BIASi, pes) = BIC_{li} + 2.BIAS_{i,pes} - BIC_{il} \end{aligned} \tag{4.5}$$

Explanation: In Step 1, we computed the "load bias" caused by the loaded process, i.e.

the extra BIC delay that may have been caused by the "loaded" process. Our goal here is to actually include that extra BIC delay in the BIC delay of the "loaded" process so that we attribute it correctly. In the equations above, we are doing exactly that. To compute the bounds for the possible imbalance, we first add the bias to the BIC delay of the "loaded" process, since we attribute it to the BIC delay of the process causing it. Then we subtract from it the adjusted BIC delay for the partner process P_i, which is equal to its BIC delay minus the "bias". This adjusted BIC delay reflects the BIC delay it might have towards the "loaded" process without the bias. So we end up subtracting the "load bias" twice in the final expression.

Since we have two values for the imbalance towards each "partner" process, the optimistic and pessimistic values, we use each of those to compute the optimistic and pessimistic imbalance values towards each partner process.

3. **Imbalance accumulation step**: After computing the extremes of possible imbalance towards each partner process, we sum up these imbalances to come up with a resultant cumulative imbalance towards all of the partner processes.

$$\begin{aligned} CumulativeImbalance_l^{opt} &= \sum_{i}^{i \neq l} Imbalance_{li}^{opt} \\ CumulativeImbalance_l^{pes} &= \sum_{i}^{i \neq l} Imbalance_{li}^{pes} \end{aligned} \tag{4.6}$$

These two values of cumulative imbalance give an estimate of the bounds by which the application may have been slowed down due the loaded process. We can then denote the range of slowdown of the application by the bounds $[CumulativeImbalance_l^{opt}$, $CumulativeImbalance_l^{pes}]$. This final range denotes the final result we want.

Notice that the above algorithm differs from the Simple Global Imbalance Algorithm in a couple of ways. First of all it uses the BIC delay information from all the "partner" processes,

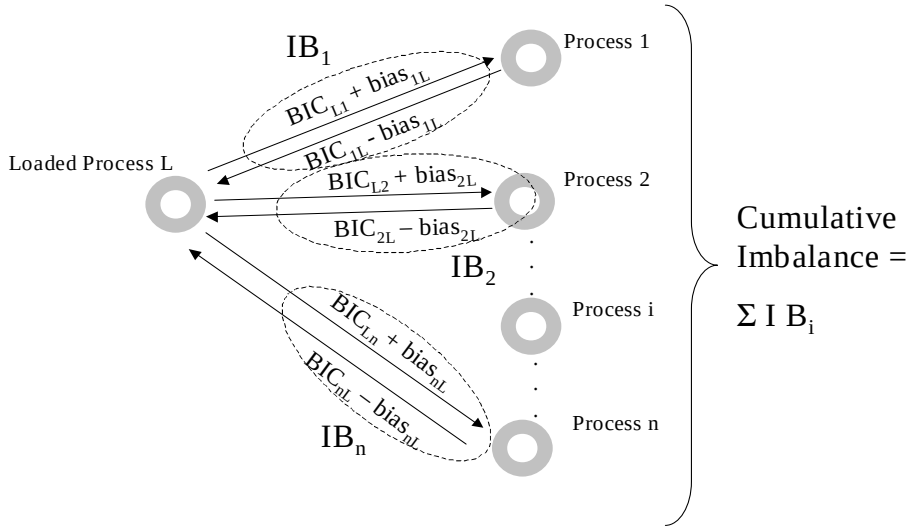


Figure 4.5: The Bias Aware Process Level BIC algo Illustrated. The Biased BIC delays for each process pair are shown. Summing these up produces the cumulative BIC imbalance

not just one partner process. Thus more measurement information is required for this algorithm. Secondly, it gives a range of Imbalance, both optimistic and pessimistic values based on how we determine the "true" BIC delay of the other "partner" process, and taking the "loaded" process bias into account. In the Global Imbalance Algorithm, we got a single value of imbalance depending on which partner process we used to compute the imbalance. A range could also be deduced in that algorithm by taking the minimum and maximum of this imbalance over various "partner" processes, but that range differs from the range computed in the current algorithm.

Complexity

The above algorithm needs to calculate the bias for each of the processes P_l communicates with. They could be n in number. For each such process, we further need to calculate the bias that may further take n steps for each of its interaction with all the other processes. Thus algorithm's time complexity is $O(n^2)$. It has the same $O(n^2)$ space complexity due to the same reasoning.

In the following sections we evaluate the above algorithm with some complex BSP applications, namely MG and IS.

BIC	172	173	175	176
172	x	0.65	0.00	1.66
173	7.72	x	0.45	0.21
175	0.01	2.95	x	0.86
176	0.53	0.00	8.61	x

Table 4.4: The BIC delays of the 4 processes for the MG application under the balanced condition, with no external load.

4.4.2 The MG application

As described earlier the MG or multi-grid benchmark from the NAS pack of BSP benchmarks uses hierarchical algorithms, such as the multi-grid method, that lie at the core of many large-scale scientific computations [25, 26, 29, 71]. It is typical of a popular class of BSP applications. We evaluate the MG Application under load and try to infer the run time of the application without load, without changing anything in the environment.

First, let's see what happens if we run the MG application under a balanced setting, without any external load.

Table 4.4 shows the inter-process BIC delays of the 4 processes (labeled 172, 173, 175 and 176 according to the last 8 bits of their IP address), when no external load is imposed. In this situation the running time of the BSP application is 13.58 seconds. It is important to understand that for each process, we break up its BIC delay according to the "partner" process corresponding to the particular BIC event. Thus we do not deal with the global BIC delay but the inter-process BIC delay. In the table we see that the inter-process BIC delays for different processes vary a bit but still contained within a certain range (2 to 8 seconds). They are not exactly the same as in the Patterns application because the way MG works is not completely balanced and somewhat unequal amounts of work may be distributed to different processes. For each process in the table, we see that it has a negligible BIC delay towards one of the processes (e.g. see the BIC delay for 173 to 176 and vice versa). This is because in MG, each process communicates with two other processes primarily, thus having significant BIC delays for only

two of the remaining 3 processes.

Now after seeing the balanced inter-process BIC delays, let's see the case where we load one of the process's (process 176) host machine with another loaded Xen VM. This would slow down the performance of the entire application drastically. In fact the run time of the application in this case is 56.25 seconds. This is a slowdown of over 42.67 seconds, or 314% over the original run time! Now let us look at the inter-process BIC delays for the loaded case. Table 4.5 shows the inter-process BIC delays for the loaded case of the MG application.

Now if we look at the BIC delays for the process 176, they are considerably larger than the other processes. This validates our theory of inflated BIC delays in the case of external load. If we compute the BIC imbalance for the loaded process with the other "partner" processes, we get a range of imbalance. We have two cases, the optimistic case and the pessimistic case. For the optimistic case, we compute the imbalance for each partner process assuming the impact of the loaded process is minimum. For each process we get the following imbalance:

- Pair 176-172: $BIC_{176-172}$ is 18.60 seconds. $BIC_{172-176}$ is 11.74 seconds (most optimistic). So the imbalance is 6.86 seconds.

- Pair 176-173: The BIC delays for this pair is negligible as they hardly communicate with each other.

- Pair 176-175: $BIC_{176-175}$ is 24.79 seconds. $BIC_{175-176}$ is 2.98 seconds (most optimistic). Note that 175 has a BIC delay of over 9.6 seconds towards 173. The total imbalance for this pair is 21.81 seconds.

From the above, the total BIC imbalance in the optimistic case is 21.81 + 6.86 = 28.67 seconds. This is the lower bound for the slowdown.

Now let's examine the pessimistic bound. We compute the inter-process imbalance as follows:

BIC	172	173	175	176
172	x	1.37	0.01	11.74
173	5.65	x	1.85	0.21
175	0.02	9.61	x	2.98
176	18.60	0.08	24.79	x

Table 4.5: The Inter-process BIC delays for the MG application in the loaded case.

- Pair 176-172: $BIC_{176-172}$ is 18.60 seconds. $BIC_{172-176}$ is 11.74 seconds. But $BIC_{172-173}$ is 1.37 seconds. In the pessimistic case, we assume that the extra delays are caused by the loaded process. In this case this bias amounts to 11.74 - 1.74 = 10.37 seconds. So we take the effective $BIC_{172-176}$ as 11.74 - 10.37 (the bias) = 1.74 seconds. We take the effective $BIC_{176-172}$ as 18.60 + 10.37 (the bias) = 28.97 seconds. So the net imbalance for this pair is 28.97 - 1.74 = 27.23 seconds.

- Pair 176-173: The BIC delays for this pair is negligible as they hardly communicate with each other.

- Pair 176-175: $BIC_{176-175}$ is 24.79 seconds. $BIC_{175-176}$ is 2.98 seconds. All the other BIC delays for 175 are only more than 2.98 seconds, so we cannot be more pessimistic in the imbalance than using this value itself. Thus the total imbalance for this pair is 21.81 seconds.

From the above adjusted pessimistic imbalance, we get the total pessimistic BIC imbalance as 28.97 + 21.81 = 50.78 seconds.

Thus from the above two steps, using our algorithm we get the slowdown range as [28.67 seconds, 50.78 seconds]. The actual slowdown 42.67 seconds, does lie within this range. This shows how the process-level BIC imbalance algorithm gives a good quantitative estimate of the slowdown of the MG application under load.

4.5 BIC Reduction Graph Approach for a Multi-load Situation

In this section we take on a slightly more complex version of the load problem: suppose more than 1 process belonging to the application is loaded. This is a multi-load situation that will cause a inflated BIC delay in more than process. Our goal is to still infer the no-load runtime of the application under such a circumstance. This can be very useful in practical situations for obvious reasons.

To solve the multiload problem, we developed an extension of the process-level BIC imbalance algorithm, that applies the process iteratively resulting in a multi-step reduction process. The goal is to balance out the BIC delays appropriately amongst processes so that there are no big imbalances in BIC delay. For example, if we load two processes with an external load, their BIC delays will inflate compared to others. Even if we compute the imbalance for one loaded process, we will still have imbalance in the system because of the other loaded process, thus giving more scope for further reducing the imbalance.

In the algorithm described earlier, we had two bounds: a pessimistic one and an optimistic one. For each bound, we resolve the imbalance in different ways. In our current extension, I develop a more generic version of the previous process-level BIC imbalance algorithm.

Looking at the algorithm at a high level, what we are trying to do is to resolve the imbalance for the most loaded process at a process level. That is, we look at its BIC delays towards its other partner processes that it may be communicating with, and then try to count the imbalance. In a more generic sense, we can actually continue the process of counting this imbalance after the first loaded process. We need not stop the balancing act after the first loaded process. Specifically, we had two bounds that we defined: the pessimistic bound and the optimistic bound. For each of these bounds, we could continue the process of balancing until we reach a "fair" amount of balance amongst the processes i.e. there are no obviously loaded processes. The downside to this approach is that if we continue the process of correcting imbalance too

far, we may end of stretching the range far too much (i.e. the upper limit may become too large) and it may reduce in its usefulness. So proper caution has to be undertaken when we keep on trying to push the bounds.

4.5.1 Multi-iteration BIC-delay Bias-Aware Imbalance Algorithm

In this section we describe the multi-step process for the previously discussed process level BIC-delay Imbalance algorithm. Each step consists of these high level steps repeated over and over:

1. Identify: Identify the *most loaded* process to balance.
2. Test: See if it meets the requirements for a *loaded process.*
3. Balance: Apply the process-level BIC imbalance algorithm balancing steps and re-label BIC delays according to the method used: optimistic or pessimistic. Also note the "imbalance" removed using this balancing step.
4. Repeat: Repeat the process until we fail to obtain a *sufficiently* loaded process.

As explained before, we want to put a limit on when to stop balancing, so as not to "over-balance" the BIC delays resulting in over-estimate of the slowdown. In any BSP application, there is bound to be some amount of inherent imbalance and we do not want to completely flatten it out.

Each of these steps requires discussion:

1. **Step 1 - Identify**: Of all the processes that are available for balancing, ideally we should pick the one that is the current bottleneck for the whole BSP application. In a symmetric BSP application, one simple way is to choose the process that has the highest global BIC delay. Another more involved way is to compute the BIC imbalance using the inter-process method and then choose the process contributing to the highest imbalance.

2. **Step 2 - Test**: After we identify the next "loaded" process, we also need to find out if its worth continuing the process of balancing for the reasons mentioned above. For this we need to set a metric. For example, if the current imbalance value is x seconds, we choose to continue this step only iff this loaded process is causing an imbalance of atleast px seconds, where $p < 1$. In our case, for example, we choose a value of $p = 0.1$.

3. **Step 3 - Balance**: In this step we do the actual balancing act, by copying over the new BIC delay values from the "partner" processes that this loaded process is imbalanced with. The new BIC delay values used to compute the imbalance actually depend on the approach used: for optimistic we could take the average of the BIC delays of the partner process and for pessimistic we could take the minimum of the all the BIC delays of the partner process.

   An important part of this step is the relabeling of the BIC delays for the loaded process. After computing the imbalance, we relabel the BIC delays of the loaded process with the new BIC delays computed from each of the partner processes that the loaded process is communicating with. Note that this step was not emphasized in the previous section where only one step was actually undertaken to compute the imbalance. However in our multi-step version of the algorithm, we need to make sure the new BIC delays are used to re-label the loaded process so that we can continue the process of balancing with the next loaded process.

4. **Step 4 - Repeat**: The above steps are repeated until we terminate at Step 2 - Test after certain number of steps.

Complexity

For each balancing step, the number of operations can be $O(n^2)$, due to the same reasons as explained in the previous section. However in this multi-step version of the algorithm, each

BIC	172	173	175	176
172	x	3.97	0.01	12.38
173	8.73	x	1.17	0.19
175	0.03	10.67	x	2.28
176	22.96	0.13	26.11	x

Table 4.6: BIC delays for the 4 processes in the MG application under a multi-load situation. Process 176 is subjected to full external computational load and Process 172 to a partial one.

balancing step can occur $n-1$ times in the worst case, though it may stop much earlier in practice owing to the test stage(Step 2). Thus the overall time complexity of the algorithm is $O(n^3)$, where n is the total number of processes.

The space complexity of the algorithm remains $O(n^2)$. As each step is independent from the previous step, there is no extra space use being introduced at each step of the iteration.

4.5.2 Evaluation with the MG application

To illustrate the above method, we run the same MG application as before, but this time under a multi-load situation. We impose a full external computational load on one of the physical hosts using a sibling Xen VM (on process 176) and a partial computational load (60% when VM is isolated) on the physical host running another process (process 172).

Under these conditions, the runtime of the application turns out to be 65.16 seconds, compared to the 13.58 seconds runtime without any load on the host machines. This amounts to a slowdown of 51.58 seconds, much more than the 42 seconds slowdown experienced in the single loaded case before.

We now apply the algorithm to the above table to compute the new BIC delays and start computing the imbalance after each of the steps.

Iteration 1

1. We note that the global BIC delay of Process 176 is the largest compared to others (49.2 seconds) so we choose it as the loaded process.

BIC	172	173	175	176
172	x	3.97	0.01	8.2/3.97
173	8.73	x	1.17	0.19
175	0.03	6.5/10.67	x	2.28
176	8.2/3.97	0.13	6.5/2.28	x

Table 4.7: The new BIC delays for different processes after Step 1. Note that two BIC delays are shown for some processes for the optimistic and pessimistic cases respectively.

2. Since currently the imbalance is zero, we continue the process of balancing.

3. Note that we compute two values of new BIC delays for the loaded process, corresponding to the optimistic and the pessimistic approaches. In this case, 176 communicates with 172 and 175, with negligible communication with 173. For the optimistic case, we compute the average BIC delay for the partner processes 172 and 175. They are 8.2 and 6.5 seconds respectively. For 176-172 pair, since the new BIC delay for 172 is 8.2 seconds, the imbalance is 22.96 - 8.2 + (12.38 - 8.2) = 18.94 seconds. Similarly for 172-175 pair, the BIC delay for 175 is 6.5 seconds. The imbalance is thus 26.11 - 6.5 = 19.61 seconds. Thus the total imbalance is 19.61 + 18.94 = 38.55 seconds, for the optimistic case, after one step. We also label the new BIC delays for 176 towards 172 and 175 as 8.2 and 6.5 seconds respectively.

   Similarly for the pessimistic case, where we take the minimum of the BIC delays of the partner processes as the new BIC delay, the new BIC delays for 172 and 175 are 3.97 and 2.28 seconds respectively. The imbalance for 176-172 is 27.4 seconds, including the bias. For 176-175, the imbalance is 23.83 seconds. Thus the total imbalance comes to 51.23 seconds. We also relabel the BIC delay of 176 with 3.98 and 2.28 respectively.

At the end of this Step 1, we have the range of slowdown as $[34.37, 51.23]$ seconds. We also have labeled the BIC delays of 176 for the two different approaches. After the re-labeling, the new BIC delay table looks as shown in Table 4.7.

Iteration 2

We now proceed to the next step in the balancing process. Our starting reference point is now Table 4.7 that we got after applying Step 1.

1. First we identify the most loaded process. In this case, we choose 172 as the most loaded process.

2. To determine if we should consider the additional BIC imbalance because of 172 we process step 3 and then apply the test for including the new loaded process.

3. Now we compute the optimistic and pessimistic imbalances with 172 as the loaded process. 172 communicates with 173 and 176. For the optimistic approach, 172 is actually balanced with 173. With 176, its BIC delay is 8.2 seconds. For 176, it's average of BIC delays would be 7.35 seconds. This leads to an imbalance of 8.2 - 7.35 = 0.85 seconds. Thus the total optimistic imbalance is 0.85 seconds seconds.

   For the pessimistic approach, in the 172-173 pair, the minimum BIC delay for 173 is 1.17 seconds. Thus the imbalance in this pair would be 3.97 - 1.17 + (8.73 - 1.17) = 10.36 seconds. For the 172-176 pair, the min delay for 176 is 2.28 seconds. Thus the imbalance is 3.97 - 2.28 = 1.69 seconds. Thus the total imbalance is 10.36 + 1.69 = 12.05 seconds.

4. We see that for the optimistic bound the increase in imbalance (0.85 seconds) is not significant. Thus we do not include it. However we do include the imbalance for the pessimistic bound (12.05 seconds). Thus the revised range for the slowdown is [34.37, 63.28] seconds.

We choose to stop the multi-step process here since the increase in imbalance has greatly slowed down at this step. We see that the actual slowdown of 52 seconds lies within this range.

4.6 Issues

While capturing the traces and processing the results to compute the BIC delay for each, there are couple of things to keep in mind:

- **Clock synchronization:** Since we want to make sure we capture the traces for the same application instance and then compare BIC delays across this instance, we want to make sure the clocks across these machines are synchronized. We have used *NTP* [147] for this purpose. The millisecond-level synchronization precision provided by ntpd (ms level) works pretty well for our purpose.

- **Anchoring while measuring the BIC delays:** When we are capturing traces, there might be other interfering packets in the trace and we want to make sure we start counting the BIC delay at the right point in time for processes belonging to the same application instance, especially if the traces are captured for a longer period than what we are measuring. For this purpose we choose one process as the "anchor" process with its correct starting and ending times. With this process as the anchor, we make sure we process the trace for other processes correspond to the same time period as the "anchor" process so that we count the BIC delays for the same time period. The key is identifying the right time period for the "anchor" process. In the offline case, this could be identified using the SYN/FIN packets in the trace. For the online case, we just capture and process the trace precisely for the period for which we are measuring the BIC delay and then process the BIC delay for the same time period for all fellow processes.

4.7 A New Metric for Global Performance Imbalance of a BSP Application

We introduced the concept of the BIC delay and then applied it to compute the imbalance across processes and ultimately a quantitative measure of the possible slowdown of the BSP applica-

tion. We now discuss another application for the BIC delay: Imbalance in BIC delay across the application. The idea is to give a quantitative measure of how imbalanced the processes of the BSP application are in local processing. This can give a great idea of the heterogeneity of the environment and possibly the input affecting the benefits of parallelization of the application. A highly imbalanced application can indicate the need for greater attention compared to other applications, for purposes of adaptation etc. It can be useful metric in adaptation algorithms so as to make sure no application is adversely affected by external factors and causing a huge imbalance in the application.

There are two possible ways to compute this imbalance: Global and at the inter-process level. We talk about some possible imbalance metrics for both of these types.

4.7.1 Global Imbalance Metrics

In the global metrics, we use the Global BIC delay for measuring the imbalance and forego the inter-process level imbalances in the system. This can give a good idea of how different the are local processing times for different processes participating in the application. There are two useful metrics:

1. **Standard deviation from average:** These are the simplest possible measures of imbalance. We can define:

$$Imbalance_{stdev} = stdev\{BIC_{p_i} \; \forall \; i\} \tag{4.7}$$

   For example for the MG application, the BIC delays for different processes are 2.31, 8.38, 3.82 and 9.14 seconds in the balanced case(Table 4.4). $Imbalance_{stdev}$ for the balanced case is thus 3.36. In the loaded case(Table 4.5), the global BIC delays are 13.12, 7.71, 12.61 and 43.47 seconds. $Imbalance_{stdev}$ for the loaded case is thus 16.34, which significantly larger than the balanced case!

2. **Squared Distance from Minimum:** This measure computes how far much larger the

BIC delays of all processes in an application are from the minimum. If the BSP application is symmetrical, it gives an idea of how much more work other processes are doing compared to the one with the minimum BIC delay. We define it as:

$$Imbalance_{min-distance} = \sqrt{\sum (BIC_i - BIC_{min})^2} - BIC_{min} \; \forall \; (i \neq min) \tag{4.8}$$

Note that we subtract BIC_{min} from the expression to account for differences in imbalances if two applications with different running times show different difference from the minimum.. If two applications show the same differences in BIC delays from the BIC_{min}, the imbalance in the application with the smaller BIC_{min} is much larger. Using the above expression, the Imbalance will actually be reported as negative if the imbalance is less than the min BIC delay.

As an example, in the balanced MG application(Table 4.4) case, $Imbalance_{min-distance}$ comes out to be 6.95. For the loaded case(Table 4.5), $Imbalance_{min-distance}$ is 28.78, much higher than the balanced case.

4.7.2 Process Level Imbalance Metric

This metric takes into account the inter-process level BIC delays to compute the imbalance amongst different processes. The idea here is to compute the difference in BIC delays among different processes. Thus, even though processes may have similar global level BIC delays, it can give us a finer grain picture of any imbalances inherent in the application because of the nature of input or the algorithm. We define it as the difference in BIC delays across all inter-process interactions:

$$Imbalance_{inter-process} = \sum \mid (BIC_{ij} - BIC_{ji}) \mid \; \forall \; i, j \; where \; i \neq j \tag{4.9}$$

For the balanced MG case 4.4, $Imbalance_{inter-process}$ comes out to be 18.66. For the loaded case 4.5, it is 40.48. We see that the difference in the measure if imbalance is not that large compared to the global measures of imbalance. This is more of a measure of the internal

	Balanced case	Loaded case	% Change
$Imbalance_{stdev}$	3.36	16.34	386%
$Imbalance_{min-distance}$	6.95	28.78	314%
$Imbalance_{inter-process}$	18.66	40.48	117%

Table 4.8: Table showing the different imbalance metric values for the balanced and the loaded case for the MG application

imbalance in the application which may be present even if the application seems to be balanced at the global level.

Table 4.8 summarizes the results from different imbalance metrics for the balanced and the loaded cases of the MG Application.

In the next section we talk about how these metrics can actually be used to understand different aspects of imbalance and performance issues with the application.

4.7.3 Which Balance Metrics are most Appropriate?

We have introduced two types of imbalance metrics: global and process-level. Which of these are more appropriate? As we noted above, it depends on the purpose for which we use them. For example, global imbalance can be used to immediately measure the amount of heterogeneity in work done by different processes. It can also be used to get an approximate idea of the extent of the slowdown of the process by comparing other processes with the fastest process. We list the suitability of different balancing metrics below:

- **Heterogeneity**: $Imbalance_{stdev}$ is a good measure of the variance in different BIC times of different processes and hence is a good metric to expose the heterogeneity in local processing times. It does not expose the *cause* of heterogeneity though: it may be induced the application itself, the input or the external environment.

- **Extent of slowdown**: $Imbalance_{min-distance}$ gives a great idea of how slow the application might be running compared to the fastest process i.e. the process with the min BIC delay. This can also indicate how much application performance could be improved.

- **Internal imbalance in the application**: The previous global metrics do not take into account different BIC delays that a single process may exhibit towards different processes. For example, process A may have vastly different BIC delays towards B and C, even though their global BIC delays may be similar. $Imbalance_{inter-process}$ gives a great idea of imbalance at that level. This can help understand any lower-level imbalances arising from skewed input or the BSP algorithm. Note that its imbalance reporting may not be correlated to the previous global slowdown metrics, hence may need to be interpreted according to what it signifies as discussed here. In other words, this metric helps us find the cause of imbalance.

4.8 Conclusion

In this chapter I proposed the novel problem of understanding how slowed down a parallel application is due to the impact of external load. I have developed novel methods and algorithms to answer this seemingly un-obvious question and evaluated them under different scenarios. I introduce the notion of The Ball In the Court delay and then design three algorithms based on the concept of computing the imbalance in the application once the Ball in the Court delays are computed for each process. These algorithms are:

1. Global Imbalance Algorithm
2. Process-level Bias Aware BIC imbalance algorithm
3. Multi-load BIC imbalance algorithm

These algorithms can greatly empower an automated adaptation system. One of the biggest decisions in such a system is that of deciding the proper placement of different processes or VMs and among the available physical hosts. Often we need to migrate processes or VMs to improve performance. However it is incredibly useful to know how useful the migration might be before actually executing the costly mechanism. The algorithms in this chapter allow the system to do just that. These algorithms can be completely automated as well and only need access to the traffic traces for the application, which is a completely black box operation.

Chapter 5

Finding Global Bottlenecks via Time Decomposition

In this chapter, we discuss the problem of finding global bottlenecks in a BSP application using black box methods, and tie in results from previous chapters and new techniques to describe a holistic method for converging to the reasons for performance slowdown in a BSP application. In a BSP application, there is high correlation amongst different processes in their activity and communications, creating inter-dependencies. For example, in a BSP application with a ring topology, if a single process in that ring is blocked because of saturated CPU, all other processes may be slowed down as well because of the cascading effect - the blocked process will result in a slowdown in communication to its neighboring processes and thus propagate the slowdown to all the processes. An entire application may be blocked due to an isolated issue that slows down a single process and thus creates slowdown in all processes in the dependency chain.

5.1 Some Questions to Answer

Some questions that should be answered to solve the problem of finding the global bottleneck for an application are as follows:

1. What are the different reasons why the application or a part of it can be blocked?

This question is addressed in the next section 5.2.

2. How can we detect if an application or a part of it is blocked because of any these reasons? We have addressed this question in Chapter 4 for host-based bottlenecks. We further explore it in this chapter, encompassing bottlenecks arising out of the network.

3. How can we infer which process(es) of the application is/are blocked? Chapter 4 describing the Ball in the Court Principle, which not only detects if the application is unbalanced in terms of performance but also is able to identify which process is the culprit behind the imbalance.

4. How can we isolate these reasons for blockage? In this chapter, we describe how once we converge to a possible location for the bottleneck (which could be a host or a part of the network), we can use performance measurement tools available in Unix/Linux to further converge to the underlying reason behind the slowdown.

5. Can we detect how *badly* the application is blocked? What speedup could be achieved if the blockage is removed? This question can greatly assist in adaptation and migration decisions. A quantitative measure of slowdown can be helpful in deciding which application to help if multiple applications or decisions can be made. Chapter 4 explored this question in detail and outlined various methods to give a quantitative estimate of how badly an application is affected by external load.

5.2 Possible Bottleneck Causes

As mentioned previously, an isolated single cause can cause the slowdown/blockage of the whole application. In order to discover these causes, we must have an idea of the possible origins and the actual nature of these causes. The causes that may cause the slowdown of the application are:

CPU saturation Another process or guest VM is using the CPU and saturating this resource along with our VM under analysis (VMA), thus slowing down the VMA. The CPU saturation may also be caused by CPU usage in the hypervisor, or dom0 in Xen, for example.

Disk Saturation A disk-intensive guest VM is interfering with the Io operations of the guest VM. Whether this interferes only with disk operations of VMA or other operations too is to be investigated.

Network contention Another guest VM is performing heavy network Io, thus affecting network Io of the VMA. This can affect both the blocking time, number of network operations and the bandwidth achievable by the VMA, thus affecting BSP applications that are more communication intensive.

Network link congestion Apart from local network affects, the network itself may have congestion that affects efficient communication amongst BSP application processes. The congestion may result in dropped packets, higher latency and reduced bandwidth for the application.

5.3 Different Models of Detecting Bottlenecks

In this section I give a high level overview of techniques that can be used to detect bottlenecks. Each approach has its benefits and issues. In my work I use one of these approaches only. However its useful to see it in the context of other possible approaches to get perspective.

5.3.1 Reactive Model

In this model, we actively monitor the performance statistics of a BSP application such as its CPU utilization, network utilization, disk utilization, execution rate per second (Chapter 3 provides these tools) and then look for any drastic changes in these metrics that might indicate a slowdown in the execution of the application. For example, if the network traffic coming out

of a host on which a particular VM is mapped falls drastically, it may indicate a performance issue on that host. It could either be a CPU or network contention issue. The change acts as a trigger to take action and figure out the reason for the slowdown - is it due to a sudden change in the internal behavior of the application or is it due to some external cause?

Pros:

- This model can give a potentially more accurate understanding of dependencies amongst a set of processes compared to *equilibrium state detection*.
- This model can be used to ask the question: "Tell me about big changes that happen for an application", "when they occur" and "the reason for the big change"

Cons:

- The model requires constant monitoring of an application, which means higher monitoring overhead.
- The model may not accurately reveal the main cause for blocking or slowdown. For example, if a certain process blocks, it will change the iteration rate of all processes at the same time and it may be difficult to isolate the original process that led to the change.

5.3.2 Equilibrium State Detection Model

This model does not assume active monitoring for triggers. The goal here is to detect any potential bottlenecks and then causes for performance slowdown in a currently running application just by passively observing its various signals. We may need to correlate signals and traces from different VMs to recognize any anomalies and use them to draw conclusions about any performance bottlenecks in a BSP application. This could be somewhat more challenging to achieve than the reactive model since there are no events that are monitored or taken advantage of to give a hint of where to look for a performance bottleneck.

Pros:

- This model can introduce a probe at any point in the system to detect any problems with execution of the current application, without doing monitoring all the time
- Using passive measurements of various signals, it may be able to figure out the actual cause behind the blocking/slowdown of the application.

Cons:

- The model may not detect any issues in execution of the application depends on the sophistication of the inference techniques. For example a sudden change in traffic for one of the VMs may indicate an increase in communication which can be caught by a reactive model. But after the event has taken place, the equilibrium model may or may not be able to detect that the application is running slower/faster after the fact, depending on the techniques themselves. Reactive approaches can indicate a change in application behavior more easily and quickly whereas in equilibrium approach it really depends on how sophisticated the inference techniques are.

In my work I have focused on the equilibrium based model because of the advantages mentioned above. Chapters 2, 3 and 4 are good examples of such an approach. All of these chapters involve passive monitoring and extracting interesting patterns from the traffic traces, giving quantitative measures of different useful quantities like application performance or slowdown caused by external load. In this chapter, I further follow this approach towards solving the global bottleneck problem.

5.4 Time Decomposition of Application Execution

One of the first steps towards identifying performance bottlenecks in an application is to get a clear picture of where the application is spending its time. For a BSP application, the total time spent is decomposed into local and network parts. If we can break down the execution time into

its various components at the process level and resource level granularity it can give us a clear picture of where the application bottleneck might be.

A BSP application's time is either spent in local computation, I/O or in network communication. We took the first step towards identifying this time decomposition in Chapter 4 by introducing the notion of BIC (Ball In The Court) delay. BIC delay is a black box measure that quantifies time spent processing locally for each inter-process interaction. We showed how this knowledge could be used to identify potential imbalance in the application, the extent of slowdown, and a quantitative estimate of the potential speedup if the imbalance is removed.

Apart from the host being a bottleneck, the only other resource which may cause a slowdown in the application's execution is the network. In this section we discuss how can we give a finer picture of the use of this resource by the BSP application based on the black box observations of the traffic being emitted from the VM. Our overall goal is to find the *time decomposition* of a BSP application to identify where it is spending most of its time and if any components are anomalous. This decomposition is a very powerful black box tool to probe into the performance characteristics of a BSP application.

What are the various metrics that we can derive for the network? Some important metrics for the network time decomposition that we output from the black box analysis are:

1. **Total number of messages sent:** This is the estimated number of messages sent by each VM hosting one of the parallel application's processes. This can give an idea of which processes are most message intensive in case not all processes are symmetrical. A message-intensive process could be better placed in order to further improve the performance of its intensive communication behavior.

2. **Average latency observed per message:** This metric gives an estimate of the one way delay observed for each pair of processes. In our implementation, we output different latencies observed for each pair if they differ drastically during the trace probing period. This metric can help identify any links that may be have unusually high latency compared

to other links

3. **Cumulative message latency:** This metric is a function of the link latency and the total number of messages sent. In some sense it is a more useful metric than average latency because it gives a sense of how much time the application is actually spending in communication for each pair of processes in the latency part. If this time dominates the overall execution time, then this part may be worth optimizing as a priority. If the cumulative latency is big, it implies lot of small messages which means the focus should be on getting VMs close together.

4. **Approximate rate of message transfer (proxy for bandwidth):** This metric serves as a proxy to indicate the bandwidth or message transfer rates of various links. It's difficult to get the actual bandwidth per message but an approximate proxy serves as a good indicator any congestion issues on the link that can indicate non-latency related problems with the network.

5. **Cumulative message transfer time:** This metric is an approximate measure of the total time spent transferring messages in the non-latency part of the communication. This is somewhat more useful than the previous rate of message transfer metric as it also takes into account the frequency of messages sent for each pair, thus giving a more complete picture of where extra time maybe going in the bandwidth component of the time. If the message transfer time is large for a particular link, the performance of the application could be improved by moving the VM pair to a faster link.

Note that each of the metrics is reported for each inter-process interaction. Thus for an application with n processes, we report $\binom{n}{2}$ instances of each metric. This can give us a very fine grain picture of where the application is spending its time and of any imbalance at the process and resource level, helping us converge to the real cause behind the slowdown.

5.4.1 Going from the Packet Level to the Message Level

An important part for deriving the metrics mentioned above is to consider the communication between the various processes at the level of messages, rather than packets. At the raw traffic capture level, we see TCP/IP packets. However, certain assumptions about these BSP applications can help us aggregate and partition these packets into individual messages. For example, for every message send, there is a message receive from the partner process. This fact can be used to partition the packet boundaries appropriately.

While processing the traffic trace for each virtual machine, we detect message boundaries whenever a series of successive send packets from one process P_1 to another process P_2 is broken by a receive event (i.e. a packet is sent from P_2 to P_1). Once we know the boundaries of individual messages, we are in a better position to estimate the metrics enumerated in the previous section. We now discuss in more detail how each of these metrics are estimated.

5.5 Methodology for Estimating Each Metric

In this section we describe the methodology used to estimate the various metrics from the traffic traces.

5.5.1 Number of Messages

For each pair of communicating processes we store the number of messages sent from the sender process to the receiver process. This is done by counting the boundaries between the send packets and receive packets for the same process pair. The number of messages can be helpful in indicating the frequency of interaction between different pairs of processes and thus guiding the network optimization part of adaptation. Apart from the latency of the link between a particular pair of processes and other link characteristics, the frequency of interaction is also quite important. Even if the latency is higher for a particular pair compared to the rest but

message frequency is low, it may not make sense to improve that particular communication link.

5.5.2 Estimating Cumulative Message Latency

For every message transfer, there are two components to the time required for the complete transfer: the *latency part* and the *bandwidth part*. These components are due to different characteristics of the link and thus measuring them separately gives a better idea of where the application is spending its time in the non-BIC or the communication component.

An important metric that is computed is the total time spent in the latency part of transferring messages for each process pair separately. This gives a great idea of the application is really imbalanced in terms of transferring messages over a high latency link. To compute this metric, we need to know the one way delay for each message sent. Computing the one-way delay for a particular pair of processes in general is not trivial. There are two general ways of going about it:

1. Measuring the time difference between message send and message arrival at both ends, requiring tight time synchronization.

2. Estimating one way delay from the round trip delay as reported by tools like ping.

The Time Difference Method

This method requires looking at the traffic traces at both ends for a particular process pair and then computing the time difference between the packets sent at the sender side and the time when the corresponding packets are received at the receiver side. This approach is generally hard because of 2 reasons:

1. Synchronization of packets across traces, and

2. Synchronization of time across different machines.

The first one is a easier problem to solve. For every packet sent at the sender side, we can locate the received packet with the same sequence number or the earliest succeeding sequence number. This requires looking up the traffic traces from both the sides in conjunction. Thus for each process's traffic trace that we examine, we go through each send packet and then locate the corresponding receive packet in the trace of the partner process. Since the sequence numbers increase monotonically for both the send and the receive packets on both sides, we can examine each send packet and find its corresponding receive packet in the partner's traffic trace with only one pass on both sides. This makes the process of traversing the trace $O(n)$ where n is the number of send packets examined at one side. The number of send packets examined in turn are directly correlated to the total number of messages sent, since we compute the one way delay of the message transfer only at the message boundary.

The second issue is much harder to solve. Achieving tight time synchronization amongst different machines can be done by various methods. The most popular of these methods is using the popular time synchronization tool NTP (network time protocol).

Using NTP to synchronize time:

Network Time Protocol (NTP) is a popular protocol designed to synchronize the clocks of computers over a network [147]. NTP 4 is a significant revision of the NTP standard, and is the current development version, but has not been formalised in an RFC. Simple NTP (SNTP) version 4 is described in RFC 2030. It provides accuracies of typically less than a millisecond on LANs and up to a few milliseconds on WANs.

As mentioned earlier, I tried to configure NTP for my LAN scenario in order to be able to calculate accurate time differences for packet departure and arrivals across different machines on the LAN. One of the main issues with NTP is getting the value of time offset (that provides a measure of how accurate the synchronization is) as low as possible. Mills [107] and some web based tutorials [5, 6] provide excellent documentation on configuring NTP properly and getting the offset down to as low as possible.

One of the primary difficulties I had with NTP were related to getting the offset down to levels where the time difference calculations actually make sense. For example, in my LAN setup, the one way delay as estimated by ping was around 150-200 microseconds. However the offset which I managed to get using NTP, though sub-millisecond, was not low enough to allow an accurate enough estimation of the one-way delay using the time difference method. Even if the synchronization error is in the range of few hundred microseconds, it is still quite large relative to the actual one way delay in a LAN situation, thus making it less useful as a measure of the one way delay for my purposes of computing the total application time spent in the network component of execution. Owing to this difficulty, I abandoned NTP for my purposes and focused on the approximate but still more reliable method of round trip based estimation of one way delay.

The round trip method

Another way to approximate one-way delay is to measure the round trip delay using a tool like *ping* and then halve it. Even if the one-way-delay is high only in one direction, the round trip time will reflect that. Using round trip time will not give the exact direction in which the delay is high but will point out the pair of processes that seem to be suffering from high latency in communicating. Then the link can be probed further to pin-point the cause for high round trip latency.

To actually store the round trip method, a background process called *rttseries* is run while the parallel application is running, on each of the physical hosts. *rttseries* is also provided with a list of other physical hosts participating in the execution of the parallel application. The tool then pings each of the hosts in the background periodically, with the interval specified when its run. It pings each host 4 times and then computes the average from the last 2 readings, and then appends the average to a file. Only the last 2 readings are used to compute the average because it was observed that occasionally the first one or two ping values were anomalous and quite high compared to the long term average. Thus on each physical machine, *rttseries* outputs four

fields: timestamp, source IP, destination IP and the average RTT computed from last two pings for the particular pair.

Note that the round trip time is measured across physical machines, not the virtual machines. This corresponds to the actual network characteristics. Measuring delay between virtual machines would actually also combine any packet scheduling effects that might be in effect between domU and dom0. That is actually accounted in the BIC delays.

The aggregated RTT delays file is then used to estimate one-way-delays during analysis for particular pair of processes at the specified time using the timestamp field.

Computing average latency using a sliding window

The methods used to estimate one way delay for a particular pair of processes are described above. I currently use the RTT based method because of its simplicity and due to the difficult to resolve complications in the NTP method.

For the observed period of traffic trace analysis (that may be a fraction of the whole runtime of the application, with the requirement that it captures the stationary average of the signal as described previously), the TCP trace for each process is processed and for each separate message, its latency is calculated for the pair of processes at the sending and the receiving end of the message. This latency can be calculated by doing a *join* of the timestamps in the traffic trace and ping output as generated by the *rttseries* script that outputs the approximate one-way-delay for each partner process along with the timestamp of measurement. For each message we compute the latency observed during that time. This latency is then also accumulated per message to count the time spent in the latency component in sending the message.

Note that this requires just one pass over the trace and for each message we need to look up the latency for the message timestamp. Since the timestamp is monotonically increasing, the ping output file also needs to be traversed only once and thus the overall time complexity is $O(n)$ where n is the number of messages in the trace.

Computation of average latency:

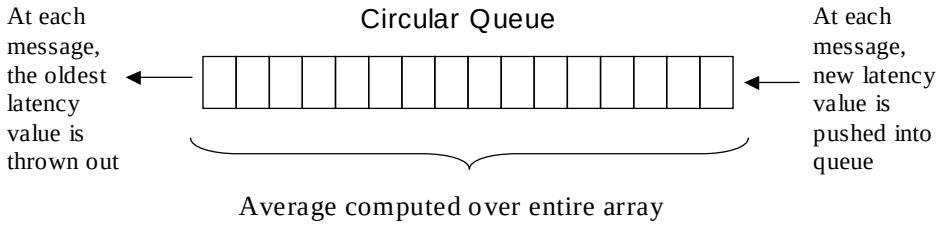


Figure 5.1: The computation of average latency uses a circular queue to compute the average of last *b* latency values

The overall goal is to output any significant changes observed in the latency for any pair of processes, during the measurement time period. This can give a good idea of the variability of the latency for that particular link also, or how stable the link is in terms of its latency.

To achieve this the latency computed for each message is also inserted into a circular queue of a bounded size *b*, to effectively compute the average of only the last *b* latency values. Each time a new latency is inserted into the queue, the average latency is computed by computing the average of the circular queue(Figure 5.1). If this average latency is significantly different from a previous output average latency ($previous - average$) by a certain threshold $avg - latency - threshold$, then this new average is output and also assigned to $previous - average$.

Note that the above is done for the traffic trace originating from each VM, and for each partner VM that the VM talks to. Thus we construct a matric for each pair of VMs, with the above metrics as elements. We output two metrics: total time spent in the latency portion of the message communication and then a series of measured average latencies which differ significantly from each other to give an idea of the latency variation for that particular link.

5.5.3 Estimating Cumulative Message Transfer Time

The bandwidth component of communication actually has two parts: time spent transferring the messages per process pair once the first packet corresponding to a message reaches the destination (refer to Figure 5.2) and the approximate measured bandwidth per message.

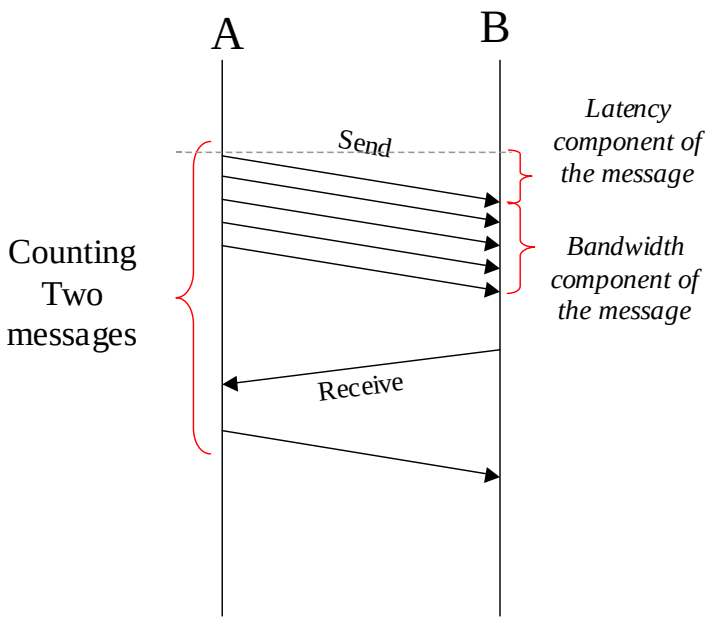


Figure 5.2: The different time components when a message consisting of multiple packets is transferred

Message Transfer Time: For measuring the approximate cumulative message transfer time, we look at the received messages of each VM's trace measured at the physical host's interface. For each received message, we compute the time difference between the last packet and the first packet. We denote this time as $message - time_{bw}$. We start accumulating $message - time_{bw}$ for each received message per sender process and thus record the total time spent transferring messages for each sender-receiver pair.

This time estimate gives us an idea of how much time was spent in receiving the message after the message initially reached the receiver i.e. the latency component was accounted for. After the first packet reaches the destination, the rest of the packets are essentially pipelined and we measure the time it takes to receive rest of the packets. If there is any congestion on the link, this time will increase regardless of the latency.

It is important to note, however, that this time is only an approximate measure of the actual message transfer time. This is because tcpdump timestamps are not completely accurate [2].

The timestamp is usually applied whenever the packet is supplied by the device driver or the networking stack to the code that is responsible for the timestamp. This is always after the last bit of the packet is received. Thus the time difference captured by the timestamps will miss the transfer time of the first packet itself and also will be an approximate measure of the time when the last bit is received in general for all packets. However it serves as a good proxy and in the relative sense serves as a good measure for comparing transfer times.

Message bandwidth proxy: As in the latency case, we also measure and output a series of average bandwidth values if they differ significantly from each other. However, as noted above, the tcpdump timestamp values are not exactly indicative of the time it takes to transfer the entire message. Especially on the receiving side, the timestamp is recorded after the last bit of packet is received, thus making the time difference actually less than what it actually is for a particular set of packets, thus inflating the bandwidth. Thus the bandwidth that we get from these timestamp computations are only proxies for the actual bandwidth and may sometimes exceed the maximum bandwidth possible on the link. However, they do permit us to *compare* the bandwidth rate of different links.

To compute the average bandwidth, we maintain a circular queue as before and measure the number of bytes transferred per message divided by the time it took for the bandwidth component of the message. We insert this bandwidth reported for that particular message in the queue. The average computed from the previous $n_{avg-bandwidth}$ readings is then output if it differs significantly from the previous average that was output. This gives an idea of the stability of the bandwidth of the link and helps identify variations amongst different links. If a link is highly fluctuating for example, it may help move one of the VMs so that the problem is alleviated.

5.6 Implementation

Currently the described above analysis has been implemented in a single script. This single script is responsible for computing the BIC as well as the non-BIC parts of the delay. It takes three input files. The first file called the *trace input* points to the tcpdump traces for all the different VMs along with their IPs. The second file points to the files containing the output of the rttseries utility for each different physical host for the corresponding VMs running the parallel processes. The third file contains a mapping from the VM IP addresses to the actual physical host IP addresses. The reason this is needed is because the output from the rttseries tool contains IP addresses for the physical hosts whereas the traces capture the IP addresses for the VMs. To translate amongst these, we need a map.

The script has two stages. First it calculates the BIC delay for each of the VMs from the traffic traces. After that it calculates the non-BIC part of the application's execution time. In the non-BIC part it processes the trace for each VM and calculates the metrics described above. At the end, the script outputs a combined summary of BIC and non-BIC delay in an ordered fashion so as to assist in quickly locating imbalances. It outputs the following summary:

1. Global BIC delays in descending order.
2. Pairwise BIC delays in descending order
3. Cumulative Message Latency in descending order
4. Average Latencies observed in descending order
5. Cumulative Message Transfer time (approximate) in descending order
6. Average Bandwidth observed (proxy) in descending order

5.6.1 Evaluation with Different Scenarios

In this section, we evaluate some of the NAS benchmark applications under several scenarios and discuss the results given by the *time decomposition* tool and how they compare across different scenarios.

The purpose of this section is to also show the mapping between different causes of slowdown (which we control in the evaluation) and the various metrics reported by the report. A knowledge of this mapping can help translate various imbalances reported by the report to the actual causes of slowdown. Dealing with different scenarios serve as case studies and give great insight into this mapping.

In each of the following scenarios we report the metrics in a table which show the main BIC and non-BIC metrics like Global BIC, Pairwise BIC delays, Pairwise Latency component times (the highest ones), Average Latency observed, and the bandwidth component times (the highest ones).

No load case

In this scenario, we run the MG NAS benchmark on four Xen VMs, running on separate physical hosts. There is no external load imposed of any kind. The total run time for the entire application is 13.84 seconds.The various BIC and non-BIC delays for this scenario are shown in Table 5.1. We see that the BIC delays are quite close to each other in absolute terms. We also see that the time spent in the latency component and the bandwidth component of the non-BIC delays is quite low, being around 0.2 seconds and around 2 seconds respectively.

One process behind high latency link (WAN)

Table 5.2 shows the numbers for the emulated scenario in which one of the VMs (corresponding to the physical host 176) is behind a high latency 3ms link (tenfold the normal latency for the link). The typical LAN inter-host latency is 300us. The latency is emulated using the Linux

Process	Global BIC	Pair	Pair BIC	Pair	Latency Time	Pair	Avg Lat	Pair	Bw time
173	9.43	173-172	8.67	175-176	0.191	175-176	1.44	176-175	1.93
172	8.15	172-176	7.58	172-176	0.135	175-176	1.00	173-172	1.74
175	5.99	175-173	5.30	173-172	0.103	172-176	0.763	175-176	1.56
176	2.44	176-175	1.75	173-175	0.050	175-176	0.690	172-176	1.51
		175-176	0.69	172-173	0.040	173-172	0.525	175-173	1.50
Exec time	**13.84s**								

Table 5.1: The various time decomposition metrics shown for the normal case where there is no external load on locally or on the network.

Process	Global BIC	Pair	Pair BIC	Pair	Latency Time	Pair	Avg Lat	Pair	Bw time	Pair	Avg Proxy Bw
176	8.25	176-175	7.28	176-175	**4.52**	173-176	3.215	176-175	**3.94**	175-176	330MB
175	7.71	175-173	6.85	175-176	**2.896**	176-175	3.18	175-176	3.79	175-173	145MB
173	7.62	173-172	6.37	172-176	0.400	175-176	2.99	172-176	2.40	175-176	143MB
172	3.34	172-176	1.68	176-172	0.397	172-176	2.67	173-172	2.23	176-172	0.574MB
		172-173	1.66	173-175	0.072	176-172	2.08	176-172	2.02	175-176	0.509MB
Exec time	**16.228s**										

Table 5.2: The various time decomposition metrics shown for the case where one of the hosts(176) is emulated to be behind a high latency 3ms link (compared to the local LAN latency of $\tilde{3}$00us). The latency times are observed to be slightly higher and so is the execution time, compared to the normal case (by approximately 2.5 seconds)

kernel's built in iproute2 traffic control framework [4]. In this scenario, we see that the execution time increases to 16.2 seconds from 13.8 seconds in the no stress case. We also observe an interesting increase in the time spent in the latency component of the non-BIC part, where we see that for certain pairs (176-175 and 175-176), the time spent is 4.52 and 2.9 seconds respectively, much higher than the time observed in the latency component of the no stress case. We see that overall the BIC delays remain similar, but see an increase in the non-BIC delay portion of the time spent. The time spent in the bandwidth component also increases as seen in the table. The highest bandwidth component in this case is 3.94 seconds compared to 1.93 seconds in the no stress case.

Working backwards, one could have easily noticed the imbalance in the latency component of various pairs and investigated the link between 176 and 175 to discover that 176 lies behind a high latency link.

Process	Global BIC	Pair	Pair BIC	Pair	Latency Time	Pair	Avg Lat	Pair	Bw time	Avg Proxy Bw
176	309.8	176-175	160.44	173-175	0.031s	173-175	0.19	176-175	**178.2**	0.07, 2, 6.06, 27.5MB
172	22.24	176-172	149.34	175-173	0.020s	173-175	0.149	176-172	**173.2**	0.068, 14.01, 30.35MB
173	18.58	172-176	21.19	176-175	0.014s	172-175	0.132	175-173	18.8	8.28, 18.3, 41.3MB
175	15.74	173-172	11.11	175-176	0.011s	173-176	0.124	173-175	10.03	18.14MB
		175-173	11.01	173-172	0.011s	173-172	0.123	175-176	2.49	25.41, 11.33MB
Exec time	**325.38s**									

Table 5.3: The various time decomposition metrics shown for the case where one of the hosts(176) is emulated to be behind a congested link using token bucket filtering with a maximum burst bandwidth of 5Mbps.

One process behind a congested link

Table 5.3 shows a more extreme scenario where one of the VMs (corresponding to 176) is emulated to be behind a low bandwidth link where the maximum burst bandwidth is restricted to 5Mbps using the traffic control framework of the Linux kernel. Here the MG application is slowed down drastically to 325.38 seconds compared to 13.84 seconds in the no load case. This drastic slowdown is because of the communication intensive nature of the application. With the bandwidth restricted, the app's short bursts of traffic get highly lengthened and high iteration phases where there are short spurts of communication interleaved with short phases of computation get slowed down drastically.

In the table we see that latency component of the non-BIC delay does not change much compared to the no load case. However the bandwidth component of the non-BIC delay is very high, with it being 178.2 and 173.2 seconds for the top 2 pairs, and highly imbalanced compared to the rest. This immediately points to a problem in the bandwidth of the links corresponding to 176-175 and 176-172, which is the actual problem here,

Also we see that global BIC delay is also very high for the host 176. This is due to the emulated nature of the slow bandwidth link. Because the bandwidth delay happens on the kernel side instead of the actual link, it also gets recorded as part of the BIC delay. This would disappear where the kernel has no role to play towards a congested link.

Process	Global BIC	Pair	Pair BIC	Pair	Latency Time	Avg Lat	Pair	Bw time	Avg Proxy Bw
176	**328.02**	176-175	164.55	172-176	0.014s	0.136ms	176-172	**191.3**	0.06, 2.19, 110MB
173	18.4	176-172	163.42	175-176	0.012s	0.118ms	176-175	**184.37**	0.06, 1.47, 25MB
172	16.79	172-176	15.96	176-172	0.012s	0.113ms	172-175	78.85	2.54, 9.58, 23.57MB
175	15.83	175-173	11.25	176-175	0.011s	0.105ms	175-173	26.86	2.55, 9.58, 23.56MB
		173-172	11.17	173-172	0.011s	0.117ms	173-175	4.75	1.47, 7.72, 26.44MB
Exec time	**346.67s**								

Table 5.4: The various time decomposition metrics shown for the case where one of the hosts(176) is emulated to be behind a congested link using token bucket filtering with a maximum burst bandwidth of 5Mbps, and its physical host is partially loaded as well (CPU load of 60% in the isolated case).

Combining high local load with a congested link

Table 5.4 shows a more stressed case than the previous one, were an additional computational load (CPU load of 60%) is also imposed on the same physical host. In this case the MG application is slowed down to 346 seconds compared to 325 seconds in the previous case, an additional slowdown of 21 seconds. Can we notice this extra stress factor in the time decomposition table? The Global BIC delay in this case is 328 seconds vs 309s in the previous case: a difference of 19 seconds. The large bandwidth times clearly tell us that the bandwidth part for host 176 is clearly imbalanced so it definitely warrants inspection. However without looking at the CPU utilization numbers (which should be probed for 176 because of its high BIC delay), it will be difficult to infer about the partial computational load.

From this example we see that the time decomposition can help isolate the bottleneck to the host or the link level. However some local diagnosis is necessary once the actual location is identified, the ascertain the actual cause(s) behind the bottleneck.

One process on a host also running a busy HTTP server

In this experiment we run the Integer Sort NAS benchmark in one of the guest VMs and the Apache web server in a sibling guest VM on the same physical host. The purpose of this scenario is to consider a realistic situation where a parallel application may be affected a non-parallel load on the same physical host. This also serves as a scenario where workloads of

different kinds are on the shared resource.

Table 5.5, 5.6 and 5.7 show the BIC and non-BIC stats for three different cases:

- The IS application is running without any load.
- The IS application is running with a busy HTTP server, with the clients on the same physical host.
- The IS application is running with a busy HTTP server, with the clients on a different physical host.

We see that the total runtime for the three cases is respectively 12.64s, 152.78s and 137.88s. The highest time is for the second case (152.78s) where the clients are on the same physical host as the Apache web server, which is running on host 176. Bombarding the web server with a large number of concurrent requests (50 clients in this case) generates lot of network contention as well as results in greatly increased CPU utilization. Each client makes a request for a static html page of size approximately 2 KBytes. Some of the statistics for loading the Apache server for the two cases are (clients on the same host (176) and a different host (175)):

- Total requests: 250000 (both cases)
- Total bytes transferred: 107,500,000 bytes (both cases)
- Requests per second: 941.04 and 1081.57 respectively [/sec] (mean)
- Transfer rate: 395.16 and 212.2 [Kbytes/sec] received

We note that the BIC time for host 176 are very high for both cases (144.3s and 129.8s respectively), indicating high local load. If we also note the bandwidth time in the last column, its very high for hosts involving 176, indicating its high network contention. Its higher for the first case when the clients are on the same physical host. Both indicate that having clients on the same physical host is bad for IS. It also shows that for a realistic non-parallel load, if we

Process	Global BIC	Pair	Pair BIC	Pair	Latency Time	Avg Lat	Pair	Bw time
175	5.8	172-176	3.97	176-173	4.35s	3.2ms,2.2ms,1.07ms	173-176	6.9s
172	5.5	175-173	3.21	175-172	2.96s	1.445ms,1.00ms	175-176	5.2s
173	4.8	173-176	2.22	173-176	1.72s	0.37ms	176-173	4.7s
176	4.5	173-172	1.82	172-175	0.56s	0.12ms	176-175	4.6s
		176-175	1.78	175-173	0.043s	2.15ms	172-176	4.6s
Exec time	12.64s							

Table 5.5: This table shows the various BIC and non-BIC delay components for the case when the IS application is running without any external load, for the purpose of comparison with other cases.

Process	Global BIC	Pair	Pair BIC	Pair	Latency Time	Avg Lat	Pair	Bw time
176	**144.3s**	176-172	50.5s	172-175	1.4s	0.65s,0.59s, 0.46s	175-176	**105.11s**
172	10.7s	176-175	47s	175-172	0.9s	1.445ms,1.00ms	172-176	**76.93s**
175	10.4s	176-173	46.8s	176-173	0.24s	0.37ms	173-176	**68.51s**
173	8.7s	172-176	8.5s	173-176	0.18s	0.12ms	172-173	38.16s
		175-176	5.8s	172-176	0.011s	2.15ms	175-172	34s
Exec time	**152.78s**							

Table 5.6: This table shows the various BIC and non-BIC delay components for the case when the IS application is running with an adjacent overloaded web server, with the clients also simulated on the same physical machine.

compute the imbalance for IS (for the case where clients are on same physical host), we get a slowdown of approximately 135 seconds (using the Global BIC balance algorithm described in Section 4.3.1). The actual slowdown delay for IS is 152.78 - 12.64 = 140.14 seconds, which is very close.

We see that using the above diagnosis, we could have converged to the cause of the problem (high load on 176 + network contention for 176) and also the amount of speedup we could have achieved if we removed the bottleneck.

5.7 Converging on the Cause of a Bottleneck

Most of this chapter has talked about converging to a bottleneck at the network or the local host level. High BIC delays point to a bottleneck at the host level whereas high non-BIC delays point to issues in the network. Moreover in the non-BIC case we can further get insight based on the latecy and the bandwidth numbers that could point to high delay, congestion issues or both.

Process	Global BIC	Pair	Pair BIC	Pair	Latency Time	Avg Lat	Pair	Bw time
176	**129.8s**	176-172	48.4s	172-175	0.53s	0.65s,0.59s, 0.46s	175-176	**94s**
172	11.6s	176-173	43.2s	175-172	0.49s	1.445ms,1.00ms	172-176	**68.5s**
175	10.7s	176-175	38.1s	176-173	0.17s	0.37ms	173-176	**64.8s**
173	10.5s	172-176	9.3s	173-176	0.14s	0.12ms	175-172	62.2s
		173-176	6.3s	175-176	0.005s	2.15ms	172-173	33.7s
Exec time	**137.88s**							

Table 5.7: This table shows the various BIC and non-BIC delay components for the case when the IS application is running with an adjacent overloaded web server, with the client being on a different physical machine (which is also running another process of the same parallel application corresponding to 175)

However, at the local level, the cause behind high BIC delay may not be immediately apparent. In this section we briefly talk about how the adaptation mechanism, the application developer or the user could further drill down to the cause of high BIC delays. High BIC delays could be caused by numerous factors such as high CPU load, disk contention, network contention on the host side and memory thrashing due to lack of physical memory to accommodate all guest machines (virtual memory performance).

A lot of effort has already happened in the UNIX community towards the development of tools that allow accurate measurement of various resources on the host side. Instead of re-inventing the wheel, I will just point to some of them and some work in the field. These tools can then be leveraged and even used in an automated setting to deduce the lower level cause of high BIC delays for example.

Some useful references for converging onto the cause of the bottleneck include Lynch [99] and Fink [44]. I briefly describe some widely available Unix/Linux tools that help isolate particular bottlenecks on the host side:

uptime: This provides system load average (SLA). It also counts as runnable all jobs waiting for disk I/O, including NFS I/O. However, uptime is a good place to start when trying to determine whether a bottleneck is CPU or I/O based.

sar: *Sar* is a very powerful command. It allows measuring the average queue length, swapping

and paging activity, time waiting for blocked I/O etc.

iostat: *iostat* is a low overhead tool that provides numerous measures of I/O activity. Iostat provides data on several important values for each physical disk. These include: percentage of the time the physical disk was busy, kilobytes per second to/from the disk, transfers per second to/from, kilobytes read and kilobytes written. This will help to determine if there is an imbalance of I/O amongst the physical volumes. *If most of the I/O is directed at the page files then memory needs to be investigated.*

vmstat: If from *iostat* we determine that the bulk of the I/O is going to paging (e.g. the page activity time is $>$ 30%) then it becomes necessary to investigate further. lsps (or pstat), vmstat and svmon are useful commands to investigate paging. *vmstat* provides information on actual memory, free pages, processes on the I/O waitq, reclaims, pageins, pageouts etc. Apart from virtual memory, it also reports kernel statistics about processes, disk, swap, and general IO statistics like disk blocks sent to disk devices in blocks per second. Fink [44] gives a good description of drilling down to specific areas using *vmstat*.

netstat: *netstat* -i shows the network interfaces along with input and output packets and errors. It also gives the number of collisions. The *collis* field shows the number of collisions since boot and can be as high as 10%. If it is greater, then it is necessary to reorganize the network as the network is overloaded on that segment. *netstat* -v is used to look at queues and other information. If max packets on the S/W transmit queue is $>$0 and is equal to current hardware transmit queue length then the send queue size should be increased. If the “no mbuf errors” is large then the receive queue size needs to be increased.

netpmon: *netpmon* gives an an overall sense of network performance. By using a combination of the above commands, it is possible to obtain a very clear view of what is happening at the network level.

Apart from the very useful and freely available Unix tools above, there are also third-party performance measurement and analysis tools that can provide even more detail and help isolate the cause of bottlenecks once a bottleneck has been identified. For example, SARCheck [7] can detect CPU Bottlenecks, runaway processes, I/O bottlenecks, improper I/O load balancing, slow disk devices, memory bottlenecks and leaks, inefficient system buffer cache sizing, improper system table sizes, inefficient PATH variables, and other problems with the way Linux and UNIX kernel parameters are set.

In summary, I think using the black box techniques described in the previous chapters and this chapter can help one identify the host or the network component where to focus one's attention on, in an automated manner. In case the issue is with a host, one can then use the specific Unix measurement tools mentioned here to identify the cause behind the bottleneck. Figure 5.3 shows the high level process of starting from performance measurement of an application and then applying the techniques mentioned in the previous chapters to figure out if the application is blocked, reason for blockage, its current performance, amount of speedup possible etc, which can be very useful in a multi application scenario.

5.8 Conclusion

In this chapter, we examined the high level problem of finding the global bottleneck in a parallel application, regardless of whether it originates from the local host or the network. We pose several important questions for the purpose of solving the global bottleneck problem. The answers to these questions are actually covered in this and the previous chapter 4.

In this chapter specifically, we looked at the problem of time decomposition, i.e. figuring out where the application spends its time, at the resource and the process level. Knowing this can immediately reveal potential sources of performance problems and can then be further probed into for revealing the exact cause of the bottleneck. Some important components of this time decomposition are:

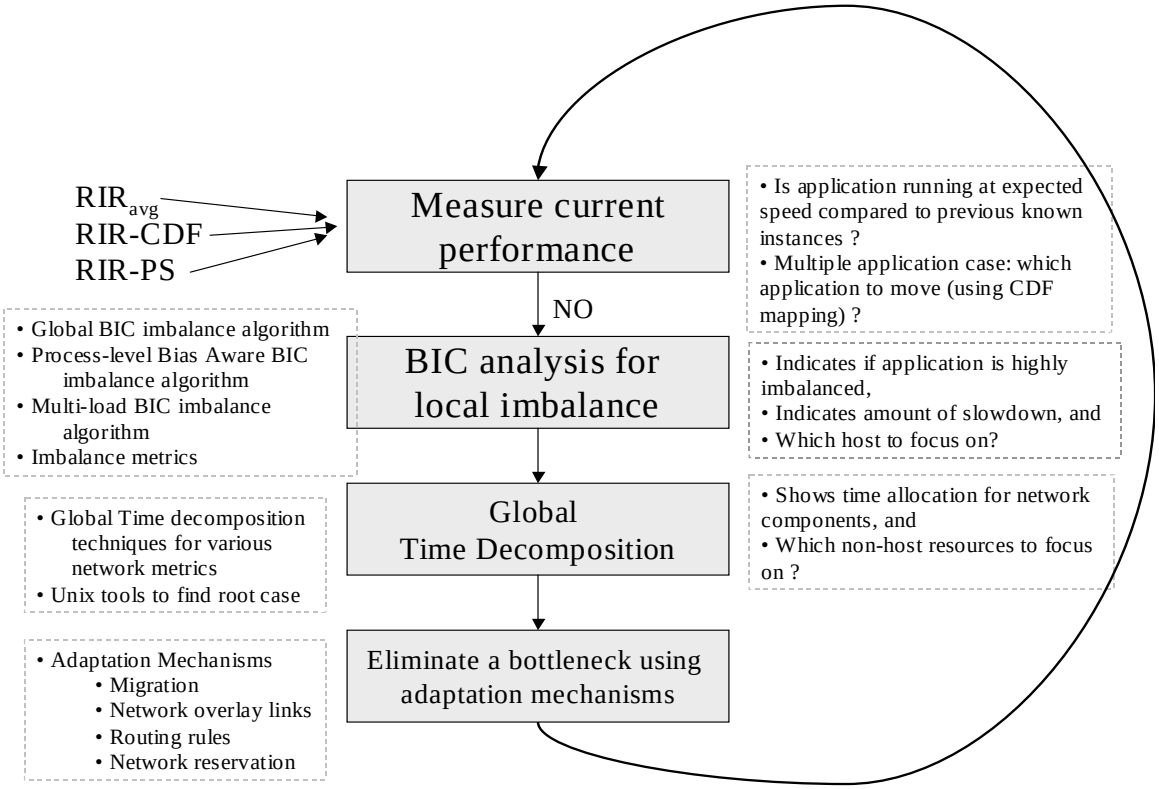


Figure 5.3: A high level overview of the steps that can be followed to converge to the cause of a performance bottleneck and then alleviate it.

1. Global BIC delays
2. Pairwise BIC delays
3. Cumulative Message Latency
4. Average Latencies observed
5. Cumulative Message Transfer time (approximate)
6. Average Bandwidth observed (proxy)

Once we can figure out which aspect of the application is contributing to performance delays, we can then use the various tools outlined in Section 5.7 to actually identify the nature of the bottleneck, whether its a CPU overload issue, I/O or network one.

Ultimately, I have demonstrated that using completely black box means, it is possible to understand how a parallel application is utilizing various resources and even quantify the impact on the application performance. This can greatly help functions like performance adaptation and application scheduling as it now allows them to peek "inside" the application execution behavior and fine tune the placement of VMs to eliminate specific issues, predict performance impact before making an adaptation decision(Chapter 4), and measure the difference in performance after different adaptation decisions (Chapter 3). Apart from performance related metrics, I also developed techniques to infer the communication behavior and the topology of the application, that can then be useful to adapting the underlying overlay network to match its topology and result in improved performance (Chapter 2).

Chapter 6

Related Work

In this chapter, I describe work done by the systems community that is directly or indirectly related to the goals of my dissertation, i.e. black box inference for parallel applications, and also its applications like adaptation. I mainly divide the related work into three areas: virtual distributed computing, adaptation, and inference related research.

6.1 Virtual Distributed Computing

The Stanford Collective is seeking to create a compute utility in which "virtual appliances" (VMs with task-specialized operating systems and applications that are intended to be easy to maintain) can be run in a trusted environment [3, 30, 47, 120]. They also support the creation of "virtual appliance networks" (VANs), which tie a group of virtual appliances to an Ethernet VLAN. The Virtuoso project is similar in that it also, in effect, ties a group of VMs together in an overlay network that behaves, from the VM perspective, as a LAN. However Virtuoso differs in the nature of the applications that it supports (parallel and distributed scientific applications). There is no focus on inference in this project especially for distributed applications since its current target is readily deployable end-user machines that work stand alone.

The Xenoserver Project [20, 67], has somewhat similar goals to Virtuoso, but they are not focused on networking and require that OS kernels be ported to their system. However their

virtualization technology could be enhanced with the inference techniques mentioned in this dissertation to aid black box inference of parallel applications.

Purdue's SODA project aims to build a service-on-demand grid infrastructure based on virtual server technology [77] and virtual networking [78]. Similar to VANs in the Collective, the SODA virtual network, VIOLIN, allows for the dynamic setup of an arbitrary private layer 2 and layer 3 virtual network among virtual servers. Both these projects can support parallel and distributed applications and support re-configuration and adaptation of applications to enhance their service. These efforts can easily utilize the various inference techniques to understand the nature of the applications and to aid the re-configuration efforts. The VioCluster project [119] proposes a virtualization based computational resource sharing platform. It allows trading of machines amongst different cluster domains to address the conflict between dynamic workload and static capacity. They have a notion of calculating demand so that *machine brokers* can allocate appropriate cluster resources to the application at runtime. When applications start sharing resources, having a black box way of measuring their performance, understanding their potential to speedup etc can greatly aid in allocating resources more intelligently to these applications.

6.2 Adaptation

Adaptation in distributed systems has a long history. The most influential work has been on load balancing of parallel applications [34, 127, 150] and load sharing in real-time [22, 66, 86] and non-realtime systems [41, 42, 68, 90] . The mechanism of control in these systems is either function and process migration [28, 108, 131], which is lighter-weight although considerably more complex than VM migration, or application-specific mechanisms such as redistributing parallel arrays.

In contrast to these approaches, some of my work focuses on adaptation using virtual machines as the lowest common denominator of migration(Appendices A and B). This results

in less complexity in terms of implementation owing to much easier migration mechanisms already in place for virtual machines as compared to function or process migration. It also allows significant other advantages for environment/operating system encapsulation and easier resource management. However, these adaptation mechanisms mentioned above can benefit nonetheless from the inference work in this dissertation when they cater to parallel applications. Every adaptation system can use black box measures of performance and way to deduce bottlenecks in a parallel application, and then take appropriate action to alleviate the problem.

For media applications in mobile computing, modulating quality has been a basic approach both commercially and in the research environment [111]. This approach requires application modification as applications must decide how to react to changing resource availability as informed by the underlying operating system. This is a different philosophy compared to my work in adaptation. There have also been efforts to factor adaptation methods and policy out of applications and into distributed object frameworks like CORBA [159]. In this project, authors aim to expose QoS functionality available in the communication substrate to distributed applications at the middleware level, so that these applications can easily adapt to scarce or variable resource availability when operating over the wide area. However, this does require the application to be aware of this new functionality and thus needs to be modified to take advantage of it. This is in contrast to my work on adaptation. It is possible that if the adaptation functionality is also delegated to the middleware, then the middleware can use the inference techniques developed in this dissertation to automatically help the application adapt to different resources in a manner most appropriate for the application.

6.3 Inference Aspects

Inference related research in the past has focused on different reasons to learn about the application or the operating system. Primary reasons have been:

1. to learn the properties of an operating system or its processes and to even control its

behavior [16, 79],

2. to classify applications in different resource demand based categories [157],

3. to dynamically adapt applications according to changing workload and application behavior [114, 153],

4. for future static configuration of applications after one-time inference of applications [37, 96], and for

5. Inference for distributed systems [14, 116].

However not all work has focused on black box inference. In fact the only work we are aware of currently is Wood et al. [153]. In this work, they focus on how black box and gray box strategies can be used to improve performance of a set of stand-alone applications running inside Xen VMs by dynamically migrating the VMs. In black-box monitoring, no knowledge of OS or application is assumed and only externally visible parameters like CPU load, disk swapping activity and network usage are used. This work does very elementary black box inference for stand alone applications. My work in contrast focuses on distributed parallel applications that are more complex collective entities composed of multiple processes, and allow much more sophisticated inference techniques to be devised, especially owing to their inter process communication behavior. Thus the main focus of my dissertation has been collective inference, that treats the application as a whole instead of looking at each process separately. My inference techniques could then be used to further extend their migration techniques to the domain of parallel applications.

6.3.1 Category I - To Learn the Properties of an Operating System or its Processes and to Even Control its Behavior

The very influential work by Arpaci-Dusseau at al [16] shows how, by leveraging some information about how the OS functions and then monitoring its various signals and correlating this

information, one can learn a lot about OS behavior, for example creating file cache activity detector or a file layout detector. They further show that it's even possible to control OS behavior by doing careful proxy activity on behalf of the application. For example by pre-fetching certain portions of the file based on past trends, one can reduce cache misses for an application. They also give an enumeration of useful inference levers that can be used to extract information. Examples include knowledge of internal algorithms used by the OS or the module, monitoring its output and signals, inserting probes, using pre-generated micro benchmark information for the system and correlating it with probes to extract useful behavior information about the OS. They also show how it's very useful to extract certain statistical properties like correlation, etc. In Chapters 2, 4 and 5, I use this insight to provide meaningful statistical properties especially in the context of a distributed application and even show how this information can be used to make interesting inferences about the system. Also, I leverage the knowledge of the structure of BSP applications to design some of my algorithms (like the BIC algorithms in Chapter 4), just as these authors leverage some of the knowledge of how the OS functions to design some of their techniques.

The Geiger project by Jones et al [79] shows how a VMM (Xen in this case) can be modified to yield useful information about a guest operating system's unified buffer cache and virtual memory system. They then leverage this capability to implement a novel working set size estimator, which allows the VMM to make more informed memory allocation decisions. They conclude that after adding such passive inference capabilities to VMMs, a whole new class VMM-level functionality can be enabled that is agnostic of the OS or the applications running inside. This philosophy can be carried over to my work as well, to allow certain black box inference primitives to be embedded with the VM itself for better performance and coupling, which can then be accessed via an API by the middleware to make adaptation decisions for the application.

6.3.2 Category II - to Classify Applications in Different Resource Demand Based Categories

The work by Zhang et al [157] shows how one can use dimensionality reduction and pattern recognition techniques over a certain chosen set of metrics to classify applications into broad categories like CPU, I/O or Memory intensive. The main focus of the work is how to simplify decisions about application scheduling and costs when faced with a multitude of metrics to observe. They also demonstrate a application resource consumption composite, that can be used to derive the cost of executing the application. They do not focus on black box monitoring but use the Ganglia Grid monitoring framework [104] to monitor certain readily available metrics like CPU load, network traffic etc across the distributed system and then aggregate it. Beyond that I also provide much more sophisticated black box metrics that can he very useful for application classification: RIR-CDF and Power spectrum based fingerprints (Chapter 3). Some of my metrics can be integrated into their pattern recognition framework for a more sophisticated application classification framework.

6.3.3 Category III - to Dynamically Adapt Applications According to Changing Workload and Application Behavior

The work discussed above [153] is an example of this. Work by Ranjan et al [114] does CPU load inference and uses it to adapt services by migrating them to appropriate servers. A particularly interesting point in this work is the impact of statistical parameters of the workload like peak/mean load ratio and auto-correlation of the load time series on the adaptive algorithm's effectiveness. They show that higher peak/mean ratio applications can benefit more from their adaptive algorithm and that poor auto-correlation can make migration decisions harder. They also provides a useful categorization of autonomic utility computing work: "Each offers a notion of utility computing where resources can be acquired and released when/where they are needed. Such architectures can be classified as employing shared server utility or full server

utility models. With the shared server utility model, many services share a server at the same time, whereas with the full server utility model, each server offers one service at a time." Their work only applies to full server utility model. In contrast, my work in adaptation and inference assumes the shared utility model. This model requires a much more detailed understanding of the application, their resource sharing dynamics and more sophisticated adaptation algorithms. My inference and adaptation work addresses these issues in many different ways.

6.3.4 Category IV - for Future Static Configuration of Applications After One-time Inference of Applications

The ECO project by Lowekamp at al [96] contains programs that analyze the network for a parallel application to understand their collective communication behavior and then establishing efficient communication patterns, which the work claims is more powerful than simply treating the network as collections of point-to-point connections. Work by Dinda et al [37] is a study on network traffic patterns exhibited by compiler-parallelized applications and it shows that the traffic is significantly different from conventional media traffic. For example, unlike media traffic, there is no intrinsic periodicity due to a frame rate. Instead, application parameters and the network itself determine the periodicity. In my work in Chapter 2, I further extend this work to look at the spatial properties of the communication amongst different processes like inferring the communication topology of the application, which can then be used to adapt the underlying overlay networks to suit the demands of the application.

6.3.5 Category V - Distributed Inference

In the work done by Aguilera et al [14], the inference focus is on distributed systems with a black box approach. The goal is to assist in detecting the points/components that contribute to high latency amongst critical/frequent message paths. This can aid in debugging such distributed systems where its not obvious where the delay is coming from. Their focus is on offline analysis of message traces for asynchronous distributed systems. They present two approaches: a RPC

nesting based approach and other is a correlation-based approach that is not dependent on the RPC protocol. This work is closest to mine. Mine differs in the following ways: It is not clear how these techniques translate to parallel applications especially since the communication is cyclic and there is no starting or ending point. The delay per node may not be constant but may actually depend on the message size. This will make the auto-correlation approach a failure as no clear correlation may be found for a long duration trace. I offer an alternative way to explore different delays exhibited by processes in the same parallel application. In Chapter 4, I develop Ball in the court principles that can give a quantitative estimate of the different local processing delays of various processes and help identify bottlenecks at the process level and network level.

In a follow-up paper [116], some aspects for inference for wide area distributed systems are discussed. The focus again is to find the "delay" culprits or processing/communication hotspots by creating a message flow graph. Wide area introduces extra challenges not covered in their earlier paper [14], like significant network latency (almost ignored in their first paper), node unreliability, higher degree parallel programming (more overlapping cause-effect message relationships) etc. They currently do not provide any modeling for barriers etc, common mechanisms in parallel/scientific applications. Their work seems to be more geared towards DHT like distributed systems. Some of the challenges they deal are:

1. Path aggregation - how to group similar path instances into one path pattern even if the instances span different nodes because of load balancing or other reasons.

2. Dealing with High parallelism - this blurs the causal relationships amongst incoming and outgoing messages at a node.

In my work, I address the problem of communication related hot-spots in Chapter 5 and provide black box methods to break down communication amongst different processes into its various constituents to better understand where the application execution time is going. I do not look at path aggregation techniques for BSP applications or causal relationships as these models

are not needed for BSP applications. Instead, since there is high dependency and correlation amongst different processes, identifying the time sinks at various resource levels is sufficient to indicate where the bottleneck for the entire application is.

Chapter 7

Conclusion

The goal of Virtuoso project has been to develop techniques for an effective virtual machine-based autonomic distributed computing framework, which users can use with the ease of using networked resources in a local area network , without worrying about the networking, scheduling and performance aspects of it. Among other things, Virtuoso targets parallel and distributed applications - applications that span multiple VMs. Virtualization removes many operational hurdles, which users face today. In this context, automated inference and performance adaptation for such applications is important for Virtuoso and systems like it to be an attractive target for users deploying their distributed applications in shared/wide area resources.

My dissertation has focused on how to achieve automated inference of various demands and behaviors of parallel applications in a *black box fashion*. This has formed the major part of the work. This dissertation has also pointed to previous work done by me in collaboration with my colleagues, that gives evidence of how some of the inferred properties can be gainfully utilized to adapt the distributed application to improve its performance. In summary I have proposed, implemented and evaluated various techniques and algorithms that automatically understand an application's needs, performance properties and bottlenecks without any external input from the user or application. All of these have been designed to work under black box assumptions, which assumes no inside knowledge of the application or even the operating system environment that might be encapsulated inside a VM.

In this chapter I summarize the contributions of the various chapters in this dissertation and also provide a high level API that gives a concrete interface to the various properties of the parallel applications that I expose via the techniques described in this dissertation.

7.1 Contributions of the Dissertation

In this section I briefly describe the overall contributions of the dissertation according to the different chapters. As mentioned above, this dissertation has focused on providing the developer, the user, the resource manager and the resource management middleware with useful parallel application properties, demands, and predictions that do not require knowledge of the particular application at hand or the operating environment. My work tries to infer as much information as possible from the externally observed signals like the TCP traffic patterns. Based on this principle I have been able to propose several novel methods for black-box inference.

In Chapter 2 I focused on inferring the traffic matrix of the parallel application and then deducing the spatial communication topology. This topology can then be used to adapt the underlying overlay virtual network to suit the application's needs. I showed that with very low overhead (around 4% on the VNET throughput for VTTIF monitoring), one can construct the topology of the application in real time and then use this as input to improve applications's performance with run-time adaptation. I have also shown the adaptation benefit as shown by my collaborative work in Appendices A and B. Adapting to topology of the application led to performance increases up to 95% in our experiments for some application scenarios.

Another aspect of topology inference is handling of dynamic situations, where the communication matrix may change with time. Such oscillations can cause unnecessary overhead for adaptation. I proposed a simple low-pass filter based solution that provide a stable and aggregate view of the topology. I discussed several implementation issues like the effect of update rate and the smoothing interval. Another important part of topology inference is when to alert the adaptation agent if a significant change in the topology has occurred. I discussed the metrics

and methods for making this change decision and provide a reactive-mechanism for automated alerting.

In Chapter 3 I proposed and provided the solution for a problem of great practical impact: How can we evaluate the performance of a BSP application under black box conditions? Performance is of critical importance to the developer, users as well as resource providers. *Adaptation mechanisms themselves usually have the goal of improving performance of the application and the system as a whole. However this can be achieved only if there is way to evaluate performance of each application and compare it across different adaptation options. This chapter solves that problem.* I have provided a set of new metrics that quantitatively represent the performance in multitude of ways. The most simplest of all is the RIR metric (Round-trip Iteration Rate) which closely correlates to the iteration rate of the application (number of super-steps executed per second). This metric is based on the intuition that inter-process interaction is a good indicator of progress in a BSP application and it only needs to look at the outgoing traffic of the processes.

Of course, for complex dynamic applications, this iteration rate keeps on changing. I suggested many derivative metrics that take into account this dynamicity. I suggested several novel metrics like long term RIR average (RIR_{avg}), the RIR cumulative distribution function (RIR-CDF), power spectrum of the RIR time series (RIR-PS), dominant power spectrum frequencies, etc. Section 3.6 provides a summary of all the metrics that I proposed. While the average RIR gives a single number to compare the overall performance of an application across multiple instances, more complex metrics like the CDF give a deeper view of the dynamic nature of the application. This can be important because depending on how dynamic the application is, it can be affected differently by external load. This can help in making more complex and informed scheduling decisions. For example, in section 3.9.2, I provided a simple proof of concept about how the CDF can be used to decide which application might be impacted more by external load based on a *slowdown mapping* of the CDF.

The power spectrum-based metrics also help uncover more useful information about the application, like the duration of its super-step and the various dominant periodicities present in the application. This can also help in fingerprinting the application and help classify applications over the long run. It can characterize and partition similar processes in a big data and task parallel application as described in 3.10.5 which can then be useful to schedule the similar processes as a group since they are more likely to be tightly coupled. It can also help in statistical scheduling where applications with different frequency characteristics may be mixed and matched according to different scheduling algorithms. For example, statistically, it may be beneficial to mix applications whose power spectrums have the least overlap so as to allow for more spread out use of resources. These techniques based on the new black box metrics I have derived, are a future area of research. Some of these concepts have not been tested in my dissertation but provide a fertile ground for further research and exploration.

In Chapter 4, I discussed and provide a solution to another very intriguing and useful problem: Can we understand how stressed out a particular application might be because of external load factors, in precise quantitative terms? Answering this question means we can actually predict how much the application will speed-up, if we are to move the application VM to a host with lower load or if we just move the load away from that host. *This can be very important in making adaptation decisions, where the flexibility and resource availability to migrate applications around may be limited. Knowing in advance the benefit of migration can greatly assist in this process.*

I suggested a novel approach to analyze the traffic trace of a BSP application and look for specific paired-events that can be accumulated to provide a metric which I call the *Ball in the Court* delay. Intuitively this metric gives a quantitative estimate of how much time the process is spending locally on the host, which may include computation or waiting for local I/O. When we compare this metric across different instances of the processes for the same application, it can give us a quantitative sense of the *imbalance* between the local processing times of these

processes. These imbalances for a symmetric BSP application are caused by local factors. By counting this imbalance, one can get an idea of how the whole application is getting slowed down because of a single process. In my evaluation, this method has been very accurate in estimating the slowdown for the application.

I also suggested more sophisticated algorithms for complex situations that take into account potential bias caused in processes that are not stressed from processes that are stressed. I identified this bias from my empirical observations of BSP applications and then suggest a more involved algorithm that takes the bias into account and gives the overall imbalance in the application. I call it the *Process-level BIC Imbalance algorithm*. This algorithm can provide a range of slowdown with upper and lower bounds. I further extended the algorithm to deal with multi-load situations as well. In the scenario where multiple processes might be stressed differently, it can iteratively apply the balancing technique to balance out the whole application and figure out the speed up gained in the balancing process.

Overall the above set of algorithms provide a remarkable way of predicting the benefits of migration and external load elimination for a parallel application and has great application in automated performance adaptation. I also suggested some metrics to quantify imbalance in a parallel application itself. The imbalance suggests that even though if some of the processes may be well-scheduled, the application is suffering as a whole because some of the processes are slowed down by external load. *This imbalance metric can serve as a warning sign to draw attention to applications that can benefit from adaptation as a priority, especially when multiple applications may be valid candidates.*

In Chapter 5, I extended the goal of Chapter 4 to find bottlenecks that may be affecting the application at a global level, i.e, going beyond the individual host. I introduced the problem of *Global Time Decomposition*, where my goal is to give an accurate picture of where the application is spending time at the process and the resource level, including the network. Having this level of understanding gives a detailed glimpse into where the bottlenecks exist and which part

of the application (including the process, the physical host, the network link) should be probed to improve the performance of the application. I defined many new metrics that isolate the network characteristics of the application at the inter-process level. Computing and comparing these metrics can help understand the network time spent at the inter-process communication level and indicate high latency or congestion. I defined composite metrics that take into account the intensity of communication as well, so that only links that are heavily used as well as slow or congested are worth investigating.

Combined with the Ball in the Court delay metrics from Chapter 4, work in this chapter gives a unified overview and will help the adaptation agent or the user identify the source of the bottleneck. I also described how various Unix performance measurement tools can be further used to isolate the real cause behind local delays for example. These causes may be CPU or IO related. Then depending on the exact cause, one can take appropriate steps to eliminate it. All this is achieved using black box assumptions.

7.2 Benefitting from Inference: Adaptation

Some of the inference work from this dissertation (from Chapter 2) has been greatly utilized in some of my work in the area of automated runtime performance adaptation, that I have done in the past with my colleagues. This work is described in Appendices A and B. The work described in Appendix A (Increasing Application Performance In Virtual Environments through Run-time Inference and Adaptation) shows how to use the continually inferred application topology and traffic to dynamically control three mechanisms of adaptation, VM migration, overlay topology, and forwarding to significantly increase the performance of two classes of applications, bulk synchronous parallel applications and transactional web ecommerce applications.

Work in Appendix B (Free Network Measurement For Adaptive Virtualized Distributed Computing) demonstrates the feasibility of free automatic network measurement by fusing the Wren passive monitoring [155] and analysis system with Virtuoso's virtual networking sys-

tem. We explain how Wren has been extended to support online analysis, and we explain how Virtuoso's adaptation algorithms have been enhanced to use Wren's physical network level information to choose VM-to-host mappings, overlay topology, and forwarding rules.

My work has been published in reputable conferences [58, 59, 61, 135, 139]. They serve as great evidence of the utility of black box inference and how properties derived from such inference can guide the application of runtime adaptation for improved performance of applications without any manual intervention or specific knowledge of the application.

7.3 Outline of an API for Inference

To give a concrete interface to the set of contributions in this dissertation, I describe a unified API that provides a consistent and complete interface to the useful properties that can be further used for other applications like adaptation or resource management/accounting.

For each API member, we mention the description, the input parameters and the output parameters.

7.3.1 Topology Inference (Chapter 2)

GetTrafficMatrix: **Description**: Gives the traffic matrix for the last n seconds for a particular BSP application.

Input: n $\Rightarrow$ the number of seconds for which to return the traffic matrix.

Output: Returns a matrix of size k x k, where k is the number of processes in the application. Element (i,j) denotes the traffic from process i to process j in the last n seconds.

GetSmoothedTopology: **Description**: Returns the current stable topology for the application where the traffic below a certain relative threshold is pruned out.

Input: None. It just returns the current topology according to the smoothing interval set separately

Output: Returns a matrix of size k x k, where k is the number of processes in the application. However the traffic is smoothed out with the smoothing parameter that is set separately.

SetSmoothingParameters: Description: Sets the period of smoothing and the size of the sliding window over which to smooth the traffic to get the resultant topology.

Input: T $\Rightarrow$ Period of smoothing

s $\Rightarrow$ Size of sliding Window. See section 2.5 for details of these parameters

Output: None.

7.3.2 Black Box Performance Measurement (Chapter 3)

GetRIRRate Description: Get the Round Trip-based Iteration rate (RIR) as measured using TCP traffic averaged over the last n seconds. This can be used to get an instantaneous measure of the application performance.

Input: n $\Rightarrow$ The last n seconds over which the RIR is averaged on.

Output: rir $\Rightarrow$ The RIR averaged over last n seconds.

GetStationaryRIRAvg Description: Gets the long term stationary average for the Round Trip-based iteration rate which captures most of the power spectrum of the signal. This metric can be used to compare application performance across instances.

Input: None. The cutoff factor c% is taken into account that is set separately.

Output: RIR_{avg} $\Rightarrow$ The long term RIR for the application. The time period for the average is calculated internally using Power Spectrum truncation to find the right t $\Rightarrow$ The time period over which the long term average is computed.

GetRIRCDF Description: Gets the Cumulative Distribution Function for the RIR over the long term time period which which captures most of the RIR signal.

Input: None

Output: Vector $[a_1, a_2, ..., a_{100}]$ that captures the CDF of the RIR time series over the time period that covers the dynamic range of the application.

GetRepresentativeTimePeriod Description: Returns the time period over which the signal behavior RIR rate for the application is mostly captured by the power spectrum. This ideally captures 1 or more super phases of the application. Note that this time period may not correspond exactly to the duration of the super phase of the application. It just captures the range of dynamic behavior of the application.

Input: None. This time period if influenced by the cutoff factor c% that is set separately.

Output: $T_{stationary} \Rightarrow$ the time period inferred from the RIR time series over which the signal is found to be stationary according the Power Spectrum truncation method.

GetPowerSpectrum Description: Returns the Power Spectrum of the RIR iteration rate over the last n seconds.

Input: n $\Rightarrow$ the last n seconds for which to return the Power Spectrum

Output: Vector of x,y pairs that indicate the power spectrum graph for the RIR time series.

GetPowerSpectrumAvg Description: Returns the Power Spectrum of the RIR iteration rate over the Representative time Period

Input: None

Output: Vector of x,y pairs that indicate the power spectrum graph for the RIR time series over $T_{stationary}$

GetPowerSpectrumPeaks Description: Returns the local maxima peaks estimated from the Avg Power Spectrum giving an idea of the various periodicities and phases of the BSP application.

Input: None

Output: Vector of x,y pairs where x_i indicates a dominant frequency in the PS and the corresponding y_i indicates the magnitude.

GetSuperPhaseEstimate Description: Returns an estimate of the duration of the super-phase of the BSP application by estimating the lowest dominant frequency in the Average Power Spectrum

Input: None

Output: $t_{super-phase} \Rightarrow$ The estimated time period of the super phase of the BSP application.

SetPowerSpectrumCutoff Description: This API call is used to decide the sensitivity of the Power Spectrum truncation mechanism that is used to decide how much of the time series to capture to cover the dynamic RIR range of the application. Higher cutoff values mean very high sensitivity and may result in longer periods of measurement.

Input: $c\% \Rightarrow$ The cutoff factor. See section 3.7.1 for details of how this is used.

Output: None

7.3.3 Ball in the Court Metrics (Chapter 4)

GetGlobalBICDelay Description: Gets the global BIC delay for the process i.

Input: $i \Rightarrow$ The process identifier of the application for which to get the BIC delay

n $\Rightarrow$ *Thelastnsecondsoverwhichtocomputetheglobal BICdelay.*

Output :BIC$_i \Rightarrow$ the BIC delay for process i.

GetPairBICDelay Description: Gets the BIC delay for the particular process pair i,j

Input: $i \Rightarrow$ The process identifier of the application for which to get the BIC delay

$j \Rightarrow$ The partner process for process i for which to count the BIC delay. n $\Rightarrow$ *Thelastnsecondsoverwhich*

Output :BIC$_{i,j} \Rightarrow$ the BIC delay for process i towards process j.

ComputeImbalanceGlobal Description: Computes the imbalance in the current application according to global BIC delays and hence the potential speedup for the last n seconds of measurement. $\frac{imbalance}{n}$ is the fraction of possible performance improvement possible after the imbalance is addressed.

Input: $n \Rightarrow$ The last n seconds over which to compute the global imbalance

Output: $t_{imbalance} \Rightarrow$ The global imbalance computed for the last n seconds.

ComputeImbalanceInterProcessBias Description: Computes the imbalance range according to the more refined inter-process load bias based algorithm for the last n seconds.

Input: $n \Rightarrow$ The last n seconds over which to compute the imbalance

Output: $[t_{optimstic}, t_{pessimistic}] \Rightarrow$ A range that indicates the possible slowdown for the application as described in Section 4.4.

ComputeImbalanceMultiload Description: Computes the possible range of imbalance for a multiple load situation. i.e. it considers BIC delays in decreasing order and computes imbalance for more than 1 global BIC delay. It stops considering more BIC delays if the percentage improvement in the imbalance estimate is less then the input t_p which is < 1.

Input: $n \Rightarrow$ The last n seconds over which to compute the imbalance

$t_p \Rightarrow$ The threshold that decides when to stop the iterations for the multi-iteration imbalance algorithm

Output: $[t_{optimstic}, t_{pessimistic}] \Rightarrow$ A range that indicates the possible slowdown for the application as described in Section 4.4.

ComputeApplicationImbalance Description: Computes the current application imbalance according to the metrics defined in Chapter 4. This can give an idea of how heterogenous the current performance and the environment of the application is.

Input: METRIC_TYPE $\Rightarrow$ Indicates which imbalance metric to choose from. Possible types are STANDARD_DEVIATION, SQUARED_DISTANCE, INTER-PROCESS_IMBALANCE.

Output: *imbalance* $\Rightarrow$ the computed imbalance according the method mentioned in the input.

7.3.4 Time Decomposition Related (Chapter 5)

GetNoMessages Description: Gets the total number of messages sent by Process P_i to Process P_j in the last n seconds.

Input: i, j, $n \Rightarrow$ as described above

Output: $m \Rightarrow$ total number of messages from P_i to P_j in last n seconds.

GetNoMessagesList Description: Gets a sorted(descending) list of the number of messages sent by the various interacting process pairs in the application in the last n seconds.

Input: $n \Rightarrow$ as above

Output: Sorted (descending) vector of tuples $[i, j, m]$ where the sorting key is m. Each tuple indicates that m messages where transferred from P_i to P_j in last n seconds.

GetLatencyTime Description: Gets the time spent in the latency component of sending messages across the process pair P_i to P_j in the last n seconds.

Input: i, j, $n \Rightarrow$ as described above

Output:$t_{latency} \Rightarrow$ time spent in the latency component for communication between P_i and P_j in last n seconds.

GetAvgLatency Description: Gets the average latency noticed over a sliding window of messages over the process pair P_i to process P_j. Note that this function returns a vector that contains the various differing avg. latencies observed in the last n seconds.

Input: i, j, $n \Rightarrow$ as described above

Output: Vector $l_1, ..., l_k]$ denoting all the significantly differing latencies observed during last n seconds.

GetLatencyTimeList Description: Returns a sorted(descending) list of times spent in the latency component of the message communication amongst all communicating pairs.

Input: $n \Rightarrow$ as above

Output: Sorted (descending) vector of tuples $[i, j, l]$ where the sorting key is l. Each tuple indicates that l seconds where spent in the latency component in the communication from P_i to P_j in last n seconds.

GetBWTime Description: Returns the bandwidth portion of the time spent transferring messages from process P_i to process P_j.

Input: i, j, $n \Rightarrow$ as described above

Output: $t_{bandwidth} \Rightarrow$ time spent in the bandwidth component for communication between P_i and P_j in last n seconds.

GetBWTimeList Description: Returns a sorted(ascending) list of the total time spent in the bandwidth portion of transferring messages by the different process pairs in the last n seconds.

Input: $n \Rightarrow$ as above

Output: Sorted (descending) vector of tuples $[i, j, b]$ where the sorting key is b. Each tuple indicates that b seconds where spent in the bandwidth component in the communication from P_i to P_j in last n seconds.

GetAvgBW Description: Returns a vector of various different average Bandwidth rates observed computed over sliding windows for the last n seconds for the input process pair P_i and P_j.

Input: i, j, $n \Rightarrow$ as described above

Output: Vector $l_1, ..., l_k]$ denoting all the significantly differing latencies observed during last n seconds.

GetMostImbalancedComponent : Out of the various places where the application is spend-

ing its time, it returns the component where the application seems to be most imbalanced compared to use of the same component amongst other processes. Based on the type and location of the component, it can lead to the actual cause of the slowdown.

Input: $n \Rightarrow$ last n seconds in which to look for imbalance.

Output: RType, RIdentifier, Imbalance $\Rightarrow$ Returns the resource and the amount of imbalance found for that resource. The resource RType can be of the type PROCESS, NETWORK LATENCY, NETWORK BANDWIDTH. The Imbalance is a number indicating the number of seconds for which the application seems imbalanced. The resource identifier RIdentifier is a tuple indicating the process number(s) for which the resource is tied to. For example a link can be identified with $[i, j]$ indicating that the link between Process i and Process j is imbalanced.

7.4 What's Ahead?

In my dissertation, I have described numerous techniques to infer useful properties for BSP applications using a black box approach. Of course this is not the end in black box inference. This dissertation shows that rich possibilities exist in this area and how even without specific knowledge of the application in the parallel application domain, one can glean useful properties about them that can then in turn be used to benefit the applications or the resource providers. Some other possibilities that may exist for further exploration into this area are:

- **Dependency Graphs:** For distributed applications, one possibility that exists is to figure out the chain of causality amongst different processes. For example in a tree topology, the causality of message propagation may be from the leaves to the roots, with every node aggregating messages from its children and forwarding them to its parent. Inferring and exposing such causality can give clues towards processes that may be delaying the whole application because of delayed forwarding.

- **Inferring Power Requirements:** A very intriguing and increasingly important issue in distributed applications especially those running in one or adjacent clusters, is of their power consumption. This is important because of the important of balancing power requirements across a set of physical hosts to balance heat distribution. It could also result in more effective use of power.

  The correlation between variable resource needs and power consumption is already established [60]. Can we exploit this correlation information with the resource inference mentioned above to also infer the power requirements of a distributed application? For example, the input might be:

  1. Correlation between the resource consumption and power consumption for various hosts
  2. Application to host mapping
  3. If available, physical proximity or co-ordinates of hosts.

  and the output would be:

  1. Power consumption of the distributed application.
  2. Power distribution amongst the hosts

  Another question that can be investigated is: How will the power consumption of an application change if processes are migrated or reconfigured amongst the physical hosts in a specific way? This can help us understand the tradeoff between performance and power consumption. Sometimes it may be preferable to trade less performance for lower power consumption and distribution for power and heating related reasons.

- **Reliability Inference:** An upcoming and interesting problem faced for massively parallel applications running on a multiple node cluster is that of its reliability needs. For a

system consisting of many independent parts, e.g. a thousand-node cluster, the mean time between failures can be very low. Moreover, this reliability can even depend on the power usage of the hardware. A direction for exploration is whether it is possible to infer the reliability needs of a distributed application and recommend any necessary action? This issue is still very new and needs a more precise study to understand the real issues and answers sought.

- **Additional VMM Assistance:** Some more questions that can be explored are: Can we greatly enhance black box inference if we can get some extra information from the VMM beyond what we can get currently ? What is the ideal data that we can get to infer useful properties about the application, while still remaining application and/or OS agnostic? What if we could embed the inference techniques developed in this dissertation into the VMM?

- **Ensuring Conflict Free Communication for Multiple Parallel Applications:** Can we sync applications so that there is no overlapping communication ?

- **Temporal Topology Inference:** If the combined topology is very complex and may pose difficulty in reservation in optical networks, can we break it up into multiple simpler topologies, based on temporal differences? For example a more complex topology may actually be composed of two simpler topologies alternating after one another in succession.

Apart from some of the above aspects for application inference, there is the other side: How can we leverage this information to help improve its performance or schedule applications better ? For example, some possible directions are:

1. Incorporate the techniques described in this dissertation in the middleware, and implement the API, so that the middleware exposes the functionality for other applications.

2. Modify adaptation algorithms to take advantage of the new inferred information. For example, the new absolute performance metrics present in Chapter 3 can be now used by adaptation algorithms to record and present the effectiveness of adaptation algorithms in a black box fashion. These records can also be used to guide the adaptation algorithms themselves. Similarly, the BIC algorithms and Global Bottleneck techniques described in Chapter 4 and 5 can result in vastly improved adaptation algorithms that can identify distressed portions of an application and migrate/re-allocate resources accordingly to eliminate the bottleneck.

3. Develop new black box statistical scheduling algorithms that take advantage of new application fingerprint data like its power spectrum or CDF. There are numerous ways in which these statistics could be combined and could result in better co-scheduling of applications together on shared resources.

The above discussion have given some idea about the rich possibilities that exist in the area of application inference and their applications.

Bibliography

[1] http://members.optushome.com.au/walshjj/single_slit_transform_pairs.gif.

[2] http://www.tcpdump.org/lists/workers/2003/08/msg00411.html.

[3] Moka5 livepc technology. www.moka5.com/.

[4] Net:iproute2. http://www.linux-foundation.org/en/Net:Iproute2.

[5] Network time protocol tutorial. http://networking.ringofsaturn.com/Protocols/ntp.php.

[6] Ntp howto. http://gentoo-wiki.com/HOWTO_NTP.

[7] Sarcheck. http://www.sarcheck.com/.

[8] tcpdump. http://www.tcpdump.org/.

[9] Vmware. http://en.wikipedia.org/wiki/VMware.

[10] Xen. http://en.wikipedia.org/wiki/Xen.

[11] Xentop. http://www.linuxcommand.org/man_pages/xentop1.html.

[12] Design, implementation, and performance of an extensible toolkit for resource prediction in distributed systems. *IEEE Trans. Parallel Distrib. Syst. 17*, 2 (2006), 160–173. Member-Peter A. Dinda.

[13] ADABALA, S., CHADHA, V., CHAWLA, P., FIGUEIREDO, R., FORTES, J., KRSUL, I., MATSUNAGA, A., TSUGAWA, M., ZHANG, J., AND ET AL. ABSTRACT — FULL TEXT + LINKS — ANONYMOUS, M. Z. From virtualized resources to virtual computing grids: the In-VIGO system. *Future Generation Computer Systems 21*, 6 (June 2005), 896–909.

[14] AGUILERA, M. K., MOGUL, J. C., WIENER, J. L., REYNOLDS, P., AND MUTHITACHAROEN, A. Performance debugging for distributed systems of black boxes. In *Proceedings of the nineteenth ACM symposium on Operating systems principles* (New

York, Oct. 19–22 2003), vol. 37, 5 of *Operating Systems Review*, ACM Press, pp. 74–89.

[15] ARABE, J., BEGUELIN, A., LOWEKAMP, B., E. SELIGMAN, M. S., AND STEPHAN, P. Dome: Parallel programming in a heterogeneous multi-user environment. Tech. Rep. CMU-CS-95-137, Carnegie Mellon University, School of Computer Science, April 1995.

[16] ARPACI-DUSSEAU, A., AND ARPACI-DUSSEAU, R. Information and control in Gray-Box systems. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP-01)* (New York, Oct. 21–24 2001), G. Ganger, Ed., vol. 35, 5 of *ACM SIGOPS Operating Systems Review*, ACM Press, pp. 43–56.

[17] BAILEY, D. NAS parallel benchmarks. Tech. Rep. 94-007, NAS Applied Research Branch (RNR), 94.

[18] BAILEY, D. H., BARSZCZ, E., BARTON, J. T., BROWNING, D. S., CARTER, R. L., DAGUM, D., FATOOHI, R. A., FREDERICKSON, P. O., LASINSKI, T. A., SCHREIBER, R. S., SIMON, H. D., VENKATAKRISHNAN, V., AND WEERATUNGA, S. K. The nas parallel benchmarks. *The International Journal of Supercomputer Applications 5*, 3 (Fall 1991), 63–73.

[19] BANERJEE, S., LEE, S., BHATTACHARJEE, B., AND SRINIVASAN, A. Resilient multicast using overlays. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (June 2003).

[20] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *ACM Symposium on Operating Systems Principles (SOSP)* (2003), pp. 164–177.

[21] BEGNUM, K., AND BURGESS, M. Principle components and importance ranking of distributed anomalies. *Machine Learning Journal 58* (2005), 217–230.

[22] BESTAVROS, A. Load profiling: A methodology for scheduling real-time tasks in a distributed system. In *17th International Conference on Distributed Computing Systems (17th ICDCS'97)* (Baltimore, MD, May 1997), IEEE, pp. 449–456.

[23] BIRRER, S., AND BUSTAMANTE, F. Nemo: Resilient peer-to-peer multicast without the cost. In *Proceedings of the 12th Annual Multimedia Computing and Networking Conference* (January 2005).

[24] BLYTHE, J., DEELMAN, E., GIL, Y., KESSELMAN, C., AGARWAL, A., MEHTA, G., AND VAHI, K. The role of planning in grid computing. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)* (2003).

[25] BRANDT, A. Multi-level adaptive solutions to boundary-value problems. *Mathematics of Computation 31,* 138 (Apr. 1977), 333–390.

[26] BRIGGS, W. L. *A Multigrid Tutorial*. SIAM Publications, 1987.

[27] BRU, R., AND MARIN, J. BSP cost of the preconditioned conjugate gradient method. In *Proceedings of the VII Jornadas de Paralelismo* (Santiago de Compostela, Spain, Sept. 1996), Universidad de Santiago de Compostela.

[28] BUSTAMANTE, F. E., EISENHAUER, G., WIDENER, P., SCHWAN, K., AND PU, C. Active streams-an approach to adaptive distributed systems. In *Eighth IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII). May 20–23, 2001, Schloss Elmau, Germany* (1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 2001), IEEE, Ed., IEEE Computer Society Press, pp. 163–163.

[29] CHAMBERLAIN, B. L., DEITZ, S. J., AND SNYDER, L. A comparative study of the NAS MG benchmark across parallel languages and architectures. In *Proceedings of Supercomputing'2000 (CD-ROM)* (Dallas, TX, Nov. 2000), IEEE and ACM SIGARCH. University of Washington.

[30] CHANDRA, R., ZELDOVICH, N., SAPUNTZAKIS, C. P., AND LAM, M. S. The collective: A cache-based system management architecture. In *NSDI* (2005), USENIX.

[31] CLARK, C., FRASER, K., HAND, S., HANSEN, J. G., JUL, E., LIMPACH, C., PRATT, I., AND WARFIELD, A. Live migration of virtual machines. In *NSDI* (2005), USENIX.

[32] CORNELL, B., DINDA, P., AND BUSTAMANTE, F. Wayback: A user-level versioning file system for linux. In *Proceedings of USENIX 2004 (Freenix Track)* (July 2004).

[33] CROWCROFT, J., FRASER, K., HAND, S., PRATT, I., AND WARFIELD, A. The inevitability of Xen. *;login: the USENIX Association newsletter 30*, 4 (Aug. 2005), ??–??

[34] CYBENKO, G. Dynamic load balancing for distributed shared memory multiprocessors. *Journal of Parallel and Distributed Computing 7*, 2 (October 1989), 279–301.

[35] DIKE, J. A user-mode port of the linux kernel. In *Proceedings of the USENIX Annual Linux Showcase and Conference* (Atlanta, GA, October 2000).

[36] DILGER, A., FLIERL, J., BEGG, L., GROVE, M., AND DISPOT, F. The PVM Patch for POV-Ray. Available at http://pvmpov.sourceforge.net.

[37] DINDA, P., GARCIA, B., AND LEUNG, K.-S. The measured network traffic of compiler-parallelized programs. *Parallel Processing, International Conference on, 2001.*.

[38] DINDA, P. A., GROSS, T., KARRER, R., LOWEKAMP, B., MILLER, N., STEENKISTE, P., AND SUTHERLAND, D. The architecture of the remos system. In *HPDC '01: Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing* (Washington, DC, USA, 2001), IEEE Computer Society, p. 252.

[39] DONNELLY, D., AND RUST, B. The Fast Fourier Transform for experimentalists, part I: Concepts. *Computing in Science and Engineering 7*, 2 (Mar./Apr. 2005), 80–88.

[40] DOUGLIS, F., AND OUSTERHOUT, J. Process migration in the Sprite operating system. In *Proceedings of the 7th International Conference on Distributed Computing Systems (ICDCS)* (September 1987).

[41] EAGER, D. L., LAZOWSKA, E. D., AND ZAHORJAN, J. Adaptive load sharing in homogeneous distributed systems. *IEEE Trans. Software Eng 12*, 5 (1986), 662–675.

[42] EAGER, D. L., LAZOWSKA, E. D., AND ZAHORJAN, J. The limited performance benefits of migrating active processes for load sharing. In *Proc. 1988 ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems* (May 1988).

[43] FIGUEIREDO, R., DINDA, P., AND FORTES, J. A case for grid computing on virtual machines. In *In Proceedings of the 23rd IEEE Conference on Distributed Computing (ICDCS 2003) May 2003* (2003), pp. 550–559.

[44] FINK, J. R. Popular unix performance-monitoring tools for linux. `http://www.informit.com/articles/article.aspx?p=29666&seqNum=1`.

[45] FOSTER, I., KESSELMAN, C., AND TUECKE, S. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of Supercomputer Applications 15* (2001).

[46] FRASER, K. A., HAND, S. M., HARRIS, T. L., LESLIE, I. M., AND PRATT, I. A. The xenoserver computing infrastructure. Tech. Rep. UCAM-CL-TR-552, University of Cambridge, Computer Laboratory, Jan. 2003.

[47] GARFINKEL, T., PFAFF, B., CHOW, J., ROSENBLUM, M., AND BONEH, D. Terra: a virtual machine-based platform for trusted computing. In *Proceedings of the nineteenth ACM symposium on Operating systems principles* (New York, Oct. 19–22 2003), vol. 37, 5 of *Operating Systems Review*, ACM Press, pp. 193–206.

[48] GEIST, A., BEGUELIN, A., DONGARRA, J., JIANG, W., MANCHECK, R., AND SUNDERAM, V. *PVM: Parallel Virtual Machine*. MIT Press, Cambridge, Massachusetts, 1994.

[49] GERBESSIOTIS, A. V., AND SINIOLAKIS, C. J. Communication efficient data structures on the BSP model with applications in computational geometry. In *Proceedings of the 2nd International Euro-Par Conference on Parallel Processing, Vol. I, EURO-PAR'96 (Lyon, France, August 26-29, 1996)*, L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, Eds., vol. 1124 of *LNCS*. Springer-Verlag, Berlin-Heidelberg-New York-London-Paris-Tokyo-Hong Kong-Barcelona-Budapest-Milan-Santa Clara-Singapore, 1996, pp. 348–351.

[50] GERBESSIOTIS, A. V., AND VALIANT, L. G. Direct bulk-synchronous parallel algorithms. *Journal of Parallel and Distributed Computing 22*, 2 (1994), 251–267.

[51] GOTZ, S. Algorithms in cgm, bsp and bsp* model: A survey. Tech. rep., Carleton University, Ottawa, 1996.

[52] GOUDREAU, LANG, AND TSANTILAS. Towards efficiency and portability: Programming with the BSP model. In *SPAA: Annual ACM Symposium on Parallel Algorithms and Architectures* (1996).

[53] GOUDREAU, M. Course: The bsp model for portable efficient parallel computing. Course at Univeristy of Central Florida, 1996. `http://www.cs.ucf.edu/csdept/faculty/goudreau/cop5937_fall96/COP5937.html`.

[54] GOVINDAN, S., NATH, A. R., DAS, A., URGAONKAR, B., AND SIVASUBRAMANIAM, A. Xen and co: communication-aware CPU scheduling for consolidated xen-based hosting platforms. In *VEE* (2007), C. Krintz, S. Hand, and D. Tarditi, Eds., ACM, pp. 126–136.

[55] GRIMSHAW, A., WULF, W., AND THE LEGION TEAM. The legion vision of a worldwide virtual computer. *Communications of the ACM 40*, 1 (1997).

[56] GRIMSHAW, A. S., STRAYER, W. T., AND P.NARAYAN. Dynamic object-oriented parallel processing. *IEEE Parallel and Distributed Technology: Systems and Applications*, 5 (May 1993), 33–47.

[57] GUPTA, A., AND DINDA, P. A. Infering the topology and traffic load of parallel programs running in a virtual machine environment. In *Proceedings of the 10th Workshop on Job Scheduling Policies for Parallel Program Processing(JSPPP)* (June 2004).

[58] GUPTA, A., AND DINDA, P. A. Inferring the topology and traffic load of parallel programs running in a virtual machine environment. In *Proceedings of the 10th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)* (June 2004).

[59] GUPTA, A., LIN, B., AND DINDA, P. A. Measuring and understanding user comfort with resource borrowing. In *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC 2004)* (June 2004). To Appear.

[60] GUPTA, A., ZANGRILLI, M., SUNDARARAJ, A., DINDA, P., AND LOWEKAMP., B. Free network measurement for adaptive virtualized distributed computing. Tech. rep., Department of Computer Science, Northwestern University, June 2005.

[61] GUPTA, A., ZANGRILLI, M. A., SUNDARARAJ, A. I., HUANG, A. I., DINDA, P. A., AND LOWEKAMP, B. B. Free network measurement for adaptive virtualized distributed computing. In *Proc. IEEE International Parallel & Distributed Processing Symposium (20th IPDPS'06)* (Rhodes Island, Greece, Apr. 2006), IEEE Computer Society.

[62] GUPTA, D., CHERKASOVA, L., GARDNER, R., AND VAHDAT, A. Enforcing performance isolation across virtual machines in xen. In *Middleware* (2006), M. van Steen and M. Henning, Eds., vol. 4290 of *Lecture Notes in Computer Science*, Springer, pp. 342–362.

[63] GUPTA, D., GARDNER, R., AND CHERKASOVA, L. Xenmon: QoS monitoring and performance profiling tool. Tech. Rep. HPL-2005-187, Hewlett Packard Laboratories, Oct. 21 2005.

[64] GUSTAVSON, F. G., AGARWAL, R. C., ALPERN, B., CARTER, L., KLEPACKI, D. J., LAWRENCE, R., AND ZUBAIR, M. High-performance parallel implementations of the NAS kernel benchmarks on the IBM SP2. *IBM Systems Journal 34*, 2 (1995).

[65] HABIB, I. Xen. *Linux Journal 2006*, 145 (May 2006), ??–??

[66] HAILPERIN, M. Load balancing for massively-parallel soft-real-time systems. Int Report STAN-CS-88-1222, also KSL-88-62, Stanford University, Dept of Computer Science, Sept. 1988.

[67] HAND, S., HARRIS, T., KOTSOVINOS, E., AND PRATT, I. Controlling the xenoserver open platform. In *Proceedings of the 6th IEEE Conference on Open Architectures and Network Programming (IEEE OPENARCH'03)* (2002), IEEE Computer Society.

[68] HARCHOL-BALTER, M., AND DOWNEY, A. Exploiting process lifetime distributions for dynamic load balancing. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer SYstems* (New York, May 23–26 1996), vol. 24,1 of *ACM SIGMETRICS Performance Evaluation Review*, ACM Press, pp. 13–24.

[69] HAROLD W. CAIN, RAVI RAJWAR, M. M., AND LIPASTI, M. H. An architectural evaluation of Java TPC-W. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture* (January 2001).

[70] HARRIS, F. J. On the use of windows for harmonic analysis with the discrete fourier transform. *Proc. IEEE 66*, 1 (Jan. 1978), 51–83.

[71] HEMKER, P. W. Sparse-grid finite-volume multigrid for 3D-problems. *Advances in computational mathematics 4* (1995), 83–110. ???

[72] HILL, J. M. D., CRUMPTON, P. I., AND BURGESS, D. A. Theory, practice, and a tool for BSP performance prediction. In *Euro-Par'96 Parallel Processing, Second International Euro-Par Conference, Volume II, (2nd Euro-Par'96)* (Lyon, France, Aug. 1996), L. Bouge, P. Fraigniaud, A. Mignotte, and Y. Robert, Eds., vol. 1124 of *Lecture Notes in Computer Science (LNCS)*, pp. 697–705.

[73] HUA CHU, Y., RAO, S., AND ZHANG, H. A case for end-system multicast. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (2000), pp. 1–12.

[74] HWANG, K., XU, Z., AND AR, M. Stap benchmark performance on the ibm sp2 massively parallel processor. In *IEEE Transactions on Parallel and Distributed Systems* (May 1996).

[75] IEEE. Special issue on fast fourier transform. *IEEE Trans. on Audio and Electroacoustics AU-17* (1969), 65–186.

[76] JIANG, H., AND DOVROLIS, C. Why is the internet traffic bursty in short time scales? In *SIGMETRICS 2005* (August 2005), ACM.

[77] JIANG, X., AND XU, D. SODA: A service-on-demand architecture for application service hosting utility platforms. In *Proc. 12th International Symposium on High-Performance Distributed Computing (12th HPDC'03)* (Seattle, Washington, USA, June 2003), IEEE Computer Society, pp. 174–183.

[78] JIANG, X., AND XU, D. Violin: Virtual internetworking on overlay infrastructure. Tech. rep., Tech. Rep. CSD TR 03-027, Department of Computer Sciences, Purdue University, July 2003.

[79] JONES, S. T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Geiger: monitoring the buffer cache in a virtual machine environment. In *ASPLOS* (2006), J. P. Shen and M. Martonosi, Eds., ACM, pp. 14–24.

[80] KALLAHALLA, M., UYSAL, M., SWAMINATHAN, R., LOWELL, D. E., WRAY, M., CHRISTIAN, T., EDWARDS, N., DALTON, C. I., AND GITTLER, F. SoftUDC: A software-based data center for utility computing. *IEEE Computer 37*, 11 (2004), 38–46.

[81] KEAHEY, K., DOERING, K., AND FOSTER, I. T. From sandbox to playground: Dynamic virtual environments in the grid. In *GRID* (2004), R. Buyya, Ed., IEEE Computer Society, pp. 34–42.

[82] KICHKAYLO, T., AND KARAMCHETI, V. Optimal resource-aware deployment planning for component-based distributed applications. In *Proceedings of the Thirteenth IEEE International Symposium on High-Performance Distributed Computing (HPDC)* (June 2004), pp. 150–159.

[83] KIRKPATRICK, S., GELATT, C. D., AND VECCHI, M. P. Optimization by simulated annealing. *Science 220* (1983), 671–680.

[84] KOBES, R. Math::fft - perl module to calculate fast fourier transforms. `http://search.cpan.org/~rkobes/Math-FFT-1.28/`.

[85] KOZUCH, M., SATYANARAYANAN, M., BRESSOUD, T., AND KE, Y. Efficient state transfer for internet suspend/resume. Tech. Rep. IRP-TR-02-03, Intel Research Laboratory at Pittburgh, May 2002.

[86] KURSONA, J. F., AND CHIPALKATTI, R. Load sharing in soft real-time distributed computer systems. *IEEE Transactions on Computers C-36*, 8 (1987), 993–1000.

[87] LABORATORY, A. N. Integrating task and data parallelism. `http://www-fp.mcs.anl.gov/fortran-m/integrate.html`.

[88] LANGE, J. R., SUNDARARAJ, A. I., AND DINDA, P. A. Automatic dynamic run-time optical network reservations. In *Proc. 14th International Symposium on High-Performance Distributed Computing (13th HPDC'05)* (Honolulu, Hawaii, USA, June 2005), IEEE Computer Society, pp. 77–86.

[89] LEIGHTON, T. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann, 1992.

[90] LELAND, W., AND OTT, T. Load-balancing heuristics and process behavior. In *ACM Performance Evaluation Review: Proc. Performance '86 and ACM SIGMETRICS 1986, Vol. 14* (May 1986), pp. 54–69.

[91] LIN, B., AND DINDA, P. A. VSched: Mixing batch and interactive virtual machines using periodic real-time scheduling. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing 2005, Seattle, WA, November 12–18 2005* (pub-ACM:adr and pub-IEEE:adr, 2005), ACM, Ed., ACM Press and IEEE Computer Society Press, pp. 8–8.

[92] LIN, B., AND DINDA, P. A. Towards scheduling virtual machines based on direct user input. In *Proceedings of the 1st International Workshop on Virtualization Technology in Distributed Computing (VTDC'06)* (2006).

[93] LIN, B., SUNDARARAJ, A. I., AND DINDA, P. A. Time-sharing parallel applications with performance isolation and control. In *ICAC '07: Proceedings of the Fourth International Conference on Autonomic Computing* (Washington, DC, USA, 2007), IEEE Computer Society, p. 28.

[94] LINUX VSERVER PROJECT. http://www.linux-vserver.org.

[95] LOPEZ, J., AND O'HALLARON, D. Support for interactive heavyweight services. In *Proceedings of the 10th IEEE Symposium on High Performance Distributed Computing HPDC 2001* (2001).

[96] LOWEKAMP, B., AND BEGUELIN, A. ECO: Efficient collective operations for communication on heterogeneous networks. In *Proceedings of the International Parallel Processing Symposium (IPPS 1996)* (1996), pp. 399–405.

[97] LOWEKAMP, B., O'HALLARON, D., AND GROSS, T. Direct queries for discovering network resource properties in a distributed environment. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing (HPDC99)* (August 1999), pp. 38–46.

[98] LOWEKAMP, B., TIERNEY, B., COTTREL, L., HUGHES-JONES, R., KIELEMANN, T., AND SWANY, M. A hierarchy of network performance characteristics for grid applications and services. Tech. Rep. Recommendation GFD-R.023, Global Grid Forum, May 2004.

[99] LYNCH, J. Unix and web performance. `http://www.incogen.com`.

[100] LYONS, R. Windowing functions improve FFT results, part I. *Test Measurement World* (1998).

[101] LYONS, R. Windowing functions improve FFT results, part II. *Test Measurement World* (1998).

[102] MAN, C. L. T., HASEGAWA, G., AND MURATA, M. A merged inline measurement method for capacity and available bandwidth. In *Passive and Active Measurement Workshop (PAM2005)* (2005), pp. 341–344.

[103] MARIN, M. Direct BSP algorithms for parallel discrete-event simulation. Tech. Rep. PRG-TR-8-97, Oxford University, Jan. 1997.

[104] MASSIE, M. L., CHUN, B. N., AND CULLER, D. E. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing 30*, 5-6 (2004), 817–840.

[105] MCCANNE, S., AND JACOBSON, V. The BSD packet filter: A new architecture for user-level packet capture. In *Prodeedings of USENIX 1993* (1993), pp. 259–270.

[106] MEDINA, A., LAKHINA, A., MATTA, I., AND BYERS, J. Brite: An approach to universal topology generation. In *Proccedings of International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS)* (2001).

[107] MILLS, D. L. The network time protocol (ntp) distribution. `http://www.eecis.udel.edu/~mills/ntp/html/index.html`.

[108] MILOJICIC, D. S., DOUGLIS, F., PAINDAVEINE, Y., WHEELER, R., AND ZHOU, S. Process migration. *ACM Comput. Surv 32*, 3 (2000), 241–299.

[109] MULLER, A., WILSON, S., HAPPE, D., AND HUMPHREY, G. J., Eds. *Virtualization with VMware ESX Server*. Syngress Publishing, Inc., pub-SYNGRESS:adr, 2005.

[110] NAZIR, F., COTTRELL, L., CHHAPARIA, M., WACHSMANN, A., AND LOGG, C. Comparison between two dimensional time-series kolmogorov smirnoff technique for anomalous variations in end-to-end internet traffic, 2006. `http://www-iepm.slac.stanford.edu/monitoring/forecast/ksvsmb/ksvsmb.htm`.

[111] NOBLE, B. D., SATYANARAYANAN, M., NARAYANAN, D., TILTON, J. E., FLINN, J., AND WALKER, K. R. Agile application-aware adaptation for mobility. In *Sixteen ACM Symposium on Operating Systems Principles* (Saint Malo, France, Oct. 1997), pp. 276–287.

[112] OSMAN, S., SUBHRAVETI, D., SU, G., AND NIEH, J. The design and implementation of Zap: A system for migrating computing environments. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* (December 2002).

[113] PRASAD, R., MURRAY, M., DOVROLIS, C., AND CLAFFY, K. Bandwidth estimation: Metrics, measurement techniques, and tools. In *IEEE Network* (June 2003).

[114] RANJAN, S., ROLIA, J., FU, H., AND KNIGHTLY, E. Qos-driven server migration for internet data centers. In *In Proc. IWQoS 2002* (2002).

[115] REED, J. N., PARROTT, K., AND LANFEAR, T. Portability, predictability and performance for parallel computing: BSP in practice. *Concurrency - Practice and Experience 8*, 10 (1996), 799–812.

[116] REYNOLDS, P., WIENER, J. L., MOGUL, J. C., AGUILERA, M. K., AND VAHDAT, A. WAP5: black-box performance debugging for wide-area systems. In *WWW* (2006), L. Carr, D. D. Roure, A. Iyengar, C. A. Goble, and M. Dahlin, Eds., ACM, pp. 347–356.

[117] RIBEIRO, V., RIEDI, R. H., BARANIUK, R. G., NAVRATIL, J., AND COTTRELL, L. pathChirp:Efficient Available Bandwidth Estimation for Network Paths. In *Passive and Active Measurement Workshop (PAM2003)* (2003).

[118] ROSENBERG, J., WEINBERGER, J., HUITEMA, C., AND MAHY, R. Stun: Simple traversal of user datagram protocol (udp) through network address translators (nats). Tech. Rep. RFC 3489, Internet Engineering Task Force, March 2003.

[119] RUTH, P., MCGACHEY, P., AND XU, D. Viocluster: Virtualization for dynamic computational domains. In *In IEEE CLUSTER, 2005* (2005).

[120] SAPUNTZAKIS, C., BRUMLEY, D., CHANDRA, R., ZELDOVICH, N., CHOW, J., LAM, M. S., AND ROSENBLUM, M. Virtual appliances for deploying and maintaining software. In *Proceedings of the Seventeenth Large Installation Systems Administration Conference (LISA 2003)* (Oct. 2003), pp. 181–194.

[121] SAPUNTZAKIS, C., CHANDRA, R., PFAFF, B., CHOW, J., LAM, M., AND ROSENBLUM, M. Optimizing the migration of virtual computers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* (December 2002).

[122] SAVAGE, S. Sting: A TCP-based network measurement tool. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems* (1999).

[123] SAVAGE, S., COLLINS, A., HOFFMAN, E., SNELL, J., AND ANDERSON, T. E. The end-to-end effects of internet path selection. In *SIGCOMM* (1999), pp. 289–299.

[124] SESHAN, S., STEMM, M., AND KATZ, R. H. SPAND: Shared passive network performance discovery. In *Proceedings of the 1997 USENIX Symposium on Internet Technologies and System (USITS)* (97).

[125] SHAKKOTTAI, S., BROWNLEE, N., AND CLAFFY, K. A study of burstiness in tcp flows. In *Passive and Active Measurement Workshop (PAM2005)* (2005), pp. 13–26.

[126] SHOYKHET, A., LANGE, J., AND DINDA, P. Virtuoso: A system for virtual machine marketplaces. Tech. Rep. NWU-CS-04-39, Department of Computer Science, Northwestern University, July 2004.

[127] SIEGELL, B. S., AND STEENKISTE, P. Automatic generation of parallel programs with dynamic load balancing. In *Proceedings of the Third International Symposium on High Performance Distributed Computing (3rd HPDC'94)* (San Francisco, CA, USA, Apr. 1994), IEEE Computer Society, pp. 166–175.

[128] SINHA, R., PAPADOPOULOS, C., AND HEIDEMANN, J. Fingerprinting internet paths using packet pair dispersion. Tech. Rep. Technical Report No. 06876, University of Southern California, 2005.

[129] S.M.WHITE, A.ALUND, AND V.S.SUNDERAM. Nas parallel benchmarks implementation. `http://web.bilkent.edu.tr/Online/pvm/CoverStuff/vaidy.html`.

[130] STASKO, J. Samba Algorithm Animation System, 1998. `http://www.cc.gatech,edu/gvu/softviz/algoanim/samba.html`.

[131] STEENSGAARD, B., AND JUL, E. Object and native code thread mobility among heterogeneous computers. In *Proceedings of the 15th Symposium on Operating Systems Principles (15th SOSP'95), Operating Systems Review (OSR)* (Copper Mountain, CO, Dec. 1995), ACM SIGOPS, pp. 68–78. Published as Proceedings of the 15th Symposium on Operating Systems Principles (15th SOSP'95), Operating Systems Review (OSR), volume 29, number 5.

[132] SUEHIRO, K., MURAI, H., AND SEO, Y. Integer sorting on shared-memory vector parallel computers. In *Proceedings of the International Conference on Supercomputing (ICS-98)* (New York, July 13–17 1998), ACM press, pp. 313–320.

[133] SUN, Y., WANG, J., AND XU, Z. Architectual implications of the NAS MG and FT parallel benchmarks. In *APDC* (1997), IEEE Computer Society, pp. 235–240.

[134] SUNDARARAJ, A., AND DINDA, P. Towards virtual networks for virtual machine grid computing. In *Proceedings of the 3rd USENIX Virtual Machine Research And Technology Symposium (VM 2004)* (May 2004).

[135] SUNDARARAJ, A., GUPTA, A., AND DINDA, P. Dynamic topology adaptation of virtual networks of virtual machines. In *Proceedings of the Seventh Workshop on Languages, Compilers and Run-time Support for Scalable Systems (LCR)* (November 2004).

[136] SUNDARARAJ, A., GUPTA, A., AND DINDA, P. Increasing distributed application performance in virtual environments through run-time inference and adaptation. In *Proceedings of the 14th IEEE International Symposium on High Performance Distributed Computing (HPDC 2005)* (2005), IEEE Computer Society.

[137] SUNDARARAJ, A., SANGHI, M., LANGE, J., AND DINDA, P. An optimization problem in adaptive virtual environmnets. In *Proceedings of the seventh Workshop on Mathematical Performance Modeling and Analysis (MAMA)* (June 2005).

[138] SUNDARARAJ, A. I. *Automatic, Run-time and Dynamic Adaptation of Distributed Applications Executing in Virtual Environments*. PhD thesis, Northwestern University, Department of Electrical Engineering and Computer Science, November 2006.

[139] SUNDARARAJ, A. I., GUPTA, A., AND DINDA, P. A. Increasing application performance in virtual environments through run-time inference and adaptation. In *Proceedings of the Fourteenth International Symposium on High Performance Distributed Computing (HPDC)* (July 2005).

[140] TAPUS, C., CHUNG, I.-H., AND HOLLINGSWORTH, J. Active harmony: Towards automated performance tuning. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing* (2002), pp. 1–11.

[141] THAIN, D., AND LIVNY, M. Bypass: A tool for building split execution systems. In *Proceedings of the Ninth IEEE Symposium on High Performance Distributed Computing (HPDC9)* (Pittsburgh, PA, August 2000).

[142] TRIDGELL, A. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, Australian National University, 1999.

[143] VALIANT, L. A bridging model for parallel computation. *Communications of the ACM 33*, 8 (Aug. 1990), 103–111.

[144] VMWARE CORPORATION. http://www.vmware.com.

[145] VMWARE INC. VMWare ESX Server: Platform for virtualizing servers, storage and networking. `http://www.vmware.com/pdf/esx_datasheet.pdf`, 2006.

[146] WALDSPURGER, C. A. Memory resource management in VMware ESX server. In *Proceedings of the 5th ACM Symposium on Operating System Design and Implementation (OSDI-02)* (New York, Dec. 9–11 2002), Operating Systems Review, ACM Press, pp. 181–194.

[147] WESSON, V. Tn200407a - network time protocol. `http://www.seis.com.au/TechNotes/TN200407A_NTP.html`.

[148] WHITE, S., ALUND, A., AND SUNDERAM, V. S. Performance of the NAS parallel benchmarks on PVM-Based networks. *Journal of Parallel and Distributed Computing 26*, 1 (Apr. 1995), 61–71.

[149] WHITE, S., ALUND, A., AND SUNDERAM, V. S. Performance of the NAS parallel benchmarks on PVM-Based networks. *Journal of Parallel and Distributed Computing 26*, 1 (1995), 61–71.

[150] WILLEBEEK-LEMAIR, M., AND REEVES, A. P. Strategies for dynamic load balancing on highly parallel computers. *IEEE Transactions on Parallel and Distributed Systems PDS-4*, 9 (Sept. 1993), 979–993. Dynamic Load Balancing Strategies for Highly Parallel Multicomputer Systems.

[151] WOLSKI, R. Forecasting network performance to support dynamic scheduling using the network weather service. In *HPDC '97: Proceedings of the 6th IEEE International Symposium on High Performance Distributed Computing* (Washington, DC, USA, 1997), IEEE Computer Society, p. 316.

[152] WOLSKI, R., SPRING, N. T., AND HAYES, J. The network weather service: A distributed resource performance forecasting system. *Journal of Future Generation Computing Systems* (1999).

[153] WOOD, T., SHENOY, P. J., VENKATARAMANI, A., AND YOUSIF, M. S. Black-box and gray-box strategies for virtual machine migration. In *NSDI* (2007), USENIX.

[154] ZANDY, V. C., MILLER, B. P., AND LIVNY, M. Process hijacking. In *Proceedings of the 8th IEEE Symposium on High Performance Distributed Computing* (Redondo Beach, CA, August 1999).

[155] ZANGRILLI, M., AND LOWEKAMP, B. Using passive traces of application traffic in a network monitoring system. In *Proc. 13th International Symposium on High-Performance Distributed Computing (13th HPDC'04)* (Honolulu, Hawaii, USA, June 2004), IEEE Computer Society, pp. 77–86.

[156] ZANGRILLI, M., AND LOWEKAMP, B. B. Applying principles of active available bandwidth algorithms to passive tcp traces. In *Passive and Active Measurement Workshop (PAM 2005)* (March 2005), LNCS, pp. 333–336.

[157] ZHANG, J., AND FIGUEIREDO, R. J. Application classification through monitoring and learning of resource consumption patterns. In *Proc. IEEE International Parallel & Distributed Processing Symposium (20th IPDPS'06)* (Rhodes Island, Greece, Apr. 2006), IEEE Computer Society. U. Florida, USA.

[158] ZHANG, Y., DU, N., PAXSON, E., AND SHENKER, S. the constancy of internet path properties, 2001.

[159] ZINKY, J. A., BAKKEN, D. E., AND SCHANTZ, R. E. Architectural support for quality of service for CORBA objects. *Theory and Practice of Object Sytems 3*, 1 (1997), 55–73.

Appendix A

Increasing Application Performance In Virtual Environments through Run-time Inference and Adaptation

Note: This work was done in collaboration with my colleague Ananth I. Sundararaj and Prof. Peter Dinda and published in High Performance Distributed Computing (HPDC) 2005

A.1 Introduction

Virtual machines can greatly simplify grid and distributed computing by lowering the level of abstraction from traditional units of work, such as jobs, processes, or RPC calls to that of a raw machine. This abstraction makes resource management easier from the perspective of resource providers and results in lower complexity and greater flexibility for resource users. A virtual machine image that includes preinstalled versions of the correct operating system, libraries, middleware and applications can make the deployment of new software far simpler. We have made a detailed case for grid computing on virtual machines in a previous paper [43]. We are developing a middleware system, Virtuoso, for virtual machine grid computing [126].

Grid computing is intrinsically about using multiple sites, with different network management and security philosophies, often spread over the wide area [45]. Running a virtual machine

on a remote site is equivalent to visiting the site and connecting a new machine. The nature of the network presence (active Ethernet port, traffic not blocked, routable IP address, forwarding of its packets through firewalls, etc) the machine gets, or whether it gets a presence at all, depends completely on the policy of the site. Not all connections between machines are possible and not all paths through the network are free. The impact of this variation is further exacerbated as the number of sites is increased, and if we permit virtual machines to migrate from site to site.

To deal with this network management problem in Virtuoso, we developed VNET [134], a simple layer 2 virtual network tool. Using VNET, virtual machines have no network presence at all on a remote site. Instead, VNET provides a mechanism to project their virtual network cards onto another network, which also moves the network management problem from one network to another. Because the virtual network is a layer 2 network, a machine can be migrated from site to site without changing its presence—it always keeps the same IP address, routes, etc. The first version of VNET is publicly available.

An application running in some distributed computing environment, must adapt to the (possibly changing) available computational and networking resources. Despite many efforts [15, 24, 34, 55, 56, 68, 82, 95, 96, 111, 127, 140, 150, 159], adaptation mechanisms and control have remained very application-specific or required rewriting applications. We claim that adaptation using the low-level, application-independent adaptation mechanisms made possible by virtual machines interconnected with a virtual network is effective. This work provides evidence for that claim.

Custom adaptation by either the user or the resource provider is exceedingly complex as the application requirements, computational and network resources can vary over time. VNET is in an ideal position to

1. measure the traffic load and application topology of the virtual machines,
2. monitor the underlying network and its topology,

3. adapt the application as measured in step 1 to the network as measured in step 2, and

4. adapt the network to the application by taking advantage of resource reservation mechanisms.

This work focuses on steps 1 and 3. There is abundant work that suggests that step 2 can be accomplished within or without the virtual network using both active [122, 152] and passive techniques [97, 124, 155]. We are just beginning to work on step 4.

These services can be done on behalf of *existing, unmodified applications and operating systems* running in the virtual machines. One previous paper [134] laid out the argument and formalized the adaptation problem, while a second paper [135] gave very preliminary results on automatic adaptation using one mechanism. Here, we demonstrate how to control three adaptation mechanisms provided by our system in response to the inferred communication behavior of the application running in a collection of virtual machines, and provide extensive evaluation.

We use the following three adaptation mechanisms:

- Virtual machine migration: Virtuoso allows us to migrate a VM from one physical host to another. Much work exists that demonstrates that fast migration of VMs running commodity applications and operating systems is possible [85, 112, 121]. Migration times down to 5 seconds have been reported [85]. As migration times decrease, the rate of adaptation we can support and our work's relevance increases. Note that while process migration and remote execution has a long history [40, 108, 131, 141, 154], to use these facilities, we must modify or relink the application and/or use a particular OS. Neither is the case with VM migration.

- Overlay topology modification: VNET allows us to modify the overlay topology among a user's VMs at will. A key difference between it and overlay work in the application layer multicast community [19, 23, 73] is that the VNET provides global control of the topology, which our adaptation algorithms currently (but not necessarily) assume.

- Overlay forwarding: VNET allows us to modify how messages are routed on the overlay. Forwarding tables are globally controlled, and topology and routing are completely separated, unlike in multicast systems.

A.2 Virtuoso

Virtuoso is a system for virtual machine grid computing that for a user very closely emulates the existing process of buying, configuring, and using an Intel-based computer or collection of computers from a web site. Virtuoso does rudimentary admission control of VMs, but the work described here additionally provides the ability for the system to adapt when the user cannot state his resource requirements, and the ability to support a mode of operation in which VMs and other processes compete for resources. In effect, the more competition, the cheaper the cost of admission. More details are available elsewhere [126].

A.2.1 VNET

VNET is the part of our system that creates and maintains the networking illusion, that the user's virtual machines (VMs) are on the user's local area network. The specific mechanisms we use are packet filters, packet sockets, and VMware's [144] host-only networking interface. Each physical machine that can instantiate virtual machines (a host) runs a single VNET daemon. One machine on the user's network also runs a VNET daemon. This machine is referred to as the Proxy.

Although we use VMware as our virtual machine monitor (VMM), VNET can operate with any VMM that provides an externally visible representation of the virtual network interface. For example, VNET, without modification, has been successfully used with User Mode Linux [35] and the VServer extension to Linux [94].

Figure A.1 shows a typical startup configuration of VNET for four hosts, each of which may support multiple VMs. Each of the VNET daemons is connected by a TCP connection (a

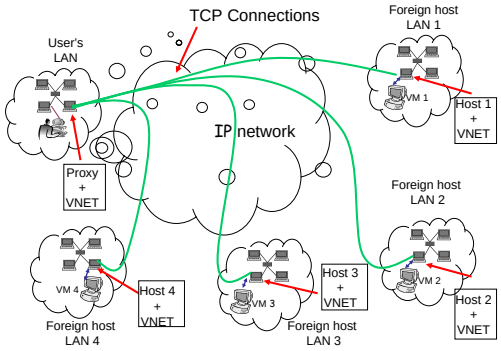


Figure A.1: VNET startup topology.

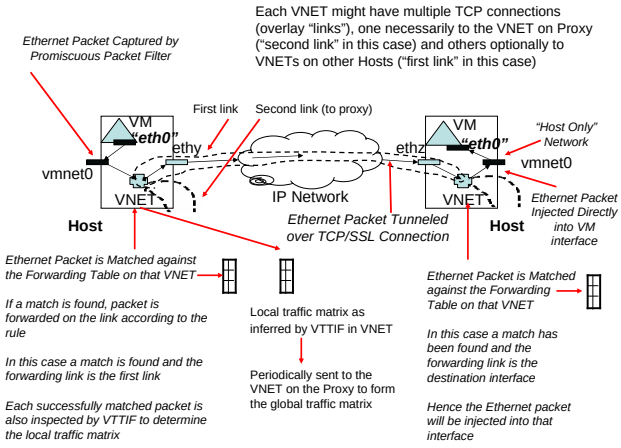


Figure A.2: A VNET link.

VNET link) to the VNET daemon running on the Proxy. We refer to this as the resilient star backbone centered on the Proxy. By resilient, we mean it will always be possible to at least make these connections and reestablish them on failure. We would not be running a VM on any of these host machines if it were not possible in some way to communicate with them. This communication mechanism can be exploited to provide VNET connectivity for a remote VM. For example, if an SSH connection can be made to the host, VNET traffic can be tunneled over the SSH connection.

The VNET daemons running on the hosts and Proxy open their virtual interfaces in promiscuous mode using Berkeley packet filters [105]. Each packet captured from the interface or

received on a link is matched against a forwarding table to determine where to send it, the possible choices being sending it over one of its outgoing links or writing it out to one of its local interfaces using libnet, which is built on packet sockets, available on both Unix and Windows.

Figure A.2 illustrates the operation of a VNET link. Each successfully matched packet is also passed to VTTIF. The Proxy, through its physical interface, provides a network presence for all the VMs on the user's LAN and makes their configuration a responsibility of the user and his site administrator.

The star topology is simply the initial configuration. Additional links and forwarding rules can be added or removed at any time. In the case of migration, the VM seamlessly maintains its layer 2 and layer 3 network presence; neither MAC nor IP addresses change and the external network presence of the VM remains on the LAN of the Proxy. Figure A.7 shows a VNET configuration that has been dynamically adapted to reflect a topology change.

A VNET client can query any VNET daemon for available network interfaces, links, and forwarding rules. It can add or remove overlay links and forwarding rules. The primitives generally execute in $\sim$20 ms, including client time. On initial startup VNET calculates an upper bound on the time taken to configure itself (or change topology). This number is used to determine sampling and smoothing intervals in VTTIF, as we describe below.

Building on the primitives, we have developed a language for describing the VM to host mapping, the topology, and its forwarding rules. A VNET overlay is usually managed using scripts that generate or parse descriptions in that language. We can

- Start up a collection of VNET daemons and establish an initial topology among them.
- Fetch and display the current topology and VM mappings.
- Fetch and display the route a packet will take between two Ethernet addresses.
- Compute the differences between the current topology, forwarding rules, and mappings and a specified topology, forwarding rules, and mappings.

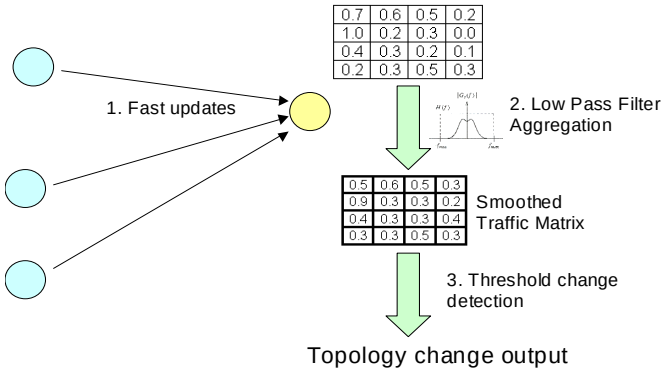


Figure A.3: An overview of the dynamic topology inference mechanism in VTTIF.

- Reconfigure the topology, forwarding rules, and VM mappings to match a specified topology, forwarding rules, and mappings.
- Fetch and display the current application topology using VTTIF.

A.2.2 VTTIF

The Virtual Topology and Traffic Inference Framework integrates with VNET to automatically infer the dynamic topology and traffic load of applications running inside the VMs in the Virtuoso system. In our earlier work [57], we demonstrated that it is possible to successfully infer the behavior of a BSP application by observing the low level traffic sent and received by each VM in which it is running. Here we show how to smooth VTTIF's reactions so that adaptation decisions made on its output cannot lead to oscillation.

VTTIF works by examining each Ethernet packet that a VNET daemon receives from a local VM. VNET daemons collectively aggregate this information producing a global traffic matrix for all the VMs in the system. The application topology is then recovered from this matrix by applying normalization and pruning techniques [57]. Since the monitoring is done below the VM, it does not depend on the application or the operating system in any manner. VTTIF automatically reacts to interesting changes in traffic patterns and reports them, driving

Figure A.4: The NAS IS benchmark running on 4 VM hosts as inferred by VTTIF.

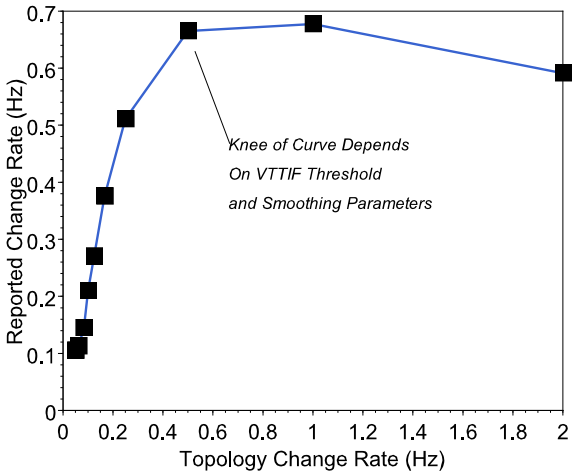


Figure A.5: VTTIF is well damped.

the adaptation process. Figure A.3 illustrates VTTIF.

VTTIF can accurately recover common topologies from both synthetic and application benchmarks like the PVM-NAS benchmarks. For example, Figure A.4 shows the topology inferred by VTTIF from the NAS benchmark Integer Sort [149] running on VMs. The thickness of each link reflects the intensity of communication along it. VTTIF adds little overhead to VNET. Latency is indistinguishable while throughput is affected by $\sim$1%.

Performance VTTIF runs continuously, updating its view of the topology and traffic load matrix among a collection of Ethernet addresses being supported by VNET. However, in the face of dynamic changes, natural questions arise: How fast can VTTIF react to topology change? If the topology is changing faster than VTTIF can react, will it oscillate or provide a damped view of the different topologies? VTTIF also depends on certain configuration parameters which affect its decision whether the topology has changed. How sensitive is VTTIF to the choice of configuration parameters in its inference algorithm?

The reaction time of VTTIF depends on the rate of updates from the individual VNET daemons. A fast *update rate* imposes network overhead but allows a finer time granularity over which topology changes can be detected. In our current implementation, at the fastest, these updates arrive at a rate of 20 Hz. At the Proxy, VTTIF then aggregates the updates into a global traffic matrix. To provide a stable view of dynamic changes, it applies a low pass filter to the updates, aggregating the updates over a sliding window and basing its decisions upon this aggregated view.

Whether VTTIF reacts to an update by declaring that the topology has changed depends on the *smoothing interval* and the *detection threshold*. The smoothing interval is the sliding window duration over which the updates are aggregated. This parameter depends on the adaptation time of VNET, which is measured at startup, and determines how long a change must persist before VTTIF notices. The detection threshold determines if the change in the aggregated global traffic matrix is large enough to declare a change in topology. After VTTIF determines that a topology has changed, it will take some time for it to settle, showing no further topology changes. The best case settle time that we have measured is one second, on par with the adaptation mechanisms.

Given an update rate, smoothing interval, and detection threshold, there is a maximum rate of topology change that VTTIF can keep up with. Beyond this rate, we have designed VTTIF to stop reacting, settling into a topology that is a union of all the topologies that are unfolding in the network. Figure A.5 shows the reaction rate of VTTIF as a function of the topology change rate and shows that it is indeed well damped. Here, we are using two separate topologies and switching rapidly between them. When this topology change rate exceeds VTTIF's configured rate, the reported change rate settles and declines. The knee of the curve depends on the choice of smoothing interval and update rate, with the best case being $\sim$1 second. Up to this limit, the rate and interval set the knee according to the Nyquist criterion.

VTTIF is largely insensitive to the choice of detection threshold, as shown in Figure A.6.

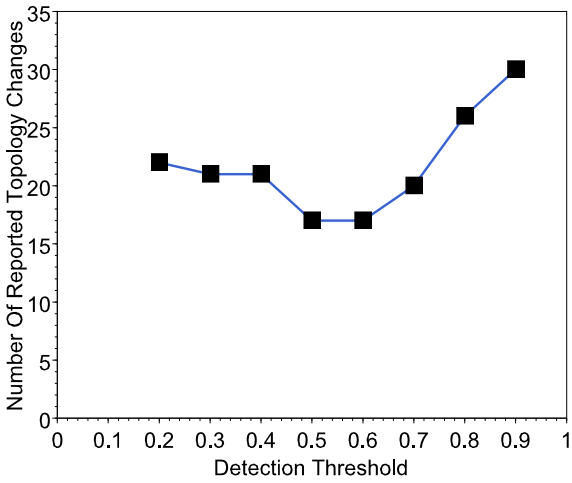


Figure A.6: VTTIF is largely insensitive to the detection threshold.

However, this parameter does determine the extent to which similar topologies can be distinguished. Note that appropriate settings of the VTTIF parameters are determined by the *adaptation mechanisms*, not the *application*.

A.3 Adaptation and VADAPT

Virtuoso uses VTTIF to determine the communication behavior of the application running in a collection of VMs and can leverage the plethora of existing work on network monitoring ([98] is a good taxonomy) to determine the behavior of the underlying resources. The VNET component of Virtuoso provides the mechanisms needed to adapt the application to the network. Beyond this, what is needed is

- the measure of application performance, and
- the algorithms to control the adaptation mechanisms in response to the application and network behaviors.

Here the measure is the throughput of the application.

The adaptation control algorithms are implemented in the VADAPT component of Virtuoso. For a formalization of the adaptation control problem, please see our previous work [134]. The full control problem, informally stated in English, is "Given the network traffic load matrix of the application and its computational intensity in each VM, the topology of the network and the load on its links, routers, and hosts, what is the mapping of VMs to hosts, the overlay topology connecting the hosts, and the forwarding rules on that topology that maximizes the application throughput?"

VADAPT uses greedy heuristic algorithms to quickly answer this question when application information is available, and VM migration and topology/forwarding rule changes are the adaptation mechanisms.

A.3.1 Topology adaptation

VADAPT uses a greedy heuristic algorithm to adapt the VNET overlay topology to the communication behavior of the application. VTTIF infers the application communication topology giving a traffic intensity matrix that is represented as an adjacency list where each entry describes communication between two VMs. The topology adaptation algorithm is as follows:

1. Generate a new list which represents the traffic intensity between VNET daemons that is implied by the VTTIF list and the current mapping of VMs to hosts.
2. Order this list by decreasing traffic intensity.
3. Establish the links in order until c links have been established.

The cost constraint c is supplied by the user or system administrator. The cost constraint can also be specified as a percentage of the total intensity reflected in the inferred traffic matrix, or as an absolute limit on bandwidth.[1]

[1]The precise details of this algorithm (and the next) can be found on our website: http://virtuoso.cs.northwestern.edu/vadapt-algs-rev1.pdf.

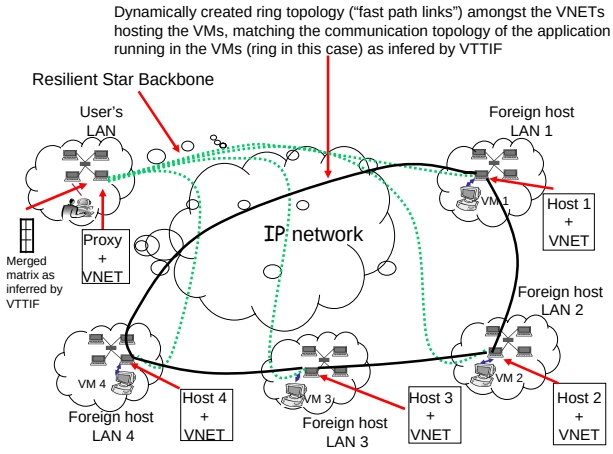


Figure A.7: As the application progresses VNET adapts its overlay topology to match that of the application communication as inferred by VTTIF leading to an significant improvement in application performance, without any participation from the user.

Figure A.7 illustrates topology adaptation. Here, an application configured with neighbor-exchange on a ring application topology of four VMs, starts executing with a VNET star topology (dotted lines) centered on the Proxy. VTTIF infers the topology and in response VADAPT tells VNET to add four links (dark lines) to form an overlay ring among the VNET daemons, thus matching the application’s topology.

We refer to these added links as the *fast path topology*, as they lead to faster communication between the application components. It is important to note that

- The links may be of different types (TCP, UDP, STUN [118], HTTP, SOAP, etc) depending on the security policies of the two sites.
- Some links may be more costly than others (for example, those that support reservations).
- Not all desired links are possible.

The resilient star topology is maintained at all times. The fast path topology and its associated forwarding rules are modified as needed to improve performance.

A.3.2 Migration

VADAPT uses a greedy heuristic algorithm to map virtual machines onto physical hosts. As above, VADAPT uses the application communication behavior as captured by VTTIF and expressed as an adjacency list as its input. In addition, we also use throughput estimates between each pair of VNET daemons arranged in decreasing order. The algorithm is as follows:

1. Generate a new list which represents the traffic intensity between VNET daemons that is implied by the VTTIF list and the current mapping of VMs to hosts.

2. Order the VM adjacency list by decreasing traffic intensity.

3. Order the VNET daemon adjacency list by decreasing throughput.

4. Make a first pass over the VM adjacency list to locate every non-overlapping pair of communicating VMs and map them greedily to the first pair of VNET daemons in the VNET daemon adjacency list which currently have no VMs mapped to them. At the end of the first pass, there is no pair of VMs on the list for which neither VM has been mapped.

5. Make a second pass over the VM adjacency list, locating, in order, all VMs that have not been mapped onto a physical host. These are the “stragglers”.

6. For each of these straggler VMs, in VM adjacency list order, map the VM to a VNET daemon such that the throughput estimate between the VM and its already mapped counterpart is maximum.

7. Compute the differences between the current mapping and the new mapping and issue migration instructions to achieve the new mapping.

A.3.3 Forwarding rules

Once VADAPT determines the overlay topology, we compute the forwarding rules using an all pairs shortest paths algorithm with each edge weight corresponding to the total load on the edge from paths we have determined. This spreads traffic out to improve network performance.

A.3.4 Combining Algorithms

When we combine our algorithms, we first run the migration algorithm to map the VMs to VNET daemons. Next, we determine the overlay topology based on that mapping. Finally, we compute the forwarding rules.

A.4 Experiments with BSP

Our evaluation of VADAPT for bulk-synchronous parallel applications examines inference time, reaction time, and benefits of adaptation using topology adaptation, migration, and both. We find that the overheads of VADAPT are low and that the benefits of adaptation can be considerable. This is especially remarkable given that the system is completely automated, requiring no help from the application, OS, or developer.

A.4.1 Patterns

Patterns [57] is a synthetic workload generator that captures the computation and communication behavior of BSP programs. In particular, we can vary the number of nodes, the compute/communicate ratio of the application, and select from communication operations such as reduction, neighbor exchange, and all-to-all on application topologies including bus, ring, *n*-dimensional mesh, *n*-dimensional torus, *n*-dimensional hypercube, and binary tree. Patterns emulates a BSP program with alternating dummy compute phases and communication phases according to the chosen topology, operation, and compute/communicate ratio.

A.4.2 Topology Adaptation

In earlier work [135] we demonstrated that topology adaptation alone can increase the performance of patterns, although the evaluation was very limited. We summarize and expand on these results here. We studied all combinations of the following parameters:

- Number of VMs: 4 and 8.
- Application topology and communication patterns: neighbor exchange on a bus, ring, 2D mesh, and all-to-all.
- Environments: (a) All VMs on a single IBM e1350 cluster[2], (b) VMs equally divided between two adjacent IBM e1350 clusters connected by two firewalls and a 10 mbit Ethernet link, (c) VMs equally divided between one IBM e1350 cluster and a slower cluster[3] connected via two firewalls and a campus network, and (d) VMs spread over the wide area hosted on performance-diverse machines at CMU, Northwestern, U.Chicago, and on the DOT network[4].

Reaction Time

For eight VNET daemons in a single cluster that is separated from the Proxy and user by a MAN, different fast path topologies and their default forwarding rules can be configured in 0.7 to 2.3 seconds. This configuration emphasizes the configuration costs. Creating the initial star takes about 0.9 seconds. Recall from Section A.2.2 that the VTTIF inference time depends on the smoothing interval chosen and other parameters, with the best measured time being about one second. In the following, VTTIF is configured with a 60 second smoothing interval.

[2]Nodes are dual 2.0 GHz Xeons with 1.5 GB RAM running Red Hat Linux 9.0 and VMware GSX Server 2.5, connected by a 100 mbit switched network.

[3]Nodes are dual 1 GHz P3s with 1 GB RAM running Red Hat 7.3 and VMware GSX Server 2.5, connected by a 100 mbit switched network

[4]www.dotresearch.org

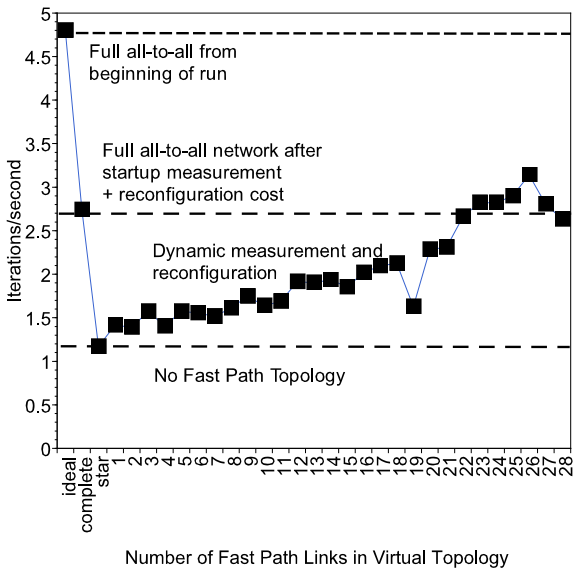


Figure A.8: All-to-all topology with eight VMs, all on the same cluster.

Benefits

If we add *c* of the *n* inferred links using the VADAPT topology adaptation algorithm, how much do we gain in terms of throughput, measured as iterations/second of patterns? We repeated this experiment for all of our configurations. In the following, we show representative results.

Figure A.8 gives an example for the single cluster configuration, here running an 8 VM all-to-all communication. Using only the resilient star, the application has a throughput of $\sim$1.25 iterations/second, which increases to $\sim$1.5 iterations/second when the highest priority fast path link is added. This increase continues as we add links, improving throughput by up to factor of two.

Figure A.9 illustrates the worst performance we measured, for a bus topology among machines spread over two clusters separated by a MAN. Even here, VADAPT did not decrease performance.

Figure A.10 shows performance for 8 VMs, all-to-all, in the WAN scenario, with the hosts spread over the WAN (3 in a single cluster at Northwestern, 2 in another cluster at Northwestern,

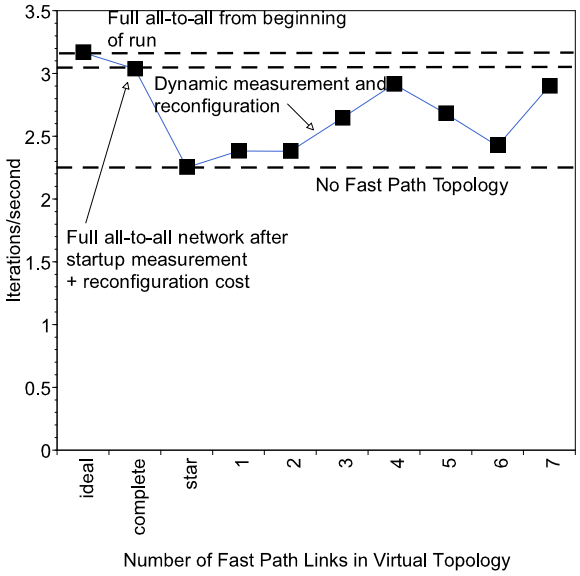


Figure A.9: Bus topology with eight VMs, spread over two clusters over a MAN.

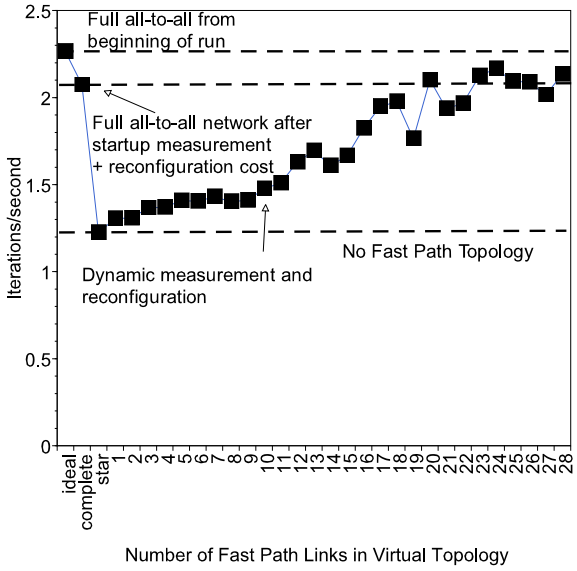


Figure A.10: All-to-all topology with eight VMs, spread over a WAN.

one in a third MAN network, one at U.Chicago, and one at CMU. The Proxy and the user are located on a separate network at Northwestern. Again, we see a significant performance improvement as more and more fast path links are added.

A.4.3 Migration and Topology Adaptation

Here we show, for the first time, results for migration and topology adaptation (Section B.4), separately and together. We studied the following scenarios:

- Adapting to compute/communicate ratio: Patterns was run in 8 VMs spread over the WAN (4 on Northwestern's e1350, 3 on the slower Northwestern cluster, and 1 at CMU). The compute/communicate ratio of patterns was varied.
- Adapting to external load imbalance: Patterns was run in 8 VMs all on Northwestern's e1350. A high level of external load was introduced on one of the nodes of the cluster. The compute/communicate ratio of patterns was varied.

In both cases, patterns executed an all-to-all communication pattern.

Reaction Time

The time needed by VNET to change the topology is as described earlier. The additional cost here is in VM migration. As we mentioned in the introduction, there is considerable work on VM migration. Some of this work has reported times as low as 5 seconds to migrate a full blown personal Windows VM [85]. Although Virtuoso supports plug-in migration schemes, of which we have implemented copy using SSH, synchronization using RSYNC [142], and migration by transferring redo logs in a versioning file system [32], in this work, we use RSYNC. The migration time is typically 300 seconds.

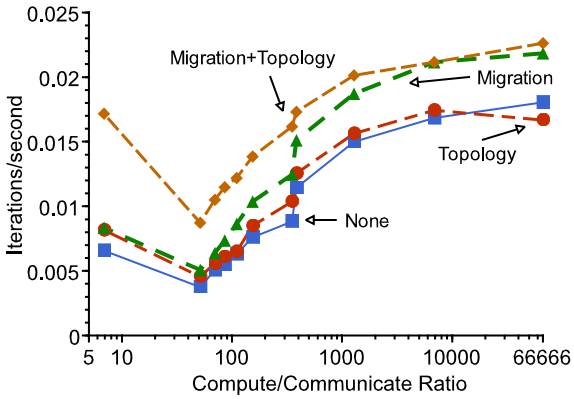


Figure A.11: Effect on application throughput of adapting to compute/communicate ratio.

Benefits

For an application with a low compute/communicate ratio, we would expect that migrating its VMs to a more closely coupled environment would improve performance. We would also expect that it would benefit more from topology adaptation than an application with a high ratio.

Figure A.11 illustrates our scenario of adapting to the compute/communicate ratio of the application. For a low compute/communicate ratio, we see that the application benefits the most from migration to a local cluster and the formation of the fast path links. In the WAN environment, adding the overlay links alone doesn't help much because the underlying network is slow. Adding the overlay links in the local environment has a dramatic effect because the underlying network is much faster.

As we move towards high compute/communicate ratios migration to a local environment results in significant performance improvements. The hosts that we use initially have diverse performance characteristics. This heterogeneity leads to increasing throughput differences as the application becomes more compute intensive. Because BSP applications run at the speed of the slowest node, the benefit of migrating to similar-performing nodes increases as the compute/communicate ratio grows.

Figure A.12, shows the results of adapting to external load imbalance. We can see that

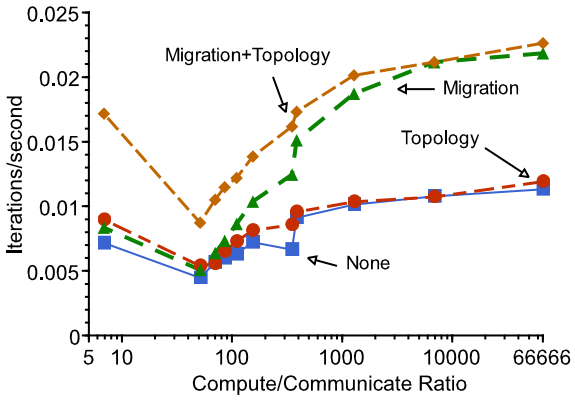


Figure A.12: Effect on application throughput of adapting to external load imbalance.

for low compute/communicate ratios, migration alone does not help much. The VMs are I/O bound here and do not benefit from being relieved of external CPU load. However, migrating to a lightly loaded host *and* adding the fast path links dramatically increases throughput. After the migration, the VM has the CPU cycles needed to drive network much faster.

As the compute/communicate ratio increases, we see that the effect of migration quickly overpowers the effect of adding the overlay links, as we might expect. Migrating the VM to a lightly loaded machine greatly improves the performance of the whole application.

A.4.4 Discussion

It is a common belief that lowering the level of abstraction increases performance while increasing complexity. In this particular case, the rule may not apply. Our abstraction for the user is identical to his existing model of a group of machines, but we can increase the performance he sees. In addition, it is our belief that lowering of the level of abstraction also makes adaptation much more straightforward to accomplish.

Clearly it is possible to use our inference tool, VTTIF, the adaptation mechanisms of VNET, and the adaptation algorithms of VADAPT to greatly increase the performance of existing, unmodified BSP applications running in a VM environment like Virtuoso.

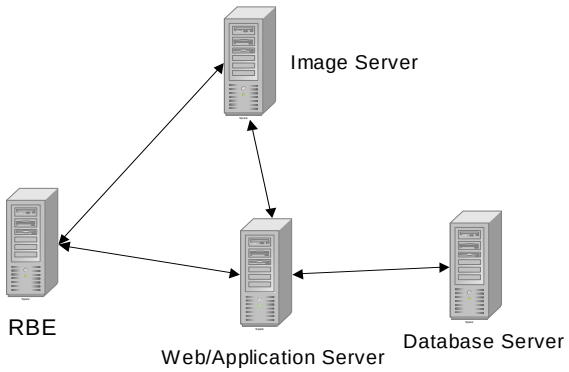


Figure A.13: The configuration of TPC-W used in our experiment.

Adaptation needs to be sensitive to the nature of the application and different or multiple adaptation mechanisms may well be needed to increase performance. The inference capabilities of tools like VTTIF play a critical role in guiding adaptation so that maximum benefit can be derived for the application. While VTTIF tells us the application's resource demands, it does not (yet) tell us where the performance bottleneck is. This is an important next step for us. Determining the application's performance goal is also a key problem. In this work, we used throughput. More generally, we can use an objective function, either given by the programmer or learned from the user.

A.5 Multi-tier Web Sites

Can VADAPT help non-parallel applications? Most web sites serve dynamic content and are built using a multi-tier model, including the client, the web server front end, the application server(s), cache(s), and the database. We are still in the early stages of applying VADAPT to this domain, but we have promising results that indicate that considerable performance gains are possible.

TPC-W is an industry benchmark[5] for such sites. TPC-W models an online bookstore.

[5]We use the Wisconsin PHARM group's implementation [69], particularly the distribution created by Jan

	No Topology	Topology
No Migration	1.216	1.76
Migration	1.4	2.52

Figure A.14: Web throughput (WIPS) with image server facing external load under different adaptation approaches.

The separable components of the site can be hosted in separate VMs. Figure A.13 shows the configuration of TPC-W that we use, spread over four VMs hosted on our e1350 cluster. Remote Browser Emulators (RBEs) simulate users interacting with the web site. RBEs talk to a web server (Apache) that also runs an application server (Tomcat). The web server fetches images from an NFS-mounted image server, alternatively forwarding image requests directly to an Apache server also running on the image server. The application server uses a backend database (MySQL) as it generates content. We run the browsing interaction job mix (5% of accesses are order-related) to place pressure on the front-end web servers and the image server.

The primary TPC-W metric is the WIPS rating. Figure A.14 shows the sustained WIPS achieved under different adaptation approaches. We are adapting to a considerable external load being applied to the host on which the image server is running. When VADAPT migrates this VM to another host in the cluster, performance improves. Reconfiguring the topology also improves performance as there is considerable traffic outbound from the image server. Using both adaptation mechanisms simultaneously increases performance by a factor of two compared to the original configuration.

A.6 Conclusions

We have demonstrated the power of adaptation at the level of a collection of virtual machines connected by a virtual network. Specifically, we can, at run-time, infer the communication topology of a BSP application or web site executing in a set of VMs. Using this information, we

Kiefer.

can dramatically increase application throughput by using heuristic algorithms to place the VMs on appropriate nodes and partially or completely match the application topology in our overlay topology. *Unlike previous work in adaptive systems and load balancing, no modifications to the application or its OS are needed, and our techniques place no requirements on the two other than they generate Ethernet packets.*

Appendix B

Free Network Measurement For Adaptive Virtualized Distributed Computing

This work was done in collaboration with my colleagues Marcia Zangrilli, Ananth I. Sundararaj, Anne I. Huang, Peter A. Dinda and Bruce B. Lowekamp and was published in IEEE International Parallel Distributed Processing Symposium (IPDPS) 2006.

B.1 Introduction

Virtual machines interconnected with virtual networks are an extremely effective platform for high performance distributed computing, providing benefits of simplicity and flexibility to both users and providers [43, 78, 81]. We have developed a virtual machine distributed computing system called Virtuoso [126] that is based on virtual machine monitors and a overlay network system called VNET [134].

A platform like Virtuoso also provides key opportunities for resource and application monitoring, and adaptation. In particular, it can:

1. Monitor the application's traffic to automatically and cheaply produce a view of the application's network demands. We have developed a tool, VTTIF [58], that accomplishes this.
2. Monitor the performance of the underlying physical network by use the application's own traffic to automatically and cheaply probe it, and then use the probes to produce characterizations. This work describes how this is done.

3. Adapt the application to the network to make it run faster or more cost-effectively. This work extends our previous adaptation work [135, 139] with algorithms that make use of network performance information.
4. Reserve resources, when possible, to improve performance [88, 91].

Virtuoso is capable of accomplishing these feats using existing, unmodified applications running on existing, unmodified operating systems.

We build on the success of our Wren passive monitoring and network characterization system [156, **?**] to accomplish (2) above. Wren consists of a kernel extension and a user-level daemon. Wren can:

1. Observe every incoming and outgoing packet arrival in the system with low overhead.
2. Analyze these arrivals using state-of-the-art techniques to derive from them latency and bandwidth information for all hosts that the present host communicates with. Earlier work described offline analysis techniques. This work describes online techniques to continuously and dynamically update the host's view of the network.
3. Collect latency, available bandwidth, and throughput information so that an adaptation algorithm can have a bird's eye view of the physical network, just as it has a bird's eye view of the application topology via VTTIF. This new work is described for the first time here.
4. Answer queries about the bandwidth and latency between any pair of machines in the virtual network. This is described for the first time here.

In the following, we begin by describing and evaluating the online Wren system (Section B.2) and how it interacts with the Virtuoso system (Section B.3). In Section B.4, we describe adaptation algorithms in Virtuoso that make use of Wren's view of the physical network. We present results showing Wren calculating available bandwidth using Virtuoso's run-time communication and present simulations of our adaptation algorithms in response to that information. Our results are promising, and we are currently integrating these components to support run-time adaptation.

B.2 Wren Online

The Wren architecture is shown in Figure B.1. The key feature Wren uses is kernel-level packet trace collection. These traces allow precise timestamps of the arrival and departure of packets

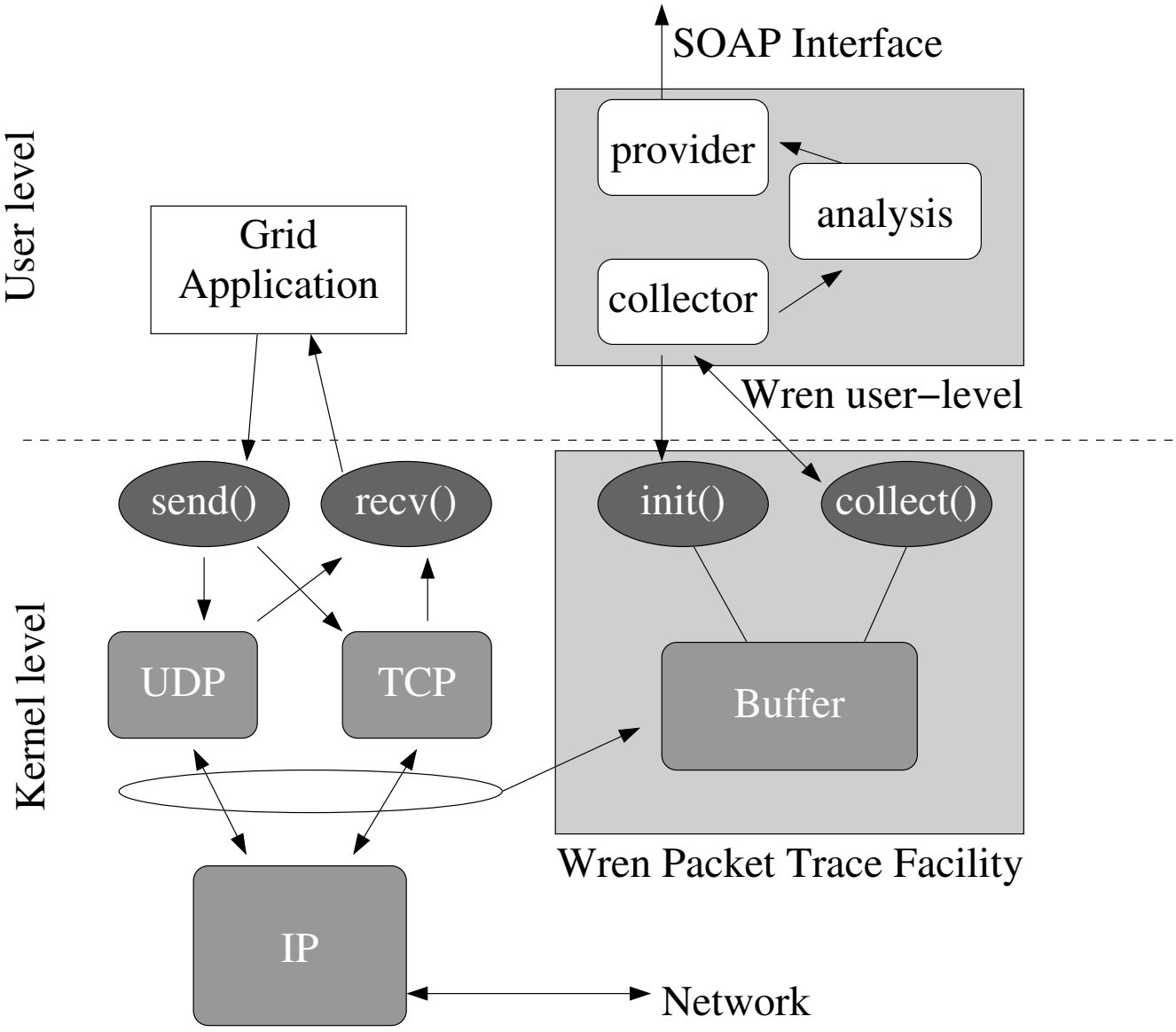


Figure B.1: Wren architecture.

on the machines. The precision of the timestamps is crucial because our passive available bandwidth algorithm relies on observing the behavior of small groups of packets on the network. A user-level component collects the traces from the kernel. Run-time analysis determines available bandwidth and the measurements are reported to other applications through a SOAP interface. Alternatively, the packet traces can be filtered for useful observations and transmitted to a remote repository for analysis.

Because we are targeting applications with potentially bursty and irregular communication patterns, many applications will not generate enough traffic to saturate the network and provide useful information on the current bandwidth achievable on the network. The key observation behind Wren is that *even when the application is not saturating the network, it is sending bursts of traffic that can be used to measure the available bandwidth of the network.*

The analysis algorithm used by Wren is based on the self-induced congestion (SIC) algorithm [113, 117]. Active implementations of this algorithm generate trains of packets at progressively faster rates until increases in one-way delay are observed, indicating queues building along the path resulting from the available bandwidth being consumed. We apply similar analysis to our passively collected traces, but our key challenge is identifying appropriate trains from the stream of packets generated by the TCP sending algorithm. ImTCP integrates an active SIC algorithm into a TCP stack, waiting until the congestion window has opened large enough to send an appropriate length train and then delaying packet transmissions until enough packets are queued to generate a precisely spaced train [102]. Wren avoids modifying the TCP sending algorithm, and in particular delaying packet transmission.

The challenge Wren addresses compared to ImTCP and other active available bandwidth tools is that Wren must select from the data naturally available in the TCP flow. Although Wren has less control over the trains and selects shorter trains than would deliberately be generated by active probing, over time the burstiness of the TCP process produces many trains at a variety of rates [76, 125], thus allowing bandwidth measurements to be made.

B.2.1 Online Analysis

Wren's general approach, collection overhead, and available bandwidth algorithm have been presented and analyzed in previous papers [155, 156]. Wren has negligible effect on throughput, latency, or CPU consumption when collecting packet header traces. To support Virtuoso's adaptation, however, two changes are required. First, previous implementations of Wren relied on offline analysis. We describe here our online analysis algorithm used to report available bandwidth measurements using our SOAP interface. Second, Wren has previously used fixed-size bursts of network traffic. The new online tool scans for maximum-sized trains that can be formed using the collected traffic. This approach results in more measurements taken from less traffic.

The online Wren groups outgoing packets into trains by identifying sequences of packets with similar interdeparture times between successive pairs. The tool searches for maximal-length trains with consistently spaced packets and calculates the initial sending rate (ISR) for those trains. After identifying a train, we calculate the ACK return rate for the matching ACKs. The available bandwidth is determined by observing the ISR at which the ACKs show an increasing trend in the RTTs, indicating congestion on the path. We have previously described this algorithm in more detail [156].

All available bandwidth observations are passed to the Wren observation thread. The observation thread provides a SOAP interface that clients can use to receive the stream of measurements produced using application traffic. Because the trains are short and represent only a singleton observation of an inherently bursty process, multiple observations are required to converge to an accurate measurement of available bandwidth.

B.2.2 Performance

We evaluated our new variable train-length algorithm in a controlled-load/controlled latency testbed environment because validating measurements on real WANs is difficult due to the lack

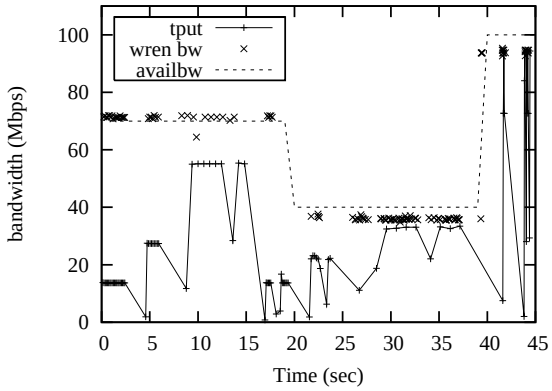


Figure B.2: Wren measurements reflect changes in available bandwidth even when the monitored application's throughput does not consume all of the available bandwidth.

of access to router information in the WAN. For this experiment, iperf generated uniform CBR cross traffic to regulate the available bandwidth, changing at 20 seconds and stopping at 40 seconds, as shown by the dashed line of Figure B.2.

We monitored application traffic that sent 20 200KB messages with .1 second inter-message spacings, paused 2 seconds, 10 500KB messages with .1 second inter-message spacings, paused 2 seconds, and then sent 10 4MB messages with .1 second inter-message spacings. This pattern was repeated twice followed by 500KB messages sent with random inter-message spacings. The throughput achieved is shown by the solid line of Figure B.2.

In the first 40 seconds of Figure B.2, we see that the throughput of the traffic generator varies according to the size of message being sent. The last 5 seconds of this graph show that the throughput of the generator also depends on the inter-message spacings. Figure B.2 shows that our algorithm produces accurate available bandwidth measurements even when the throughput of the application we are monitoring is not saturating the available bandwidth, as seen particularly well at the beginning and 20 seconds into the trace. The reported available bandwidth includes that consumed by the application traffic used for the measurement.

In our next experiment, we simulated a WAN environment using Nistnet to increase the latencies that the cross traffic and monitored application traffic experienced on our testbed. We

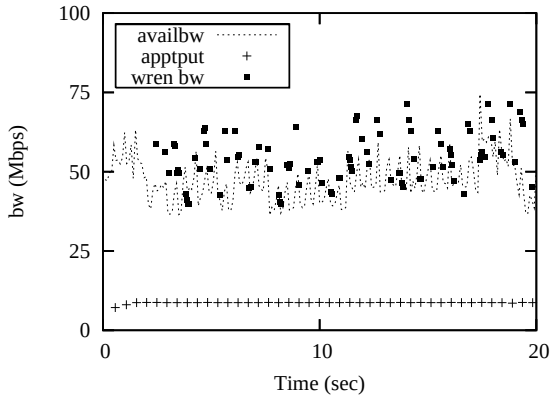


Figure B.3: Wren measurements from monitoring application on simulated WAN accurately detect changes in available bandwidth. The cross traffic in the testbed is created by on/off TCP generators.

used on/off TCP traffic generators to create congestion on the path, with Nistnet emulating latencies ranging from 20 to 100ms and bandwidths from 3 to 25Mbps for the TCP traffic generators. The application traffic that was monitored sent 700K messages with .1 second inter-message spacing, with Nistnet adding a 50ms RTT to that path. SNMP was used to poll the congested link to measure the actual available bandwidth. Figure B.3 demonstrates how the Wren algorithm can measure the available bandwidth of larger latency paths with variable cross traffic.

We have shown that our online Wren can accurately measure available bandwidth by monitoring application traffic that does not consume all of the available bandwidth. Furthermore, Wren can be used to monitor available bandwidth on low latency LANs or high latency WANs.

B.2.3 Monitoring VNET Application Traffic

To validate the combination of Wren monitoring an application using VNET we ran a simple BSP-style communication pattern generator. Figure B.4 shows the results of this experiment, with the throughput achieved by the application during its bursty communication phase and Wren's available bandwidth observations. Although the application never achieved significant

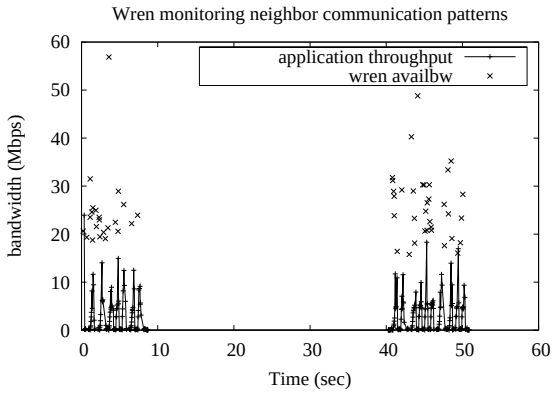


Figure B.4: Wren observing a neighbor communication pattern sending 200K messages within VNET.

levels of throughput, Wren was able to measure the available bandwidth. Validating these results across a WAN is difficult, but iperf achieved approximately 24Mbps throughput when run following this experiment, which is in line with our expectations based on Wren's observations and the large number of connections sharing W&M's 150Mbps Abilene connection.

B.3 Virtuoso and Wren

Virtuoso [43, 126], is a system for virtual machine distributed computing where the virtual machines are interconnected with VNET, a virtual overlay network. The VTTIF (virtual traffic and topology inference framework) component observes every packet sent by a VM and infers from this traffic a global communication topology and traffic load matrix among a collection of VMs. Wren uses the traffic generated by VNET to monitor the underlying network and makes its measurements available to Virtuoso's adaptation framework, as seen in Figure B.5.

B.3.1 VNET

VNET [134, 135] is the part of Virtuoso that creates and maintains the networking illusion that the user's virtual machines (VMs) are on the user's local area network. Each physical machine

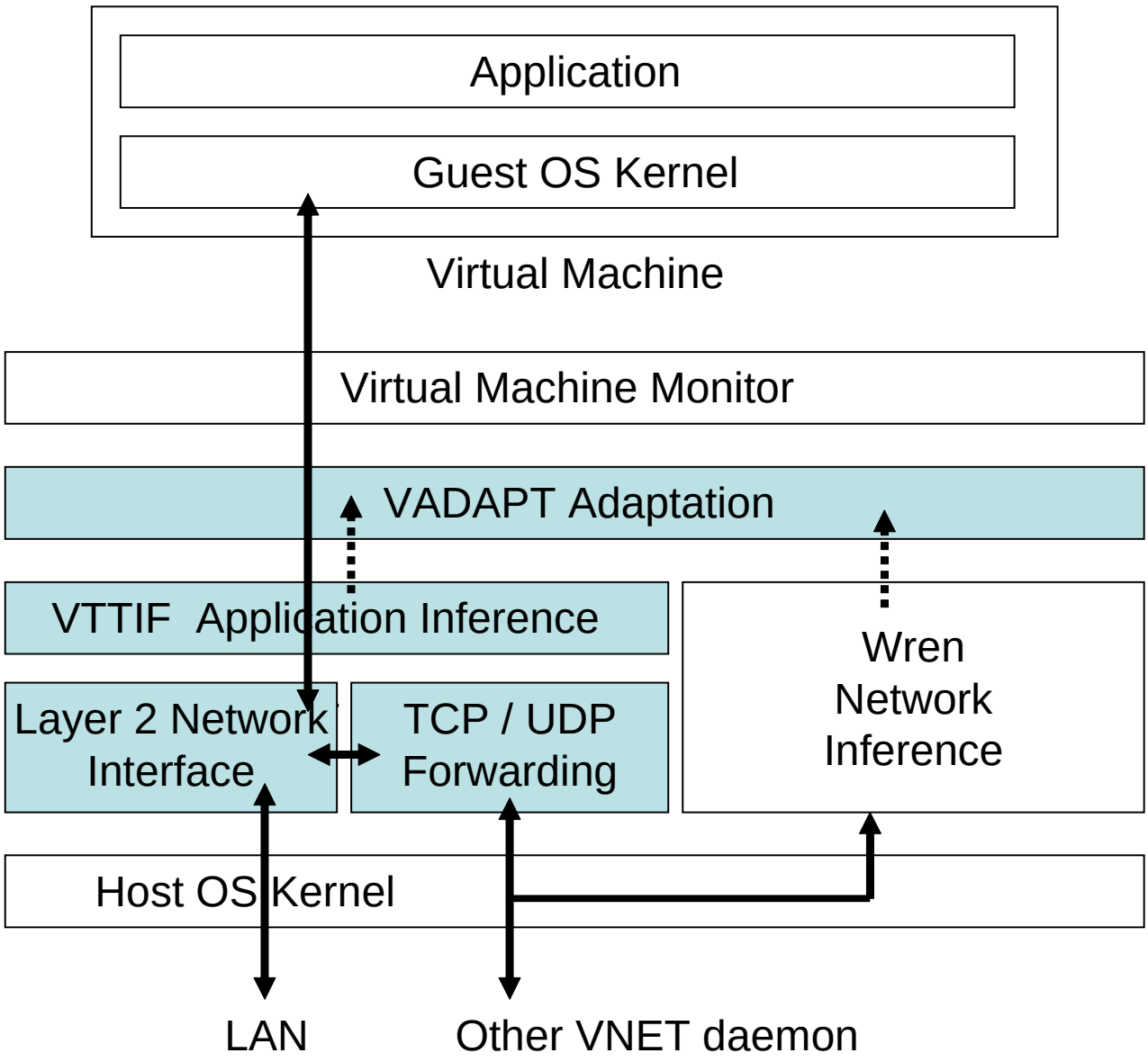


Figure B.5: Virtuoso's interaction with Wren. The highlighted boxes are components of Virtuoso.

that can instantiate virtual machines (a host) runs a single VNET daemon. One machine on the user's network also runs a VNET daemon. This machine is referred to as the Proxy. Each of the VNET daemons is connected by a TCP or a virtual UDP connection (a VNET link) to the VNET daemon running on the Proxy. This is the initial star topology that is always maintained. Additional links and forwarding rules can be added or removed at any time to improve application performance.

The VNET daemon running on a machine opens the machine's virtual (i.e., VMM-provided attachments to the VMs' interfaces) and physical Ethernet interfaces in promiscuous mode. Each packet captured from an interface or received on a link is matched against a forwarding table to determine where to send it, the possible choices being sending it over one of the daemon's outgoing links or writing it out to one of the local interfaces. Each successfully matched packet is also passed to VTTIF to determine the local traffic matrix. Each VNET daemon periodically sends its inferred local traffic matrix to the VNET daemon on the Proxy. The Proxy, through its physical interface, provides a network presence for all the VMs on the user's LAN and makes their configuration a responsibility of the user and his site administrator.

B.3.2 VTTIF

The VTTIF component integrates with VNET to automatically infer the dynamic topology and traffic load of applications running inside the VMs in the Virtuoso system. In our earlier work [58], we demonstrated that it is possible to successfully infer the behavior of a BSP application by observing the low level traffic sent and received by each VM in which it is running. We have also shown [135] how to smooth VTTIF's reactions so that adaptation decisions made on its output cannot lead to oscillation. The reaction time of VTTIF depends on the rate of updates from the individual VNET daemons and on configuration parameters. Beyond this rate, we have designed VTTIF to stop reacting, settling into a topology that is a union of all the topologies that are unfolding in the network.

VTTIF works by examining each Ethernet packet that a VNET daemon receives from a local VM. VNET daemons collectively aggregate this information producing a global traffic matrix for all the VMs in the system. To provide a stable view of dynamic changes, it applies a low pass filter to the updates, aggregating the updates over a sliding window and basing its decisions upon this aggregated view. The application topology is then recovered from this matrix by applying normalization and pruning techniques.

Since the monitoring is done below the VM, it does not depend on the application or the operating system in any manner. VTTIF automatically reacts to interesting changes in traffic patterns and reports them, driving adaptation.

B.3.3 Integrating Virtuoso and Wren

Virtuoso and Wren are integrated by incorporating the Wren extensions into the Host operating system of the machines running VNET. In this position, Wren monitors the traffic between VNET daemons, not between individual VMs. Both the VMs and VNET are oblivious to this monitoring, except for a negligible performance degradation.

The local instance of Wren is made visible to Virtuoso through its SOAP interface. VTTIF executes nonblocking calls to Wren to collect updates on available bandwidth and latency from the local host to other VNET hosts. VTTIF uses VNET to periodically send the local matrices to the Proxy machine, which maintains global matrices with information about every pair of VNET hosts. In practice, only those pairs whose VNET daemons exchange messages have entries. Through these mechanisms, the Proxy has a view of the physical network interconnecting the machines running VNET daemons and a view of the application topology and traffic load of the VMs.

B.3.4 Overheads

The overheads of integrating Wren with Virtuoso stem from the extra kernel-level Wren processing each VNET transmission sees, Wren user-level processing of data into bandwidth and latency estimates, and the cost of using VNET and VTTIF to aggregate local Wren information into a global view. Of these, only the first is in the critical path of application performance. The Wren kernel-level processing has no distinguishable effect on either throughput or latency [155]. With VTTIF, latency is unaffected, while throughput is affected by $\sim$1%. The cost of local processing is tiny and can be delayed.

B.4 Adaptation Using Network Information

As shown in Figure B.5, the VADAPT component of Virtuoso, using the VTTIF and Wren mechanisms, has a view of the dynamic performance characteristics of the physical network interconnecting the machines running VNET daemons and a view of the the demands that the VMs place on it. More specifically, it receives:

1. A graph representing the application topology of the VMs and a traffic load matrix among them, and
2. Matrices representing the available bandwidth and latency among the Hosts running VNET daemons.

VADAPT's goal is to use this information to choose a configuration that maximizes the performance of the application running inside the VMs. A configuration consists of

1. The mapping of VMs to Hosts running VNET daemons,
2. The topology of the VNET overlay network,
3. The forwarding rules on that topology, and
4. The choice of resource reservations on the network and the hosts, if available.

In previous work [135, 139], we have demonstrated heuristic solutions to a subset of the above problem. In particular, we have manipulated the configuration (sans reservations) in response to application information. In the following, we expand this work in two ways. First,

we show how to incorporate the information about the physical network in a formal manner. Second, we describe two approaches for addressing the formal problem and present an initial evaluation of them.

B.4.1 Problem formulation

VNET Topology: We are given a complete directed graph $G = (H, E)$ in which H is the set of all of the Hosts that are running VNET daemons and can host VMs.

VNET Links: Each edge $e = (i, j) \in E$ is a prospective link between VNET daemons. e has a real-valued capacity c_e which is the bandwidth that the edge can carry in that direction. This is the available bandwidth between two Hosts (the ones running daemons i and j) reported by Wren.

VNET Paths: A path, $p(i, j)$, between two VNET daemons $i, j \in H$ is defined as an ordered collection of links in E, $\langle (i, v_1), (v_1, v_2), ..., (v_n, j) \rangle$, which are the set of VNET links traversed to get from VNET daemon i to j given the current forwarding rules and topology, $v_1, \ldots v_n \in H$. P is the set of all paths.

VM Mapping: V is the set of VMs in the system, while M is a function mapping VMs to daemons. $M(k) = l$ if VM $k \in V$ is mapped to Host $l \in H$.

VM Connectivity: We are also given a set of ordered 3-tuples $A = (S, D, C)$. Any tuple, $A(s_i, d_i, c_i)$, corresponds to an entry in the traffic load matrix supplied by VTTIF. More specifically, if there are two VMs, $k, m \in V$, where $M(k) = s_i$ and $M(m) = d_i$, then c_i is the traffic matrix entry for the flow from VM k to VM m.

Configurations: A configuration $CONF = (M, P)$ consists of the VM to VNET daemon mapping function M and the set of paths P among the VNET daemons needed to assure the connectivity of the VMs. The topology and forwarding rules for the daemons follow from the set of paths.

Residual Capacity of a VNET Link: Each tuple, A_i, can be mapped to one of multiple paths, $p(s_i, d_i)$. Once a configuration has been determined, each VNET link $e \in E$ has a real-valued residual capacity rc_e which is the bandwidth remaining unused on that edge.

Bottleneck Bandwidth of a VNET Path: For each mapped paths $p(s_i, d_i)$ we define its bottleneck bandwidth, $b(p(s_i, d_i))$, as $(min(cr_e)).\forall e \in p(s_i, d_i)$.

Optimization Problem: We want to choose a configuration *CONF* which maps every VM in V to a VNET daemon, and every input tuple A_i to a network path $p(s_i, d_i)$ such that the total bottleneck capacity on the VNET graph,

$$\sum_{p \in P} b(p(s_i, d_i)) \tag{B.1}$$

is maximized or minimized subject to the constraint that

$$\forall e \in E : rc_e \geq 0 \tag{B.2}$$

The intuition behind maximizing the residual bottleneck capacity is to leave the most room for the application to increase performance within the current configuration. Conversely, the intuition for minimizing the residual bottleneck capacity is to increase room for other applications to enter the system. This problem is NP-complete by reduction from the edge disjoint path problem [137].

B.4.2 Greedy Heuristic Solution

In an online system of any scale, we are unlikely to be able to enumerate all possible configurations to choose a good one. Our first approach is necessarily heuristic and is based on a greedy strategy with two sequential steps: (1) find a mapping from VMs to Hosts, and (2) determine paths for each pair of communicating VMs.

Mapping VMs to Hosts

VADAPT uses a greedy heuristic algorithm to map virtual machines onto physical hosts. The input to the algorithm is the application communication behavior as captured by VTTIF and available bandwidth between each pair of VNET daemons, as reported by Wren, expressed as adjacency lists. The algorithm is as follows:

1. Generate a new VM adjacency list which represents the traffic intensity between VNET daemons that is implied by the VTTIF list and the current mapping of VMs to hosts.
2. Order the VM adjacency list by decreasing traffic intensity.
3. Extract an ordered list of VMs from the above with a breadth first approach, eliminating duplicates.
4. For each pair of VNET daemons, find the maximum bottleneck bandwidth (the widest path) using the adapted Dijkstra's algorithm described in Section B.4.2.
5. Order the VNET daemon adjacency list by decreasing bottleneck bandwidth.
6. Extract an ordered list of VNET daemons from the above with a breadth first approach, eliminating duplicates.
7. Map the VMs to VNET daemons in order using the ordered list of VMs and VNET daemons obtained above.
8. Compute the differences between the current mapping and the new mapping and issue migration instructions to achieve the new mapping.

Mapping Communicating VMs to Paths

Once the VM to Host mapping has been determined, VADAPT uses a greedy heuristic algorithm to determine a path for each pair of communicating VMs. The VNET links and forwarding rules derive from the paths. As above VADAPT uses VTTIF and Wren outputs expressed as adjacency lists as inputs. The algorithm is as follows:

1. Order the set *A* of VM to VM communication demands in descending order of communication intensity (VTTIF traffic matrix entry).
2. Consider each 3-tuple in the ordered set *A*, making a greedy mapping of it onto a path. The mapping is on the current residual capacity graph *G* and uses an adapted version of Dijkstra's algorithm described in Section B.4.2. No backtracking is done at this stage.

Adapted Dijkstra's algorithm

We use a modified version of Dijkstra's algorithm to select a path for each 3-tuple that has the maximum bottleneck bandwidth. This is the "select widest" approach. Notice that as there is no backtracking, it is quite possible to reach a point where it is impossible to map a 3-tuple at all. Furthermore, even if all 3-tuples can be mapped, the configuration may not minimize/maximize Equation B.1 as the greedy mapping for each 3-tuple doesn't guarantee a global optimum.

Dijkstra's algorithm solves the single-source shortest paths problem on a weighted, directed graph $G = (H, E)$. Our algorithm solves the single-source widest paths problem on a weighted directed graph $G = (H, E)$ with a weight function $c : E \rightarrow \mathbb{R}$ which is the available bandwidth in our case. The full algorithm description and a proof of correctness is available in a technical report [60].

B.4.3 Simulated Annealing Solution

Simulated annealing [83] (SA) is a probabilistic evolutionary method that is well suited to solving global optimization problems, especially if a good heuristic is not known. SA's ability to locate a good, although perhaps non-optimal solution for a given objective function in the face of a large search space is well suited to our problem. Since the physical layer and VNET layer graphs in our system are fully connected there are a great many possible forwarding paths and mappings. Additionally, as SA incrementally improves its solution with time, there is some solution available at all times.

The basic approach is to start with some initial solution to the problem computed using some simple heuristic such as the adapted Dijkstra based heuristic described above. SA iterations then attempt to find better solutions by perturbing the current solution and evaluating its quality using a cost function. At any iteration, the system state is the set of prospective solutions. The random perturbations of the SA algorithm make it possible to explore a diverse range of the search space including points that may appear sub-optimal or even worse then previous options but may lead

to better solutions later on. The probability of choosing options that are worse than those in the present iteration is reduced as the iterations proceed, focusing increasingly on finding better solutions close to those in the current iteration. The full algorithm is available in our technical report [60]. The problem-specific elements of our application of SA, the perturbation function and the cost evaluation function are described below.

Perturbation Function

The role of the perturbation function (*PF*) is to find neighbors of the current state that are then chosen according to a probability function $P(dE,T)$ of the energy difference $dE = E(s') - E(s)$ between the two states, and of a global time-varying parameter T (the temperature). The probability function we use is $e^{dE/T}$ if dE is negative, 1 otherwise. As iterations proceed T is decreased which reduces the probability of jumping into states that are worse than the current state.

Given a configuration $CONF = (M,P)$, where P is a set of forwarding paths $p(i,j)$ and each $p(i,j)$ is a sequence of $k_{i,j}$ vertices $v_i, v_1, v_2, ..., v_j$, the perturbation function selects a neighbor $N(CONF)$ of the current configuration with the following probabilities: For each $p(i,j) \in P$:

1. With probability $1/3$ *PF* adds a random vertex v_r into the path sequence where $v_r \in V$ and $\notin p(i,j)$. Note that the set V consists of all potential physical nodes which are running VNET and hence are capable of routing any VNET traffic. This step attempts to modify each path by randomly adding a potential overlay node in the existing forwarding path.
2. With probability $1/3$ *PF* deletes a random vertex v_r from the path sequence where $v_r \in p(i,j)$.
3. With probability $1/3$ *PF* swaps two nodes v_x and v_y where $x \neq y$ and $v_x, v_y \in p(i,j)$.

On a typical iteration, our algorithm only perturbs the current forwarding paths. To also explore new mappings of the VMs to different VNET hosts, we also perturb that mapping. However, as perturbing a mapping effectively resets the forwarding paths, we perturb the mappings with a lower probability.

Cost Evaluation Function

The cost evaluation function *CEF* computes the cost of a configuration C using Equation B.1. After a neighbor $N(C)$ is found using the perturbation function, a cost difference $CEF(N(C)) - CEF(C)$ is computed. This is the energy difference used to compute the future path in the simulated annealing approach using a probability $e^{(CEF(N(C))-CEF(C))/t}$ if the difference is negative, 1 otherwise. As iterations proceed and temperature decreases, the SA algorithm finally converges to the best state it encounters in its search space.

B.4.4 Performance

Because we have not yet coupled the entire real-time toolchain, our evaluation is done in simulation, using Wren measurements collected from observing VNET data to the extent possible. We also evaluate our algorithms by posing a challenging adaptation problem, and evaluate their scalability using a large-scale problem. In each scenario the goal is to generate a configuration consisting of VM to Host mappings and paths between the communicating VMs that maximizes the total residual bottleneck bandwidth (Section B.4.1). We compare the greedy heuristic (GH), simulated annealing approach (SA) and simulated annealing with the greedy heuristic solution as the starting point (SA+GH). In addition at all points in time we also maintain the best solution found so far with (SA+GH), we call this (SA+GH+B), where 'B' indicates the best solution so far. The W&M and NWU setup had a solution space small enough to enumerate all possible configurations to find the optimal solution.

Wren Measurements for Testbed

We have created a testbed of Wren-enabled machines: two at William and Mary and two at Northwestern as shown in Figure B.6. We have successfully run VNET on top of Wren on these systems with Wren using VM traffic to characterize the network connectivity, as shown in Figure B.4. At the same time Wren provides its available bandwidth matrix, VTTIF provides

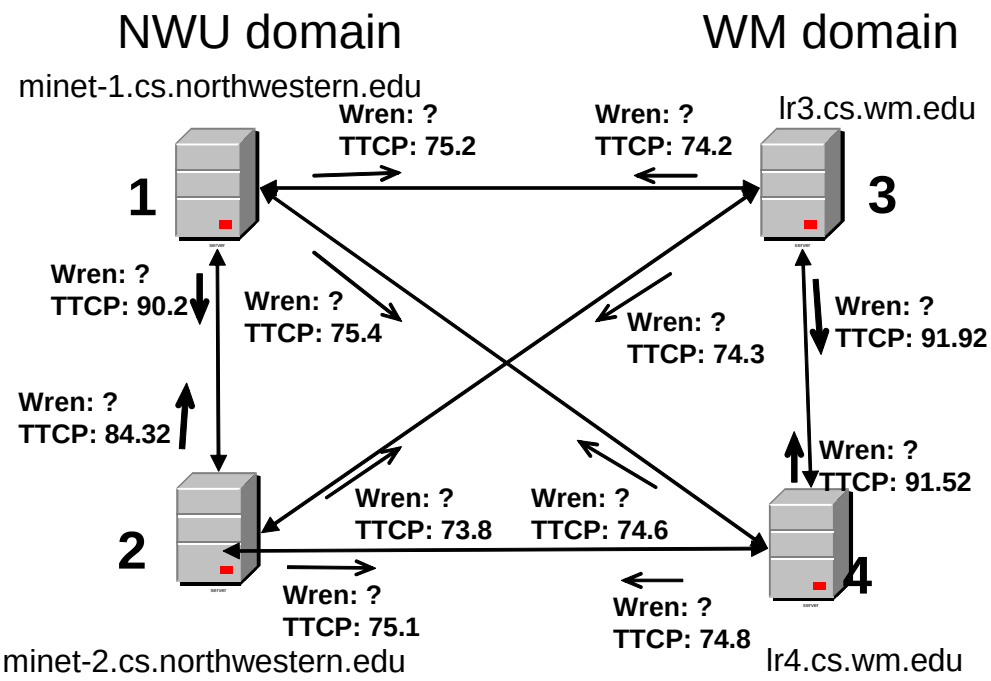


Figure B.6: Northwestern / William and Mary testbed. Numbers are Mb/sec.

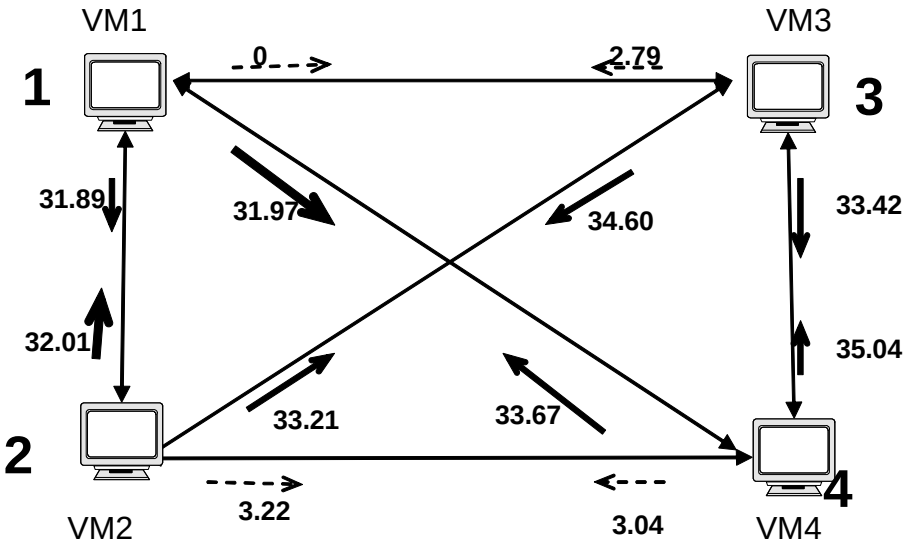


Figure B.7: VTTIF topology inferred from NAS MultiGrid Benchmark. Numbers are Mb/sec.

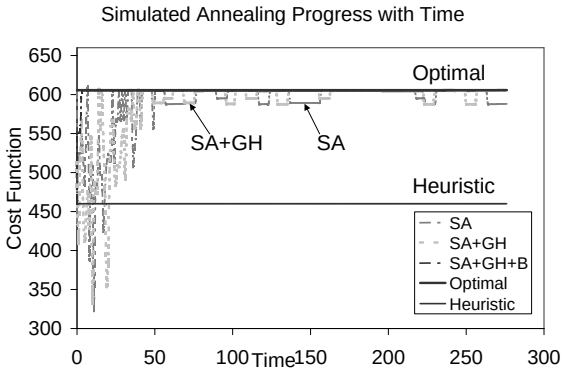


Figure B.8: Adaptation performance while mapping 4 VM all-to-all application onto NWU / W&M testbed.

the (correct) application topology matrix. The full Wren matrix is used in Section B.4.4.

Adaptation in W&M and NWU Testbed

We evaluated our adaptation algorithms for an application running inside of VMs hosted on the W&M and NWU testbed in simulation. The VMs were running the NAS MultiGrid benchmark. Figure B.7 shows the application topology inferred by VTTIF for a 4 VM NAS MultiGrid benchmark. The thickness of the arrows are directly proportional to the bandwidth demand in that direction.

Figure B.8 shows the performance of our algorithms as a function of time. The two flat lines indicate the heuristic (GH) performance and the optimal cost of the objective function (evaluated by hand). Since the solution space is small with 12 possibilities for the VM to VNET mapping, we were able to enumerate all possible configurations and thus determine the optimal solution. The optimal mapping is $VM1 \rightarrow 2$, $VM2 \rightarrow 4$, $VM3 \rightarrow 3$, $VM4 \rightarrow 1$ with an optimal CEF value of 605.66.

There is a curve for the simulated annealing algorithm, SA+GH (annealing algorithm starting with heuristic as the initial point) and the best cost reached so far, showing their values over time. We see that the convergence rate of SA is crucial to obtaining a good solution quickly.

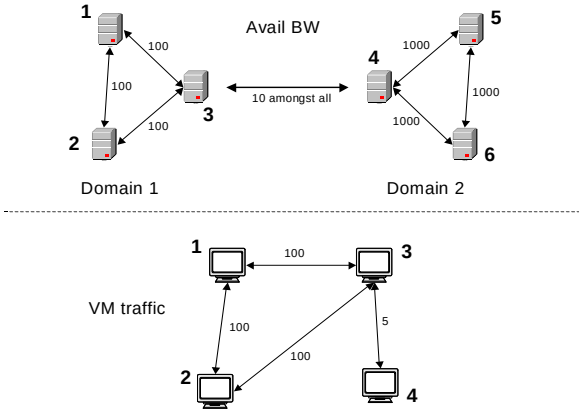
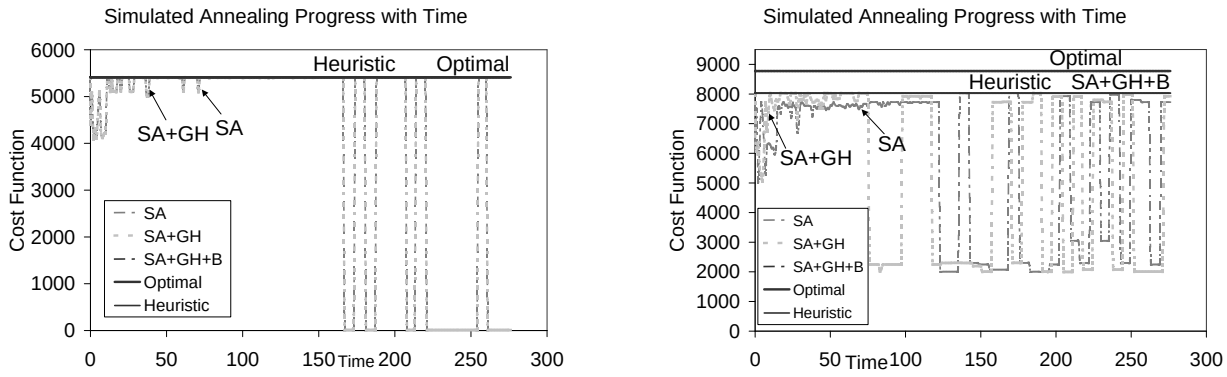


Figure B.9: A challenging scenario that requires a specific VM to Host mapping for good performance.

Notice that SA is able to find close to optimal solutions in a reasonably short time, while GH completes almost instantaneously, but is not able to find a good solution. SA+GH performs slightly better than SA. Note that the graph shows, for each iteration, the best value of the objective function of that iteration. SA+GH+B shows the best solution of all the iterations up to the present one by SA+GH.

Challenge

We also designed a challenging scenario, illustrated in Figure B.9, to test our adaptation algorithms. The VNET node topology consists of two clusters of three machines each. The domain 1 cluster has 100 Mbps links interconnecting the machines, while domain 2 cluster has 1000 Mbps links. The available bandwidth on the link connecting the two domains is 10 Mbps. This scenario is similar to a setup consisting of two tightly coupled clusters connected to each other via WAN. The lower part of the figure shows the VM configuration. VMs 1, 2 and 3 communicate with a much higher bandwidth as compared to VM 4. An optimal solution for this would be to place VMs 1, 2 and 3 on the three VNET nodes in domain 2 and place VM 4 on a VNET node in domain 1. The final mapping reported by GH is $VM1 \rightarrow 5$, $VM2 \rightarrow 4$, $VM3 \rightarrow 6$, $VM4 \rightarrow 1$. The final mapping reported by SA+GH is $VM1 \rightarrow 4$, $VM2 \rightarrow 5$, $VM3 \rightarrow 6$, $VM4 \rightarrow 1$. Both are



(a) Residual BW Only

(b) Residual BW and Latency

Figure B.10: Adaptation performance while mapping 6 VM all-to-all in the challenging scenario.

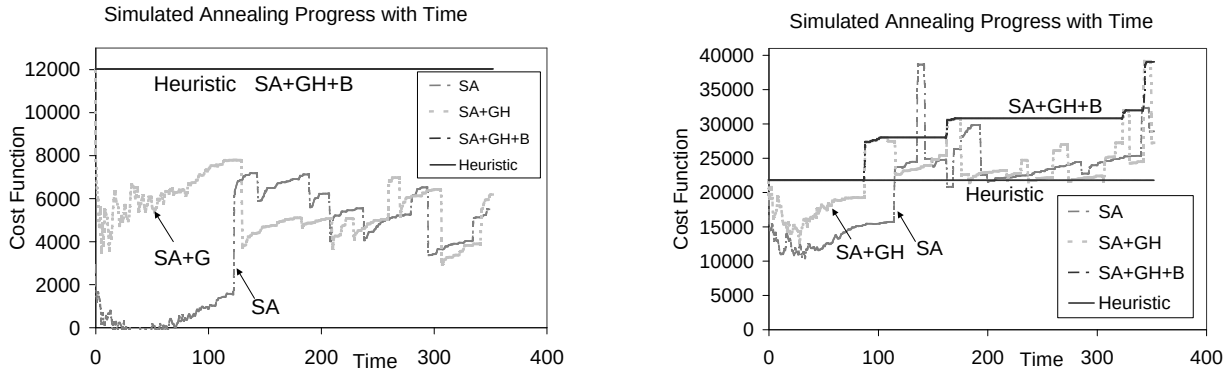
optimal for the metric described before with a final CEF value of 5410.

For this scenario, both GH and SA are able to find the optimal mappings quickly. Figure B.10(a) illustrates the performance of our adaptation algorithms. The physical and application topologies have been constructed so that only one valid solution exists. We see that GH, SA, and SA+GH all find the optimal solution quickly with very small difference in their performance. The large fluctuations in the objective function value for SA curves is due to the occasional perturbation of VM to Host mapping. If a mapping is highly sub-optimal, the objective function value drops sharply and remains such until a better mapping is chosen again.

Multi-constraint Optimization: In this scenario, we also use the annealing algorithm to perform multi-constrained optimization. In Figure B.10(b), we show the performance of our algorithms with an objective function that takes into account both bandwidth and latency. Specifically, we have changed Equation B.1 to be

$$\sum_{p \in P} b(p(s_i, d_i)) + \frac{c}{l(p(s_i, d_i))} \tag{B.3}$$

where $l(p)$ is the path latency for path p and c is a constant. This penalizes the paths with



(a) Residual BW Only

(b) Residual BW and Latency

Figure B.11: Adaptation performance while mapping 8 VM all-to-all to 32 hosts on a 256 node network.

large latencies. We see that SA and SA+GH find better solutions than GH. GH provides a good starting point for SA which further explores the search space to improve the solution based on the defined multi-constraint objective.

Large topology

To study scalability of our adaptation algorithms we generated a 256 node BRITE [106] physical topology. The BRITE topology was generated using the Waxman Flat-router model with a uniformly varying bandwidth from 10 to 1024 units. Each node has an out-degree of 2. In this topology, we chose 32 hosts at random to run VNET daemons, hence each is a potential VM host.

A VNET link is a path in the underlying BRITE physical topology. We calculated the bandwidths for the VNET overlay links as the bottleneck bandwidths of the paths in the underlying BRITE topology connecting the end points of the VNET link.

Figure B.11 shows the performance of our algorithms adapting a 8 VM application communicating with a ring topology to the available network resources. It illustrates the point that the simple greedy heuristic is more suited to smaller scenarios, while simulated annealing is

best used when the quality of the solution is most important and more time to find a solution is available.

GH completes very quickly and produces a solution that we unfortunately cannot compare with optimiality since enumerating solutions is intractable. Simulated annealing on the other hand takes much longer, but produces a much better result in the end. Figure B.11 shows the scenario for the time during which the SA solution is inferior to the GH solution. However, given more time the SA solution meets and exceeds the GH solution.

Figure B.11(b) shows the performance using the combined bandwidth/latency objective function of Equation B.3. Here, in the allotted time, SA's performance greatly exceeds that of GH. This is not particularly surprising as GH does not consider latency at all. However, the point is that it is trivial to change the objective function in SA compared to in ad hoc heuristic techniques. Simulated annealing is very effective in finding good solutions for larger scale problems, and for different complex objective functions.

B.5 Conclusions

We have described how the Virtuoso and Wren systems may be integrated to provide a virtual execution environment that simplifies application portability while providing the application and resource measurements required for transparent optimization of application performance. We have described extensions to the Wren passive monitoring system that support online available bandwidth measurement and export the results of those measurements via a SOAP interface. Our results indicate that this system has low overhead and produces available bandwidth observations while monitoring bursty VNET traffic. VADAPT, the adaptation component of Virtuoso uses this information provided by Wren along with application characteristics provided by VTTIF to dynamically configure the application, maximizing its performance. We formalized the adaptation problem, and compared two heuristic algorithms as solutions to this NP-hard problem. We found the greedy heuristic to perform as well or better than the simu-

lated annealing approach, however, if the heuristic was taken as the starting point for simulated annealing it performed much better than the greedy heuristic.