Workload Characterization of Nondeterministic Programs Parallelized by STATS

Enrico A. Deiana
Northwestern University
ead@u.northwestern.edu

Simone Campanoni
Northwestern University
simonec@eecs.northwestern.edu

***Abstract*—Chip Multiprocessors (CMP) are everywhere, from mobile systems, to servers. Thread Level Parallelism (TLP) is the characteristic of a program that makes use of the parallel cores of a CMP to generate performance. Despite all efforts for creating TLP, multiple cores are still underutilized even though we have been in the multicore era for more than a decade. Recently, a new approach called STATS has been proposed to generate additional TLP for complex and irregular nondeterministic programs. STATS allows a developer to describe application-specific information that its compiler uses to automatically generate a new source of TLP. This new source of TLP increases with the size of the input and it has the potential to generate scalable performance with the number of cores. Even though STATS obtains most of its potential, some of it is still unreached. This paper identifies and characterizes the sources of overhead that are currently blocking STATS parallelized programs to achieve their full potential. To this end, we characterized the workloads generated by the STATS compiler on a 28 core Intel-based machine (dual-socket). This paper shows that the performance loss is due to a combination of factors: some can be optimized via engineering efforts and some require a deeper evolution of STATS. We also highlight potential solutions to significantly reduce most of this overhead. Exploiting these insights will unblock scalable performance for the parallel binaries generated by STATS.**

***Keywords*-nondeterminism; parallelizing compiler; speculation; workload characterization;**

I. Introduction

The thread-level parallelism (TLP) of a program defines the performance and the energy efficiency of a commodity system in this multicore era. Unfortunately, typical programs suffer from low—and non-scalable—TLP due to data exchanges between threads. These data exchanges are called *actual dependences*. This is in contrast with *apparent dependences*, which usually exist (and require synchronization) because a developer (or a tool) could not prove that data motion is not actually necessary (e.g., the data generated by one thread is not actually read by the other). These two types of dependences are the main cause of low TLP in today's software, which leads to underutilized hardware cores in commodity platforms and, therefore, low performance.

To date, tremendous research effort has resulted in techniques, such as thread-level speculation, that alleviate the impact of apparent dependences [24], [29], [35], [37], [48], [49], [51], [53], [58], [60]. The results of this effort landed in some today's commodity processors, which now include transactional memories that are required by these techniques. This is not enough. Cores are still underutilized. To truly unleash the performance potential present in current (and even more so future) hardware, we need to address actual dependences as well.

Recently, a new compiler-based approach called STATS has identified a new opportunity to generate additional TLP from nondeterministic workloads while preserving the original program semantics [20]. To this end, STATS targets a subset of actual dependences called *state dependences*: non-transitive dependences that update the state of a computation based on the previous state. Hence, a state dependence forms a dependence chain constraining the TLP of its program. Often, an update related to a state dependence does not depend on its previous updates that are old enough. STATS exploits this short memory property by speculating on how many state-updates a given one depends on and it splits accordingly the dependence chain. The so-generated dependence chain segments are linked back together at run-time exploiting the variations of intermediate data measured while a nondeterministic program runs. The short memory property allows STATS to generate a new source of TLP that has the potential of scaling linearly with the amount of inputs (state updates) that a program needs to process.

Intuitively, the TLP generated by STATS should translate in performance that scales linearly with the number of cores when enough inputs are processed. However, the performance increase measured on commodity multicores is sub-linear. This paper characterizes the workloads generated by STATS to understand what are the limiting factors that are blocking STATS to fulfill its potential. We show that the performance loss is composed by a combination of factors. Some of them can be eliminated by relatively simple engineering efforts, others require a deeper evolution of the STATS approach.

After further describing STATS (Section II), we expose the sources of overhead that are blocking the generated parallel binaries to achieve linear performance scaling (Section III). Then, we show the empirical evaluations of these overhead sources on a 28 core, 2 sockets Intel-based platform (Section V). Finally, motivated by our evaluations, we conclude suggesting the most promising evolutions of STATS for obtaining its full potential.

II. The STATS Approach to Extract TLP

Modern nondeterministic workloads often possess dependences that serialize large portions of these programs. STATS

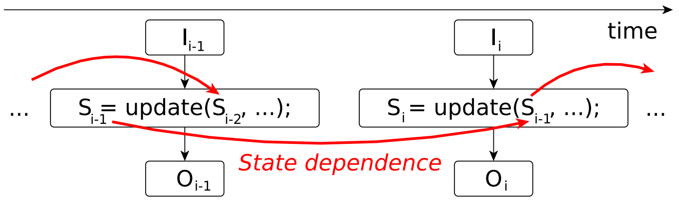


Fig. 1: Code pattern followed by a state data dependence.

performs a new compiler transformation that reorganizes these dependences to create additional TLP for these nondeterministic programs. This section describes what programs STATS targets, which properties it takes advantage of to perform its transformation (Section II-A), what is the execution model STATS follows to generate additional TLP (Section II-B), and how this execution model is enforced (Section II-C).

A. State Dependence

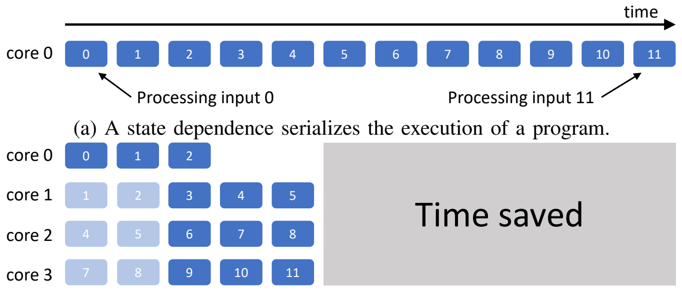
Program's dependences are either actual or apparent. Apparent dependences are those that a compiler (or developer) could not prove their non-existence, so the compiled program is forced to satisfy them. Dependences created by the conservativeness of alias analysis are examples of apparent dependences. Actual dependences, instead, are those that require actual data transfers while the program executes. Producer-consumer dependences are examples of actual dependences.

STATS focuses on a subset of actual dependences that are found in nondeterministic programs. These dependences are called *state dependences* because the data transfers that are necessary to satisfy them is related to the *state* of a computation [20].

A state dependence is a read-after-write actual dependence that updates the current state of the computation based on the previous computational state. Fig. 1 shows the pattern followed by a state dependence. The computation performed by the i^{th} call of `update()` takes as input the state $\mathtt{S}_{i-1}$ of the previous computation along with the input $\mathtt{I}_i$. This invocation produces an output $\mathtt{O}_i$ and updates the computational state to $\mathtt{S}_i$. The new state $\mathtt{S}_i$ is then transferred to the subsequent call of `update()`. These data transfers create the state dependence.

STATS allows the developer to explicitly expose state dependences. Its compiler automatically extracts a new source of TLP from the described state dependences. An example of a nondeterministic program that possess a state dependence is `bodytrack`. We will use this program as a driving example in the rest of this section to describe the properties of state dependences in a real workload as well as the related TLP that STATS generates from them.

`bodytrack` analyzes a stream of images [5]. For each image, `bodytrack` identifies the human bodies found in it and it draws the edges of their body parts (e.g., legs, arms, head) over the original image. `bodytrack` does not know the exact position of a body within an image. Therefore, the program takes multiple guesses on where the body could be in the space. To increase the accuracy of the algorithm, `bodytrack` exploits the fact that where the body is at a



(a) A state dependence serializes the execution of a program.

(b) STATS generates additional TLP that reduces the execution time.

Fig. 2: The STATS execution model.

given instant is likely to be close to where the body was an instant before. To this end, `bodytrack` takes random guesses on where the body is within an image $\mathtt{I}_i$ by distributing such guesses close to where the body was found in the previous image $\mathtt{I}_{i-1}$. This is performed every time there is a previous image to exploit (i.e., every image but the first one). Taking advantage of where the body was in the previous image is essential: the accuracy obtained without it is too low for practical uses. Finally, using the state (i.e., body part locations) of the computation of the previous image to process the current one creates a chain of read-after-write data dependences throughout the processing of the whole sequence of images given as input. This chain of actual dependences blocks the TLP of the original `bodytrack` and, therefore, its performance on multicore platforms. This is an example of state dependence.

While the location of a body depends on where it was the instant before, it does not depend on where the body was a long time ago. In other words, where the body is at image $\mathtt{I}_i$ does not depend on where it was in the image $\mathtt{I}_{i-k}$ with high k. STATS takes advantage of this short memory property to generate additional TLP by enforcing the parallel execution model described next.

B. The STATS Execution Model

STATS creates additional TLP for a nondeterministic program by exploiting the short memory property of state dependences. To do so, STATS enforces a parallel execution model in the generated binary that unconventionally satisfies such dependences.

Additional TLP generated by STATS. STATS divides the computation of the original program (serialized by a state dependence as shown in Fig. 2a) into non-overlapping chunks of computation. These chunks are distributed on different threads, which run on different cores (Fig. 2b). Except for the first chunk, which is the first part of the original program, the others need an initial state to start their computation. In the original code the initial state of an input chunk comes from the processing of the last input of its respective previous chunk. STATS takes advantage of the short memory property by creating an alternative producer of the initial state for each chunk. The computation performed by an alternative producer is shown in light colored boxes in Fig. 2b; the actual program computation, instead, is represented by dark colored boxes.

An alternative producer processes only k inputs ($k = 2$ in the example of Fig. 2b) prior to the first one of the related chunk, instead of processing all previous inputs like the original producer does. Therefore, the alternative producer is able to feed the initial state to its related chunk faster than the original producer could have done. Also, the chunks of computation are now independent from each other because they consume the state (e.g., where the body is in the image before the first one of the current chunk) generated by the alternative producer. This enable STATS to run these computation chunks in parallel generating additional TLP.

Preserving the original semantics. STATS preserves the original semantics of the program being compiled. To this end, the chunks of computation after the first one are considered speculative because it is unknown whether or not the state coming from an alternative producer (speculative state) is as good as the state that would have come from the original computation (original state). To understand whether or not these speculations can commit, STATS relies on its runtime, which commits or aborts speculations in the sequential order dictated by the original program semantics. To decide whether or not a speculation can commit, the STATS runtime compares the speculative state generated by the related alternative producer with the original state generated at the end of the previous chunk of computation. If the speculative state matches the original one, then the STATS runtime commits the current chunk of computation and move on to check the next one. If these states differ, then this difference can be due to either (i) the alternative producer did not process enough inputs prior to the current chunk of computation (i.e., the length of the short memory property was incorrectly estimated by STATS) or (ii) the data fluctuations generated by the nondeterminism of the original code. In the first case (i), the STATS runtime aborts the computation and runs it again starting from the original state generated by the previous chunk of computation (original producer). In the second case (ii), the STATS runtime commits the computation and move on to check the next one. To distinguish between these two cases, the STATS runtime processes the set of inputs prior to the current speculative chunk multiple times generating multiple original states. These original states differ because of the nondeterminism of the original algorithm. For example, in Fig. 2b the computation related to `input 1` and `input 2` of the first chunk that runs on `core 0` can be executed multiple times, leading to multiple different original states. Finally, the STATS runtime checks if the current speculative state (generated by the alternative producer, the light colored boxes in Fig. 2b) matches one of the original states. If no original state matches the speculative one, then we are in the first case (i) and the STATS runtime aborts and restarts the current computation. Otherwise, we are in the second case (ii) and the STATS runtime commits the current computation.

STATS design space. The number of inputs that alternative producers process to comply with the short memory property, the maximum number of extra original states that are generated by the STATS runtime, and the length of each computation chunk are parameters related to a state dependence. Independently on how these parameters are set, the original program semantics is preserved thanks to the checks performed by the STATS runtime. However, to increase the program's performance, these parameters need to be properly tuned to balance the likelihood of committing speculative computation with the extra computation required by the STATS execution model. The parameters create a design space that STATS automatically explores. This process is described next.

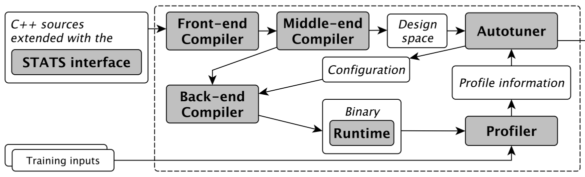


Fig. 3: STATS is composed by three compilers, a runtime, a profiler, and an autotuner. The autotuner is in charge to find the configuration that optimizes the program's performance.

C. The STATS System

The STATS system is composed by three compilers, an autotuner, and a runtime. This system enforces the previously described execution model using the compilation flow shown in Fig. 3. STATS takes as input the source code of a program, extended with the STATS interface. The STATS interface is a language extension that makes state dependences explicit to STATS compilers.

The front-end compiler translates the language extension to standard C++ code, which is then given to the middle-end compiler along with the program's sources. The middle-end compiler automatically generates alternative producer code and the design space. The design space is composed by all values that state dependence parameters can assume (e.g., number of inputs the alternative producer of a given state dependence will process).

The autotuner chooses a configuration in this design space, which is passed to the back-end compiler. The back-end compiler implements the given configuration in the program and it embeds the STATS runtime in the generated binary. This binary is then given to the profiler.

The profiler executes the binary using the developer provided training inputs and collects profiling information such as execution time and energy consumption of the program. These information are given back to the autotuner, which uses them to choose the next configuration. STATS keeps iterating the autotuner, back-end, profiler loop until it outputs the best binary that corresponds to the best seen configuration with respect to the objective that the autotuner is optimizing for (e.g., execution time).

III. Sources of Overhead

Parallel binaries generated by STATS include an additional source of TLP that has the potential to scale with the amount of input that needs to be processed. To this end, STATS introduces additional computation and inter-core communication to enforce the execution model described in Section II-B.

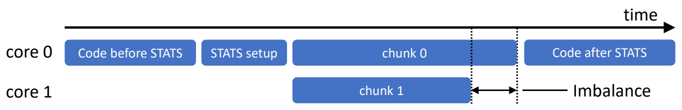


Fig. 4: Processing different number of inputs leads to imbalance computation.

This additional computation and communication generates overhead in the program's execution that blocks STATS to completely fulfill its potential. To understand how much of this overhead can be eliminated by engineering efforts and how much it requires a deeper evolution, it is important to understand and study the components of this overhead and their relative importance. This section describes such components and Section V empirically evaluates them.

A. Imbalance computation

The thread that runs the longest is the one that defines the performance of a parallel program. Therefore, the performance lost because of imbalance execution is the amount of time spent when all threads but one is running. This loss needs to be erased to scale the performance of a parallel program linearly with the number of cores. Namely, the computation needs to be perfectly balanced, so that at any point in time every core is executing some computation.

Parallel binaries generated by the STATS compiler can show imbalance computation at run-time. This is shown in Fig. 4 and it is created by having an imbalance division of the computation between threads. For example, imbalance can be created if different threads in the STATS execution model shown in Fig. 2b process different number of inputs. Another potential source of imbalance for the STATS execution model is generated by having different computation latencies for different inputs distributed between threads. Finally, imbalance can be generated if different threads start to execute their chunk of computation at different times because of thread synchronization overhead.

B. Extra computation

To implement the execution model described in Section II-B, the binary generated by the STATS back-end compiler performs extra work at run-time that was not part of the original program. What follows is a description of the components of this extra computation.

Generating speculative states. The STATS execution model (Fig. 2b) processes in parallel a sequence of inputs by splitting it into chunks. Each chunk is processed in parallel. A thread generated by STATS that processes a chunk needs to start from a computational state. Such computational state is computed by an alternative producer exploiting the short memory property of the related state dependence. In other words, the goal of an alternative producer is to predict the state that will be generated at the end of the computation of the previous chunk.

Alternative producers enable the extraction of additional TLP from the original code. To do so, they perform extra computation that was not included in the original program. An

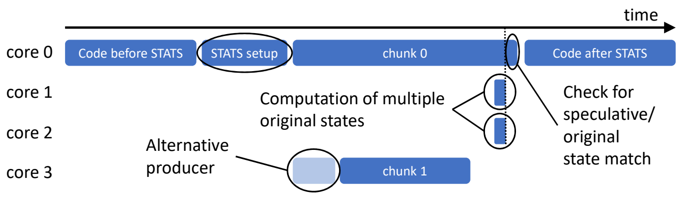


Fig. 5: STATS parallelized programs perform extra work because of the execution model that STATS enforces.

example of execution of an alternative producer is highlighted in Fig. 5. In this example, the STATS thread running on core 3 can start processing chunk 1 only after its alternative producer has generated its initial speculative state. To do so, this alternative producer processes a few inputs before the first one of chunk 1, which are the last inputs of chunk 0. The STATS thread that processes chunk 0 does not need an alternative producer because this is the first chunk of inputs and, therefore, it starts from the initial state defined by the original code. All other STATS threads process at run-time a few inputs before the beginning of the chunk assigned to them (like for the one running on core 3 of Fig. 5). The computation performed by all alternative producers is overhead tightly coupled with the STATS execution model and, as such, hard to reduce. While this source of extra work is hard to remove, Section V includes our empirical evaluation that suggests this is not the dominant overhead.

Generating multiple original states. STATS generates multiple original states at the end of each chunk of inputs to explore the space of acceptable states. For example, Fig. 5 shows that the computation of the last few inputs of chunk 0 is repeated three times, one on core 0, one on core 1, and the last one on core 2. This extra computation generates three states that differ because of the nondeterminism of the original code executed.

Having multiple states at the end of an input chunk allows the STATS runtime to decide whether or not the speculative state generated by the alternative producer of the next chunk can be accepted (and therefore the chunk that starts from it can commit). For example, the alternative producer running on core 3 in Fig. 5 generates the speculative state that will be used to start the computation of chunk 1. Before starting the computation of chunk 1, this speculative state is copied and sent to the STATS runtime, which will compare it against the multiple original states computed on cores 0, 1, and 2.

The multiple original states are generated in parallel (e.g., they are generated in parallel between cores 0, 1, and 2 in the example shown in Fig. 5). However, this computation requires to replicate the original computation for a few inputs, multiple times (twice in the example of Fig. 5). Generating multiple original states at the end of each chunk of inputs can be an important source of overhead for STATS and, therefore, it can limit the overall performance obtained.

State comparisons. Once the multiple original states are computed as described in the previous paragraph, the STATS runtime compares them with the speculative state generated

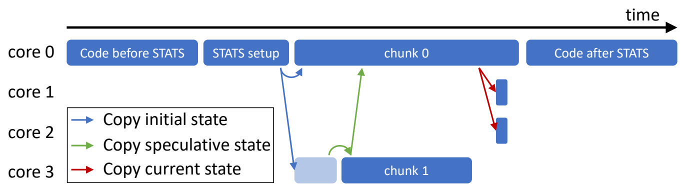


Fig. 6: STATS enforces an execution model that requires copies of the computational state to execute the computation in parallel.

by the alternative producer of the subsequent chunk of inputs. For example, when the thread running on core 0 in Fig. 5 has finished processing its chunk (i.e., chunk 0), it compares the multiple original states of that computational point with the speculative state generated by the alternative producer that has run on core 3. This comparison is needed to allow the STATS runtime to decide whether the subsequent chunk (e.g., chunk 1 of Fig. 5) can commit. These state comparisons are extra computation that was not included in the original program and, as such, can reduce the overall performance improvements obtained.

Setup. STATS needs to allocate and initialize supporting data structures to enforce the execution model described in Section II-B. These extra data structures (input lists, states, outputs, synchronization mechanisms such as mutexes and conditional variables) are allocated and initialized at the beginning of the STATS runtime (as Fig. 5 shows), before STATS threads start their computation. Moreover, they are freed when all STATS threads have ended their computation. These extra operations is what we consider the STATS setup overhead.

State copying. STATS splits the original sequential computation in different chunks and it processes them in parallel. Hence, the STATS execution model needs multiple copies of the computational state. This is in contrast with the original program where only a single computational state is needed.

The multiple states are created on demand by copying another one. For example, Fig. 6 shows that the first copy of the state is done at STATS setup time, when the system prepares the initial state that will then be passed to the STATS threads. Another copy of the state (speculative state in this case) is done by a STATS thread (chunk 1 in Fig. 6) to its previous STATS thread (chunk 0) that has committed, so that it can later check the quality of the speculative state. In order to check the quality of the given speculative state, chunk 0 needs to compute multiple original states, and does this in parallel. So, other state copies are necessary. All these state copies can limit the overall performance improvements obtained.

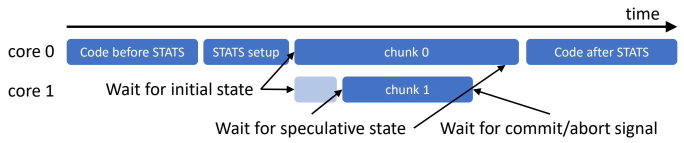


Fig. 7: Threads created by STATS need to synchronize among each others to send or receive data or signals.

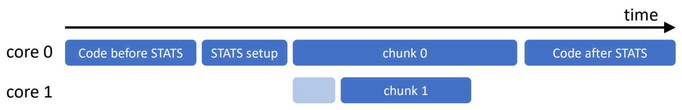


Fig. 8: Sequential code outside the code region of STATS does not benefit from the additional TLP that STATS generates.

C. Threads synchronization

Synchronizing threads can require the program to go to the kernel (e.g., to wakeup another thread), which takes several hundreds of clock cycles. On top of that, threads usually need to wait at the synchronization point for data or signals. The combination of the extra work (going to the kernel) and waiting time, is the synchronization overhead.

STATS generates a new source of TLP that follows a fork-join model as Fig. 7 shows. These threads need to synchronize with each other to comply with the execution model described in Section II-B. Initially, all STATS threads are spawned. A synchronization overhead is created by having all STATS threads to wait for their initial state. Another one is created by having a thread that has already committed its execution (like chunk 0 in Fig. 7) waiting for the speculative state that the thread processing the subsequent chunk of computation (chunk 1) used as initial state. This is necessary because chunk 0 is in charge of checking whether or not there is a match between the speculative state and its original states. Finally, once a speculative thread (chunk 1 in the example) finishes to process all its inputs, it has to wait for a commit/abort signal that comes from the previous committed thread (chunk 0) before it can end the execution and join the parent thread.

D. Sequential code

Speedup benefits coming from a parallelization scheme can only come from the program's region that is parallelized. Everything outside that region creates overhead that prevents a given parallelization scheme to reach ideal performance improvements.

STATS creates additional TLP only for the code region of the program affected by a state dependence. Everything that is outside this region of interest does not benefit the additional TLP generated by STATS, hence we consider it as overhead. Fig. 8 shows the computation outside the STATS region of interest with the boxes `Code before STATS` and `Code after STATS`.

E. Mispeculation and Unreachability

Mispeculation. The STATS autotuner decides the number of parallel chunks to generate based on the balance observed between the amount of parallelism extracted and the amount of mispeculation generated. The more parallel chunks, the more speculations are performed, the more potential mispeculations there are. In other words, STATS could generate more parallel chunks (and therefore performance) if all speculations commit. We classify as mispeculation the speedup lost due to having a lower number of parallel chunks (chosen by the autotuner) because some speculations abort.

Unreachability. Linear speedup is often not reached even if the parallelization does not add computation or communication to the execution as well as all speculations commit. This can happen because there are not enough parallel chunks to fully utilize all cores even when all speculations commit. We classify the speedup lost due to this aspect as unreachable.

IV. Experimental setup

This paper evaluates the potential performance roadblocks for parallel binaries generated by STATS. To this end, we run multiple experiments on an Intel-based platform. This section describes the experimental setup we have used to perform our empirical evaluations. Section V describes such evaluations.

A. Platform Setup

Our evaluation is done on a dual socket Dell PowerEdge R730 server with two Intel Xeon E5-2695 v3 Haswell processors running at 2.3GHz and capable of 9.60GT/s on the QPI interface. Each processor has 14 cores with 2-way hyper-threading, 35MB of last-level cache, and has a peak power consumption of 120W. The cores are supported by 256GB of main memory in 16 dual rank RDIMMs at 2133MHz. The OS is Red Hat Enterprise Linux Server 7.2 (kernel 3.10.0-693.21.1), with no CPU frequency governors enabled (all cores run at maximum frequency). Hyper-Threading is turned off for all experiments. Moreover, Turbo Boost is disabled. STATS is built on top of LLVM 7.0.0 [33], Racket 6.8 [23], and OpenTuner 0.8 [4].

B. Statistics

Convergence. Each data point we show is an average of repeated runs. We run the relevant configuration as many times as necessary to achieve a tight confidence interval where 95% of the measurements are within 5% of the median.

Autotuning time and configurations explored. Each benchmark has a unique design space. This impacts the time the autotuner needs to find a good configuration and the number of configurations explored. To address this issue, we customized the autotuning time on a per benchmark basis, which ranged from 2 to 72 hours. Within this autotuning time window, the number of configurations analyzed varied from 89 to 342.

States and threads created. The STATS execution model creates additional computational states and threads that were not present in the original benchmarks. These extra resources are needed to create additional TLP, generate the speculative state and extra original states, and produce the auxiliary code. Table I shows the total number of threads, computational states, and their size in bytes, that STATS creates for each benchmark when using 28 cores. Notice that the number of threads created is greater than the number of cores. This increases the core utilization of STATS parallelized benchmarks compared to the original ones. The only exception is `facetrack` where STATS only creates 14 parallel chunks of computation to avoid mispeculation.

TABLE I: Total number of threads, computational states, and state size of the "Par. STATS" version of the benchmarks shown in Fig. 9b.

Benchmark	#Threads	#States	State size [Bytes]
swaptions	36	36	24
streamclassifier	28	28	104
streamcluster	280	280	104
bodytrack	74	12	500000
facetrack	14	14	8000
facedet-and-track	70	70	8000

C. Benchmarks

We considered the POSIX multi-threaded versions of the PARSEC (version 3.0) benchmarks as well as their sequential version. We have considered five out of the six benchmarks that have been evaluated by the authors of STATS [20]. We did not consider `fluidanimate` because the STATS parallelization had no significant impact in the program's performance. We substituted `facedet` with `facetrack`, which performs the same task of tracking a person's face using a newer version of OpenCV (3.2.0). We considered a new benchmark called `facedet-and-track`, which uses a particle filter to track a person's face only when the OpenCV face detection API fails to do so.

Inputs. We used the native inputs provided by the PARSEC suite for our evaluation in Section V. In some cases native inputs are too small to properly test performance scalability on today's platforms. This has been already observed by prior work [31]; we thus extended the native inputs in the same fashion. For `streamclassifier`, we used the inputs from [50]. For `swaptions`, we increased the number of simulations to 32 millions and decreased the number of swaptions to 4, in order to allow the benchmark's bottlenecks to manifest. For `facetrack`, we used a video of a person moving in front of a camera, which includes 600 frames. For `facedet-and-track`, we used a longer video (1,050 frames) to compensate for the faster execution of the OpenCV face detection API with respect to the particle filter. To find the best configuration for a benchmark we used training inputs, which are different from the native inputs previously described.

Output quality. We relied on the same well-known output quality metrics used by the authors of STATS [20]. For the output quality of `facetrack` and `facedet-and-track` we used the average Euclidean distance between the boxes containing the detected faces.

V. Experimental Evaluation

Our evaluation examines the impact of the overhead described in Section III on the parallel execution of nondeterministic programs compiled with STATS. We analyze the performance of these programs in Section V-A. We evaluate the overhead of combining the TLP that the benchmarks had originally with the TLP generated by STATS in Section V-B. Then, we further analyze the overhead introduced by STATS by focusing on the performance scalability roadblocks that such overhead generates. We also investigate the total amount of additional work that STATS introduces in terms of number of instructions in Section V-C. Moreover, we describe the impact of the STATS parallelization in terms of data locality and branch prediction in Section V-D. Finally, we analyze the impact of STATS on the inherent output variability of the considered nondeterministic programs in Section V-E.

A. Performance Obtained by TLP Sources

The TLP that is expressed explicitly by developers via parallel programming APIs is not enough to utilize all cores included in our platform. Fig. 9 shows the performance obtained by a parallel binary compared to the sequential execution of that program. The black bars correspond to the parallel binary generated by only using the original TLP. The performance obtained by this TLP source is only 3.7× on 14 cores and 3.76× on 28 cores.

The TLP that is extracted by STATS scales more than the original TLP. The grey bars of Fig. 9 correspond to the parallel binary generated by STATS when no original TLP is used. In other words, these binaries rely only on the TLP extracted from state dependences. The performance obtained in this case is 8.45× on 14 cores and 11.65× on 28 cores.

Combining the original TLP with the STATS TLP generates important performance improvements. The red bars of Fig. 9 correspond to the parallel binary generated by STATS when the TLP extracted from state dependences is combined with the original TLP. The performance obtained in this situation is 10.61× on 14 cores and 14.77× on 28 cores. These results show that STATS improves significantly the overall performance, but it is not able to reach speedups that scale linearly with the number of cores. The rest of the section analyzes the reasons behind this limitation.

B. Performance Effects of STATS Overhead

To understand what is limiting the STATS binaries to obtain speedups that scale linearly with the number of cores, we have evaluated the performance impact of each of the six categories of overhead described in Section III. Notice that these sources of overhead are only related to the STATS execution model. These six overhead categories are evaluated as follows.

First we run the parallel binary generated by STATS and we keep track of the time in CPU cycles of each critical point of the STATS execution model. For example, we measure the CPU cycles that passed since the beginning of the program's execution and the time the main thread starts the code region parallelized by STATS. Another example of execution point we keep track of is when each STATS thread starts the execution of their chunk of inputs. Other examples are the beginning and the end of (i) each alternative producer, (ii) each code block that computes original states, (iii) the STATS setup block (all shown in Fig. 5), (iv) each thread synchronization code block (Fig. 7), (v) each code block that clones computational states (Fig. 6), and (vi) each code region parallelized by STATS. After obtaining these timestamps, we compute post-mortem the critical path of the parallel execution similar to what proposed in [26]. Finally, to evaluate the performance loss due to a given overhead, we compute the speedup obtainable if that overhead would be removed. To do this, we emulate the parallel execution removing only the part of the overhead targeted that is in the critical path, similar to what proposed in [26].

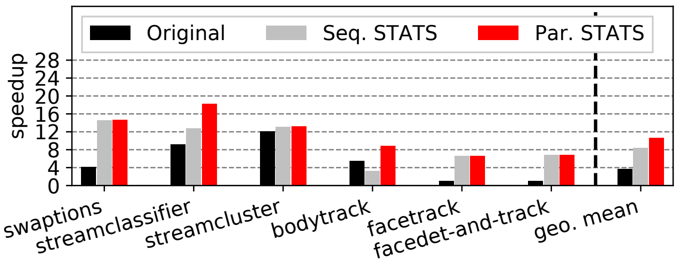
We measure the performance loss of each source of overhead of the STATS execution model following the approach described above. We performed this analysis in two situations. First we consider the scenario that STATS has been designed for. We let STATS combining the original TLP with the TLP extracted from state dependences. Then, we perform the same type of evaluation but forcing STATS to only rely on the TLP that comes from state dependences. This last analysis is needed to analyze the parallelism extracted by STATS at its limit.

Combining Original and STATS parallelism. Fig. 10 shows the performance loss for 28 cores when both original and STATS parallelism are used. It is interesting to notice that different benchmarks have different principal sources of overhead. For example, STATS is not able to achieve linear speedup with the number of cores for `facedet-and-track` mainly because of the synchronization overhead, which is required to implement the STATS execution model. `facetrack` is mainly limited by mispeculation because STATS only creates 7 parallel chunks to avoid aborting the computation. `bodytrack` is evenly limited by unreachability, mispeculation, and the overhead related to the STATS execution model (synchronization and extra computation). `streamclassifier` is mainly limited by synchronization and the code outside the regions parallelized by STATS. `streamcluster` is also limited by the sequential code outside the STATS parallel region, but also by the imbalance and synchronization between STATS threads. On the other hand, `swaptions` parallelized by STATS reaches linear speedup on 28 cores.

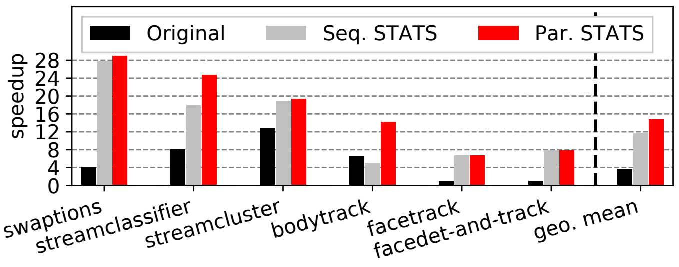
Fig. 11 shows the breakdown of the extra computation performed by the parallel binaries. The two main sources of extra computation are related to (and required by) the speculation scheme implemented by STATS: generating the speculative state and the multiple original states.

Only TLP from state dependences. This section analyzes the performance loss when the parallel binaries do not include the original TLP of the benchmark to better understand the impact of the sources of overhead described in Section III to the parallelism that STATS generates. To do this, we run STATS forcing it to create 14 and 28 STATS-threads (i.e., parallel chunks of computation) without using the original TLP. We performed the performance loss analysis for both 14 and 28 cores to highlight how each overhead source scales with the number of cores. Fig. 12 shows these results.

The difference between Fig. 12 and Fig. 10 highlights that extracting more TLP from state dependences generates significantly more extra computation. The more TLP is extracted from a state dependence, the more extra code is required to implement the STATS execution model. This makes the extra computation overhead more dominant than when the STATS TLP is combined with the original one. This is because when



(a) 14 cores



(b) 28 cores

Fig. 9: For most benchmarks, STATS generates a significant amount of extra parallelism. “Original” is the out-of-the-box benchmark that has been parallelized by traditional means. “Seq. STATS” (“Par. STATS”) is the binary generated by STATS starting from the sequential (multi-threaded) version of a benchmark.

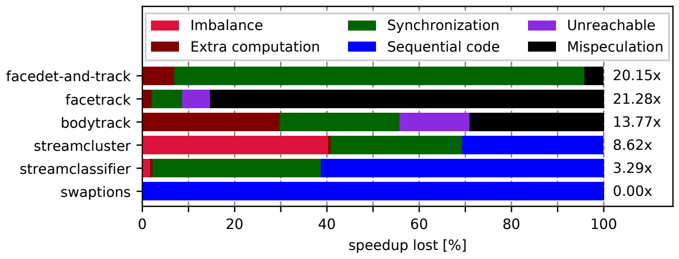


Fig. 10: Percentage of speedup lost by benchmarks that take advantage of both original TLP and STATS TLP, on 28 cores. The number at the right of each bar is the amount of speedup lost with respect to the ideal speedup. Every benchmark is limited by different sources of overhead.

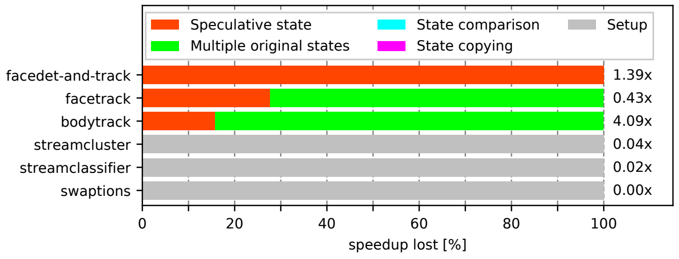


Fig. 11: Percentage of speedup lost due to the “Extra computation” fraction of Fig. 10. The number at the right of each bar is the amount of speedup lost only because of “Extra computation”. The two main sources of overhead are related to the generation of the speculative state and multiple original states. The overhead due to the STATS setup phase only accounts for a small fraction of the speedup lost.

STATS can combine the two sources of TLP, it is not forced to break state dependences too often.

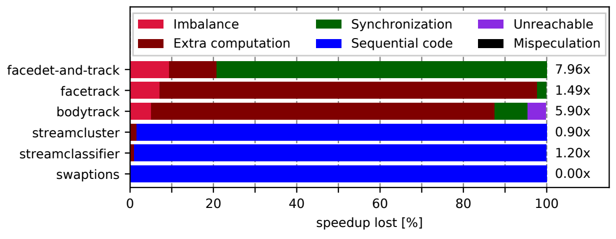
To further understand the extra computation generated, we broke it down in the 5 components described in Section III-B. This analysis is shown in Fig. 13. As for the case when both sources of TLP are used (Fig. 11), the two main sources of extra computation are related to the speculation scheme implemented by STATS: generating the speculative state and the multiple original states. The importance of these two sources of overhead suggests that the STATS execution model should evolve to implement a more scalable speculation scheme that requires less extra computation.

C. Extra Computation

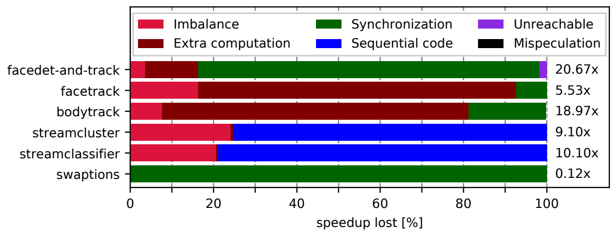
The previous empirical analysis suggests that the extra computation performed to implement the STATS execution model is an important source of overhead. To understand whether this is due to an abundant amount of extra work or because this extra work was performed in the critical path of the parallel execution, we further analyze it. To this end, we computed the total amount of extra work performed in terms of number of instructions executed at run time.

Fig. 14 shows the amount of extra instructions executed to implement the STATS execution model using 28 cores. The benchmarks that execute a considerable amount of extra instructions are `bodytrack` and `facedet-and-track`, and have respectively 107.4% and 43.8% extra instructions that are due to the extra computation described in Section III-B. This result combined with the previous analyses suggest that the extra computation overhead is an important performance roadblock for STATS, and that this extra computation is often performed in the critical path of the parallel execution. Finally, `streamclassifier` and `streamcluster` execute less instructions than the baseline because the TLP extracted from state dependences leads the execution to find their clustering solutions faster. This was already noticed by the authors of STATS [20].

Most of the extra instructions added by STATS are executed to copy computational states and to generate speculative states. Fig. 15 shows the breakdown of these extra instructions. Combining these results with the loss in performance shown in Fig. 13b, it is clear that instructions related to “State copying” are not in the critical path of the parallel execution, since the performance lost because of that are negligible. Contrarily, the amount of committed instructions to generate the speculative state (and to create multiple original states in the `bodytrack` case), have an impact on performance as well. However, we believe that improving STATS by accelerating the state copy operator is still valuable. This is because in the design space explored by the autotuner, there might be configurations that would scale well, but they are not chosen because copying computational states has a negative impact on their performance. A more efficient state copying would solve this problem. Improving the state copying could be implemented by compiler optimizations that exploit STATS

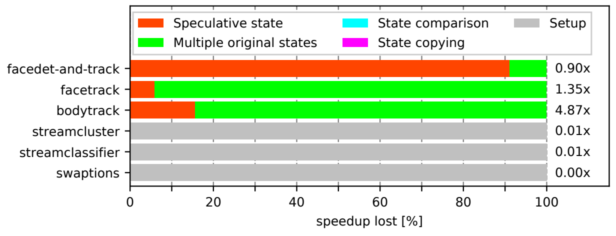


(a) 14 cores

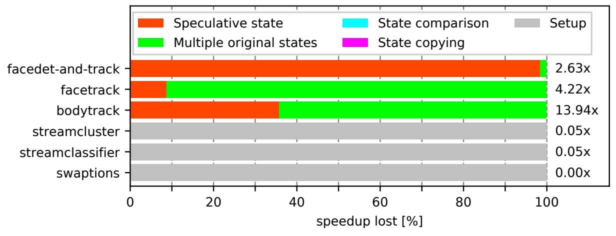


(b) 28 cores

Fig. 12: Percentage of speedup lost by benchmarks that take advantage of STATS TLP only, on both 14 and 28 cores. The number at the right of each bar is the amount of speedup lost with respect to the ideal speedup of 28× and 14× respectively. The fraction of speedup lost due to STATS “Extra computation” dramatically increases when more TLP is generated from state dependences.



(a) 14 cores



(b) 28 cores

Fig. 13: Percentage of speedup lost due to the “Extra computation” fraction of Fig. 12. The number at the right of each bar is the amount of speedup lost only because of “Extra computation”. As in Fig. 11, the main overhead components are related to the STATS speculation scheme (speculative state and multiple original states), while the speedup lost because of the STATS setup is negligible.

specific knowledge to consider a transformation space of the state copy operator larger than what general-purpose code transformations could reach. Another solution could be to exploit hardware accelerations for this task.

D. *Architecture Effects of STATS TLP*

TLP can have a negative effect to some architecture-specific characteristics of the underlying platform such as data locality and branch prediction. To evaluate these effects, we measured the total number of cache misses (absolute and percentage compared to the total number of cache accesses) for the L1D cache, L2 cache, and for the last level cache (LLC) of our platform. Furthermore, we measured the total number of branch mispredictions (absolute and percentage). These values are computed by adding all of the per-core counters of that hardware event. For example, the number of cache L1D misses is computed by counting all cache misses of all L1D of our 28 core platform.

Table II shows this analysis for the baseline code when no source of TLP is used (i.e., sequential execution), when only the original TLP is used (and 28 cores are considered), and when only STATS TLP is used (again, on 28 cores).

`facetrack` and `facedet-and-track` lose some data locality when STATS is used. This is because the STATS execution model runs in parallel the computation of input chunks breaking both the temporal and spatial locality between these chunks. Contrarily, `streamcluster` and `streamclassifier` have less cache misses and branch mispredictions compared to their out-of-the-box version because they execute less code. As described in Section V-C, the STATS version of these benchmarks converges faster to their solution. Finally, `swaptions` and `bodytrack` maintain a similar misprediction rate between the original and the STATS version of the benchmark. However, the number of absolute misses in `bodytrack` grows in the STATS version because the number of instructions executed is greater than the original version of the benchmark.

E. *Output Variability Due to Nondeterminism*

STATS preserves the original semantics: each output generated by a STATS binary could have been generated by the original program. However, the distribution of outputs generated by the nondeterminism of the original program can be affected by the parallelization STATS performs.

We run the original program two hundreds times and we compared all the outputs with an oracle one (i.e., highest output quality). The result is a distribution of output qualities between runs shown in Fig. 16. This figure also shows the same analysis for the parallel binaries generated by STATS. This comparison allows us to understand the impact of the STATS transformation to the output variance of the nondeterministic benchmarks considered. Counterintuitively, Fig. 16 shows that STATS tends to improve the quality of the outputs.

VI. Related Work

This paper characterizes the performance loss that is blocking the parallelizing compiler STATS [20], [21] to achieve speedups that scale linearly with the number of cores. To do so, this work characterizes the effects of the STATS parallelization scheme on a 28 core (dual-socket) Intel-based platform. Therefore, this paper relates with studies that characterize parallel workloads as well as parallelizing compilers.

TABLE II: Cache and branch mispredictions of the original and STATS transformed benchmarks. For each entry the value on the left is the total number of mispredictions (in billions), the value on the right is the misprediction rate.

Benchmark	Sequential original code				Parallel original code				STATS on 28 cores			
	L1D	L2	LLC	BR	L1D	L2	LLC	BR	L1D	L2	LLC	BR
swaptions	5.5 (1.6%)	0.3 (10.2%)	0.0006 (7.5%)	2.3 (1.7%)	5.7 (1.6%)	0.4 (12.7%)	0.001 (19.9%)	2.3 (1.7%)	5.7 (1.6%)	0.4 (12.1%)	0.01 (29.9%)	2.3 (1.7%)
streamclassifier	68 (30%)	5.5 (98%)	4.5 (87%)	0.293 (0.35%)	65 (28%)	8.7 (93%)	0.8 (11%)	0.316 (0.35%)	49 (29%)	6 (89%)	1 (17%)	0.198 (0.32%)
streamcluster	351 (32%)	6.2 (97%)	5 (90%)	0.688 (0.25%)	392 (35%)	32 (97%)	27 (98%)	0.724 (0.26%)	305 (27%)	17 (97%)	2 (11%)	0.752 (0.29%)
bodytrack	7.3 (5%)	1.6 (25%)	0.005 (0.4%)	0.447 (0.64%)	8.4 (5.7%)	2.1 (30%)	0.032 (2.2%)	0.543 (0.78%)	16.4 (5.4%)	3.5 (25.4%)	0.049 (1.7%)	0.994 (0.69%)
facetrack	13.8 (1%)	2.7 (44%)	0.004 (0.5%)	3 (0.13%)	13.8 (1%)	2.7 (44%)	0.004 (0.5%)	3 (0.13%)	17.2 (1%)	3.6 (47%)	0.06 (5.9%)	3.5 (0.13%)
facedet-and-track	6.1 (1%)	1.3 (47%)	0.005 (1.4%)	1.5 (0.17%)	6.1 (1%)	1.3 (47%)	0.005 (1.4%)	1.5 (0.17%)	17 (1.9%)	2.9 (56%)	0.03 (5.4%)	2.4 (0.18%)

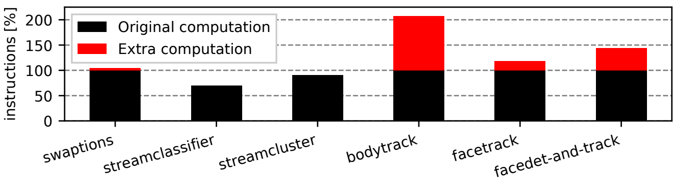


Fig. 14: Extra amount of instructions executed by STATS parallelized benchmarks on 28 cores. The benchmarks `bodytrack` and `facedet-and-track`, execute a considerable amount of extra instructions than their original version.

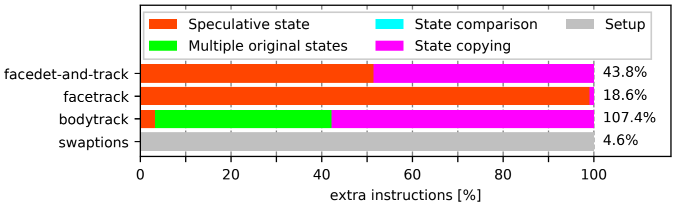


Fig. 15: Extra instructions breakdown related to the "Extra computation" of Fig. 14. Instructions related to the generation of the "Speculative state" by the alternative producer, and "State copying" dominate the other sources of extra instructions.

Parallel Workload Characterization. Most of the benchmarks [5] considered by this paper already include some TLP that was expressed manually by developers using parallel programming APIs like POSIX threads, OpenMP [19], and Intel TBB [41]. This TLP has been studied and characterized by prior work on multiple platforms [5], [6], [7], [8], [9], [10]. The STATS compiler adds the parallelism related to state dependences to the original TLP. This paper characterizes this additional parallelism both in isolation with the original TLP and when both sources of TLP are combined.

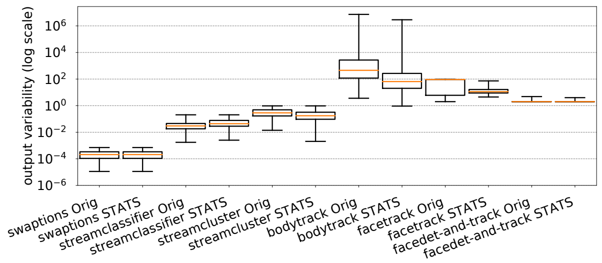


Fig. 16: Output variability before and after the transformation performed by STATS (lower values are better).

Parallelizing Compilers. Many parallelizing compilers have been proposed. Some of them satisfy all dependences without making the distinction between actual and apparent [2], [11], [12], [13], [15], [16], [17], [18], [22], [27], [28], [34], [36], [40], [44], [54], [57]. Others avoid the overhead of apparent dependences by speculating they do not exist and rolling back the parallel execution when they are wrong [1], [24], [25], [29], [32], [35], [39], [42], [43], [52], [53], [56], [58], [60]. The parallel binaries generated by these compilers execute sequentially all actual dependences. There are also parallelizing compilers that break actual dependences [3], [14], [38], [46], [47], [55], but do not preserve the original program semantics. Other systems [30], [45], [59], instead, break actual dependences by speculating on the state of a computation and roll back if that speculation was wrong to preserve the program semantics. However, their speculation techniques do not work for the irregular nondeterministic programs that STATS targets because they do not take advantage of the short memory property and nondeterminism of these programs. The binaries produced by STATS execute in parallel the code involved in state dependences, and preserve the original program semantics. This new type of parallelization performed by STATS introduces different and new sources of overhead that are not present in parallel binaries generated by other parallelizing compilers. This paper is the first study that identifies and characterizes these new sources of overhead.

VII. Conclusion

Thread level parallelism is the most important aspect of a program that defines its performance in the multicore era. TLP is typically obtained by executing independent code blocks in parallel. Recently, an additional source of TLP for nondeterministic workloads has been identified and exploited: code blocks that depend on each other via a state dependence can also run in parallel. This new source of TLP is generated by enforcing a new execution model that the compiler STATS embeds in the compiled code. This paper characterizes the impact of such execution model on a commodity platform. We have identified and characterized the main factors that can potentially block the performance obtained by the STATS parallel binaries. This analysis suggests that STATS can benefit from additional engineering efforts to reduce some of these factors. Finally, our characterization also suggests that the STATS execution model needs to evolve to remove the remaining performance roadblocks.

References

[1] Wonsun Ahn, Shanxiang Qi, M Nicolaides, Josep Torrellas, J-W Lee, Xing Fang, S Midkiff, and David Wong. BulkCompiler: high-performance sequential consistency through cooperative compiler and hardware support. In *International Symposium on Microarchitecture (MICRO)*, 2009.

[2] Alexander Aiken and Alexandru Nicolau. Perfect Pipelining: A New Loop Parallelization Technique. In *European Symposium on Programming (ESOP)*, 1988.

[3] Riad Akram, Mohammad Mejbah Ul Alam, and Abdullah Muzahid. Approximate Lock: Trading off Accuracy for Performance by Skipping Critical Sections. In *International Symposium on Software Reliability Engineering (ISSRE)*, 2016.

[4] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. OpenTuner: An Extensible Framework for Program Autotuning. In *Parallel Architectures and Compilation Techniques (PACT)*, 2014.

[5] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[6] Christian Bienia, Sanjeev Kumar, and Kai Li. Parsec vs. splash-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors. In *International Symposium on Workload Characterization (IISWC)*, September 2008.

[7] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[8] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[9] Christian Bienia and Kai Li. Characteristics of workloads using the pipeline programming model. In *Workshop on Emerging Applications and Many-core Architecture*, June 2010.

[10] Christian Bienia and Kai Li. Fidelity and scaling of the parsec benchmark inputs. In *International Symposium on Workload Characterization (IISWC)*, December 2010.

[11] S. Campanoni, T. M. Jones, G. Holloway, G. Y. Wei, and D. Brooks. HELIX: Making the Extraction of Thread-Level Parallelism Mainstream. In *IEEE MICRO*, 2012.

[12] Simone Campanoni, Kevin Brownell, Svilen Kanev, Timothy M. Jones, Gu-Yeon Wei, and David Brooks. HELIX-RC: An Architecture-compiler Co-design for Automatic Parallelization of Irregular Programs. In *International Symposium on Computer Architecuture (ISCA)*, 2014.

[13] Simone Campanoni, Kevin Brownell, Svilen Kanev, Timothy M. Jones, Gu-Yeon Wei, and David Brooks. Automatically accelerating non-numerical programs by architecture-compiler co-design. *Communication ACM*, 2017.

[14] Simone Campanoni, Glenn Holloway, Gu-Yeon Wei, and David Brooks. HELIX-UP: Relaxing Program Semantics to Unleash Parallelization. In *Code Generation and Optimization (CGO)*, 2015.

[15] Simone Campanoni, Timothy Jones, Glenn Holloway, Vijay Janapa Reddi, Gu-Yeon Wei, and David Brooks. HELIX: Automatic Parallelization of Irregular Programs for Chip Multiprocessing. In *Code Generation and Optimization (CGO)*, 2012.

[16] Simone Campanoni, Timothy Jones, Glenn Holloway, Gu. Y. Wei, and David Brooks. The helix project: Overview and directions. In *Design Automation Conference (DAC)*, 2012.

[17] Ding-Kai Chen and Pen-Chung Yew. On Effective Execution of Nonuniform DOACROSS Loops. In *Transactions on Parallel and Distributed Systems (TPDS)*, 1996.

[18] Ding-Kai Chen and Pen-Chung Yew. Redundant Synchronization Elimination for DOACROSS Loops. In *Transactions on Parallel and Distributed Systems (TPDS)*, 1999.

[19] Leonardo Dagum and Ramesh Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. In *IEEE Comput. Sci. Eng.*, 1998.

[20] Enrico A. Deiana, Vincent St-Amour, Peter A. Dinda, Nikos Hardavellas, and Simone Campanoni. Unconventional parallelization of nondeterministic applications. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[21] Enrico Armenio Deiana, Vincent St-Amour, Peter A. Dinda, Nikos Hardavellas, and Simone Campanoni. POSTER: the liberation day of nondeterministic programs. In *Parallel Architectures and Compilation Techniques (PACT)*, 2017.

[22] Kemal Ebcioglu and Alexandru Nicolau. A Global Resource-constrained Parallelization Technique. In *International Conference on Supercomputing (ICS)*, 1989.

[23] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. The Racket Manifesto. In *Summit on Advances in Programming Languages (SNAPL)*, 2015.

[24] Lance Hammond, Benedict A. Hubbert, Michael Siu, Manohar K. Prabhu, Michael K. Chen, and Kunle Olukotun. The Stanford Hydra CMP. In *International Symposium on Microarchitecture (MICRO)*, 2000.

[25] Liang Han, Wei Liu, and James M. Tuck. Speculative Parallelization of Partial Reduction Variables. In *Code Generation and Optimization (CGO)*, 2010.

[26] Jeffrey K Hollingsworth. Critical path profiling of message passing and shared-memory programs. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):1029–1040, 1998.

[27] Jialu Huang, Arun Raman, Thomas B. Jablin, Yun Zhang, Tzu-Han Hung, and David I. August. Decoupled Software Pipelining Creates Parallelization Opportunities. In *Code Generation and Optimization (CGO)*, 2010.

[28] A.R. Hurson, Joford T., LimKrishna M., and KaviBen Lee. Parallelization of DOALL and DOACROSS Loops - A Survey. In *Advances in Computers*, 1997.

[29] Troy A. Johnson, Rudolf Eigenmann, and T. N. Vijaykumar. Speculative Thread Decomposition Through Empirical Optimization. In *Principles and Practice of Parallel Programming (PPoPP)*, 2007.

[30] K. Kelsey, T. Bai, C. Ding, and C. Zhang. Fast Track: A Software System for Speculative Program Optimization. In *Code Generation and Optimization (CGO)*, 2009.

[31] Hanjun Kim, Nick P Johnson, Jae W Lee, Scott A Mahlke, and David I August. Automatic speculative DOALL for clusters. In *Code Generation and Optimization (CGO)*, 2012.

[32] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic Parallelism Requires Abstractions. In *Programming Language Design and Implementation (PLDI)*, 2007.

[33] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization (CGO)*, 2004.

[34] Duo Liu, Zili Shao, Meng Wang, Minyi Guo, and Jingling Xue. Optimal Loop Parallelization for Maximizing Iteration-level Parallelism. In *Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, 2009.

[35] Wei Liu, James Tuck, Luis Ceze, Wonsun Ahn, Karin Strauss, Jose Renau, and Josep Torrellas. POSH: A TLS Compiler That Exploits Program Structure. In *Principles and Practice of Parallel Programming (PPoPP)*, 2006.

[36] Kathryn S. McKinley. Evaluating Automatic Parallelization for Efficient Execution on Shared-memory Multiprocessors. In *International Conference on Supercomputing (ICS)*, 1994.

[37] Mojtaba Mehrara, Jeff Hao, Po-Chun Hsu, and Scott Mahlke. Parallelizing Sequential Applications on Commodity Hardware Using a Low-cost Software Transactional Memory. In *Programming Language Design and Implementation (PLDI)*, 2009.

[38] Sasa Misailovic, Deokhwan Kim, and Martin Rinard. Parallelizing Sequential Programs with Statistical Accuracy Tests. In *ACM Trans. Embed. Comput. Syst. (TECS)*, 2013.

[39] Niall Murphy, Timothy Jones, Robert Mullins, and Simone Campanoni. Performance implications of transient loop-carried data dependences in automatically parallelized loops. In *Compiler Construction (CC)*, 2016.

[40] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. Automatic Thread Extraction with Decoupled Software Pipelining. In *International Symposium on Microarchitecture (MICRO)*, 2005.

[41] Chuck Pheatt. Intel&Reg; Threading Building Blocks. In *J. Comput. Sci. Coll.*, 2008.

[42] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. The Tao of Parallelism in Algorithms. In *Programming Language Design and Implementation (PLDI)*, 2011.

[43] Arun Raman, Hanjun Kim, Thomas R. Mason, Thomas B. Jablin, and David I. August. Speculative Parallelization Using Software Multi-threaded Transactions. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[44] Easwaran Raman, Guilherme Ottoni, Arun Raman, Matthew J. Bridges, and David I. August. Parallel-stage Decoupled Software Pipelining. In *Code Generation and Optimization (CGO)*, 2008.

[45] Easwaran Raman, Neil Va hharajani, Ram Rangan, and David I. August. Spice: Speculative Parallel Iteration Chunk Execution. In *Code Generation and Optimization (CGO)*, 2008.

[46] Lakshminarayanan Renganarayana, Vijayalakshmi Srinivasan, Ravi Nair, and Daniel Prener. Programming with relaxed synchronization. In *Relaxing synchronization for multicore and manycore scalability (RACES)*, 2012.

[47] Martin C Rinard. Using early phase termination to eliminate load imbalances at barrier synchronization points. In *Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2007.

[48] Behnam Robatmil, Dong Li, Hadi Esmaeilzadeh, Sibi Govindan, Aaron Smith, Andrew Putnam, Doug Burger, and Stephen W. Keckler. How to Implement Effective Prediction and Forwarding for Fusable Dynamic Multicore Architectures. In *High-Performance Computer Architecture (HPCA)*, 2013.

[49] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Nitya Ranganathan, Doug Burger, Stephen W. Keckler, Robert G. McDonald, and Charles R. Moore. TRIPS: A polymorphous architecture for exploiting ILP, TLP, and DLP. In *Transactions on Architecture and Code Optimization (TACO)*, 2004.

[50] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. Managing Performance vs. Accuracy Trade-offs with Loop Perforation. In *European Conference on Foundations of Software Engineering (ESEC/FSE)*, 2011.

[51] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar Processors. In *International Symposium on Computer Architecture (ISCA)*, 1995.

[52] J. Steffan and T Mowry. The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization. In *High-Performance Computer Architecture (HPCA)*, 1998.

[53] J. Gregory Steffan, Christopher Colohan, Antonia Zhai, and Todd C. Mowry. The STAMPede Approach to Thread-level Speculation. In *Transactions on Computer Systems (TOC)*, 2005.

[54] Georgios Tournavitis, Zheng Wang, Björn Franke, and Michael F.P. O'Boyle. Towards a Holistic Approach to Auto-parallelization: Integrating Profile-driven Parallelism Detection and Machine-learning Based Mapping. In *Programming Language Design and Implementation (PLDI)*, 2009.

[55] Abhishek Udupa, Kaushik Rajan, and William Thies. ALTER: Exploiting breakable dependences for parallelization. In *Programming Language Design and Implementation (PLDI)*, 2011.

[56] Cheng Wang, Youfeng Wu, Edson Borin, Shiliang Hu, Wei Liu, Dave Sager, Tin-fook Ngai, and Jesse Fang. Dynamic Parallelization of Single-threaded Binary Programs Using Speculative Slicing. In *International Conference of Supercomputing (ICS)*, 2009.

[57] Cheng-Zhong Xu and Vipin Chaudhary. Time Stamp Algorithms for Runtime Parallelization of DOACROSS Loops with Dynamic Dependences. In *Transactions on Parallel and Distributed Systems (TPDS)*, 2001.

[58] Antonia Zhai, J. Gregory Steffan, Christopher B. Colohan, and Todd C. Mowry. Compiler and Hardware Support for Reducing the Synchronization of Speculative Threads. In *Transactions on Architecture and Code Optimization (TACO)*, 2008.

[59] Zhijia Zhao and Xipeng Shen. On-the-Fly Principled Speculation for FSM Parallelization. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[60] Hongtao Zhong, Mojtaba Mehrara, Steven A. Lieberman, and Scott A. Mahlke. Uncovering hidden loop level parallelism in sequential applications. In *High-Performance Computer Architecture (HPCA)*, 2008.