Unconventional Parallelization of Nondeterministic Applications

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

Vincent St-Amour
Northwestern University
stamourv@northwestern.edu

Peter A. Dinda
Northwestern University
pdinda@northwestern.edu

Nikos Hardavellas
Northwestern University
nikos@northwestern.edu

Simone Campanoni
Northwestern University
simonec@eecs.northwestern.edu

Abstract

The demand for thread-level-parallelism (TLP) on commodity processors is endless as it is essential for gaining performance and saving energy. However, TLP in today's programs is limited by dependences that must be satisfied at run time. We have found that for nondeterministic programs, some of these actual dependences can be satisfied with alternative data that can be generated in parallel, thus boosting the program's TLP. Satisfying these dependences with alternative data nonetheless produces final outputs that match those of the original nondeterministic program. To demonstrate the practicality of our technique, we describe the design, implementation, and evaluation of our compilers, autotuner, profiler, and runtime, which are enabled by our proposed C++ programming language extensions. The resulting system boosts the performance of six well-known nondeterministic and multi-threaded benchmarks by 158.2% (geometric mean) on a 28-core Intel-based platform.

Keywords dependences; parallelization; speculation

ACM Reference Format:
Enrico A. Deiana, Vincent St-Amour, Peter A. Dinda, Nikos Hardavellas, and Simone Campanoni. 2018. Unconventional Parallelization of Nondeterministic Applications . In *ASPLOS '18: 2018 Architectural Support for Programming Languages and Operating Systems, March 24–28, 2018, Williamsburg, VA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3173162.3173181

1 Introduction

Increasing thread-level parallelism (TLP) is the chief way to improve performance on multi-core systems. However, the inter-thread data movements present in most programs constrain their TLP and hence their performance. These data movements are necessary for satisfying dependences, a taxonomy for which Figure 1 illustrates. Ideally, data movements satisfy only *actual dependences*, i.e., those which are fundamental to the program. Unfortunately, they may also occur to satisfy *apparent dependences*, i.e., those that are not in fact necessary, but for which the compiler or developer were unable to prove unnecessary.

The research community has had tremendous success using techniques like thread-level speculation to reduce the impact of apparent dependences [35, 43, 52, 54, 68, 70, 73, 77, 89, 90]. These achievements have been proven productive enough to be implemented in both high-end [20, 37, 42, 49] and commodity processors [34]. Uncovering and ignoring apparent dependences, however, has reached the point of diminishing returns. To climb from this parallelism plateau, we must turn our attention to actual dependences.

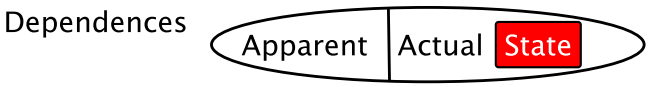


Figure 1. Taxonomy of dependences. State dependences are those that can be satisfied with auxiliary code.

While all actual dependences must be satisfied to preserve program semantics, strict semantics preservation is not always necessary. Prior work has shown that carefully selecting which actual dependences to break can lead to performance gains [16, 57, 69, 79, 81], but the output inaccuracies these techniques introduce make them unsuitable for settings where output quality must be preserved.

Nondeterministic applications naturally produce different results from run to run given the same input. We see this as an opportunity and we ask the question: can we exploit the output variability of nondeterministic programs to liberate additional TLP while preserving the output quality? This paper aims to answer this question by considering alternative ways of satisfying actual dependences.

Specifically, we identify a subset of actual dependences—*state dependences*—that can be satisfied via alternative, per-dependence code which we dub *auxiliary code*. Such auxiliary code aims to produce results that match the original code, but much faster. This code is compiler-generated enabled by developer-supplied algorithm-specific information. Auxiliary code runs in parallel with the source of its state

ASPLOS '18, March 24–28, 2018, Williamsburg, VA, USA

ACM ISBN 978-1-4503-4911-6/18/03…$15.00
https://doi.org/10.1145/3173162.3173181

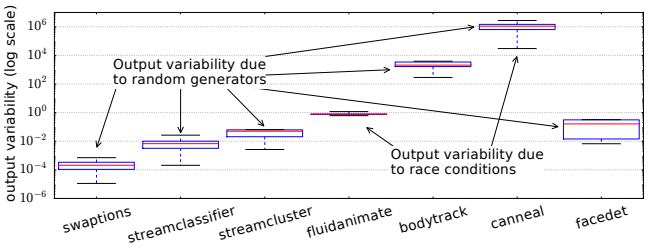


Figure 2. Output variability of nondeterministic PARSEC benchmarks. Several exhibit very high variability and are particularly amenable to STATS.

dependence, which generates additional TLP. To preserve the program's original semantics, the auxiliary code's execution is monitored by our runtime system, to check that its speculative results match those expected from the original nondeterministic code. To increase the probability of having a match among speculative and original results, we take advantage of the program's nondeterminism and compute multiple original results that we compare against the speculative one. When these checks succeed, the additional TLP generated can be safely used. When these checks fail, our runtime aborts the computation that relied on the inadequate auxiliary code. Then, the runtime reverts the execution satisfying that state dependence the usual way, falling back to the original code using the first generated original result. Our experiments show that it is possible to generate auxiliary code such that these checks generally succeed, with the TLP benefits that this entails. This work is the first to automatically generate auxiliary code able to satisfy particular actual dependences while preserving output quality.

Our work is embodied in a system named *STATS (STAte Transition Speculator)* that takes advantage of state dependences throughout its compilers, autotuner, profiler, and runtime system. To assess the effectiveness of STATS, we evaluated it using the nondeterministic programs of the PARSEC suite [8] as well as the well-known, industrial-strength codebase OpenCV [13]. **Without loss in output quality** (guaranteed via run-time checks), STATS increases their performance by 158.2% on average (geometric mean) on a dual-socket Intel-based platform with 14 cores per socket. Alternatively, STATS can save 71.35% on average (geometric mean) of the system-wide energy consumption.

The contributions of the paper are as follows. (1) We identify and describe the concept of state dependence. (2) We identify a code pattern that is commonly found in nondeterministic programs, and which corresponds to a state dependence. (3) We describe a technique for determining whether a state dependence can be satisfied with auxiliary code while preserving the original output quality. (4) We analyze the considerable opportunity for increasing TLP that state dependences following this pattern offer. (5) We develop a methodology for exploiting state dependences to increase TLP in a nondeterministic program while preserving the output quality. (6) We describe the design and implementation of a system that embodies our methodology to further

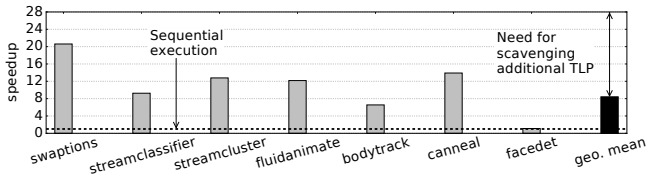


Figure 3. Highest speedup obtained by nondeterministic PARSEC benchmarks on a 28 cores Intel-based platform.

parallelize multi-threaded C++ programs. (7) We evaluate the system on well-known, nondeterministic multi-threaded benchmarks. Despite these benchmarks having been heavily parallelized already, we obtained considerable gains in both performance and energy.

2 Opportunity

The output of a deterministic program is determined solely by its input. To preserve its output quality, all of its actual dependences must be satisfied by generating and forwarding the intermediate data according to these dependences. Some programs, however, are *nondeterministic by design*. Such a program may exhibit variation in its output across runs for the same input. Figure 2 shows such variation over 100 runs for six well-known benchmarks. [1]

Output variations of nondeterministic programs originate from variations in their program's intermediate data. A given intermediate datum generated by a producer and forwarded to a consumer may vary across runs for the same input. This suggests a degree of freedom (i.e., any of these data can be forwarded to its consumer) in satisfying the related dependence. This work is the first that takes advantage of this opportunity.

The rest of this section shows the performance limitations of the considered nondeterministic benchmarks. Then, it uses one of these benchmarks to demonstrate the described opportunity as well as to give the intuition behind our solution. We end this section by describing a code pattern we found in these programs that our approach targets to take advantage of this opportunity.

2.1 Today's Limits

To understand the need for additional parallelizations for nondeterministic programs, we studied the PARSEC benchmark suite [8], which features multi-threaded implementations of emerging workloads, as well as the industrial-strength codebase OpenCV [13]. These programs have been manually parallelized extensively leaving no room for simple additional parallelizations. We measure the performance of only the nondeterministic benchmarks that successfully compiled with *clang* [48] (i.e., `bodytrack`, `fluidanimate`, `swaptions`, `streamcluster`, `streamclassifier`, `canneal`, and `facedet`) in a 28 cores, Intel-based platform. [1]

All benchmarks considered have limited TLP. Figure 3 shows the highest speedup obtained by each benchmark compared to their sequential execution. The distance from

[1] Details about these experiments are in Section 4.

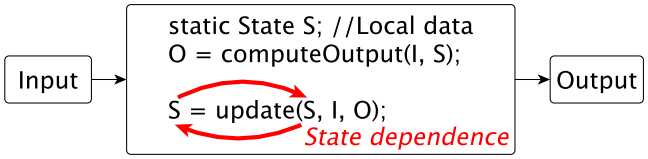


Figure 4. Code pattern that includes a state dependence.

an ideal speedup of 28× (number of cores) shows the need for scavenging additional TLP.

2.2 Code Example

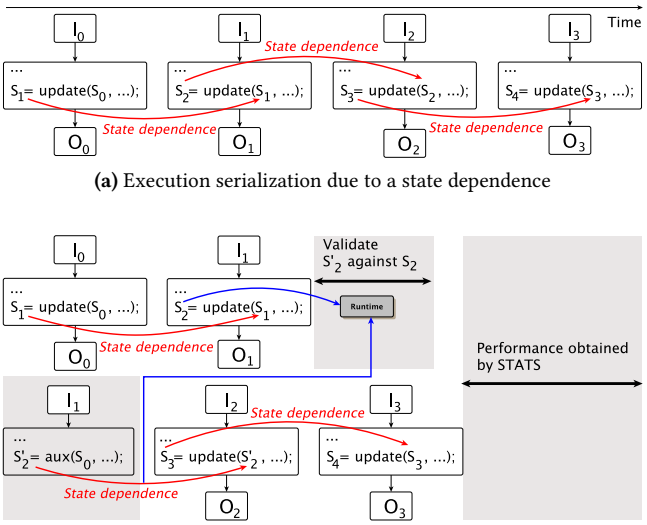
Benchmark. Let us consider the nondeterministic program `bodytrack`. This program tracks a person's body as captured by four cameras that target the same space (e.g., an office). To do so, `bodytrack` analyzes the stream of four pictures, called quadruples, one quadruple at a time.

The analysis of a quadruple generates a datum, which represents the current belief of where the body is in the 3D space. This datum is consumed by the analysis of the next quadruple to exploit that is likely that the person in quadruple $i + 1$ is relatively close to where he/she was in quadruple i. The computation performed to analyze quadruples is computationally intensive (i.e., it consumes 97% of the total execution time) and randomized (i.e., nondeterministic). This randomization is responsible for the generation of slightly different positions of the body parts for the same quadruple over multiple and independent runs, any of which are acceptable.

The TLP, and therefore performance, of `bodytrack` is constrained by a single sequential chain of dependences. The analysis of the quadruple $i+1$ can start only when the datum generated by the analysis of the quadruple i is available.

Opportunity. This limiting dependence chain between quadruples can be broken by injecting an alternative producer for the forwarded datum. The intuition is that where a human is at quadruple i is likely to be independent of where he/she was in the quadruple $i - k$ with high k. This can be exploited as follows: rather than blocking the analysis of i until the analysis of all previous quadruples ends, we can overlap it with them. To preserve the output quality, however, we perform extra computation before analyzing the quadruple i. This extra computation aims to be an alternative producer of the datum required by the quadruple i and, therefore, it needs to consume (only) a few previous quadruples of the i^{th} one. We call the alternative producer *auxiliary code* because we use it as a substitute in case of need—when there is not enough TLP for the target platform.

Safeguard. The assumption under which the auxiliary code can safely substitute the original producer of the related datum is that the last few inputs (how many is determined by STATS) are enough for this goal. This assumption is checked at run time to preserve the original output quality by comparing the datum generated by the auxiliary code with the ones generated by the original producer. In more detail, when all



(a) Execution serialization due to a state dependence

(b) Additional TLP generated by auxiliary code

Figure 5. Alternative execution model obtained by using auxiliary code to satisfy a state dependence.

quadruples before i are analyzed by the original code, an actual datum that the auxiliary code aims to reproduce is now available. These two data are compared to check whether the analysis of the quadruple i (and therefore the next ones) matches the original semantic. If not, then we can either generate another datum from the non-deterministic original producer and repeat the checks or we abort the analysis of i (and the subsequent ones) restarting it using the correct datum. Our hypothesis (confirmed by STATS) is that often the auxiliary code generates an acceptable datum, therefore, liberating additional TLP.

2.3 State Dependences

The dependence chain described earlier between quadruples in `bodytrack` is an example of state dependence. State dependences are the actual dependences related to a piece of computational state used in the code pattern shown in Figure 4. A piece of code (e.g., a basic block, a loop iteration, a loop, a hammock [25], or an entire function) computes an output `O` from a given input `I`, consulting some local state `S`. As part of computing `O`, the code also updates `S` which then feeds forward to the next invocation of the code. Hence, there is a dependence between invocation i's write of `S` and invocation $i + 1$'s read of `S`, which serializes invocations of the code. This is shown in Figure 5a where multiple invocations of the code pattern of Figure 4 run sequentially because of the state dependence between the producer of an `S` (invocation i) and its consumer (invocation $i + 1$).

3 The STATS Solution

The STATS tool-chain increases the TLP (and thus performance or energy efficiency) of nondeterministic C++ programs that exhibit the pattern of Figure 4. It does so relying on additional algorithm-specific information and training inputs from the developer. These inputs are only used to

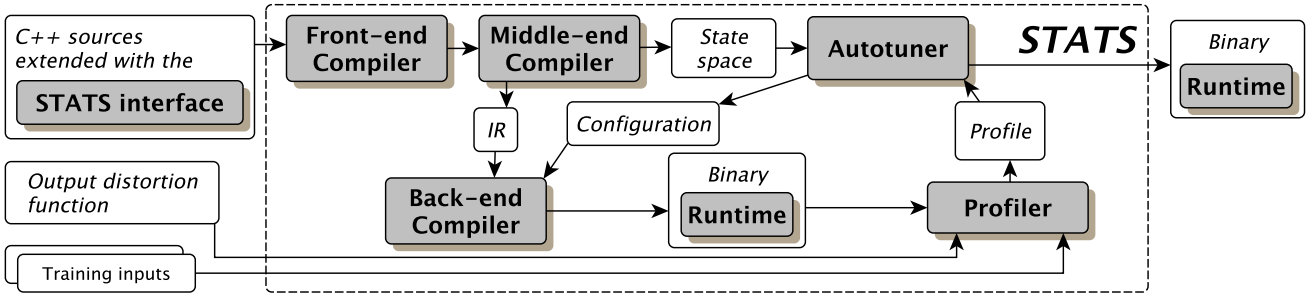


Figure 6. STATS includes three compilers, a runtime, an autotuner, and a profiler to optimize a nondeterministic C++ program for which the developer has identified state dependences via the STATS Interface.

explore the design space described by state dependences and find a configuration of the program with the best profile (e.g., highest performance). Our runtime preserves the output quality regardless of the representativeness of training inputs, but the more representative of actual workloads they are, the more performant STATS's output program will be in production. The rest of this section describes the execution model generated by STATS and its compilation flow.

3.1 Execution Model

STATS extracts additional TLP by grouping inputs of the code pattern shown in Figure 4 in ordered blocks and by overlapping their computations. This additional TLP can be exploited only when auxiliary code can satisfy the related state dependence. In other words, when the auxiliary code generates a datum that matches one of the many possible outputs (due to the non-determinism) that can be generated by the original producer.

For example, consider the original execution shown in Figure 5a. Here, all inputs are sequentially processed. STATS, in this example, generates the execution model shown in Figure 5b. Inputs are grouped in pairs (e.g., I_0, I_1 and I_2, I_3) and each group is processed in parallel (STATS automatically decides what is the most convenient group cardinality). The first invocation of the first group starts with the initial state S_0 to process its first input (e.g., I_0). Instead, the first invocation of each subsequent group starts with the state generated by its auxiliary code (e.g., S_2'), which we call speculative because it is based on the assumption it will match the one that will be generated by the original producer (e.g., S_2). The auxiliary code generates its speculative state (e.g., S_2') starting from the initial state S_0 and by using a few (decided by STATS) previous inputs (e.g., I_1 in our example). When the last invocation of the previous group of inputs ends (the second invocation in the example shown in Figure 5b), the runtime compares its final state (e.g., S_2) with the speculative state used by the first invocation of the subsequent input group (e.g., S_2'). If these states match (like in the example shown in Figure 5b), then the computation of the subsequent group stops being speculative and its outputs can be used. If these states do not match, then the execution of the previous group of inputs goes back a few inputs (STATS decides how many inputs to go back) and it repeats the computation. This new computation might lead to a different final state (e.g., a new S_2) because of the non-determinism of the target code (i.e., `computeOutput()` of Figure 4). If the new final state matches the speculative state (e.g., S_2'), then the computation of the subsequent group stops being speculative and its outputs can be used. Otherwise, either the execution of the previous group of inputs goes back a few inputs one more time and checks again the final state (STATS decides how many times the execution of the previous group can be repeated) or the computation of all subsequent groups of inputs aborts. If the computation aborts, then all the outputs generated by processing subsequent inputs (e.g., input I_2 and after) are squashed, the execution restarts from the first not-speculative state generated by the previous group of inputs (e.g., S_2), and no other speculation is performed until all the current inputs are processed.

3.2 Architecture

STATS enforces the execution model described in Section 3.1 via its architecture shown in Figure 6. Developers provide descriptions of state dependences, as well as algorithm-specific tradeoffs needed to generate auxiliary code, through the STATS *interface* implemented as C++ extensions.

The *front-end* compiler translates extended C++ codebases to standard C++ source, encoding STATS-specific information in APIs calls understood by the other STATS compilers. The *middle-end* compiler translates the output of the front-end to our intermediate representation (IR), which represents the design space explicitly, which we call *state space*.

The description of the state space is used by the *autotuner*, which explores it by choosing the next configuration to test. A configuration describes which state dependences to consider to satisfy with auxiliary code and its parameters. Parameters include the inputs to each auxiliary code, how to set the auxiliary-code tradeoffs, how many times the original producer of a state dependence can re-execute, and how far back the original execution needs to go.

The *back-end* compiler translates our IR to the binary that corresponds to a configuration chosen by the autotuner. The back-end also embeds the STATS *runtime* into the binary after having specialized it for each state dependence that

```
void estimateLocations() {
  vector<int> frameIds(numFrames);
  vector<Particle> model(numParticles);
  vector<BodyPart> positions;
  for(auto frameId : frameIds) {
    Frame f = getFrame(frameId);
    model = updateModel(numAnnealingLayers,
                        model, f);
    positions = getPositions(model);
  }
}
```

Figure 7. Original code of bodytrack.

will be satisfied by auxiliary code. The runtime determines whether to accept the speculative state and enforces the execution model described in Section 3.1.

The *profiler* runs the binary generated by the back-end using the provided training inputs, measuring its energy consumption and performance. It provides such information to the autotuner. The autotuner then decides whether or not to test other configurations. When enough information has been obtained, the autotuner generates the most performant binary. Finally, the autotuner stores the results of its exploration in the description of the state space, which allows them to be reused should the specific optimization objective change (e.g., changing the optimization goal from performance to energy).

3.3 The STATS Interface

The STATS Interface enables developers to describe state dependences and algorithm-specific tradeoffs.

State Dependence Interface (SDI). Identifying state dependences requires algorithmic knowledge that is beyond the purview of automatic tools. Hence, developers provide STATS with a set of state dependences. It may turn out that auxiliary code cannot satisfy some of them; STATS automatically detects and discards such cases.

The SDI allows developers to encode instances of the pattern in Figure 4, thereby asserting that the inter-invocation dependence on `State` is a state dependence. The STATS autotuner will decide whether or not such state dependence can be satisfied with auxiliary code. The SDI encoding replaces the corresponding pattern instance in the program. The API for the SDI is shown in Figure 9. Developers need to create classes corresponding to `Input`, `State`, and `Output`, then instantiate a state dependence object parameterized with these classes. The `start()` method of a state dependence object begins the execution model described in Section 3.1 in parallel with the invoking thread. The `join()` method waits until all inputs provided to the state dependence object are correctly processed.

Making state dependence patterns explicit has two main advantages. First, the STATS compilers immediately identify that the inter-invocation dependence on `State` is actually a state dependence. Second, it allows the compilers to enforce a rigid dependence structure, which they then

```
class Input { int frameId; };
class Output { vector<BodyPart> positions; };
class State {
  vector<Particle> model;
  State& operator=(State&);
  bool doesSpecStateMatchAny(set<State*>);
};
Output* computeOutput(Input *i, State *s){
  Frame f = getFrame(i->frameId);
  s->model = updateModel(TO_numAnnealingLayers,
                         s->model, f);
  Output *o = new Output();
  o->positions = getPositions(s->model);
  return o;
}
void estimateLocations() {
  vector<Input*> i(numFrames);
  vector<Particle> model(numParticles);
  State s; s.model = model;
  StateDependence<Input, State, Output>
                stateDep(&i,&s,computeOutput);
  stateDep.start(); stateDep.join();
}
```

Figure 8. Use of SDI in bodytrack.

exploit. Specifically, they need to enforce that computing `Output` only depends on `Input` and `State`, and that the only inter-invocation dependence in this code is that on `State`. Most importantly, STATS explicitly manages which values of `State` each invocation sees, which makes it possible to execute multiple instances of `computeOutput()` (that contains the computation related to the state dependence) in parallel. This involves privatizing `State` for each thread by cloning it, the code for which is provided by developers by overriding `State`'s assignment method (`operator=()`). With the SDI encoding, the STATS runtime thus clones `State` whenever it is necessary.

Finally, developers need to provide the state comparison method (`doesSpecStateMatchAny()`). This function compares the speculative state coming from the auxiliary code with a set of original states, and returns whether the speculative state should be considered equivalent to an original state. This API allows developers to decide how strict the matching between speculative and original states needs to be. We describe how the state comparison method is used in Section 3.4.

Figure 8 shows how a state dependence in `bodytrack` is encoded using the SDI. Figure 7 shows the original version of the benchmark.

Tradeoff Interface (TI). TI is used to describe tradeoffs specific to an algorithm, which are used to balance quality and performance in auxiliary code. Identifying such tradeoffs requires knowledge beyond the reach of automatic tools.

A tradeoff is a piece of program text (constant, data type, function) whose value is chosen from a range supplied by developers. Tradeoff values are sorted by their index (e.g., first value, second value). A tradeoff example from `bodytrack`

```
template<class Input, class State, class Output>
class StateDependence {
  StateDependence(
    vector<Input*> *inputs,
    State *initialState,
    function<Output* (Input*, State*)>
      computeOutput
  );
  void start(void);
  void join(void);
};
```

Figure 9. The State Dependence Interface makes the pattern of Figure 4 explicit to the compiler.

```
class AnnealingLayers_options:Tradeoff_options{
  int64_t getMaxIndex(){ return 10; }
  auto getValue(int64_t i){ return i+1; }
  int64_t getDefaultIndex() { return 4; }
};
tradeoff TO_numAnnealingLayers {
  {AnnealingLayers_options};
};
```

Figure 10. Use of TI in bodytrack.

is the number of annealing layers to use when computing an estimation of the human body position. The higher the tradeoff value, the better the estimation, but at the cost of a longer computation time. Tradeoffs (and the ranges of values that they can assume) are specific to particular algorithms.

Figure 10 shows this tradeoff described using the TI. A tradeoff provides three methods: `getMaxIndex()` returns the number of possible values; `getValue()`, given a valid index i, returns the i-th possible value; and, finally, the method `getDefaultIndex()` returns the index to use when the tradeoff is used outside auxiliary code. To obtain the original version of the program (our baseline), we set all tradeoffs to their default value and satisfy all state dependences conventionally (i.e., no auxiliary code).

The target of a state dependence requires `State` to compute its output (c.f. Figure 4). The auxiliary code computes at run time an alternative (`State'`) of `State` for that purpose. Tradeoffs are used to strike the right balance between the quality of `State'` and its computational cost. The better `State'` is, the more likely it will match `State`.

The state space. The state space is defined by all tradeoffs, by how often a state dependence is satisfied with auxiliary code, by the number of previous inputs an auxiliary code will consider, by the maximum number of times the STATS runtime can execute an original producer of a given state dependence, and by the number of threads to dedicate to the TLP already available in the original program. We found natural to express all of these using TI and SDI.

Each of these aspects represent one dimension of the state space. A program configuration, therefore, corresponds to picking one value for each of these dimensions. STATS explores this space to find the most performant configuration, using the developer-provided training inputs.

3.4 Compilers and Runtime

STATS includes three compilers called the front-end, the middle-end, and the back-end compilers.

Generating standard C++ code. The front-end compiler translates C++ with the SDI and TI extensions to standard C++ code which includes a description of the tradeoffs. Figure 11 shows the code generated from Figures 8 and 10, which gets `#included` by all source files. Each tradeoff is described with an entry in the array (`TO`), which includes the name of the C++ functions generated from the relevant TI (e.g., `T_42_size`), and the name of the function used as a placeholder for a tradeoff value (e.g., `T_42`). [2]

Generating IR with auxiliary code. The middle-end compiler translates the C++ code generated by the front-end to LLVM IR extended with extra metadata, which encodes the information in the extra header file generated by the front-end. This solution is inspired by the DotNET compilation framework, which encodes source level information in metadata tables included in CIL bytecode files [29]. This is implemented as a new compilation pass in *clang*.

After translating C++ code to the IR, and before producing its output, the middle-end compiler generates auxiliary code. For each state dependence d, the middle-end compiler clones d's `computeOutput()` (c.f., Figure 8) and links it to d's metadata entry. The compiler also clones the included tradeoffs (to distinguish them from the original ones) by creating new entries (one per cloned tradeoff) in the metadata. Cloning tradeoffs allows STATS to control the quality of the auxiliary code's results independently from the rest of the code.

Finally, the middle-end sets the tradeoffs that are outside auxiliary code to their default value, by scanning the tradeoff descriptions in the metadata, then deletes their metadata entries. The resulting IR is the middle-end's output, which includes only tradeoffs that are part of auxiliary code.

Generating a binary. The back-end compiler takes as input the IR generated by the middle-end and a configuration (from the autotuner) in the state space. This configuration lists the state dependences to be satisfied using auxiliary code and how to set their tradeoffs. The back-end compiler uses the following algorithm for each state dependence. First, it reads the metadata to find the auxiliary code specific to the current state dependence as well as its related runtime (described next), then links them. Second, it sets the tradeoffs left in the IR based on their index in the input state space configuration.

[2] These names are generated to avoid conflicts with the rest of the code.

```
#pragma once                                                    1
int64_t T_42 (int64_t p) { return p;}                           2
#define TO_numAnnealingLayers T_42(42)                          3
char *TO[] = { "T_42_getValue T_42_size                         4
                T_42_getDefaultIndex T_42" }                    5
auto T_42_getValue (int64_t i){ return i+1; }                   6
int64_t T_42_size() { return 10;}                               7
int64_t T_42_getDefaultIndex() { return 8;}                     8
```

Figure 11. C++ code generated by the front-end compiler from Figures 8 and 10.

Setting a tradeoff. Setting a tradeoff t requires two compile-time steps: fetching the value v identified by an index i, and setting references of t to v.

We rely on LLVM's dynamic compiler for the former. We generate machine code from the IR code of the function `getValue()` related to t, then invoke it with input i. Finally, we store v and its type for the next step.

A tradeoff reference (e.g., `TO_numAnnealingLayers`, line 10 of Figure 8) is set to a value v depending on the tradeoff type of v. If v is a constant (e.g., number of layers in `body-track`), the tradeoff reference is a call to a placeholder (e.g., `T_42()`); setting this tradeoff replaces that call with the constant v. If v is a type (e.g., float), setting a tradeoff changes the type of the related variable accordingly. When needed, extra casts are added according to the variable's uses. Finally, if v refers to a function (e.g., a specific implementation of `sqrt`), a tradeoff reference is a call to a placeholder function; setting this tradeoff replaces its callee with v.

Runtime. The execution of code that leverages state dependences relies on the STATS runtime. Its main goal is to implement efficiently the execution model described in Section 3.1. To do so, it includes low-level implementations of thread synchronization primitives. It also includes an efficient thread pool implementation (shared with all state dependences) to minimize thread creation overhead.

Design choices. Next we describe the main compiler-related design choices we made.

We divided the translation from the extended C++ language to the IR in two compilers (front-end and middle-end) for engineering reasons. We preferred to avoid adding complexity to the already-complex C++ parser in *clang*. Note also that C++ is a moving target (C++11, 14, 17); modifying the mainline parser would also introduce maintenance costs. Our solution does not modify the *clang* C++ parser and avoids these extra costs. The middle-end compiler uses the unmodified parser. Finally, the front-end compiler only needs to partially parse C++ programs, which made it possible to use a simple implementation based on Racket [30].

We decoupled the generation of the IR code that describes the state space (middle-end) from instantiation of a given configuration (back-end) to reduce the overall compilation time. As it evaluates the state space, the autotuner must instantiate the same IR to multiple configurations, which makes it necessary for instantiation to be efficient. We achieve this by leaving only simple code changes to the back-end.

The middle-end performs deep cloning of the function `computeOutput()` of a state dependence. It balances the amount of extra code generated (lower is better) with the number of degrees of freedom (i.e., number of tradeoffs cloned) available in auxiliary code (higher is better). In more detail, it clones functions reachable by `computeOutput()` only if they, or some of their callees, include a tradeoff (found using a bottom-up analysis of the call graph). The middle-end stops cloning when it reaches a maximum number of instructions per `computeOutput()`.

3.5 Autotuner

The goal of our autotuner is to find a performant (or energy efficient) configuration for the developer-provided training inputs. The state space is composed, on average, of 1.3 million points in our benchmarks, which makes exhaustive exploration impossible. Therefore, we use OpenTuner 0.7 [6] to explore this space using a set of statistical analyses already available in this framework. We describe each tradeoff in OpenTuner extending its class "IntegerParamsTuner" as the values of a tradeoff can always be enumerated.

4 Evaluation

Our evaluation tests the hypothesis behind our work: state dependences can be satisfied with carefully-generated auxiliary code creating additional TLP. Next we show that this additional TLP generates significant performance and energy efficiency improvements. We also compare to related approaches; thanks to the generation of auxiliary code, STATS is the only approach that gains performance while preserving output quality for complex benchmarks. We also relate the benefits obtained by STATS with the number of tradeoffs encoded by a developer. We show that most benefits are already obtained with only two tradeoffs, which suggests developers gain most of the benefits with a minimum effort. Finally, we show that only a small fraction of performance improvements is lost if the training inputs are not representative of the ones used in production.

4.1 Experimental Setup

Platform. 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 6.7 (kernel 2.6.32-573.18.1), with no CPU frequency governors enabled (all cores run at maximum frequency). Hyper-Threading is turned off for all experiments unless explicitly specified. Moreover, Turbo Boost is disabled. We evaluate the energy consumption using a Watts Up Pro energy monitor measuring the (120 V / 60 Hz) AC-side total system power consumption at 1-second

Benchmark	Original LOC	State dependences	Lines of code modified/added per tradeoff									LOC for the state comparison code	LOC generated by compilers	Binary size increase	Extra committed x86_64 instructions
			1	2	3	4	5	6	7	8	9				
swaptions	1120	1	10 / 15	20 / 120	3/9	3/9						< 5	45066	715%	3.4%
streamclassifier	1770	2	70 / 180	10 / 20	60 / 130	0 / 15	0 / 15	0 / 15	0 / 15			< 5	62414	1073%	< 0.1%
streamcluster	1770	2	80 / 215	10 / 20	60 / 174	0 / 15	0 / 15	0 / 15	0 / 15			< 5	62969	1076%	< 0.1%
fluidanimate	4350	1	5 / 10	5 / 10	100 / 130	0 / 10	0 / 30	0 / 10	0 / 15	0 / 10	0 / 10	< 5	61619	880%	< 0.1%
bodytrack	16430	1	60 / 95	5 / 10	0 / 15	0 / 10	0 / 10					19	87844	50%	7.1%
facedet	606472	1	70 / 150	5 / 10	5 / 10	3 / 10	0 / 10	0 / 10				29	44993	189%	1.4%

Table 1. Most code changes required to take advantage of static dependences are automatically performed by STATS compilers. The lines of code (LOC) modified/added by a developer through the STATS interface is negligible compared to the ones automatically generated by STATS compilers. Moreover, the auxiliary code and the STATS runtime add only a small amount of extra instructions at run-time ($\leq$ 7.1%).

intervals. STATS is built on top of LLVM 3.9.1 [48], Racket 6.8 [30], and OpenTuner 0.7 [6].

Statistics and 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 mean.

4.2 Benchmarks

We considered the POSIX multi-threaded versions of the PARSEC version 3.0 benchmarks as well as their sequential version. The only benchmarks we could not consider are `vips` and `dedup` because they did not compile using the vanilla *clang* compiler. Moreover, the binary generated by *clang* for `ferret` produced incorrect outputs. We considered only the remaining benchmarks that exhibit nondeterminism: `bodytrack`, `canneal`, `fluidanimate`, `swaptions`, and two variants of `streamcluster` (clustering, called `streamcluster`, and classification, which we called `streamclassifier`). Moreover, to test STATS in a large codebase, we considered OpenCV [13] for detecting faces in a video stream (`facedet`). Out of these benchmarks, we could not find a state dependence that STATS can target only in `canneal` and, as our technique does not apply, we do not consider it in the rest of this section. In more detail, STATS needs to know the number of inputs that the code pattern of Figure 4 has to process at run time just before the first invocation of this code pattern. This information is unfortunately unavailable in the `canneal` benchmark: the number of inputs depends on the evolution of the computation state.

Inputs. We used the native inputs provided by the PARSEC suite for our evaluation. 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 [46]; we thus extended the native inputs in the same fashion. `swaptions`, on the other hand, has native inputs large enough to show performance bottlenecks only after 128 cores. Therefore, we decreased the swaptions inputs (34 swaptions rather than 128) to allow bottlenecks in the program to manifest that would otherwise have remained hidden. For `streamclassifier`, we used the inputs from [72]. For `facedet`, we used a 40 seconds video of a person moving in front of a camera. Finally, we used a fraction of the evaluation inputs to compile our benchmarks.

Output quality. We used well-known domain-specific output quality metrics to measure output variability. These metrics (next described) were computed against an oracle. The oracle was computed using a benchmark version generated by setting its tradeoffs to maximize output quality. The generated output is significantly more accurate than the output of the (significantly faster) unmodified benchmark versions.

`bodytrack`'s metric is the relative mean square error of the body parts vectors [58]. `fluidanimate`'s metric is the average Euclidean distance between the position of the particles. `streamcluster`'s metric is the difference of the Davies-Bouldin indices of the clusterings [27]. `streamclassifier`'s metric is the difference in B^3 metrics [58]. `swaptions` uses the average relative difference between the prices generated [38]. `facedet` uses the average Euclidean distance between the faces detected.

Nondeterminism. While the actual programs from which the PARSEC benchmarks are drawn are nondeterministic, some of them have been made deterministic to facilitate experiments. [3] This was accomplished via the use of pseudo random value generators (PRVG) with constant and predefined seeds. Therefore, the outputs of such generators are deterministic and constant across runs with the same inputs. To properly study the effect of nondeterminism in these programs, we restored the use of PRVGs with random seeds as it is done in a real scenario. We also adapted the benchmarks to use the STATS interface.

State dependences, tradeoffs, and state comparison methods. We now describe the state dependences we found, the tradeoffs we encoded in auxiliary code, and the state comparison functions we implemented for every benchmark. The tradeoffs described next do not include the number of original threads and the number of threads to use for state dependences, which all benchmarks naturally have.

`bodytrack` accesses a model of the location of human body parts in a frame, updates this model with the results for the current frame, and passes it to the computation for the next frame. Frame i thus depends on the model update of frame $i-1$, which serializes the execution. The state is the model of the human body in the 3D space, which includes the position of the body parts. The state dependence is on the updates of this model. Tradeoffs are the number of simulated annealing layers, the data type (and therefore precision) of one variable used for this simulation, and the number of particles. The

[3] This is common practice, for result reproducibility reasons.

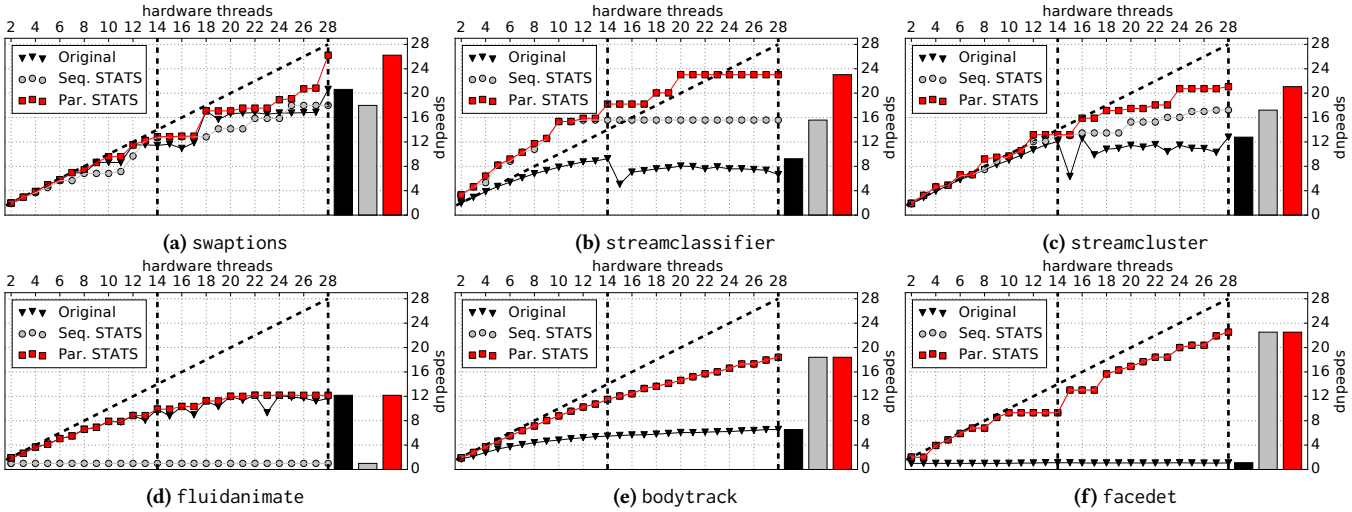


Figure 12. For most benchmarks, STATS generates a significant amount of extra parallelism that saturates the hardware resources of our platform. "Original" is the out-of-the-box benchmark that has been parallelized by traditional means. "Seq. STATS" ("Par. STATS") is the binary generated by STATS starting from the sequential (multi-threaded) version of a benchmark. The bar graphs show maximum speedup.

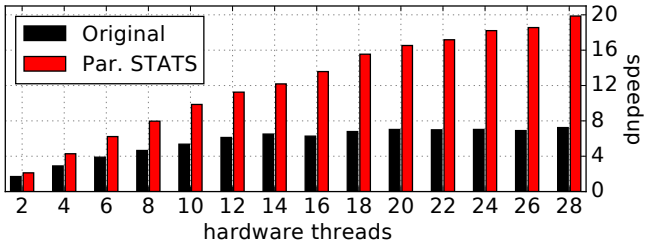


Figure 13. Geometric mean of speedups shown in Figure 12.

state comparison function computes the distances of the speculative state with the given set of original states, and the distances among all the original states. The distance measure we use is the sum of the absolute differences of every body part position between two states. If the distance of the speculative state `S'` with an original state `S` is less or equal the distance of another original state and `S`, then we consider the speculative state as valid and commit the results of the auxiliary code computation. In other words, if the body positions encoded in `S'` are between (in the 3D space) two original states, then we accept and commit `S'`.

`fluidanimate` simulates a fluid in time frames. The state is the condition of the fluid during the simulation (i.e., the position and velocity of the particles that compose the fluid). The state dependence is on the updates of the fluid condition between frames. Tradeoffs are the version of `sqrt` (different accuracies for different versions), the data type for three variables used for the simulation, and the x, y, and z dimensions of the per-thread prism where the simulation happens. The state comparison function behaves like the `bodytrack` one, but the distance measure is the average Euclidean distance among the position of the particles.

`facedet` updates the position of the detected faces at each frame. To do so, it takes advantage of the position of the faces found in the previous frame by applying a randomized particle filter. This create a dependence where the state is the position of the human face on a frame. Tradeoffs are the number of particles and the number of times Gaussian noise is added to the particles. The state comparison function operates as described in the previous benchmarks, but the distance measure is the average Euclidean distance of the four points of the box that contains the person's face.

`streamcluster` and `streamclassifier` consider adding the candidate centroids one by one depending on the status of the current solution. They update the current solution if the current centroid is added; these updates serialize the execution. The state dependence is on updating the status of the current solution. Tradeoffs are the data type of three variables used to estimate the quality of the current solution, and both the maximum and minimum number of clusters.

`swaptions` executes Monte Carlo simulations for each swaption. The simulation of a swaption is performed sequentially. The state dependence is on updating the price of a swaption during the simulation. Tradeoffs are the data type of two values used during the Monte Carlo simulation.

These last three benchmarks do not require a state comparison function because, by construction of the state dependence, the speculative state could have already been generated by an execution of the original program.

4.3 Taking Advantage of State Dependences

Exploiting multiple cores. Satisfying state dependences with auxiliary code liberates important additional TLP. Figure 12 compares the scalability and peak speedup of three approaches to parallelizing the benchmarks. The first, "Original", is the out-of-the-box benchmark that has been parallelized by traditional means. The second, "Seq. STATS", uses only the TLP obtained by satisfying state dependences with auxiliary code. The third, "Par. STATS", combines these two

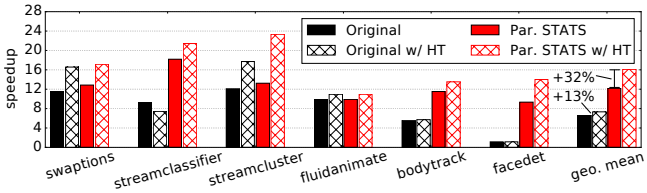


Figure 14. The performance obtained by STATS using a single socket with Hyper-Threading is constrained by hardware resources and not by low TLP.

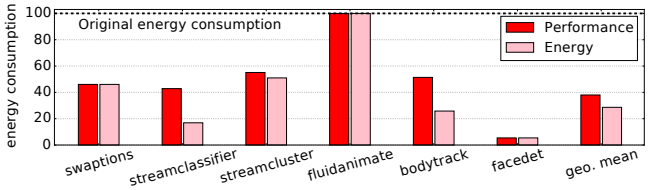


Figure 15. The binaries generated by STATS use considerably less energy compared to the original benchmarks.

sources of TLP by performing a state space search for a number of cores, the default mode of operation for STATS. On the left is the speedup graph, while on the right, the adjoining bar graph compares the maximum speedups of the three approaches. All speedup values were computed using the single-threaded version of the out-of-the-box benchmark as baseline. Figure 12 shows that taking advantage of state dependences doubles the performance of the considered benchmarks (the geometric mean speedup increases from 7.75× to 20.01×) on a 28 core platform. This empirically supports our hypothesis: state dependences can be satisfied with auxiliary code.

Both sources of TLP ("Original" and "Seq. STATS") are important. Figure 12 shows that, with the only exception of `bodytrack` and `facedet`, none of the two sources of TLP is enough to fulfill the parallelism requirements alone. It is necessary, instead, to properly combine them considering, therefore, the state space. Our work is the first to do so.

`swaptions` and `bodytrack` exhibit interesting behavior. In the former, at low core counts, Seq. STATS underperforms the original code. At 10 cores, the original achieves a respectable 8.7× speedup, while Seq. STATS achieves only 6.8×. Par. STATS, on the other hand, does not suffer from this drawback and produced a version of `swaptions` that outperforms the other two. This indicates that considering both sources of TLP is necessary. In `bodytrack`, on the other hand, the TLP generated by satisfying state dependences with auxiliary code generates higher performance than the original TLP. This is because the latter requires more frequent inter-thread synchronizations creating a bottleneck that the former does not have. While this was the case for our platform, we expect STATS to combine both TLPs when more cores are available.

The original parallelism available in `facedet` is used to aggressively vectorize the code (performed for the baseline as well). When possible, vectorization is preferred compared to TLP because it is more energy efficient. A significant amount of TLP is extracted from `facedet` by STATS thanks to its state dependence. Combining the aggressive vectorization performed in the original code and the significant TLP extracted by STATS led to a highly performant code.

STATS obtains speedups higher than the number of cores for `streamclassifier` (Figure 12b) from 2 to 22 cores as well as for `streamcluster` (Figure 12c) for 6, 8, and 12 cores. This is because of the following two effects. First, the threads generated by STATS better take advantage of the multiple L1s of the multiple cores; instead, the original multi-threaded code distributes the computation differently leading to a worse L1 hit rate. Second, the state dependences of these benchmarks are in a loop that ends when the current clustering solution is above a threshold. Satisfying these state dependences with auxiliary code leads both benchmarks to consider the potential centroids that compose a solution in a different order. This led the program to converge to the final solution faster.

Finally, `fluidanimate` (Figure 12d) shows little/no improvement with STATS. The auxiliary code for this benchmark almost always aborted at profiling time leading the STATS autotuner to prefer the original TLP rather than the one generated by state dependences. This is because `fluidanimate` is the only benchmark we considered where the state that the auxiliary code needs to generate requires all previous inputs (the result of a simulation of a fluid at time t requires the simulation of all previous time steps).

Exploiting Intel Hyper-Threading (HT). To study the impact of HT on STATS binaries, we constrained their execution to stay within a single socket of our platform. Figure 14 shows the extra performance obtained by STATS using HT.

We consider the additional speedup obtained by STATS using HT to be a success. The speedup (geometric mean) increased from 12.18× to 16.13×. Due to sharing computational and storage resources, Intel suggests that a successful use of HT should generate an extra performance of 30% [19, 82]. STATS obtained a 32% performance improvement. Hence, the performance obtained by STATS is constrained by hardware resources and not by low TLP.

The multi-socket effect. `fluidanimate`, `swaptions`, and `streamcluster` exhibited near-linear speedup within a single socket. STATS continues to improve performance on two sockets, but sub-linearly. An Intel VTune analysis demonstrated that this is due to the NUMA memory system—a common problem whose known solutions [11, 50, 62, 74, 75] apply to STATS, but go beyond the scope of this paper.

In more detail, when an application uses a single socket, the system tends to allocate memory pages served by the memory controllers within the chip, exhibiting low memory latency. However, when the application is spread over two sockets, memory references often have to cross from one socket to another to get to the relevant memory controller. This increases the latency for memory accesses and obstructs further performance improvements. Nonetheless, STATS continues to deliver increasing performance.

Saving energy. So far we have used STATS only to decrease the execution time. STATS can also be used to decrease the

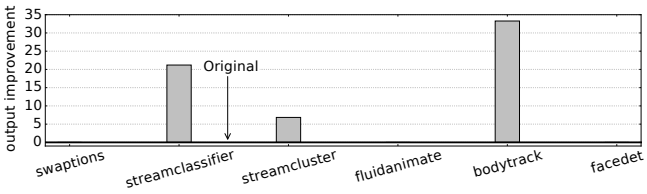


Figure 16. STATS can increase the original output quality by spending the saved time to iterate more over the same dataset.

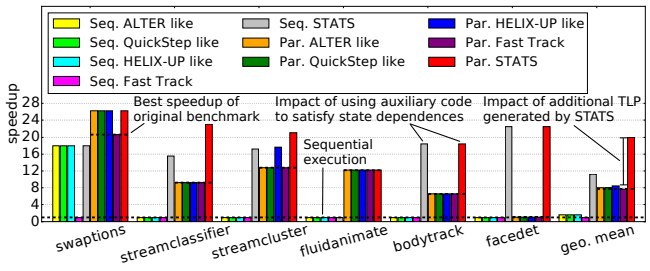


Figure 17. Only STATS takes advantage of non-trivial state dependences: they require the auxiliary code only STATS generates.

energy consumption. In this case, STATS autotuner minimizes energy rather than time leading to a different binary. To compare the energy reduction in these two operating modes, we used two sockets of our platform.

Figure 15 compares the system-wide energy consumption obtained in these two modes relative to the energy consumed by the peak-performing original version. When targeting time, STATS saves 61.98% of the baseline energy as a direct result of finishing the execution earlier. Moreover, STATS saves even more energy (71.35%) in energy mode by avoiding using extra cores if the additional performance obtained by them is not significant.

Improving output quality. For nondeterministic applications where speed of computation or energy is of utmost importance, the developer may decide to use STATS as described so far. However, for applications where quality matters most, STATS can also be used to improve the output. By making the computation several times faster than the original, STATS allows the application to spend the saved time to iterate more over the same dataset, thereby increasing the final output's quality. Figure 16 shows the quality improvements from running the STATS versions for the same amount of time as the original versions. Three benchmarks show quality increases from 6.84× to 33.27×.

4.4 STATS and its Related Work

We implemented related approaches able to target the considered benchmarks on our infrastructure and configured them to target only the state dependences we identified. Both prior work and STATS can generate TLP starting from both sequential and multi-threaded versions of a program. We applied them to both versions, exploring their configurations (e.g., dependences to break, how to break them) and kept the highest speedups obtained without exceeding the original output variability (Figure 2). Figure 17 shows the results that we obtained.

Prior approaches were only able to take advantage of the state dependence in `swaptions`. Its producer and consumer are single instructions and the state (a register) is implicitly cloned by running them on different cores. Every other state dependence has larger producers and consumers and their states must be explicitly cloned. They also require auxiliary code to preserve output quality. No prior work either explicitly clones the state of actual producer-consumer dependences or produces auxiliary code. Hence, only STATS is able to take advantage of non-trivial state dependences (Figure 17).

The "ALTER like" approach [81] breaks dependences to execute loop iterations out-of-order with optional stale reads. It also exploits reduction variables whose values at the end of the broken-dependence computation are guaranteed to be the same as those produced by a serial execution. In our case, these variables represent the state of the parallel computation when a dependence is broken. The reduction variables can only be updated using a limited number of operators and the update instruction must be of the form: *variable = variable operator value*. `swaptions` is the only benchmark we considered where "ALTER like" was applicable. All state dependences of the other benchmarks have more complicated states (i.e., complex data structures and objects with methods) and update operations on the state variables for the "ALTER like" approach to be applicable.

Both "QuickStep like" [57] and "HELIX-UP like" [16] broke several state dependences. They improved performance only for `swaptions`; other benchmarks require both state cloning and auxiliary code (not generated by either technique) to preserve output quality.

"Fast Track" [44] applied code transformations that broke state dependences speculating no changes in the final state. Its runtime evaluates this speculation comparing the so-generated final state with the (single) unspeculative state loosing, therefore, the opportunity created by the nondeterminism of the original code that could have created (multiple) different unspeculative states. For these reasons, "Fast Track" always aborted its speculations in our experiments.

4.5 Developer Effort

Identifying and encoding tradeoffs requires developer effort, but we consider the amount of work reasonable for two reasons. First, the number of lines of code (LOC) edited when encoding a tradeoff is reasonably low. Table 1 shows, for each benchmark, the LOC in the original program and the LOC modified and added for each tradeoff.

Second, our approach yields benefits even when we encode only a subset of the tradeoffs we identified, which suggests that our approach is "pay as you go". Figure 18 shows the geometric mean of speedups as additional tradeoffs are encoded. The first point after 0 is the mean speedup across all benchmarks after encoding one tradeoff for each of them, the second, two, and so on. We picked the orderings of tradeoffs starting with the ones for which we expected the highest

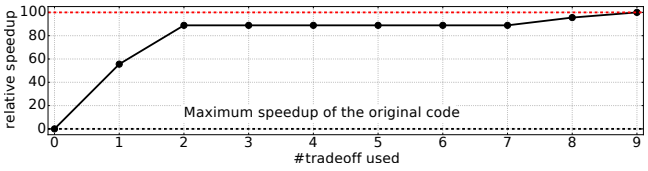


Figure 18. Developers gain most of the STATS benefits with a minimum effort (by encoding only two tradeoffs). This figure shows the average performance (geometric mean) relative to the best STATS speedup, by number of tradeoffs encoded.

payoff; just as a developer using STATS would. The orderings in Figure 18 correspond to the ones in Table 1. On average, encoding a single tradeoff yields around 55% of the speedup of encoding all, and encoding two yields around 95%.

For all benchmarks considered, the first two tradeoffs that yield most benefits are the most obvious ones to target. In other words, it is unlikely that a reasonable developer would encode the third (or next ones) tradeoffs before the first two.

4.6 Non-representative Inputs

STATS relies on training inputs at compile time. The representativeness of these inputs, however, only affects performance; correctness is guaranteed by the STATS runtime.

When its training inputs are not representative, STATS looses only a small fraction of the performance obtained when representative inputs are used. To estimate the loss in performance from non-representative training inputs, we generated non-representative training data for each benchmark. Specifically, the subject does not move across quadruples for `bodytrack`, points overlap in the multidimensional space for both `streamcluster` and `streamclassifier`, unrealistic swaption parameters like market strikes and maturity dates for `swaptions`, the detected face in `facedet` does not move. We used these as training inputs and tested the resulting binaries using the same evaluation inputs used in the previous experiments. Figure 19 shows the lowest performance among the binaries generated by STATS using the least-representative training inputs we could find.

4.7 Auto-tuning in STATS

The autotuner consistently and rapidly converges to the best program. Figure 20 shows that evaluating 88 configurations (less than 1%) is sufficient to find the best binary (in less than 20 minutes on our platform), which is used in Figure 12 for 28 cores. 2,000 additional evaluations (and 15 hours of additional time in our platform) did not improve it. The autotuner uses nondeterminism for better exploration; different searches for the same program may find different best configurations. Figure 20 shows that the variance in best speedups disappears after exploring 46 configurations.

4.8 When STATS Should Be Used

Invocation i of `computeOutput()` of Figure 4 depends on the previous invocation $i-1$. This generates a chain of dependences from the first invocation to the last one. We observed that some nondeterministic computations have the following

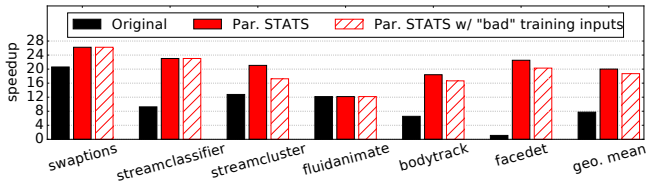


Figure 19. STATS looses only a small amount of performance when not representative inputs are used.

property: an invocation of `computeOutput()` requires only a few previous invocations to generate a correct output. In other words, the computation converges to a correct state after processing a window of inputs that starts in the middle of this dependence chain. The auxiliary code is responsible to converge to a correct state.

The computation performed by the `bodytrack` benchmark, for example, has this property. The position of a human body at quadruple i can be computed by detecting where the body was in the last few quadruples (rather than all previous ones). We found that some computations, however, do not have the property required by STATS. For example, the main state dependence we found in `fluidanimate` does not have this property — the simulation of a fluid at instant i requires the simulation of it in all previous instants. This is perhaps not surprising given the properties of the Navier-Stokes equations underlying `fluidanimate`'s model [9].

We included `fluidanimate` to test the limits of STATS. We wanted to see what happen when a developer uses the SDI interface to describe a state dependence that does not have the property STATS needs. Results show that the STATS autotuner empirically finds that every time the main state dependence of `fluidanimate` was satisfied with auxiliary code, the STATS runtime aborted the speculative execution. Hence, the STATS autotuner chose a configuration where such dependence is always satisfied with the original code (rather than with the auxiliary code).

More broadly, we believe time-step simulations like `fluidanimate` are not good fit for STATS. A better fit for STATS are applications that analyze a long stream of data (e.g., `bodytrack`, `facedet`, `streamcluster`) where the information about inputs that is automatically computed (e.g., 3D location of bodies, 2D location of faces, centroids of multidimensional points) has the "short memory" dependence property described above.

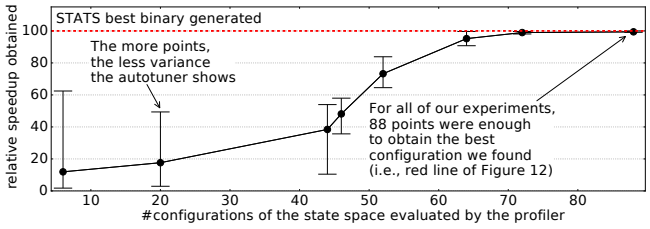


Figure 20. Average performance (geometric mean) of the final binary identified by the STATS autotuner after exploring a number of configurations.

5 Related Work

STATS is related to prior work that either extracts TLP from programs or uses search to optimize program configurations.

5.1 Extracting TLP

Automatic TLP extraction from sequential programs has a rich history, in which we identify two relevant categories.

TLP Extraction with Cost-reduced Actual Dependences
Earlier work addresses the cost of actual dependences by accelerating data exchanges or by avoiding some altogether.

Multiple techniques [12, 14, 15, 68, 70, 71, 86] attempt to reduce the cost of actual dependences by making them cheaper individually, while still preserving all of them. Such techniques include hardware support to accelerate data exchanges between threads running on parallel cores. While these techniques reduce the costs of *data transfer*, they still force *synchronization* between threads for all actual dependences. Our approach, instead, avoids the producer-consumer synchronizations related to state dependences altogether.

Some techniques break actual dependences [3, 16, 57, 66, 67, 81]. These approaches do not generate auxiliary code and they do not take advantage of developers' algorithm-specific knowledge. This limits their applicability to simple dependences and Figure 17 measures empirically this limit for some of them. One of these approaches generates compensation code [67], which is executed after the code involved in a dependence. While compensation code can avoid high inaccuracies, it does not preserve the output quality. Our approach generates auxiliary code, which is executed before the code involved in a dependence, taking advantage of algorithm-specific knowledge, which makes it more broadly applicable. STATS is the first system to do so.

Other approaches have been proposed that break dependences for specific class of algorithms [23, 55, 56, 83] These approaches do not generate auxiliary code because it is not required thanks to the characteristics of the specific class of algorithms they target. However, our benchmarks require auxiliary code to preserve the output quality.

Galois [47, 63] introduces TLP by optimistically assuming that ignoring an actual dependence will not lead to an invalid execution, then dynamically checks whether that is the case, and aborts the erroneous computation if not. This approach does not cover the state dependences we identified in the PARSEC benchmarks, which are not related to the kind of data-parallelism Galois targets.

Fast Track [44] generates TLP by creating an unsafe optimization of a program, which runs in parallel with the safely optimized code. The system checks whether the results of the unsafe execution match the results of the safe one. This technique does not take advantage of the nondeterminism of a program. It does not compute multiple results to increase the probability of a match.

TLP Extraction With Complete Dependence Preservation
Approaches that preserve all dependences can be considered along two axes: speculative/not, and manual/automatic.

Automatic non-speculative approaches: The many approaches in this category [2, 17, 18, 21, 22, 28, 39, 40, 51, 53, 59, 65, 80, 88] all rely on accurate data dependence analyses to identify code regions that can run safely in parallel. These systems preserve all the dependences they find. In contrast, our work relies on algorithm-specific knowledge provided by developers to satisfy actual dependences with auxiliary code. Moreover, STATS automatically combines the TLP that arises from state dependences with that already present in the program, leading to more TLP overall.

Automatic speculation-based approaches: Several parallelizing compilers rely on thread-level speculation techniques to reduce the cost of dependences that turn out to be false at run time [1, 35, 36, 43, 52, 64, 76, 77, 85, 89, 90]. These approaches, while effective, only address the cost of *apparent* dependences—not the cost of *actual* dependences, as we do in this work. Finally, some techniques speculate on data values [32, 33]. However, they do not rely on algorithm-specific knowledge and are limited to simple data dependences of scalar values.

Manual approaches: In many multi-threaded programs (including those of PARSEC), TLP has been introduced manually using parallel programming APIs [26, 60, 78]. These programs preserve all Read-After-Write actual dependences (including state dependences of Figure 4), which constrains TLP and overall program performance (as shown by the black lines of Figure 12). STATS goes beyond this limit.

5.2 Autotuning/Search-based Optimization

Considerable effort has gone into the general area of autotuning. A number of systems focus on tuning libraries in specific domains [4, 10, 31, 41, 45, 84, 87]. Others are designed as general auto-tuning frameworks [5–7, 24, 61]. The STATS autotuner is built on top of the most recent one, OpenTuner, and it is used for the specific task performed by STATS, i.e., combining the original TLP with the one generated by targeting state dependences.

6 Conclusion

Actual dependences have been either satisfied or broken by prior work. This paper proposes a middle ground for nondeterministic programs: satisfying state dependences with auxiliary code. This work is the first step in exploiting state dependences. It demonstrates that it is possible to achieve large performance gains, energy savings, or output quality increases by doing so within a prototype system. We have focused on using state dependences to optimize a particular code pattern that is common within the benchmarks we considered. More generally, we believe that actual dependences should be studied more carefully by our community to find other subsets that might yield important benefits.

Acknowledgments

This project is made possible by support from the US NSF through grants CCF-1453853, CCF-1533560, and the Department of Energy's Sandia National Laboratories through the Hobbes Project.

References

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

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

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

[4] Ayaz Ali, Lennart Johnsson, and Jaspal Subhlok. 2007. Scheduling FFT Computation on SMP and Multicore Systems. In *International Conference on Supercomputing (ICS)*.

[5] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. 2009. PetaBricks: A Language and Compiler for Algorithmic Choice. In *Programming Language Design and Implementation (PLDI)*.

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

[7] Jason Ansel, Yee Lok Wong, Cy Chan, Marek Olszewski, Alan Edelman, and Saman Amarasinghe. 2011. Language and Compiler Support for Auto-tuning Variable-accuracy Algorithms. In *Code Generation and Optimization (CGO)*.

[8] Christian Bienia. 2011. *Benchmarking Modern Multiprocessors*. Ph.D. Dissertation. Princeton University.

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

[10] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. 1997. Optimizing Matrix Multiply Using PHiPAC: A Portable, High-performance, ANSI C Coding Methodology. In *International Conference on Supercomputing (ICS)*.

[11] Sergey Blagodurov, Sergey Zhuravlev, Alexandra Fedorova, and Ali Kamali. 2010. A case for NUMA-aware contention management on multicore systems. In *Parallel Architectures and Compilation Techniques (PACT)*.

[12] Shekhar Borkar, Robert Cohn, George Cox, Sha Gleason, Thomas Gross, H. T. Kung, Monica Lam, Brian Moore, Craig Peterson, John Pieper, Linda Rankin, P. S. Tseng, Jim Sutton, John Urbanski, and Jon Webb. 1988. iWarp: An Integrated Solution to High-Speed Parallel Computing. In *International Conference on Supercomputing (ICS)*.

[13] Gary Bradski and Adrian Kaehler. 2008. *Learning OpenCV: Computer vision with the OpenCV library*. "O'Reilly Media, Inc.".

[14] Scott E. Breach, T. N. Vijaykumar, and Gurindar S. Sohi. 1994. The Anatomy of the Register File in a Multiscalar Processor. In *International Symposium on Microarchitecture (MICRO)*.

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

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

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

[18] S. Campanoni, T. M. Jones, G. Holloway, G. Y. Wei, and D. Brooks. 2012. HELIX: Making the Extraction of Thread-Level Parallelism Mainstream. In *International Symposium on Microarchitecture (MICRO)*.

[19] Shawn D. Casey. 2011. How to Determine the Effectiveness of Hyper-Threading Technology with an Application. https://goo.gl/ycuL6E. (2011). Accessed: 2018-01-14.

[20] Shailender Chaudhry, Robert Cypher, Magnus Ekman, Martin Karlsson, Anders Landin, Sherman Yip, Håkan Zeffer, and Marc Tremblay. 2009. Rock: A high-performance sparc cmt processor. In *International Symposium on Microarchitecture (MICRO)*.

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

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

[23] Trishul M Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. 2014. Project Adam: Building an Efficient and Scalable Deep Learning Training System.. In *Operating Systems Design and Implementation (OSDI)*.

[24] Cristian Ţăpuş, I-Hsin Chung, and Jeffrey K. Hollingsworth. 2002. Active Harmony: Towards Automated Performance Tuning. In *Supercomputing Conference (SC)*.

[25] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)* (1991).

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

[27] D. L. Davies and D. W. Bouldin. 1979. A Cluster Separation Measure. In *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*.

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

[29] ECMA 2005. *Standard ECMA-335 Common Language Infrastructure (CLI)* (3rd ed.). ECMA, Rue du Rhone 114 CH-1204 Geneva. http://www.ecma-international.org/publications/standards/Ecma-335.htm

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

[31] Matteo Frigo and Steven G. Johnson. 2005. The design and implementation of FFTW3. In *International Symposium on Microarchitecture (MICRO)*.

[32] Chao-Ying Fu, Matthew D Jennings, Sergei Y Larin, and Thomas M Conte. 1998. Value speculation scheduling for high performance processors. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[33] José González and Antonio González. 1998. The potential of data value speculation to boost ILP. In *International Conference on Supercomputing (ICS)*.

[34] P. Hammarlund, A. J. Martinez, A. A. Bajwa, D. L. Hill, E. Hallnor, H. Jiang, M. Dixon, M. Derr, M. Hunsaker, R. Kumar, R. B. Osborne, R. Rajwar, R. Singhal, R. D'Sa, R. Chappell, S. Kaushik, S. Chennupaty, S. Jourdan, S. Gunther, T. Piazza, and T. Burton. 2014. Haswell: The fourth-generation intel core processor. In *International Symposium on Microarchitecture (MICRO)*.

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

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

[37] R. Haring, M. Ohmacht, T. Fox, M. Gschwind, D. Satterfield, K. Sugavanam, P. Coteus, P. Heidelberger, M. Blumrich, R. Wisniewski, a. gara, G. Chiu, P. Boyle, N. Chist, and C. Kim. 2012. The IBM Blue Gene/Q Compute Chip. In *International Symposium on Microarchitecture (MICRO)*.

[38] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. 2011. Dynamic Knobs for Responsive Power-aware Computing. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[39] Jialu Huang, Arun Raman, Thomas B. Jablin, Yun Zhang, Tzu-Han Hung, and David I. August. 2010. Decoupled Software Pipelining

Creates Parallelization Opportunities. In *Code Generation and Optimization (CGO)*.

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

[41] Eun-Jin Im and Katherine A. Yelick. 2001. Optimizing Sparse Matrix Computations for Register Reuse in SPARSITY. In *International Conference on Computational Sciences (ICCS)*.

[42] Christian Jacobi, Timothy Slegel, and Dan Greiner. 2012. Transactional memory architecture and implementation for IBM System z. In *International Symposium on Microarchitecture (MICRO)*.

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

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

[45] C. Kessler and W. Löwe. 2012. Optimized Composition of Performance-aware Parallel Components. In *Concurr. Comput. : Pract. Exper.*

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

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

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

[49] Hung Q Le, GL Guthrie, DE Williams, Maged M Michael, BG Frey, William J Starke, Cathy May, Rei Odaira, and Takuya Nakaike. 2015. Transactional memory support in the IBM POWER8 processor. In *IBM Journal of Research and Development*.

[50] Baptiste Lepers, Vivien Quéma, and Alexandra Fedorova. 2015. Thread and Memory Placement on NUMA Systems: Asymmetry Matters.. In *USENIX Annual Technical Conference (USENIX ATC)*.

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

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

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

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

[55] Jiayuan Meng, Srimat Chakradhar, and Anand Raghunathan. 2009. Best-effort parallel execution framework for recognition and mining applications. In *International Symposium on Parallel and Distributed Processing (IPDPS)*.

[56] Jiayuan Meng, Anand Raghunathan, Srimat T. Chakradhar, and Surendra Byna. 2010. Exploiting the forgiving nature of applications for scalable parallel execution. In *International Symposium on Parallel and Distributed Processing (IPDPS)*.

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

[58] Sasa Misailovic, Stelios Sidiroglou, Henry Hoffmann, and Martin Rinard. 2010. Quality of Service Profiling. In *International Conference on Software Engineering (ICSE)*.

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

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

[61] Phitchaya Mangpo Phothilimthana, Jason Ansel, Jonathan Ragan-Kelley, and Saman Amarasinghe. 2013. Portable Performance on Heterogeneous Architectures. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[62] Guilherme Piccoli, Henrique N Santos, Raphael E Rodrigues, Christiane Pousa, Edson Borin, and Fernando M Quintão Pereira. 2014. Compiler support for selective page migration in NUMA architectures. In *Parallel Architectures and Compilation Techniques (PACT)*.

[63] 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. 2011. The Tao of Parallelism in Algorithms. In *Programming Language Design and Implementation (PLDI)*.

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

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

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

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

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

[69] Mehrzad Samadi, Janghaeng Lee, D Anoushe Jamshidi, Amir Hormati, and Scott Mahlke. 2013. Sage: Self-tuning approximation for graphics engines. In *International Symposium on Microarchitecture (MICRO)*.

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

[71] Steven L. Scott. 1996. Synchronization and Communication in the T3E Multiprocessor. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

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

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

[74] Sharanyan Srikanthan, Sandhya Dwarkadas, and Kai Shen. 2015. Data Sharing or Resource Contention: Toward Performance Transparency on Multicore Systems. In *USENIX Annual Technical Conference (USENIX ATC)*.

[75] Sharanyan Srikanthan, Sandhya Dwarkadas, and Kai Shen. 2016. Coherence stalls or latency tolerance: informed CPU scheduling for socket and core sharing. In *USENIX Annual Technical Conference (USENIX ATC)*.

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

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

[78] John E. Stone, David Gohara, and Guochun Shi. 2010. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. In *IEEE Des. Test.*

[79] Xin Sui, Andrew Lenharth, Donald S. Fussell, and Keshav Pingali. 2016. Proactive Control of Approximate Programs. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

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

[81] Abhishek Udupa, Kaushik Rajan, and William Thies. 2011. ALTER: Exploiting Breakable Dependences for Parallelization. In *Programming Language Design and Implementation (PLDI)*.

[82] Antonio Valles, M Gillespie, and G Drysdale. 2009. Performance Insights to Intel® Hyper-Threading Technology. http://software.intel.com/en-us/articles/performance-insights-to-intel-hyper-threading-technology. (2009). Accessed: 2017-07-01.

[83] Keval Vora, Sai Charan Koduru, and Rajiv Gupta. 2014. ASPIRE: Exploiting asynchronous parallelism in iterative algorithms using a relaxed consistency based DSM. In *Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

[84] Yevgen Voronenko, Frédéric de Mesmay, and Markus Püschel. 2009. Computer Generation of General Size Linear Transform Libraries. In *Code Generation and Optimization (CGO)*.

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

[86] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F. Brown, III, and Anant Agarwal. 2007. On-Chip Interconnection Architecture of the Tile Processor. In *International Symposium on Microarchitecture (MICRO)*.

[87] R. Clint Whaley and Jack J. Dongarra. 1998. Automatically Tuned Linear Algebra Software. In *Supercomputing Conference (SC)*.

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

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

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