NORTHWESTERN UNIVERSITY

Generating Thread-Level Parallelism in Nondeterministic Programs

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Field of Computer Science

By

Enrico Armenio Deiana

EVANSTON, ILLINOIS

November 2023

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 improve performance. Programming language abstractions are a way to generate TLP, which allows CMP to reach their full potential.

This dissertation focuses on two main contributions:

- STATS, a parallelizing compiler designed to extract TLP from nondeterministic programs by leveraging a novel programming language abstraction.
- CARMOT, a tool intended to aid programmers in effectively utilizing this innovative programming language abstraction.

TLP in today's programs is limited by data dependences that must be satisfied as the program executes. Nondeterministic programs suffer from this same limitation, but the nondeterminism gives them an additional degree of freedom that deterministic programs do not have: the ability to satisfy some dependences with many different data, which results in different outputs even when they run with the same input. Some of these outputs can be generated more quickly in parallel than others can.STATS is the first compiler to generate a new source of parallelism that has never been explored before, and it does so by taking advantage of this extra degree of freedom in nondeterministic programs. This resulted in STATS being able to achieve significant performance improvements in nondeterministic programs. To use STATS, developers have to express this additional degree of freedom in the code explicitly. STATS enables developers to encode this knowledge by extending the C++ language with a new abstraction.

While STATS allows developers to obtain significantly more performance, using the C++ abstraction we introduced can be challenging in large code-bases. To assist developers in using this

abstraction, we created a new tool called CARMOT. We found that CARMOT can also support developers in using many other programming language abstractions beyond the STATS abstraction. Hence, we developed an approach that generalizes the STATS-specific needs to reach many other modern programming language abstractions such as those offered by OpenMP pragmas and C++ features such as smart pointers.

Thesis Statement:

Thread-Level Parallelism of programs is limited by data dependences that must be satisfied in their intended sequential order to preserve program semantics. However, nondeterministic programs have an additional degree of freedom that deterministic programs do not have: the ability to satisfy some dependences with different data. We explore this additional degree of freedom and uncover a new subset of data dependences that can be satisfied in an alternative way, which generates Thread-Level Parallelism while preserving the semantics of nondeterministic programs.

ACKNOWLEDGEMENTS

I want to express my appreciation and profund gratitude to the following people who have played an essential role in my academic journey and the completion of my PhD.

To my advisor: Simone Campanoni, for his guidance, dedication, expertise, and mentorship throughout my research, for the opportunities he gave me, and the knowledge he shared.

To my thesis committee members and collaborators: Peter Dinda, Nikos Hardavellas, Robby Findler, Margo Seltzer, and Arthur "Barney" Maccabe, for their feedback, support, and insightful ideas.

To my friends and lab mates: Tommy McMichen, Brian Homerding, Yian Su, Federico Sossai, Atmn Patel, Nathan Greiner, David Dlott, Brian Suchy, Mike Wilkins, Nick Wanninger, Vijay Kandiah, Lukas Lazarek, Sanchit Kalhan, Madhav Suresh, Michalis Mamakos, and my Italian friends Ettore M. G. Trainiti and Simone Bianconi, for all the fun we had, and the beers at Sketchbook.

To my family, for their boundless love, encouragement, and sacrifices, and for supporting me throughout my studies for many, many, oh so many years.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1
INTRODUCTION

Since approximately 2004, the increase of sequential code performance has reached a plateau due to the end of Moore's law and Dennard scaling (Figure 1.1 shows the speedup of SPEC CINT benchmarks over the years). The scaling of CMOS technology has slowed down and the increase in power consumption is the main barrier to improving sequential code performance. To tackle this problem, single-core computer architectures evolved into multi-core architectures. Nowadays, multi-core systems are everywhere, from mobile systems to servers. But, to fully take advantage of these additional cores, programs need Thread-Level Parallelism (TLP). Unfortunately, most of today's workloads have low and non-scalable TLP because of their irregularity and complexity. This complexity comes from the data that need to be moved from and to different parts of a program, which creates a data dependence. Data dependences are the main obstacle that is blocking programs' TLP and making multi-core systems mostly underutilized. However, data dependences must be satisfied in their intended sequential order to preserve programs semantics.

In this dissertation, we focus on nondeterministic programs, which are a substantial subset of today's workloads. Nondeterministic programs also suffer from low and non-scalable TLP. These programs are widely used in many fields, from machine learning techniques (e.g., centroid selection in clustering, choice of initial weights in neural networks) to multimedia analysis (e.g., video analysis, image retrieval) and financial applications (e.g., Monte Carlo simulations for options pricing). Their nondeterminism is often used to avoid local minima, decrease the average computational complexity of algorithms, model stochastic variables, and more.

Nondeterministic programs naturally produce different outputs from run to run given the same

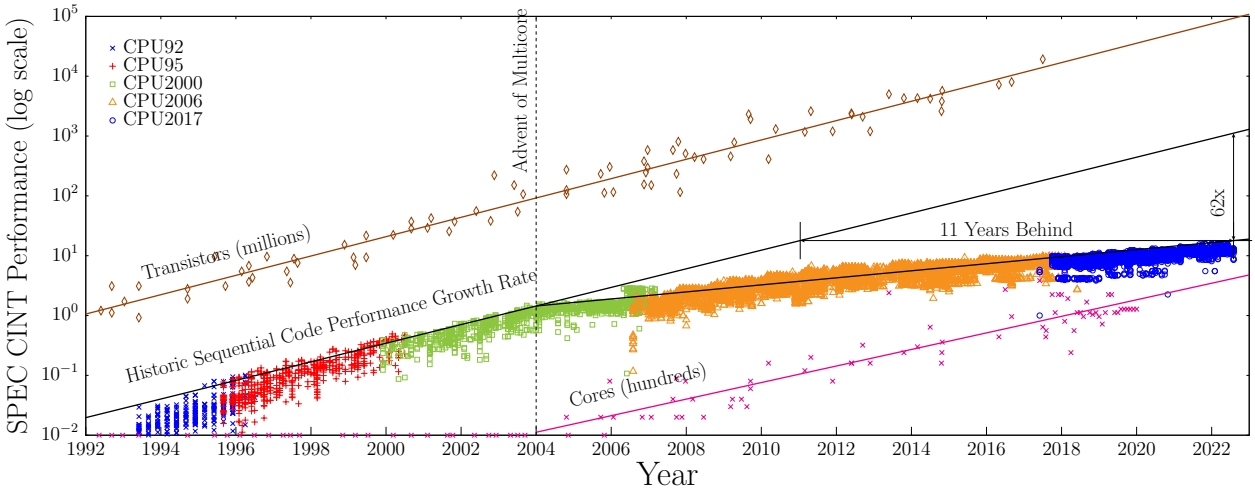


Figure 1.1: Sequential code performance has reached a plateau and it is now 62× (or 11 years) behind what it should have been. Furthermore, the performance improvements since approximately 2015 have not been significant.

input. This gives them an additional degree of freedom that deterministic programs do not have and that has never been exploited before. We see this as an opportunity, and we ask the question: Can we take advantage of the output variability of nondeterministic programs to liberate additional TLP while preserving the output quality? This dissertation answers this question by considering alternative ways of satisfying data dependences in nondeterministic programs.

We identify a subset of dependences in nondeterministic programs that can be satisfied via alternative, dependence-specific code. We take advantage of these dependences to generate TLP in nondeterministic programs following a parallel execution model that we implemented in a system we call *STATS (STAte Transition Speculator)*. STATS extends the C++ language with a programming language abstraction that developers use to provide information to the STATS system about the computational state and the nondeterminism of the program region that will be parallelized. Furthermore, we implemented a tool called *CARMOT (Compiler And Runtime Memory Observation Tool)* that aids developers in using the STATS programming language abstraction (and many

others) by collecting essential information about the computational state of a program. Next, we describe how STATS generates TLP in nondeterministic programs and how CARMOT assists developers in using the STATS programming language abstraction.

1.1 Generating TLP in Nondeterministic Programs with STATS

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, hence to preserve the semantics of a program and its output quality. To generate parallelism we must break dependences in order to make the producer of these data independent from the consumer. Once the producer is independent from the consumer, both of those parts of the program can run in parallel. However, we still need to preserve the semantics of the program. In other words, we need a way to generate the data that will be fed to the consumer, independently from the producer of those data. This is where nondeterminism becomes essential. In nondeterministic programs, the data that flow from a producer to a consumer change at every run of the program because of the nondeterminism in the computation. So, in nondeterministic programs we can generate any of the data that might have been generated by the nondeterministic producer. This gives us more chances to predict these data correctly and break the dependence between the original producer and the consumer.

We perform the prediction of the data that breaks the dependence between producer and consumer with our system *STATS (STAte Transition Speculator)*, a parallelizing compiler for nondeterministic programs. For STATS to work, the producer, the consumer, and the data exchange between them, need to be expressed explicitly in the source code of the program. This knowledge is program-specific and needs to be encoded by the developer. We enable developers to do so by

exposing a new programming language abstraction that interfaces with STATS.

1.2 Aiding Programmers in Using Programming Language Abstractions with CARMOT

Programming languages evolve to give programmers powerful abstractions that improve performance, energy savings, and code clarity. For example, the abstraction we introduced with STATS enables programmers to obtain additional performance by taking advantage of the multiple cores available in a chip. Unfortunately, programmers often struggle to use abstractions properly, which leads to performance and correctness issues. The difficulty in using abstractions lies in the implicit requirement that programmers must be omniscient regarding the behavior of the whole program. For example, in the STATS abstraction, programmers need to understand what data (i.e., variables and memory locations) flow from the producer to its consumer for the code region the abstraction is applied to. More generally, programmers need to understand how variables and memory locations evolve as the program executes, from the point of view of the target code region of an abstraction (e.g., a memory object is always written before being read in a code region) to properly use modern programming language abstractions. This can be a time-consuming and error-prone process.

Variables and memory locations form the state of a program. We refer to them as Program State Elements (PSEs). We observe that many abstractions rely on a common piece of information related to the access pattern of PSEs. Our approach studies this access pattern for the code region where the abstraction is to be applied. We define a new concept that summarizes the impact of this access pattern, which we call Program State Element Characterization (PSEC). PSEC describes: (a) which, where, and how PSEs are used in a code region, (b) how data of PSEs flows across code region boundaries, and (c) the reachability relationships between different PSEs. Intuitively, the PSEC of a code region assists programmers by formalizing their mental process when thinking about abstractions. PSEC presents this information to programmers in human-readable form by

reporting source code level information as instances of the target abstraction.

We are the first to automate PSEC. We do it with *CARMOT (Compiler And Runtime Memory Observation Tool)*, a compiler-runtime, co-designed tool that efficiently performs PSEC. A programmer invokes CARMOT with the abstraction they would like to use on a given code region (e.g., the STATS abstraction). CARMOT then generates abstraction recommendations by synthesizing an instance of the target abstraction using the PSEC of the target program.

1.3 Dissertation Contributions

The contributions of this dissertation are as follows.

- We show how we can take advantage of the nondeterminism of programs to generate TLP through our programming language abstraction offered by ouri STATS system (§3).
- We perform an in-depth analysis of STATS-generated overhead to understand its limiting factors from a performance point of view (§3.3).
- We observe that PSEC is the common information needed to use numerous abstractions correctly and at their full potential, and we illustrate how CARMOT performs PSEC efficiently to provide programmers with support for the STATS abstraction and many others (§4)).

Published work:

The STATS parallelizing compiler has been published at ASPLOS 2018 [1], our in-depth study of STATS-generated overhead has been published at ISPASS 2019 [2], while CARMOT has been published at CGO 2023 [3].

CHAPTER 2
BACKGROUND AND MOTIVATION

We begin with an introduction of the basic concepts that we will often refer to in this dissertation and that are in common between our parallelizing compiler STATS and our tool for programming language abstractions CARMOT. We first provide a definition of these concepts in §2.1, then we describe the opportunities that motivated STATS in §2.2 and CARMOT in §2.3.

2.1 Definitions

Source Code Region of Interest. In general, a code region is a delimited section of source code. In the context of this dissertation we limit the general definition by considering only single-entry single-exit code regions that are compound statements. In imperative languages such as C++, a compound statement is formed using scopes (or code-blocks), which are delimited by curly braces "{...}". Examples of compound statements are functions, *if-then-else* blocks, and loops (e.g., Figure 2.1).

Program State. The computational state of a program is comprised of all its variables and memory locations (i.e., globals, heap, and stack) and the data contained them. We call these components *Program State Elements (PSEs)* [3]. In the context of a code region, the program state of the region is the subset of PSEs that are used (i.e., read or written) in that region; we show an example in Figure 2.1.

Dependences. A dependence is a directed binary relation between two instructions [4].

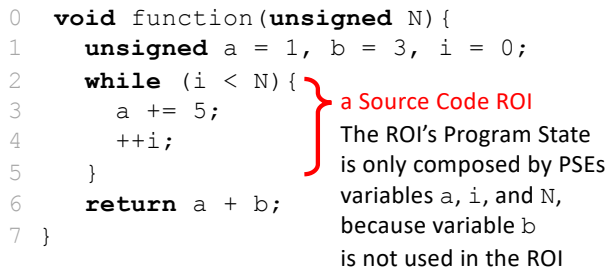


Figure 2.1: Example of a source code region of interest and its program state.

Dependences can be classified in two sets: *control dependences* and *data dependences*. Control dependences are related to the control-flow of a program (i.e., conditional branches such as *if-then-else* and *switch* statements, loops, *goto*, etc.). An instruction `i` control-depends on another instruction `j` if the outcome of `j` determines whether `i` will be executed or not. For example, in Figure 2.2 instruction `4` control-depends on instruction `3` because `4` will be executed depending on the outcome of `3`. On the other hand, data dependences are related to the data-flow of a program, which involves instructions that produce data and instructions that consume data. An instruction is data-dependent on another instruction if they both access the same data (e.g., the same variable or memory location) and at least one of them writes it. Data dependences can be further classified in three categories: *Read-After-Write (RAW)*, *Write-After-Read (WAR)*, and *Write-After-Write (WAW)* dependences. As the name suggests, a RAW dependence implies that an instruction first writes some data, and a subsequent instruction reads that same data. WAR and WAW dependences are defined similarly. Furthermore, when dependences can cross one or more loop iterations they are defined as *loop-carried* RAW, WAR, and WAW. Examples of RAW, WAR, and WAW dependences are shown in Figure 2.2.

Dependedences can also be orthogonally classified as *apparent dependences* or *actual dependences*. Apparent dependences are those that are not in fact necessary, and do not need to be

satisfied in order to correctly execute a program an preserve its semantics, however their existence could not be disproved. We show an example of apparent dependence in Figure 2.2, where we cannot prove that a RAW dependence between instructions `10` and `13` does not exist, even though intuitively it is clear that the condition at line `12` can never be true because of `11`. Conversely, all actual dependences in a program need to be satisfied for the semantics of a program to be preserved, and generally they need to be satisfied in the intended sequential order defined by the instructions of the program. An example of actual dependence is the RAW dependence shown in Figure 2.2, where instruction `6` is the producer of variable `a` (i.e., writes a value into `a`) and instruction `7` is the consumer of such variable (i.e., reads the value from `a`). We define *state dependences* as the subset of actual dependences that involve the update of the state of a code region and do not need to follow a sequential execution to be satisfied. More details on state dependences are provided in §2.2 and §3. We illustrate a taxonomy for this dependence classification in Figure 2.3.

Programming Language Abstractions. An abstraction is the representation of a concept or a process that has been simplified by removing unnecessary details. Programming language abstractions hide the complex details of computer architectures, allowing developers to produce more performant and easy to maintain code. In this dissertation, we explore: the STATS abstraction that allows developers to generate TLP without worrying about the details of how this is done; several OpenMP abstractions (parallel for, critical/ordered section, task), which allow developers to declare TLP and asynchronous units of computation while hiding the complex implementation details; and the C++ smart pointers abstraction, which performs automatic memory management relieving the developer of having to do it manually.

Software Systems. In general, a system is a software program that manages hardware resources and provides a platform to run applications. In the context of this dissertation, we focus on systems

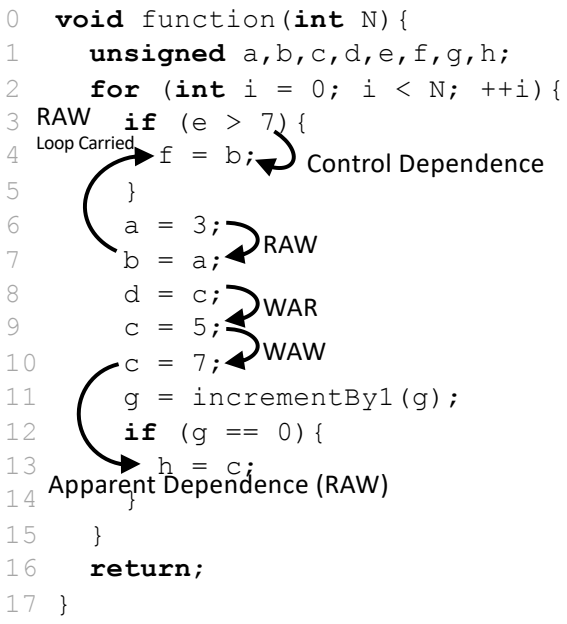


Figure 2.2: Examples of control and data dependences, along with apparent and actual dependences.

that have a compiler and a runtime component.

A runtime provides an execution environment that a program can run in. Depending on the task, a runtime can provide: memory management and garbage collection, input-output-error handling, execution control, multithreading and synchronization support (e.g., the OpenMP runtime), access and security control, and more. In STATS, the runtime provides support for generating TLP and for predicting the data to satisfy a dependence. While in CARMOT, the runtime is responsible for collecting information about the behavior of PSEs and for performing PSEC.

A compiler is responsible for analyzing, transforming, and translating high-level programming language source code into machine-readable code that a processing unit (e.g., a CPU) can understand and execute. Typically, a compiler has three components: front-end, middle-end, and back-end. The front-end parses the high-level, human-readable source code into a representation

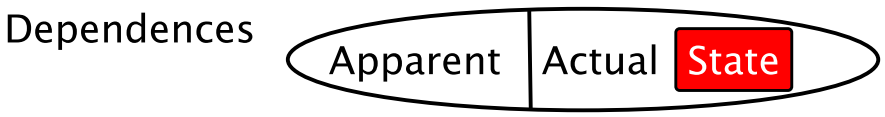


Figure 2.3: Taxonomy of dependences. State dependences are those that can be satisfied in an alternative way.

more amenable for analyses and transformations called Intermediate Representation (IR). Then, the middle-end is responsible for analyzing, transforming, and optimizing the IR. Finally, the back-end translates the IR into machine code for the target computer architecture. The STATS compiler inserts in the IR the program-specific knowledge embedded by a developer through the STATS programming language abstraction. While the CARMOT compiler inserts calls to the CARMOT runtime (this practice is called "code instrumentation") to collect information on PSEs and performs several middle-end optimizations to reduce the amount of instrumentation in order to reduce CARMOT's overhead.

In this dissertation we will touch upon other well-known compiler concepts such as: backward and forward Data-Flow Analysis (and its GEN, KILL, IN, and OUT sets), basic blocks, invariants, induction variables, callgraph, the LLVM mem2reg compiler optimization and others. Explaining these concepts in detail is outside the scope of this dissertation. We refer readers who are not familiar with these concepts to the canonical references [5], [6].

2.2 State Dependences Allow STATS to Generate a New Source of TLP

With the definition of the basic concepts that we build upon, we now motivate the research novelty of this dissertation. We first motivate STATS, our parallelizing compiler for nondeterministic programs the generate a new source of TLP by satisfying a subset of actual dependences, which we call state dependences, in an alternative way.

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*. For example, Monte Carlo simulations use random sampling to approximate a solution or make predictions. These programs may exhibit variation in their output across runs for the same input. Figure 2.4 shows such variation over 100 runs for six well-known nondeterministic, parallel benchmarks of the PARSEC benchmark suite [7].

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. If we were able to predict any of these intermediate data, we could forward them to their respective consumer. 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. We describe our solution in §3.

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.

2.2.1 Today's Limits

To understand the need for additional parallelizations for nondeterministic programs, we studied the PARSEC benchmark suite [7], which features multi-threaded implementations of modern workloads, as well as the industrial-strength and widely adopted codebase OpenCV [8] (which is composed of millions of lines of code). These programs have been manually parallelized extensively leaving no room for simple additional parallelizations.

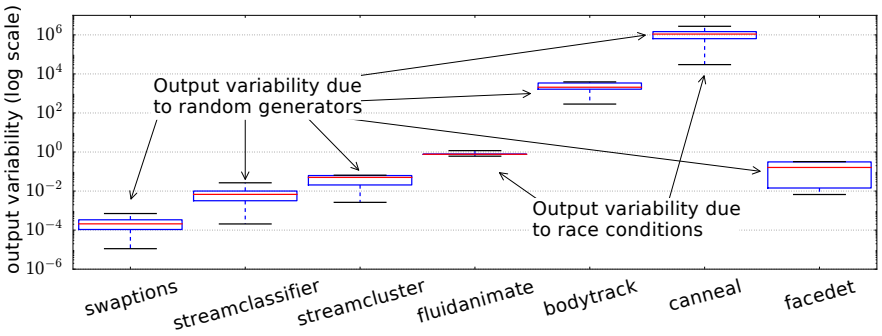


Figure 2.4: Output variability of nondeterministic PARSEC benchmarks. Several exhibit high variability and are particularly amenable to STATS.

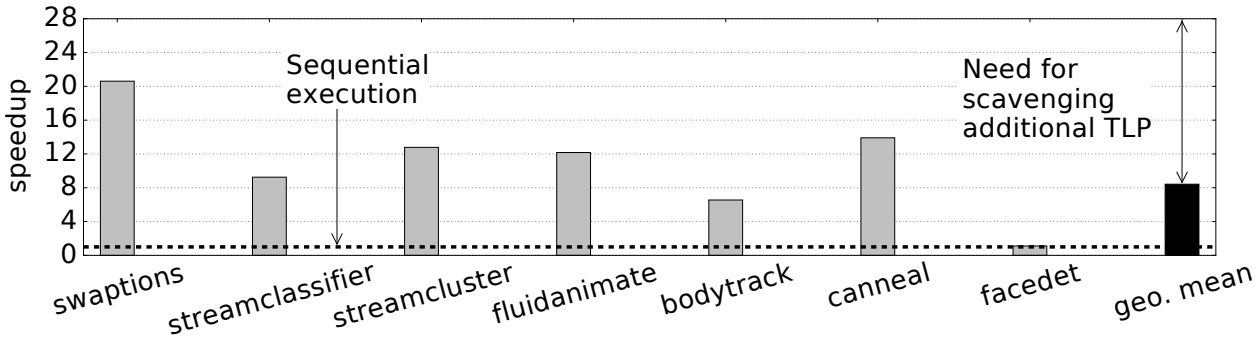


Figure 2.5: Highest speedup obtained by nondeterministic PARSEC benchmarks on a 28-core Intel-based platform.

All benchmarks considered have limited TLP. Figure 2.5 shows the highest speedup obtained by each benchmark compared to their sequential execution. The distance from an ideal speedup of 28× (the total number of cores available in our Intel-based machine) shows the need for creating additional TLP.

2.2.2 Code Example

Now that we have shown that there is a need to generate additional TLP out of modern, nondeterministic workloads (Figure 2.5). We are going to explain how we can extract an additional

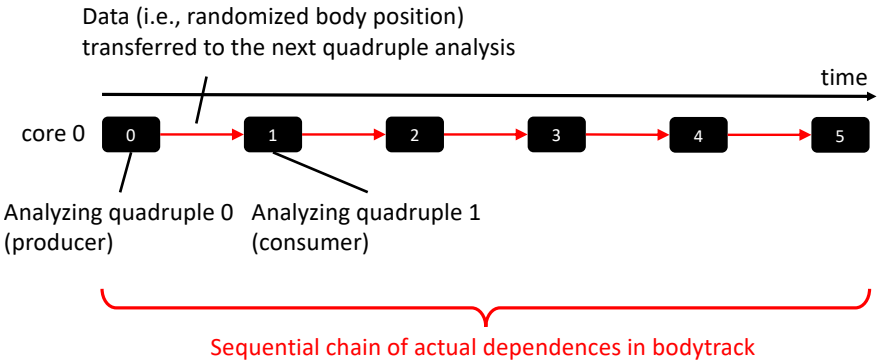


Figure 2.6: `bodytrack` execution is serialized by a chain of actual dependences.

source of TLP out of these benchmarks. To do so, we concretize the discussion using a specific benchmark, which is a good representative of the others.

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 the fact that is likely that the person in quadruple $i + 1$ is relatively close to where he/she was in quadruple i. By exploiting this correlation, `bodytrack` is capable of obtaining high accuracy in its output. However, the TLP, and therefore performance, of `bodytrack` is constrained by a single sequential chain of dependences, because the analysis of the quadruple $i + 1$ can start only when the datum generated by the analysis of the quadruple i is available (Figure 2.6).

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. These randomized body positions are the data that need to be transferred from the analysis of quadruple i (producer) to $i + 1$ (consumer) to satisfy the sequential chain of dependences. We take advantage of the fact that any of these randomized body positions are acceptable to satisfy the dependence.

State Dependence. The dependence chain described earlier between quadruples in `bodytrack` shown in Figure 2.6 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 2.7. A code region (e.g., a basic block, a loop, 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.

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 do so we have to solve the problem of predicting the incoming data that the analysis of i would have received if the code had run sequentially. To

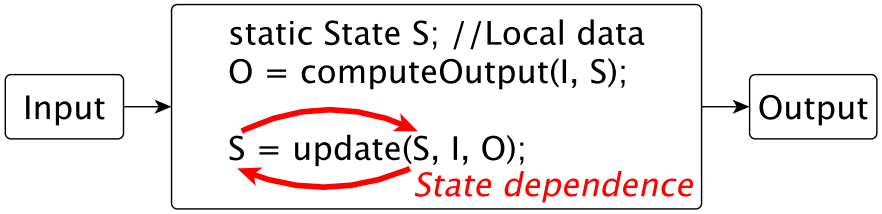


Figure 2.7: Code pattern that includes a state dependence.

solve this problem we perform extra computation before analyzing the quadruple i. This extra computation is 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*.

Short Memory Property. The property 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. In other words, an update related to a state dependence only depends on a few previous updates, not all of them (like in regular actual dependences). We call this the Short Memory Property. This property 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 quadruples before i are analyzed by the original code, an actual datum that the auxiliary code produces 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.

To take advantage of state dependences and their short memory property, STATS introduces a new programming language abstraction that developers use to encode information about state dependences.

2.3 PSEs Analysis is the Key for Programming Language Abstractions Support

Using some programming language abstractions in a large codebase can be challenging and error-prone due to the lack of tools to assist programmers [9]. Here we show three use cases that explore five different abstractions and describing how they support and challenge programmers. The programming language abstractions we target are: the STATS abstraction, several TLP-generating OpenMP abstractions, and the C++ smart pointer abstraction. We show how an analysis of PSEs behavior that we call PSEC is the common information necessary to build tools that help programmers use these abstractions. Finally, we show that prior work is insufficient as they use either static analysis only [10]–[13] or limited dynamic strategies [14]–[17] that limit them to a few simple abstractions.

2.3.1 Challenges in Adopting Abstractions

Declaring State Dependences with STATS. STATS requires a programmer to follow a given code structure, which makes the compiler aware of a program's state dependence. To do so, a programmer needs to classify the PSEs accessed by the code region where STATS operates into three classes: 1) Input class (PSEs that are only read), 2) Output class (PSEs that are written first), 3) State class (PSEs that are read first and then written). Understanding which PSE goes into which class often requires a programmer to understand the behavior of the entire program. Misclassifications lead to performance degradation or an incorrect program.

Program Parallelization/Synchronization. OpenMP has high-level abstractions to parallelize loops (*#pragma omp parallel for*), synchronize parallel accesses (*#pragma omp critical/ordered*), and asynchronously execute units of computation (*#pragma omp task*). Using these pragmas to their full potential quickly becomes complex as they often require both the specification of their

attributes and extra code to prepare the target code region for efficient parallel execution. For example, *#pragma omp parallel for* requires programmers to understand which PSE variable needs to be privatized per thread (using the *private* attribute), which can be shared (*shared* attribute), and which code statement uses PSEs involved in true data dependences that should be in a *#pragma omp critical/ordered* section. Also, programmers need to understand how a *private* PSE variable interacts with the code outside the target loop. Variables that are written before the loop and read inside need to be declared as *first private*, while variables written inside the loop and read after need to be *last private*. Furthermore, programmers might have to write additional preparation code to, for example, clone PSEs that are more complex than variables (e.g., arrays, objects). This requires knowledge of where and how these PSEs are allocated (e.g., their size, type, alignment). Similarly, *#pragma omp task* requires an understanding of which PSEs are consumed/produced by the task through the *depend(in/out)* attribute. Failing to correctly classify PSEs or their code statements results in invalid or inefficient code.

Managing Dynamic Memory. C++ programmers used to manually manage the dynamic memory of a program. To help with this task, modern C++ standards (as of C++11) have added the smart pointer abstraction. Smart pointers manage dynamic memory using reference counting, which tracks the number of pointers to a dynamic PSE object and deletes it when the count drops to zero. Unfortunately, using smart pointers can lead to memory leaks when there are cycles in the reference counting graph of PSEs. Programmers have limited tools support to detect when cycles occur and no support to identify and break them. This is particularly challenging when reference cycles cross many functions and source code files, making manual detection difficult.

2.3.2 Benefits of PSEC

PSEC conveniently summarizes the knowledge of how the program state is affected by a code region that abstractions of §2.3.1 require. We now give an intuition about how to collect and apply PSEC to use the STATS abstractions (further detail are described in §4). Consider the loop in Figure 2.8 to be the code region where a programmer wants to apply the STATS abstraction. PSEC would classify the PSEs affected by the loop's body as follows: the memory locations of array `inputs` and variable `N` are always only read, the memory locations of array `outputs` are always only written once, and variable `state` is always read and then written. With this information, CARMOT automatically generates the Input, Output, and State classes necessary to implement the code structure needed by the STATS abstraction as Figure 2.9 shows.

```
int N = {...};
int state = {...};
int outputs[N];
int* function(int inputs[], int N) {
  for(int i = 0; i < N; ++i) {
    state = stateDependenceUpdate(inputs[i], state);
    outputs[i] = getOutput(state);
  }
  return outputs;
}
```

Figure 2.8: A generic state dependence code structure. PSEC is performed on the loop body code region where the STATS abstraction will be applied.

```
class Input { int inputs[N]; int N; };
class Output { int outputs[N]; };
class State { int state; };
```

Figure 2.9: Resulting classes necessary to utilize the STATS abstraction, generated from PSEC-extracted information.

2.3.3 Overhead of PSEC

On top of tracking memory accesses (i.e., reads and writes) like other memory-tracking tools do [18], [19], PSEC needs to track accesses to function variables for a target code region to obtain complete information of the program state. We measured the increase of accesses and observed $8\times$ more accesses on average that need to be tracked for PSEC. Other tools do not have to track function variables, because their only goal is to validate memory accesses. Hence, these tools are able to invoke many general-purpose compiler optimizations, which are not compatible with PSEC (e.g., the mem2reg compiler transformation disrupts the mapping between variables in the source code and in the intermediate representation of the compiler, as we show in §A). This is why our approach requires a more involved compiler-based solution including several PSEC-specific compiler and runtime optimizations.

2.3.4 Limitations of Current Dynamic Analyses

As §2.3.1 shows, PSEs are explicitly used in many modern programming language abstractions. Despite their important role, we are the first one to consider PSEs as first class citizens. Dynamic analyses of prior works [14], [15], [17] are instead based on dependences or memory footprint of instructions. This limits prior work to imprecise and overly-conservative information that can

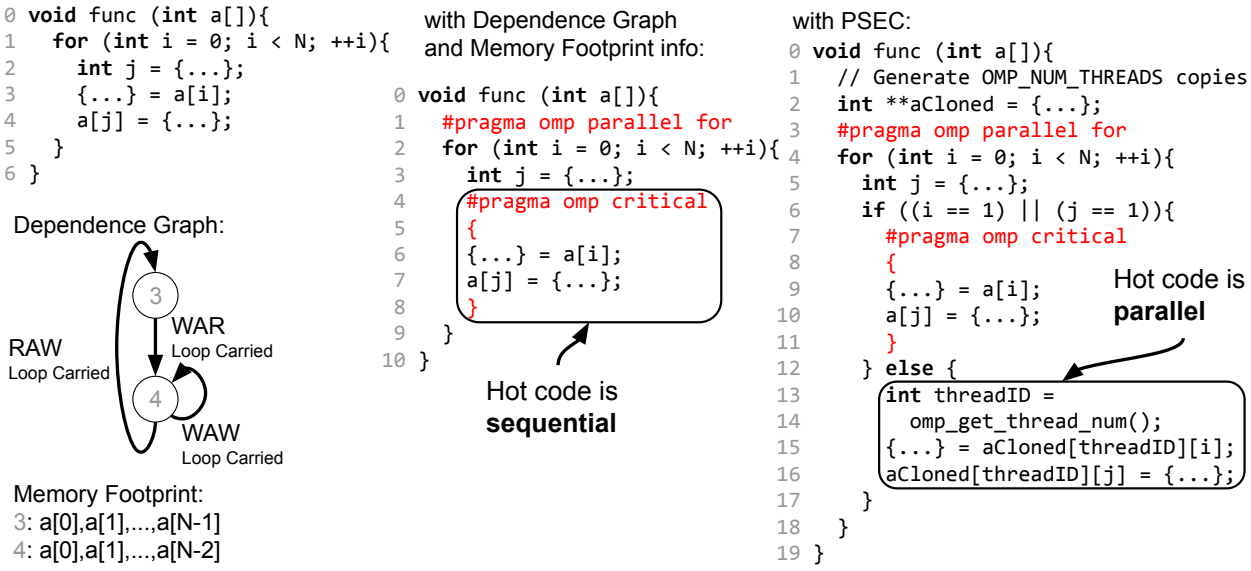


Figure 2.10: State of the art dynamic analyses based on dependence graph and/or memory footprint of instructions miss important parallelization opportunities compared to PSEC.

defeat the purpose of using an abstraction altogether. Figure 2.10 shows an example of the fundamental limitations of such dynamic analyses. In this example *i* spans from *0* to *N-1*, while *j* assumes the values {*1,0,0,2,3,...,N-2*}. The corresponding dependence graph and memory footprint for the relevant instructions (i.e., *3* and *4*) are reported in Figure 2.10. If programmers want to parallelize the for-loop in the example following the information provided by the dependence graph and memory footprint of the program, they need to be conservative and assume that any element of the memory object *a* can be involved in the loop-carried RAW dependence. This results in placing the most computationally intensive part of the loop body in a critical section, which runs sequentially and defeats the purpose of parallelizing the loop in the first place. This fundamental limitation becomes even more severe as the size of *a[N]* grows. PSEC instead reports to the programmer that only a small portion of *a* (*a[1]* in our example) is involved in the loop-carried RAW dependence. Hence, the programmer can considerably shrink the critical section and clone the rest of *a* to remove the loop-carried WAR and WAW dependences, regaining parallelism.

CHAPTER 3
UNCONVENTIONAL PARALLELIZATION OF NONDETERMINISTIC APPLICATIONS

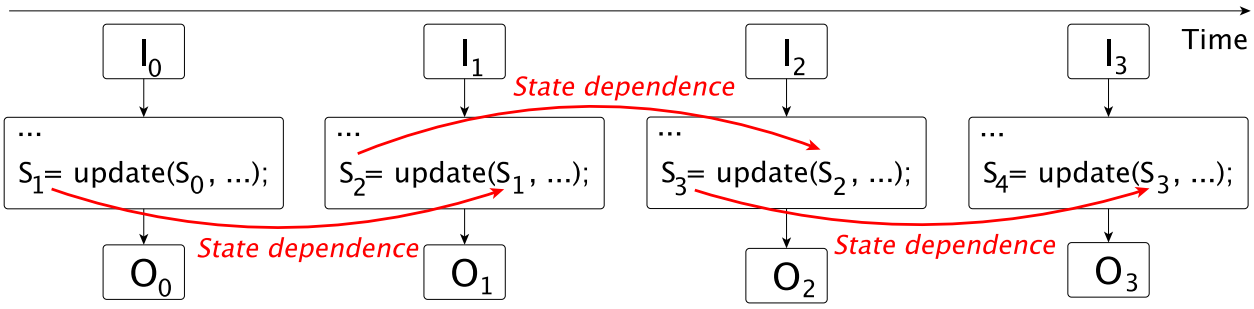
We now describe the execution model that STATS follows, the programming language abstraction offered by the STATS interface that allows developers to describe state dependences, and how STATS works as a system (§3.1). Then, we evaluate STATS on six nondeterministic benchmarks in §3.2. Finally, we perform an in-depth analysis of the overhead introduced by the execution model of STATS in §3.3 and evaluate its impact on the benchmarks in §3.4.

3.1 The STATS Solution

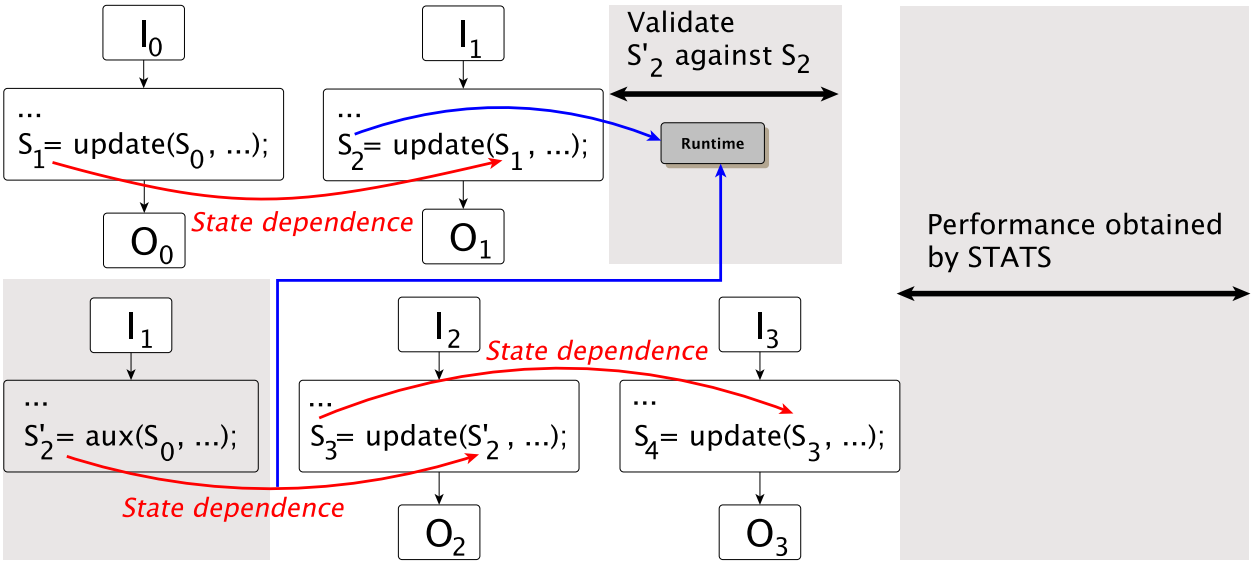
The STATS tool-chain increases the TLP (and thus performance) of nondeterministic C++ programs that exhibit the pattern of a state dependence as shown in Figure 2.7. It does so relying on additional algorithm-specific information and training inputs from the developer. These inputs are only used to explore 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 the STATS's output program will be in production. The rest of this section describes the execution model generated by STATS and its compilation flow.

3.1.1 Execution Model

STATS extracts additional TLP by grouping inputs of the code pattern shown in Figure 2.7 in ordered blocks and by overlapping their computations. This additional TLP can be exploited only



(a) Execution serialization due to a state dependence



(b) Additional TLP generated by auxiliary code

Figure 3.1: Alternative execution model obtained by using auxiliary code to satisfy a state dependence.

when the auxiliary code can satisfy the related state dependence. In other words, TLP can be generated only 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 3.1a. Here, all inputs are sequentially processed. STATS, in this example, generates the execution model shown in Figure 3.1b. Inputs are grouped in pairs (e.g., $\texttt{I}_0$, $\texttt{I}_1$ and $\texttt{I}_2$, $\texttt{I}_3$) and each group is processed in parallel (STATS

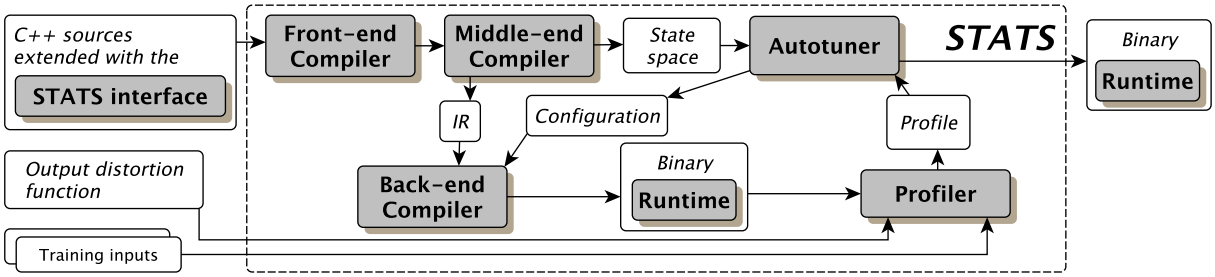


Figure 3.2: STATS includes three compilers, a runtime, an autotuner, and a profiler to optimize a nondeterministic C++ program for which the developer has identified state dependences via the STATS Interface.

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). While 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 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 3.1b), 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 3.1b), 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 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 2.7). 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 ex-

ecution 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 $\texttt{I}_2$ and after) are squashed, the execution restarts from the first not-speculative state generated by the previous group of inputs (e.g., $\texttt{S}_2$), and no other speculation is performed until all the current inputs are processed.

3.1.2 Software Architecture

STATS enforces the execution model described in §3.1.1 via its architecture shown in Figure 3.2. 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 will be satisfied by auxiliary code. The runtime determines whether to accept the speculative state and enforces the execution model described in

```
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 3.3: Original code of `bodytrack`.

§3.1.1.

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).

3.1.3 The STATS Interface

The STATS Interface is a programming language abstraction that extends the C++ language and 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 2.7, 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 3.5. 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 §3.1.1 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 exploit. Specifically, they need to enforce that computing `Output` depends only 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 origi-

```
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 3.4: Use of SDI in `bodytrack`.

nal 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 §3.1.4.

Figure 3.4 shows how a state dependence in `bodytrack` is encoded using the SDI. Figure 3.3 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` 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 3.6 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 2.7). 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 it 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

```
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 3.5: The State Dependence Interface makes the pattern of Figure 2.7 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 3.6: Use of TI in `bodytrack`.

space to find the most performant configuration, using the developer-provided training inputs.

3.1.4 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 3.7 shows the code generated from Figures 3.4 and 3.6, 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`). [1]

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 [20]. This is implemented as a new compilation pass in the *clang* compiler.

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 3.4) 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

[1] 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 3.7: C++ code generated by the front-end compiler from Figures 3.4 and 3.6.

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.

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 3.4) is set to a value v depending on the tradeoff type of v. If v is a constant (e.g., number of layers in `bodytrack`), 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 §3.1.1. 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 needs to only partially parse C++ programs, which made it possible to use a simple implementation based on Racket [21].

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()`.

3.1.5 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 [22] 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.

3.2 Evaluating STATS-Generated TLP

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 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. Also, we 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.

3.2.1 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

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 3.1: 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 ($\leq 7.1\%$).

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 intervals. STATS is built on top of LLVM 3.9.1 [23], Racket 6.8 [21], and OpenTuner 0.7 [22].

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.

3.2.2 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`). In the case of `bodytrack` we substituted the single (fixed) initial body position provided by the benchmark suite with a set of random (but still plausible) initial body positions, because of the sensitivity to the initial body position of particle filter-based tracking approaches (such as `bodytrack`). This limitation of particle filter-based tracking is well

known in the literature [24]–[33]. We chose an approach similar to the "big data initialization" method of [24], which uses a set of initial positions to start the tracking.

Moreover, to test STATS in a large codebase, we considered OpenCV [8] 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 2.7 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 [34]; 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 [35]. 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 [36]. `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 [37]. `streamclassifier`'s metric is the difference in B^3 metrics [36]. `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. [2] 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

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

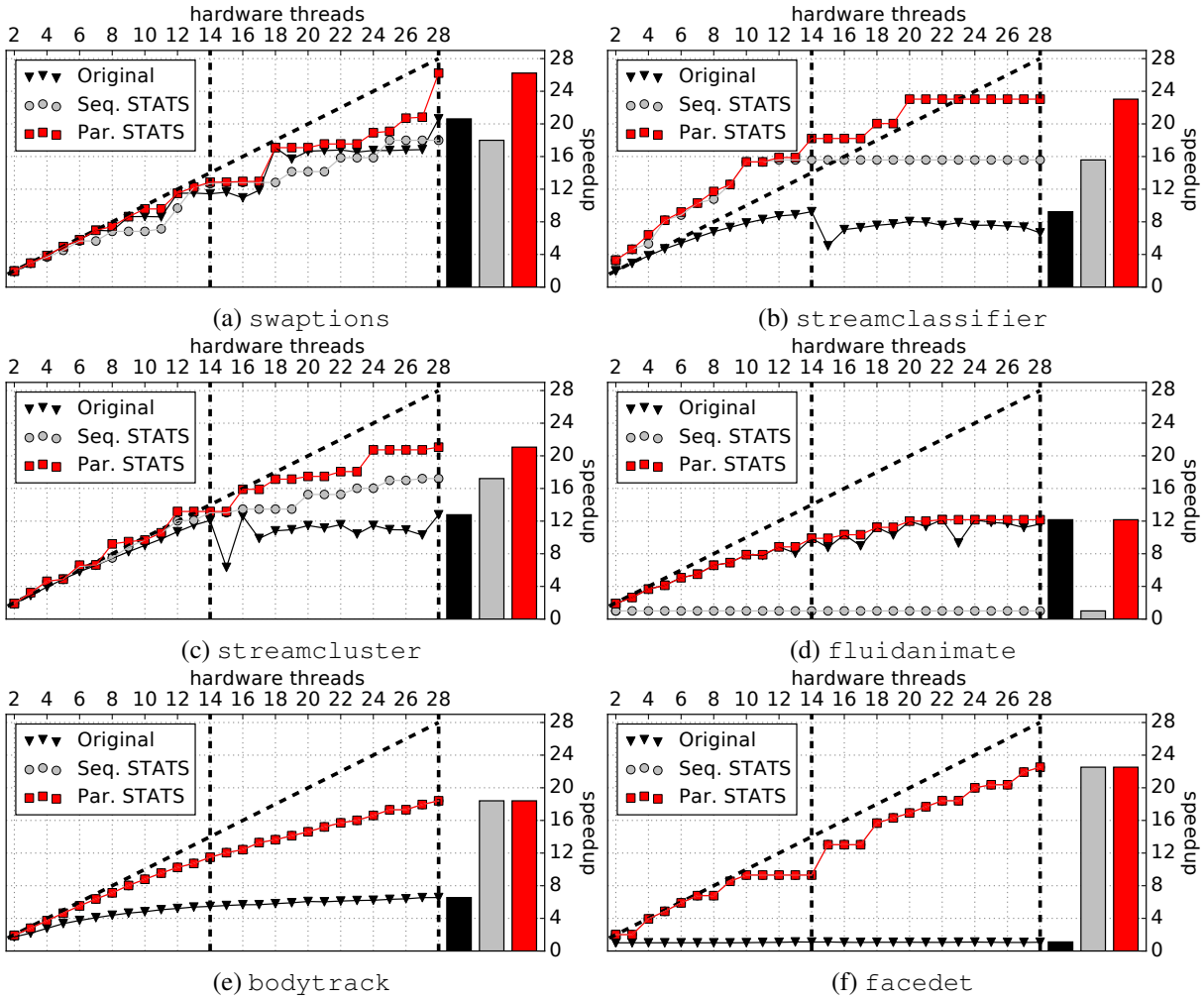


Figure 3.8: For most benchmarks, STATS generates a significant amount of extra parallelism that saturates the hardware resources of our platform. "Original" is the out-of-the-box benchmark that has been parallelized by traditional means. "Seq. STATS" ("Par. STATS") is the binary generated by STATS starting from the sequential (multi-threaded) version of a benchmark. The bar graphs show maximum speedup.

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 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

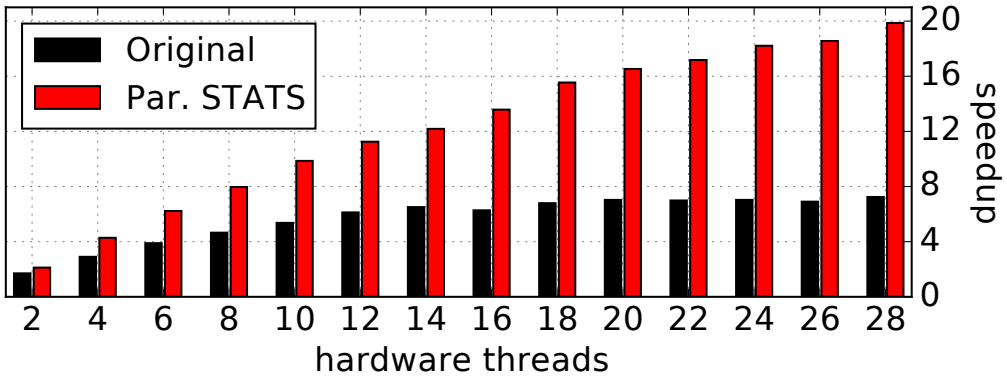


Figure 3.9: Geometric mean of speedups shown in Figure 3.8.

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.

3.2.3 Taking Advantage of State Dependences

Exploiting Multiple Cores. Satisfying state dependences with auxiliary code liberates important additional TLP. Figure 3.8 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 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 3.8 shows that

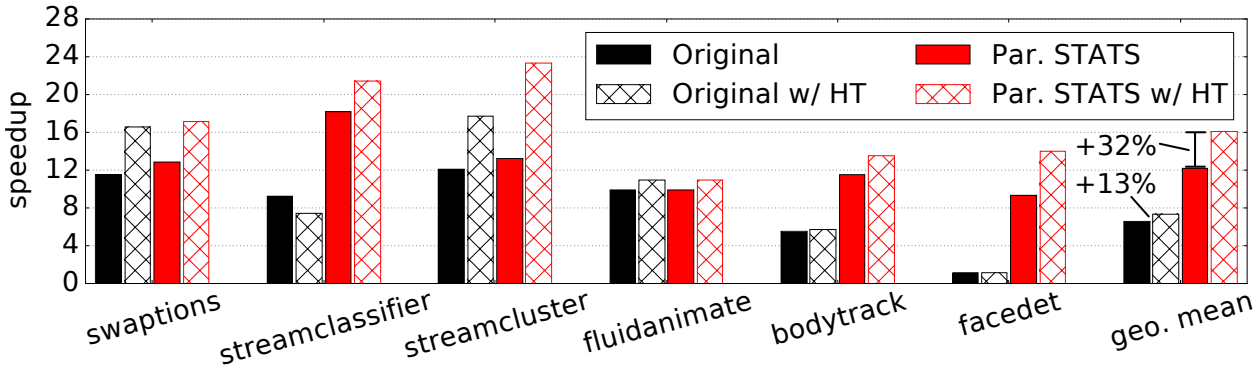


Figure 3.10: The performance obtained by STATS using a single socket with Hyper-Threading is constrained by hardware resources and not by low TLP.

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.

Figure 3.8 shows that both sources of TLP ("Original" and "Seq. STATS") are important to fulfill the parallelism requirements and that they need to be properly combined 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, because the `bodytrack` requires more frequent inter-thread synchronizations creating a bottleneck that `swaptions` 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 (per-

formed 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 3.8b) from 2 to 22 cores as well as for `streamcluster` (Figure 3.8c) for 6, 8, and 12 cores, because of the following two effects. First, the threads generated by STATS take better 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 more quickly.

Finally, `fluidanimate` (Figure 3.8d) 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 happens because `fluidanimate` has low output variability and 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 3.10 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

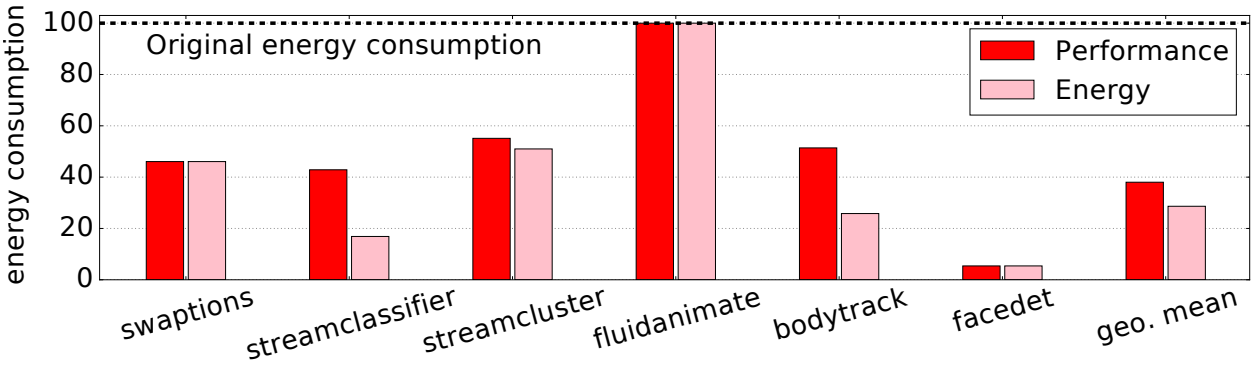


Figure 3.11: The binaries generated by STATS use considerably less energy compared to the original benchmarks.

(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 extra performance of 30% [39], [40]. STATS obtained a 32% performance improvement. Hence, the performance obtained by STATS is likely constrained by hardware resources and not by low TLP.

The Multi-Socket Effect. `fluidanimate`, `swaptions`, and `streamcluster` exhibited near-linear speedup on a single socket. STATS continues to improve performance on two sockets, but sub-linearly. An Intel VTune analysis indicated that this is due to the NUMA memory system—a common problem whose known solutions [41]–[45] apply to STATS, but go beyond the scope of this work.

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 energy consumption. In this case, the 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 3.11 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 might 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 3.12 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×.

3.2.4 STATS and its Related Work

We ask how well prior work is able to capture and take advantage of state dependences. To this end, we implemented related approaches able to target the considered benchmarks on our infrastructure and configured them to target 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

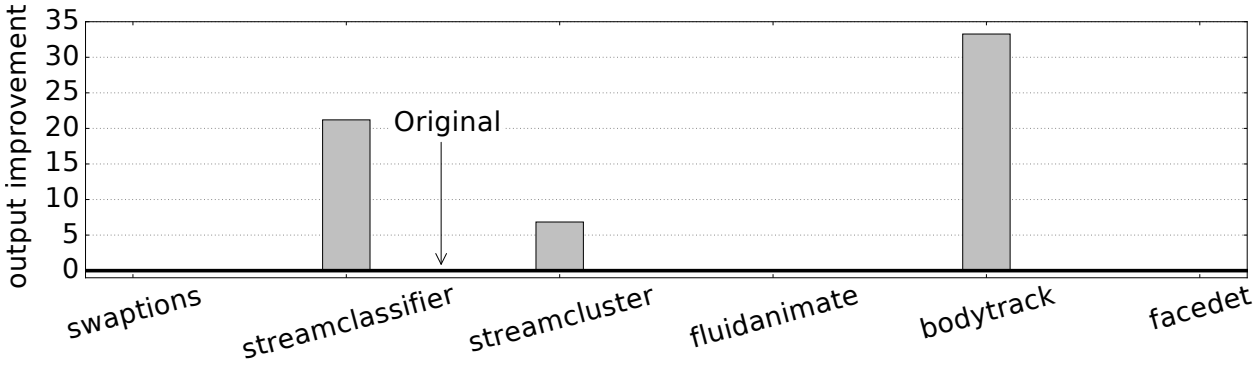


Figure 3.12: STATS can increase the original output quality by spending the saved time to iterate more over the same dataset.

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 2.4). Figure 3.13 shows the results that we obtained.

Prior approaches were able to take advantage of the state dependence only 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 3.13).

The "ALTER like" approach [46] 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 be updated only using a limited number of operators and the update instruction must be of the form: $variable = variable\ operator\ value$. `swaptions` is

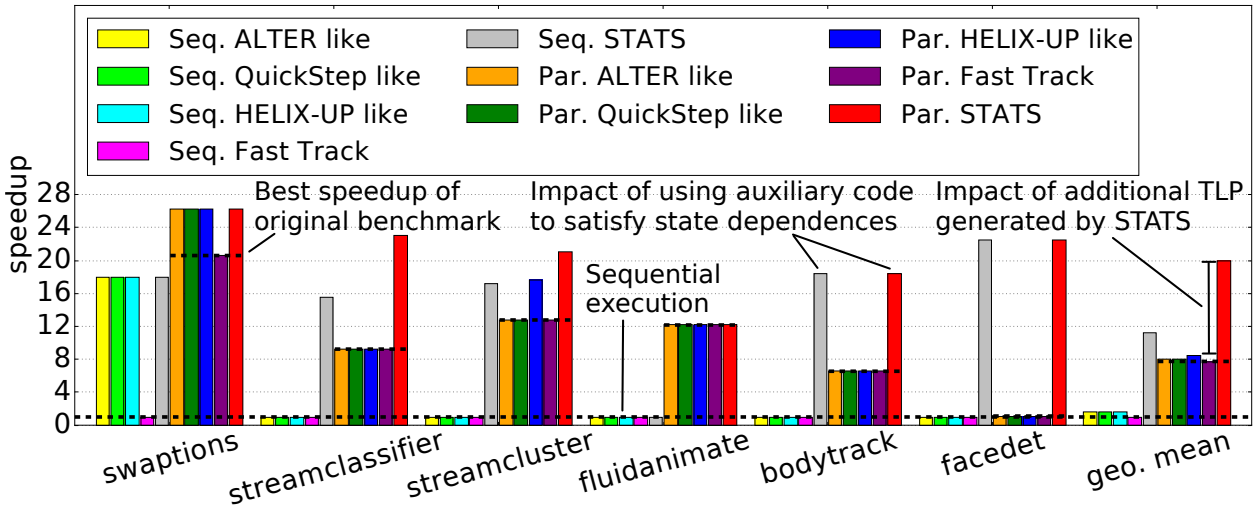


Figure 3.13: Only STATS takes advantage of non-trivial state dependences: they require the auxiliary code only STATS generates.

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" [16] and "HELIX-UP like" [47] 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" [48] 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 non-determinism of the original code that could have created (multiple) different unspeculative states. For these reasons, "Fast Track" always aborted its speculations in our experiments.

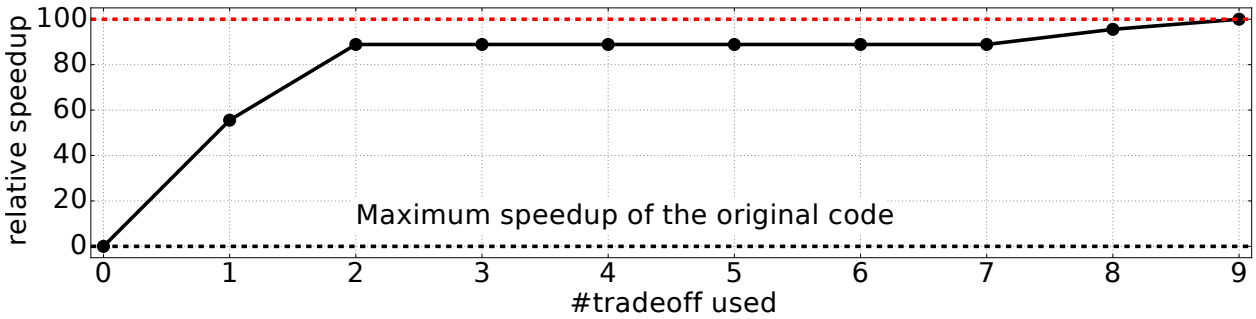


Figure 3.14: Developers gain most of the STATS benefits with a minimum effort (by encoding only two tradeoffs). This figure shows the average performance (geometric mean) relative to the best STATS speedup, by number of tradeoffs encoded.

3.2.5 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 3.1 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 3.14 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 payoff; just as a developer using STATS would. The orderings in Figure 3.14 correspond to the ones in Table 3.1. 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 control the amount of STATS-generated parallelism and original parallelism already encoded in the benchmark. These tradeoffs yield the

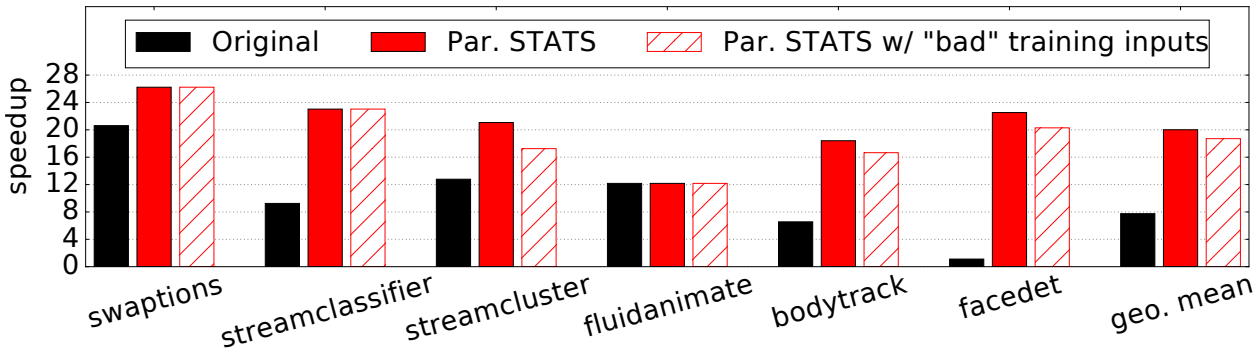


Figure 3.15: STATS loses only a small amount of performance when not representative inputs are used.

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.

3.2.6 Non-Representative Inputs

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

When its training inputs are not representative, STATS loses 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 3.15 shows that STATS loses only a moderate amount of performance when using the least-representative training inputs we could find.

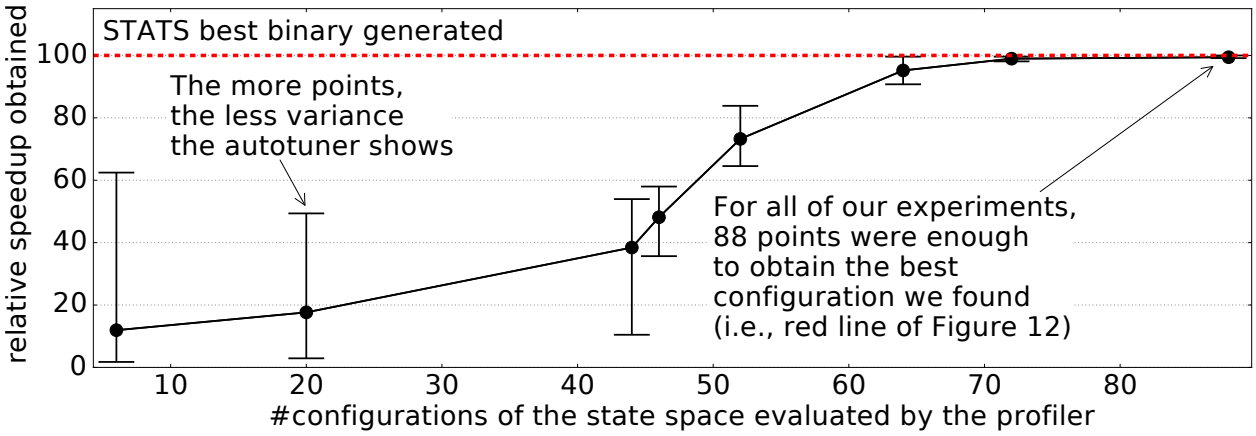


Figure 3.16: Average performance (geometric mean) of the final binary identified by the STATS autotuner after exploring a number of configurations.

3.2.7 Autotuning in STATS

The autotuner consistently and rapidly converges to the best program. Figure 3.16 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 3.8 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 3.16 shows that the variance in best speedups disappears after exploring 46 configurations.

3.2.8 When STATS Should Be Used

Invocation i of `computeOutput()` of Figure 2.7 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 property: an invocation of `computeOutput()` requires only a few previous invocations to generate a correct output. In other words, the compu-

tation 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 for converging 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 [49].

We included `fluidanimate` to test the limits of STATS. We wanted to see what happens 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 such as `fluidanimate` are not a 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 multi-dimensional points) has the "short memory" dependence property described above.

3.3 STATS 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

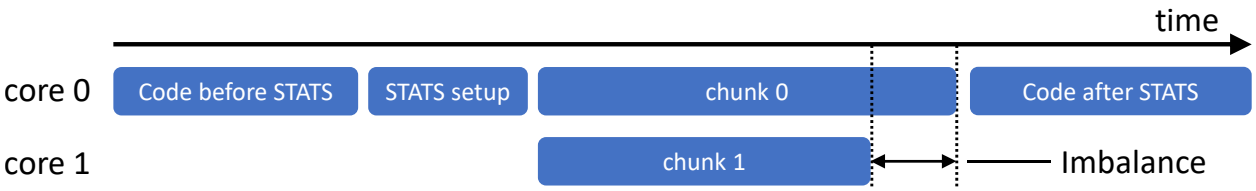


Figure 3.17: Processing different number of inputs leads to unbalanced computation.

additional computation and inter-core communication to enforce the execution model described in §3.1.1. 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 §3.4 empirically evaluates them.

3.3.1 Unbalanced Computation

The thread that runs the longest is the one that defines the performance of a parallel program. Therefore, the performance lost because of unbalanced 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 unbalanced computation at runtime. This is shown in Figure 3.17, and it is created by having an unbalanced division of the computation between threads. For example, imbalance can be created if different threads in the STATS execution model shown in Figure 3.1 process different numbers 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 due to thread synchronization overhead.

3.3.2 Extra Computation

To implement the execution model described in §3.1.1, 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 (Figure 3.1) processes, in parallel, a sequence of inputs by splitting it into chunks. 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 example of execution of an alternative producer is highlighted in Figure 3.18. 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 Figure 3.18). The computation performed by all alternative producers is overhead

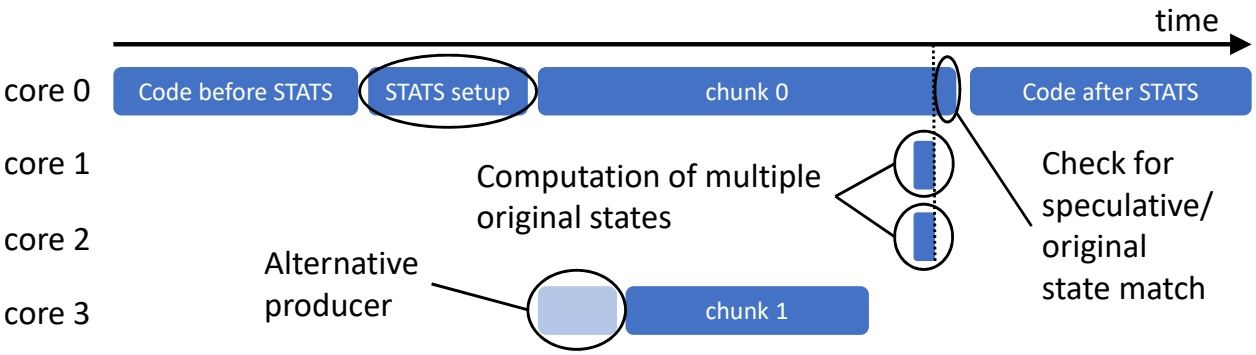


Figure 3.18: STATS parallelized programs perform extra work because of the execution model that STATS enforces.

tightly coupled with the STATS execution model and, as such, hard to reduce. While this source of extra work is hard to remove, §3.4 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, Figure 3.18 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 Figure 3.18 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.

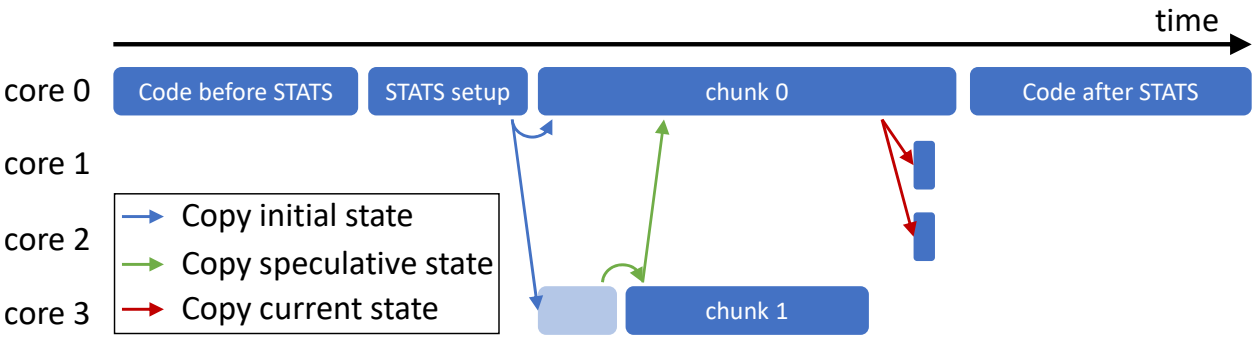


Figure 3.19: STATS enforces an execution model that requires copies of the computational state to execute the computation in parallel.

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 Figure 3.18). However, this computation requires replicating the original computation for a few inputs, multiple times (twice in the example of Figure 3.18). 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 by the alternative producer of the subsequent chunk of inputs. For example, when the thread running on core 0 in Figure 3.18 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 Figure 3.18) 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 §3.1.1. 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 Figure 3.18 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, in contrast to the original program where only a single computational state is needed.

The multiple states are created on demand by copying another one. For example, Figure 3.19 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 Figure 3.19) to its previous STATS thread (chunk 0) that has committed, so that it can later check the quality of the speculative state. 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.

3.3.3 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.

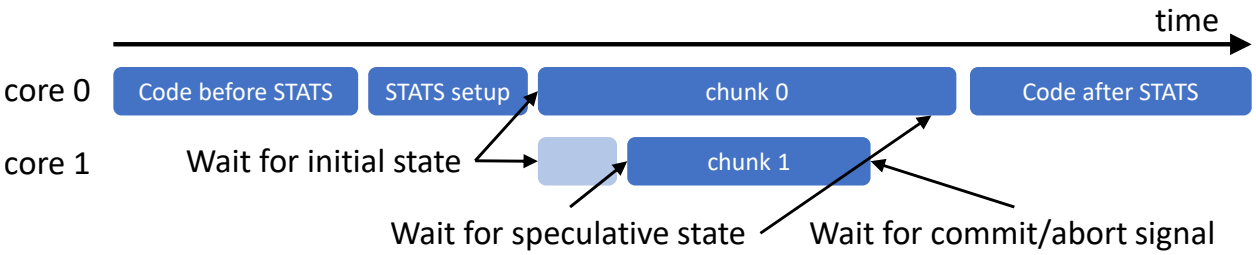


Figure 3.20: Threads created by STATS need to synchronize among each others to send or receive data or signals.

STATS generates a new source of TLP that follows a fork-join model as Figure 3.20 shows. These threads need to synchronize with each other to comply with the execution model described in §3.1.1. Spawning all the STATS threads at initialization introduces synchronization overhead as all the threads wait to receive their initial state. Another source of overhead is created by having a thread that has already committed its execution (like chunk 0 in Figure 3.20) 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.

3.3.4 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

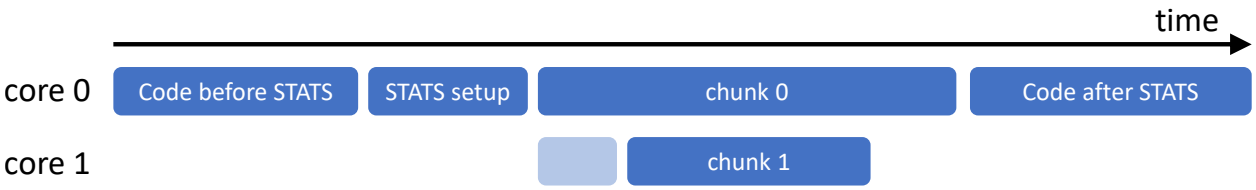


Figure 3.21: Sequential code outside the code region of STATS does not benefit from the additional TLP that STATS generates.

dependence. Everything that is outside this region of interest does not benefit the additional TLP generated by STATS, hence we consider it as overhead. Figure 3.21 shows the computation outside the STATS region of interest with the boxes `Code before STATS` and `Code after STATS`.

3.3.5 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; the more speculations that are, the more misspeculations 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.

3.4 Evaluating the Impact of STATS-Generated Overhead

Here we evaluate 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 changes to the experimental setup we used compared to our previous evaluation of STATS-generated TLP in §3.2.1.

3.4.1 Experimental Setup

Platform. We used the same machine described in §3.2.1, with a few upgrades. The OS is Red Hat Enterprise Linux Server 7.2 (kernel 3.10.0-693.21.1), still with no CPU frequency governors enabled (all cores run at maximum frequency). Hyper-Threading and Turbo Boost is turned off for all experiments. STATS is now built on top of LLVM 7.0.0 [23], Racket 6.8 [21], and OpenTuner 0.8 [22].

3.4.2 Statistics

Convergence. As in §3.2.1, each data point we show is an average of repeated runs, and configurations are executed 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.

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

Table 3.2: Total number of threads, computational states, and state size of the "Par. STATS" version of the benchmarks shown in Figure 3.22b.

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 3.2 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 creates only 14 parallel chunks of computation to avoid mispeculation.

3.4.3 Benchmarks

We considered the same POSIX multi-threaded versions of the PARSEC (version 3.0) benchmarks and their sequential versions described in §3.2.1 with a few exceptions. In this case we considered five out of the six benchmarks that have been evaluated in §3.2.1. 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

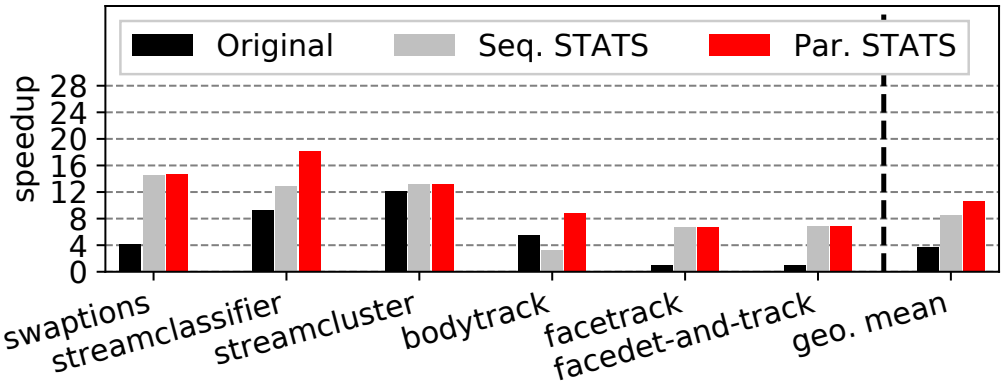
OpenCV face detection API fails to do so.

Inputs. To run the benchmarks, we used the same inputs described in §3.2.1 except for the additional `facetrack` and `facedet-and-track` benchmarks. 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.

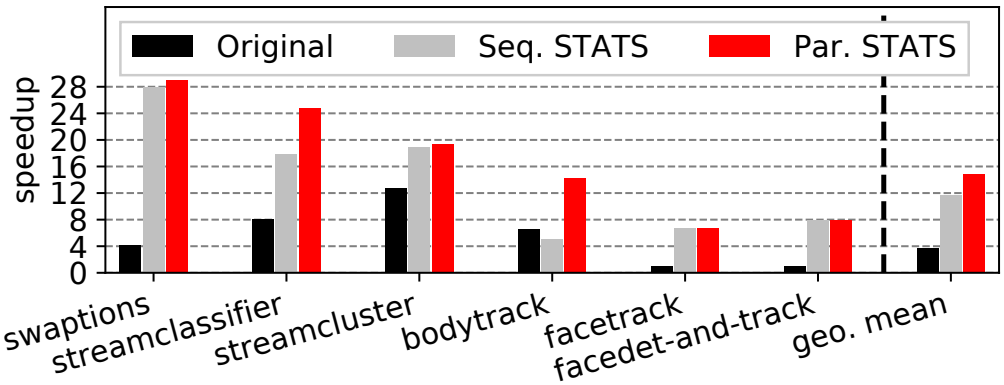
Output Quality. We relied on the same well-known output quality metrics we describe in §3.2.1. For the output quality of `facetrack` and `facedet-and-track` we used the average Euclidean distance between the boxes containing the detected faces.

3.4.4 Impact of STATS-Generated Overhead on Benchmarks

Our evaluation examines the impact of the overhead described in §3.3 on the parallel execution of nondeterministic programs compiled with STATS. We analyze the performance of these programs in §3.4.5. We evaluate the overhead of combining the TLP that the benchmarks had originally with the TLP generated by STATS in §3.4.6. 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 §3.4.7. Moreover, we describe the impact of the STATS parallelization in terms of data locality and branch prediction in §3.4.8. Finally, we analyze the impact of STATS on the inherent output variability of the considered nondeterministic programs in §3.4.9.



(a) 14 cores



(b) 28 cores

Figure 3.22: 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.

3.4.5 Performance Obtained by TLP Sources

The TLP that is expressed explicitly by developers via parallel programming APIs is not enough to fully utilize all cores included in our platform. Figure 3.22 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 on average.

The TLP that is extracted by STATS scales more than the original TLP. The grey bars of Fig-

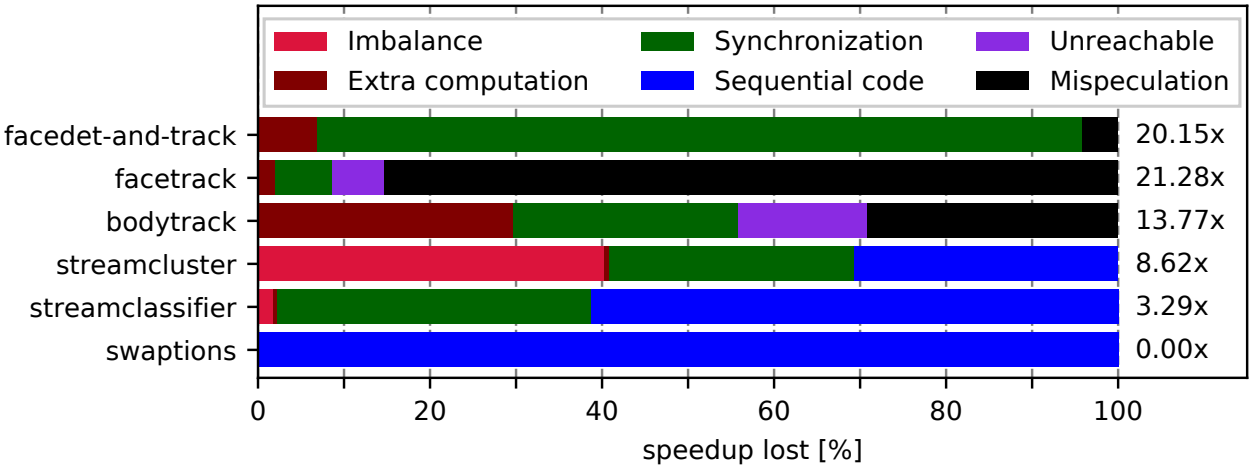


Figure 3.23: Percentage of speedup lost by benchmarks that take advantage of both original TLP and STATS TLP, on 28 cores. The number at the right of each bar is the amount of speedup lost with respect to the ideal speedup. Every benchmark is limited by different sources of overhead.

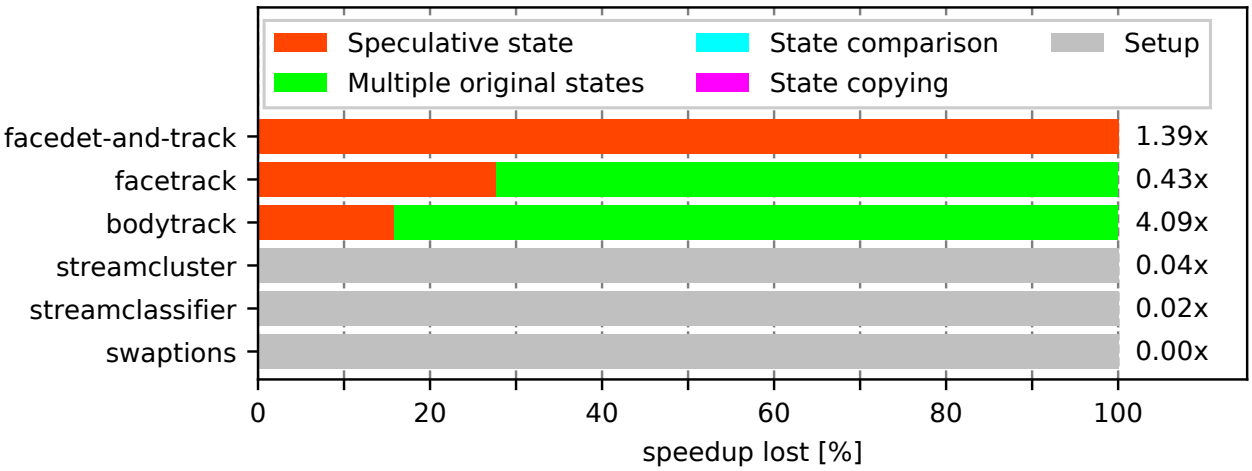


Figure 3.24: Percentage of speedup lost due to the "Extra computation" fraction of Figure 3.23. The number at the right of each bar is the amount of speedup lost only because of "Extra computation". The two main sources of overhead are related to the generation of the speculative state and multiple original states. The overhead due to the STATS setup phase only accounts for a small fraction of the speedup lost.

ure 3.22 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 Figure 3.22 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.

3.4.6 Performance Effects of STATS Overhead

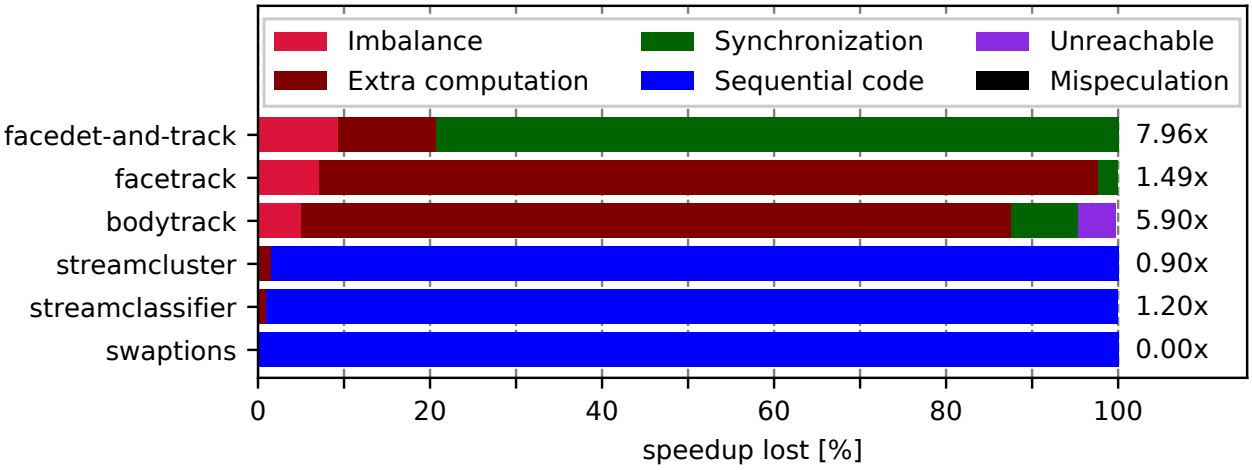
To understand what is limiting the STATS binaries to obtain speedups that scale linearly with the number of cores, we evaluated the performance impact of each of the six categories of overhead described in §3.3. Notice that these sources of overhead are related only to the STATS execution model. These 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 Figure 3.18), (iv) each thread synchronization code block (Figure 3.20), (v) each code block that clones computational states (Figure 3.19), and

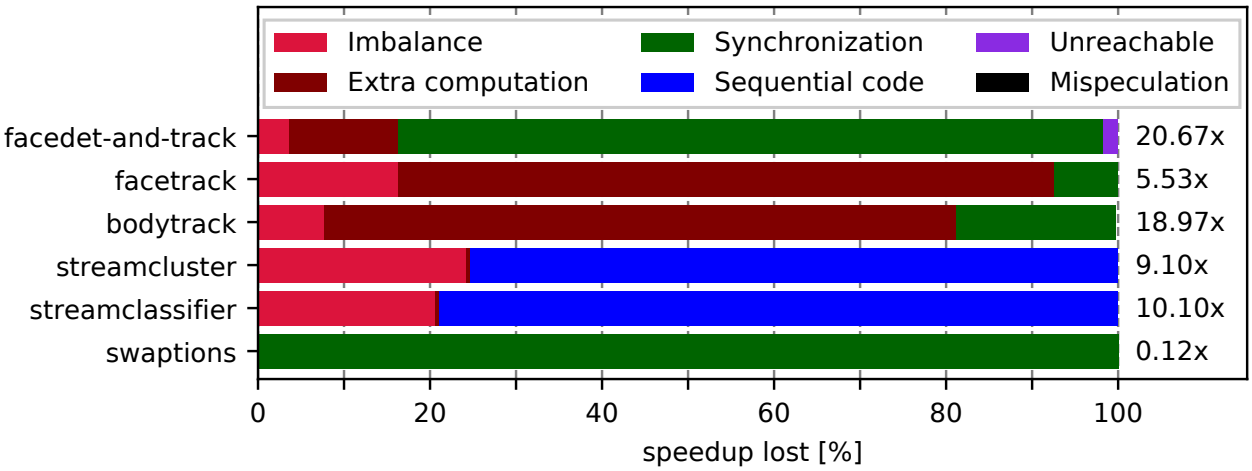
(vi) each code region parallelized by STATS. After obtaining these timestamps, we compute post-mortem the critical path of the parallel execution similar to what proposed in [50]. Finally, to evaluate the performance loss due to a given overhead, we compute the speedup obtainable if that overhead were 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 [50].

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 force STATS to rely only 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. Figure 3.23 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 creates only 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.



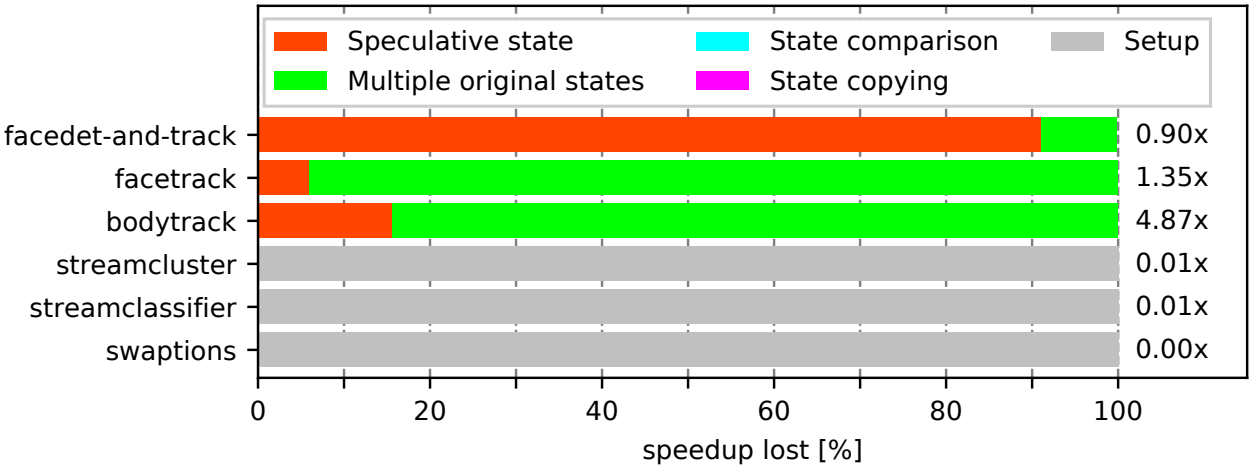
(a) 14 cores



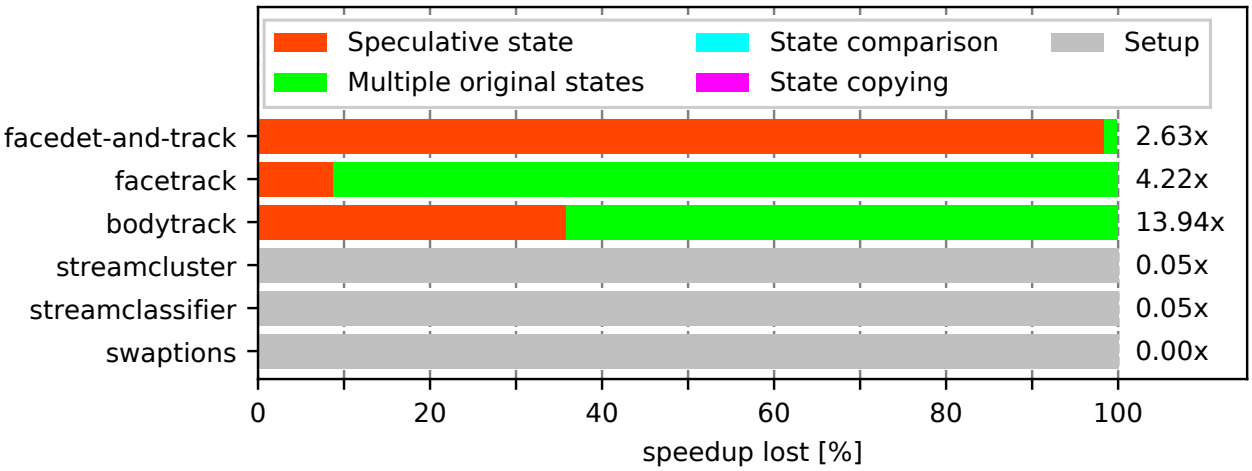
(b) 28 cores

Figure 3.25: 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.

Figure 3.24 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.



(a) 14 cores



(b) 28 cores

Figure 3.26: Percentage of speedup lost due to the "Extra computation" fraction of Figure 3.25. The number at the right of each bar is the amount of speedup lost only because of "Extra computation". As in Figure 3.24, 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.

Only TLP from State Dependences. Here we analyze 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 §3.3 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. Figure 3.25 shows these results.

The difference between Figure 3.25 and Figure 3.23 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, because when 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 §3.3.2. This analysis is shown in Figure 3.26. As for the case when both sources of TLP are used (Figure 3.24), 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.

3.4.7 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.

Figure 3.27 shows the amount of extra instructions executed to implement the STATS execution

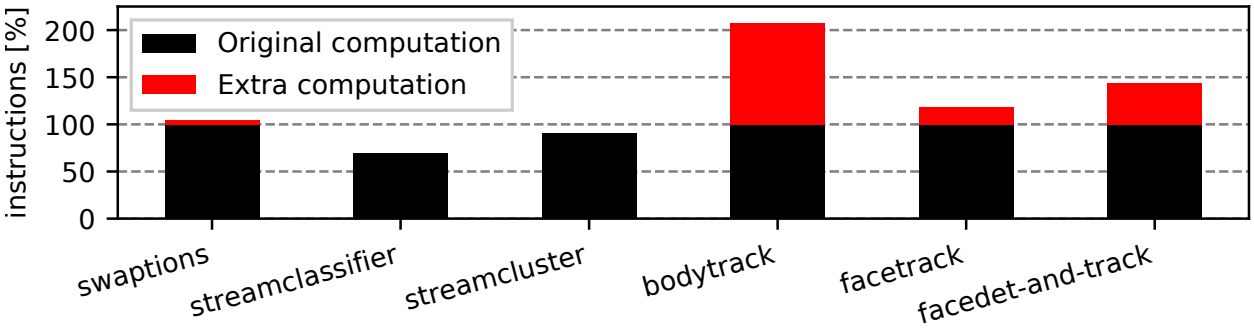


Figure 3.27: Extra amount of instructions executed by STATS parallelized benchmarks on 28 cores. The benchmarks `bodytrack` and `facedet-and-track`, execute a considerable amount of extra instructions than their original version.

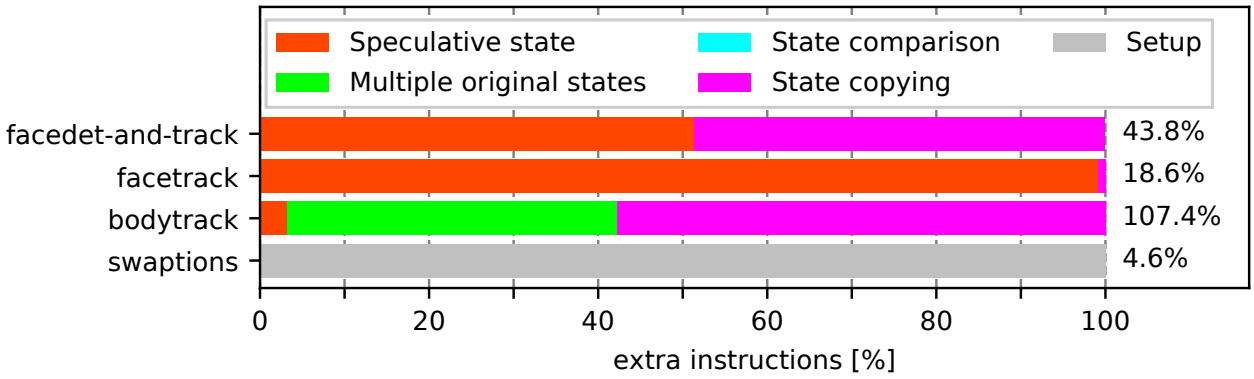


Figure 3.28: Extra instructions breakdown related to the "Extra computation" of Figure 3.27. Instructions related to the generation of the "Speculative state" by the alternative producer, and "State copying" dominate the other sources of extra instructions.

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 §3.3.2. 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 fewer instruc-

tions than the baseline, because the TLP extracted from state dependences leads the execution to find their clustering solutions faster.

Most of the extra instructions added by STATS are executed to copy computational states and to generate speculative states. Figure 3.28 shows the breakdown of these extra instructions. Combining these results with the loss in performance shown in Figure 3.26b, 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. In contrast, the number of committed instructions needed 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, 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. More efficient state copying would solve this problem. Improving the state copying could be implemented by compiler optimizations that exploit STATS 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.

3.4.8 Architecture Effects of STATS-Generated 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 3.3 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, 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 fewer cache misses and branch mispredictions compared to their out-of-the-box version because they execute less code. As described in §3.4.7, the STATS version of these benchmarks converges more quickly 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.

3.4.9 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 Figure 3.29. 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, Figure 3.29 shows that STATS tends to improve the quality of the outputs.

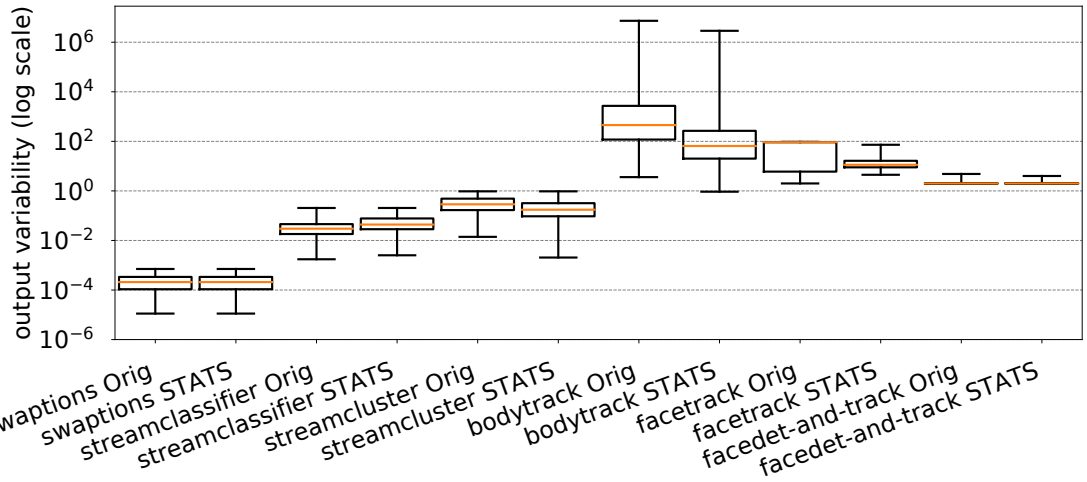


Figure 3.29: Output variability before and after the transformation performed by STATS (lower values are better).

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%)

Table 3.3: 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.

CHAPTER 4
PROGRAM STATE ELEMENT CHARACTERIZATION

We now describe Program State Element Characterization (PSEC) by outlining its components and illustrating the process that translates PSEC into programming language abstraction recommendations (§4.1). Then, we discuss CARMOT, our compiler-runtime, co-designed tool that implements PSEC and aids developers in using the programming language abstractions that we currently target: the STATS abtraction, some of the abstractions offered by OpenMP, and the C++ smart pointer abstraction (§4.2). Finally, we evaluate the overhead of CARMOT and its PSEC, and show the benefits of the optimizations we developed, which make PSEC feasible (§4.3).

4.1 Performing PSEC

Generating the programming language abstractions we discussed in §2.3.1 (i.e., the STATS abstraction, the OpenMP pragmas, and the C++ smart pointers abstraction) requires three essential pieces of information about PSEs: classification (e.g., only read PSEs), contextualization (when and where PSEs are used in the program), and reachability (PSEs that reference other PSEs). We now describe how PSEC provides this information and how it is used to automatically generate these abstractions.

4.1.1 Components of PSEC

PSEC has three components (Table 4.1): *Sets* to classify PSEs, *Use-callstacks* to contextualize computation, *Reachability Graph* to represent reachability relationships between PSEs.

Sets. In §2.1 we defined Program State Elements as the set of memory locations (stack and heap)

and variables (local and global) of a program at the source code level. Also, we defined a Source Code Region of Interest (*ROI*) as a single-entry, single-exit code region [51]. Examples of an ROI are a single statement, an if-then-else code block, a loop, or a function. PSEC is related to an ROI and contains information about how PSEs are read and/or written by that ROI.

PSEC classifies the ROI's access to PSEs into four *Sets*. Each set indicates how an ROI in the source code interacts with (i.e., reads/writes) PSEs. The sets that comprise a PSEC for a dynamically invoked ROI *Z* are:

Input set: PSEs read by a dynamic invocation of *Z* before being written by any invocation of *Z*. This set represents the input of *Z* as these data are generated by the code outside *Z* and consumed by *Z*.

Output set: PSEs written in a dynamic invocation of *Z* and read outside *Z*. This set represents the output of *Z* as this data is generated by *Z* and consumed by the code outside *Z*.

Cloneable set: PSEs written by more than one invocation of *Z* where no subsequent invocation reads them before overwriting them. This set represents data locations reused by invocations of *Z* without triggering RAW data dependences.

Transfer set: PSEs written by an invocation of *Z* and then read by a subsequent invocation of *Z* before any potential overwrites. This set represents the data generated by an invocation of *Z* and consumed by a subsequent invocation of *Z*, triggering a RAW data dependence.

Three pieces of information are necessary to classify PSEs in the correct set of a PSEC. First, we need to know *where* PSEs are allocated in the source code. Second, we need the *context* of such allocations. As context we use the callstacks that lead to the code statements that performed such

allocations. The context of allocations is necessary, because the same static code statement that generates PSEs can be used in different parts of the program, and the programmer must be able to distinguish them. For example, custom allocators are widely used in large codebases. Without knowing the callstack all allocations would look like they are coming from the allocation statement in the custom allocator, which is not useful information. Third, we need to *record reads and writes* the ROI performs on all PSEs to characterize them correctly. We call these accesses *uses* of PSEs.

Use-Callstacks. The program statements in an ROI (i.e., the uses of PSEs) can be executed multiple times from different parts of a program. To take this into account, we record the callstack of each statement invocation. We refer to these statements and their recorded callstacks as *Use-callstacks* of a PSEC. Knowing *Use-callstacks* enables us to disambiguate a static statement when invoked from different parts of the program, which can lead to a PSE being classified in different *Sets* of a PSEC. This is useful, for example, to report precisely which statement must be in a critical section.

Reachability Graph. PSEs can reference other PSEs in different points of a program. PSEC collects reference information through its *uses* of PSEs. Specifically, recording pointer escapes of PSEs. Escapes are recorded in the PSEC *Reachability Graph* where nodes are PSEs allocated within the PSEC's ROI and edges are references that point to other PSEs. We use this information to keep track of how PSEs reference each other to, for example, identify reference cycles.

4.1.2 From PSEC to Abstractions

Programmers declare the abstraction to apply to a given ROI to CARMOT. Then, CARMOT uses an ROI's PSEC to automatically generate new source code with the requested abstraction in it and customizes it with the correct attributes (Table 4.1 illustrates which parts of PSEC are necessary to

Abstraction	PSEC		
	Sets (I,O,C,T)	Use-callstacks	Reachability Graph
OMP parallel for (and critical/ordered)	✓	✓	✗
OMP task	✓	✗	✗
Smart Pointers	✓	✗	✓
STATS	✓	✗	✗

Table 4.1: Different abstractions need different parts of PSEC.

generate each abstraction). Next we describe the generation of abstractions from PSEC.

Declaring State Dependences with STATS. The STATS abstraction Input-Output-State can be mapped directly from the *Sets* of an ROI's PSEC. PSEs classified in the Input, Output, or Transfer sets are respectively mapped to the Input, Output, State classes of the STATS abstraction. The STATS abstraction requires the target ROI to be explicitly moved into a separate function; hence, PSEs in the Cloneable set are declared locally in that function. This localization enables the STATS compiler to spawn independent parallel threads to execute the related ROI.

Program Parallelization/Synchronization. To generate a *#pragma omp parallel for* with the correct attributes, CARMOT uses the *Sets* of the PSEC as follows. For every PSE e in the Cloneable set, CARMOT extracts the callstack for the element's allocation. These PSEs and their callstacks tell us what needs to be cloned to remove WAR and WAW data dependences between invocations of the related ROI (e.g., the body of a loop). If e is a variable, then CARMOT privatizes it in the generated pragma. Variables that are also in the Output set are declared as *lastprivate*, since they can be read after the ROI. Similarly, variables that are also in the Input set are declared as *firstprivate*, since they were first read inside the ROI. If e is a memory location, then CARMOT advises programmers to clone the PSE (CARMOT's output provides the allocation site and

its callstack to help programmers understanding how to perform the cloning) and use the OpenMP API *omp_get_thread_num()* to access the correct clone of that allocation in the ROI. PSEs that belong only to the Input set are declared as *shared* in the pragma because they are only read. Finally, for each PSE e in the Transfer set CARMOT retrieves its *Use-callstacks*. If e is a variable, then CARMOT checks each use of e to understand if the computation performed on e is reducible (i.e., the statement uses one of the OpenMP-supported reduction operators such as $+$). If the computation is reducible, then CARMOT includes e and the supported operation in the *reduction(operator:variable)* attribute. Otherwise, all statements that access e are wrapped in a *#pragma omp critical* or *#pragma omp ordered* section. Note that CARMOT leaves the decision as to which abstraction to use, either critical or ordered, to programmers as they know whether it is necessary to preserve the loop iteration order. CARMOT generates dependences for *#pragma omp task* as follows. The Input and Output sets of a computational spoor are mapped to the *depend* attribute of *#pragma omp task*. All PSEs e in the Input set are declared as *depend(in:*e*)*. Similarly, all PSEs e in the Output set are declared as *depend(out:*e*)*.

Managing Dynamic Memory. Cycles between PSEs allocated in an ROI are detected using the PSEC *Reachability Graph*, which tracks references between PSEs. CARMOT reports detected reference cycles to programmers and can suggest which reference should become a weak pointer[1] to break a detected reference cycle. It does so by identifying the node in that cycle that has the oldest access time. This enables programmers to gradually port ROIs within a large codebase to use smart pointers without introducing cycles.

[1] A weak pointer does not increment an allocation reference counter.

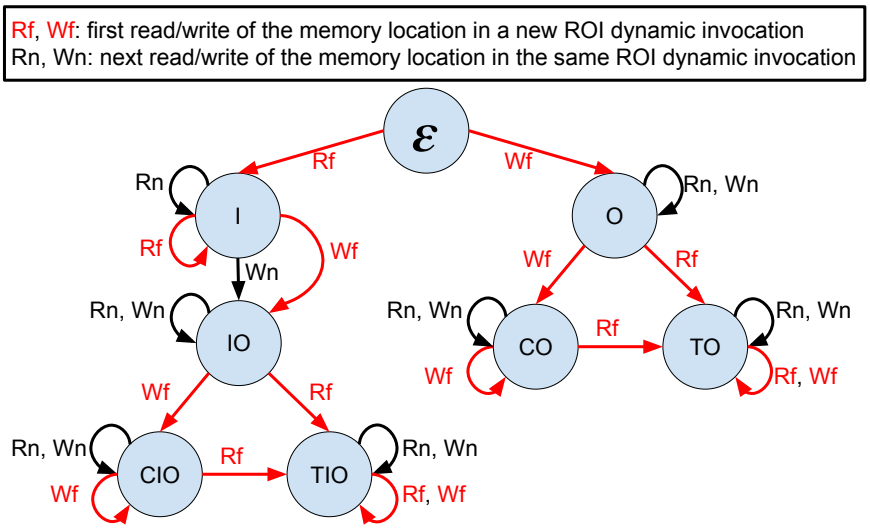


Figure 4.1: PSEC follows a Finite State Automaton (FSA).

4.2 CARMOT

Here we describe how CARMOT performs PSEC, its compiler, Pintool, runtime, and PSEC-specific optimizations.

4.2.1 PSEC with CARMOT

CARMOT performs PSEC of an ROI independently of other ROIs. When a PSE (e.g., variable) is accessed within an ROI, CARMOT classifies it into the *Sets* of that ROI's PSEC following the Finite State Automaton (FSA) shown in Figure 4.1. Each PSE has an instance of this FSA. PSEs start in the ε state. A PSE is added to the PSEC of an ROI Z upon its first access within Z. Subsequent accesses of a PSE in Z might change its FSA's state for Z. At the end of a program's execution, the final FSA state of a PSE for Z reflects the set (or sets) that the PSE belongs to with respect to ROI Z. In more detail, if the terminal FSA state includes an `I`, `O`, `C`, and/or `T`, then the related PSE belongs to the Input, Output, Cloneable, and/or Transfer set respectively. Note that a

```
0  int work(int a, int b){
1      int i, x, y;
2      y = 42;
3      for (i = 0; i < 10; ++i){
4          #pragma carmot roi{
5              x = i/(a + b);
6              y /= a*x + b;
7          }
8      }
9      return y;
10 }
```

Figure 4.2: CARMOT automatically builds the PSEC containing the information to parallelize this for-loop.

PSE can never be both in the Cloneable and Transfer sets ($C \cap T = \emptyset$).

Let us consider the loop in Figure 4.2 and the PSE variable y. In the first dynamic invocation of ROI Z, PSE y is first read and then written in line 6, hence y transitions from ε to `I` (R_f) and then to `IO` (W_n). In the subsequent dynamic invocation of Z a read of y happens (R_f), which causes a trasition to `TIO`. `TIO` is a sink state, so when the program finishes, CARMOT classifies y in the Transfer, Input, and Output sets.

The FSA operates only on reads and writes that happen within ROIs. This design decision enables CARMOT to avoid profiling code outside ROIs, but it also makes the assumption that PSEs written in an ROI will be read outside the ROI, so they will be part of the Output set. This assumption is conservative and does not affect the correctness of the PSEC.

4.2.2 Advantages of CARMOT's Dynamic Approach

CARMOT performs PSEC by profiling a specific run of the target program. We envision CARMOT users will perform PSEC on a program multiple times to cover many program inputs and combine the generated PSEC. Combining PSECs can be done through set union. For example, if PSE e is classified in the Input and Output sets in the first run, and in the Cloneable and Output sets in the

second run, the PSEC across runs classifies *e* in the Cloneable, Input, and Output sets. The only exception to this union rule is when *e* is in the Cloneable set for one run and in the Transfer set for another run. In this case, the conservative answer is to classify *e* in the Transfer set. Currently, and only for engineering reasons, CARMOT's users need to manually apply these rules to merge multiple PSEC to gain a more comprehensive understanding of the target program.

CARMOT's dynamic approach goes beyond what can be determined with static code analyses and provides programmers recommendations and support for programming language abstractions at the source code level for a specific program execution. The advantage of providing recommendations, as opposed to making automatic semantic changes to the code, is that it makes CARMOT more accessible to programmers. These recommendations allow for a better understanding of code behaviour and provide a starting point to tune abstractions to the programmer's needs. The disadvantage is that verifying the correctness of such recommendations for all prossible program executions has to be done manually. However, we argue that such a process is more suitable for humans rather than tools. Programmers can leverage domain specific knowledge about a program to make a decision, while an automatic, semantic-changing tool has to make conservative assumptions when trying to build an abstraction that is sound for all possible program executions. This conservativeness hides the true behavior of the execution of a program, which prevents programmers from reasoning about their programs and the abstractions they want to use.

4.2.3 CARMOT as a System

CARMOT implements the compilation flow in Figure 4.3. CARMOT's compiler (§4.2.4) generates a binary from C/C++ source files including code instrumentation. Complementarily, CARMOT loads its Pintool (§4.2.5) into memory to cover code that lacks available source. CARMOT's runtime (§4.2.6) is embedded within the generated binary as a static library. The runtime processes

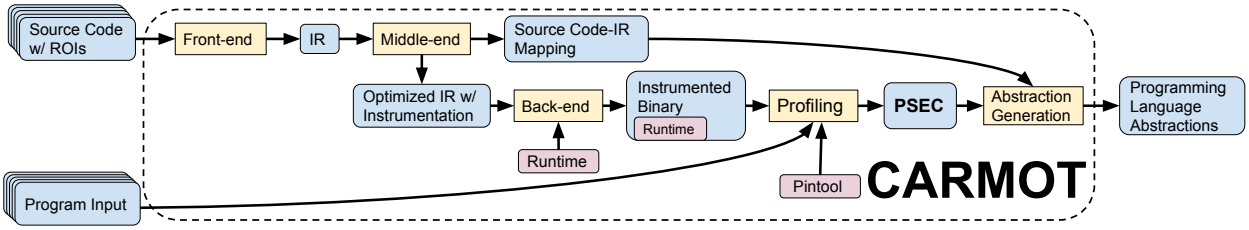


Figure 4.3: CARMOT produces the mapping between source/IR code and runs an instrumented binary to build the PSEC, which is then used to generate the target abstraction information for the programmer at the source code level.

the reads and writes provided both by the instrumentation generated by CARMOT's compiler and by its Pintool. This generates the PSEC of each ROI specified by CARMOT's pragma included in the program's source code (Figure 4.2). The PSEC is then translated into programming language abstraction recommendations for the abstraction chosen by the CARMOT user.

4.2.4 Compiler

CARMOT's compiler uses `clang` with debugging symbols enabled, but without optimizations, to translate a C/C++ program to LLVM's IR and guarantee a reversible mapping between source code and IR. The advantage of performing PSEC at the IR level rather than at the binary level is two-fold. First, we can easily implement precise and effective specialized code analyses and optimizations. Second, the amount of instrumentation is considerably reduced compared to binary instrumentation, where spilling of variables onto memory already occurred, generating extra memory loads and stores. The disadvantage of performing PSEC on unoptimized IR is the high overhead of the profiling phase, making PSEC infeasible for a large codebase. For this reason, CARMOT uses the following PSEC-specific code analyses and optimizations.

1) Subsequent accesses. The FSA of Figure 4.1 shows that transitions of PSEs to a different state happen only upon the first read or write (R_f, W_f) of a new dynamic invocation of an ROI.

The only exception is a subsequent write (W_n) in the same ROI dynamic invocation when the PSE is in the `I` state. Following this observation, this optimization aims to instrument only the first read and write of a PSE and avoid instrumenting subsequent accesses that are proved to always access the same PSE.

We developed a new intra-procedural data-flow analysis to identify where a PSE must have been accessed already since the beginning of an ROI. For this data-flow analysis, predecessors and successors of basic blocks that are outside or leave an ROI are not followed during the data-flow value propagation as only instructions within an ROI need to be considered. We do so by considering the entry point of an ROI as the entry point for our analysis. Elements in the GEN, IN, and OUT sets are the variables and memory locations (i.e., PSEs) of the target program. Given an instruction i that is either a load or a store, the sets for the data-flow analysis are defined as follows. The GEN set of i is the PSE a that a load is guaranteed to access or a store must write to ($GEN[i] = \{a\}$). The IN set of i is first initialized to be the union of all PSEs, and then refined to be $IN[i] = \bigcap_{\forall p \in preds(i)} OUT[p]$, where p are the predecessors of i. The OUT set of i is initially empty, and then refined to be $OUT[i] = IN[i] \cup GEN[i]$. This data-flow analysis runs until a fixed point is reached for each ROI. Elements in the IN set of an instruction i are the PSEs that must have been accessed between the entry of the ROI and i. Hence, CARMOT reduces profiling overhead by avoiding instrumenting instructions i where the PSE accessed by i belongs to $IN[i]$.

2) PSEs aggregation. Normally, uses of PSEs are instrumented singularly. However, contiguous PSEs that can be indexed (e.g., arrays), for which the same operation is performed at every ROI's dynamic invocation (e.g., they are always only read or only written) are instrumented altogether at once. Currently we limit this optimization on ROIs that wrap the body of a loop for which the loop governing induction variable indexes the contiguous PSEs.

3) Fixed setting of FSA state for PSEs. The FSA in Figure 4.1 shows that PSEs that are always only read will always be classified as Input in the PSEC. Hence, PSEs that can be verified to be only read at compile time can be instrumented only once and still be correctly classified in the Input set. Although an ROI is a general code region, we currently enable this optimization only for ROIs that wrap the body of a loop. We determine whether a PSE is only read by verifying that the corresponding load instruction is loop invariant. Similarly, the FSA classifies PSEs that are always only written as Output or Cloneable. At compile time, we determine whether a PSE is only written using the PDG. If the store instruction that writes the PSE has no incoming memory dependence edge where the source of the edge is an instruction in the ROI, we set the FSA state of that PSE to be Output. Then, if the considered ROI wraps the body of a loop, we use the loop governing induction variable to determine whether the store to the PSE is executed more than once. If so, we set the FSA state of that PSE to also be Cloneable.

4) Selective mem2reg. The LLVM mem2reg optimization [6] promotes the uses of local variables of a function into virtual LLVM registers. This optimization is extremely beneficial for performance and to enable further analysis and transformations, but it cannot be generally applied when performing PSEC (§2.3.3). However, some allocations of PSEs that are local variables can be promoted to registers without affecting the correctness of PSEC. For example, local variables that are never used in any ROIs can be safely promoted to registers because they will not be part of a PSEC, and instrumentation of such variables can be safely removed. Also, local variables with a specific role in an ROI have to be promoted to registers to be identified (e.g., loop governing induction variables). We built a wrapper around the LLVM mem2reg optimization that allows the promotion of specific local variables to registers only when it is safe to do so.

5) Call graph-based optimization. This optimization selects functions that can be optimized with conventional transformations while preserving the aforementioned IR-to-source-code mapping needed for PSEC. This optimization is based on the observation that if a function f cannot be in the callstack when any ROI starts, then f can be optimized with conventional optimizations (such as `-O3`) without breaking the IR-to-source-code mapping, because any PSE allocated in the stack by f will not be part of the PSEC of any ROI. This holds even if f is invoked within an ROI, as its stack is freed before returning to its caller and therefore such stack PSEs cannot be involved in any data dependences that cross the boundaries of an ROI. Therefore, only PSEs that are heap allocated by f need to be tracked, which are preserved by the optimizations included in `-O3` of `clang`.

To perform this optimization, CARMOT identifies the functions that cannot be in the callstack when any ROI starts by computing the complete callgraph of a program (i.e., a callgraph where the lack of an edge (f_i, f_j) means f_i cannot invoke f_j). Unfortunately, the callgraph provided by LLVM is not complete. To generate the complete callgraph, CARMOT computes the program dependence graph (PDG) to automatically discover the possible callees to which a pointer could refer. CARMOT uses the same memory alias analyses used by the previous optimization. Armed with the complete callgraph, CARMOT identifies the set of functions that can be optimized. For every ROI, CARMOT takes the function f where the ROI belongs to and traverses the edges of the callgraph backwards from f and tags all functions reached, including f. All functions in the program that are not tagged are optimized by CARMOT invoking the `-O3` optimizations of `clang`.

6) Reducing Pin instrumentation. CARMOT uses the call graph to also reduce Pin instrumentation by enabling the Pintool only when it cannot guarantee that a call will not jump to precompiled

code.

7) Callstack clustering. CARMOT needs to record the callstack of every PSE allocation. In a typical function, many PSEs are allocated. In a naive implementation, every time an allocation of a PSE occurs, the callstack must be computed and assigned to that PSE. However, allocations made within the same invocation of a function share the same callstack. To avoid recomputing the callstack for each PSE allocation within a function, CARMOT computes the callstack only once at the beginning of the function. The instrumentation that documents allocations can now collectively share the computed callstack instead of producing redundant callstack records that are clones of one another.

4.2.5 Pin Instrumentation

When the target program includes code outside the available sources (e.g., precompiled libraries), it is impossible to track all PSEs information with a purely compiler-based approach. However, the activity of PSEs outside the available sources must be tracked in order for the PSEC to be correct and complete. To perform this tracking, CARMOT uses dynamic binary instrumentation through a Pintool that builds upon the Pinatrace memory access tracing tool from Intel [52]. The key challenge is to efficiently communicate between the Pintool and the compiler/runtime environment of CARMOT. To overcome these challenges we use compiler injected calls to invoke our Pintool, which tracks allocations and accesses of PSEs in precompiled code and communicates them to the CARMOT runtime. This is a costly operation, but necessary to generate a correct PSEC.

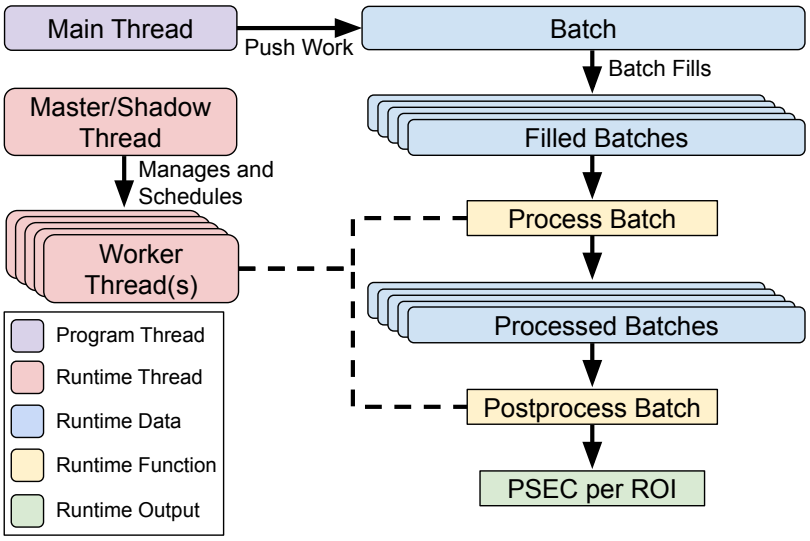


Figure 4.4: The runtime utilizes batching, shadow profiling, and pipeline parallelism to efficiently perform PSEC.

4.2.6 Runtime

CARMOT's runtime processes the uses of PSEs provided by either compiler-injected instrumentation or the Pintool. This information needs to be processed at run-time as the large amount of data collected makes storage a bottleneck.

The primary structures the runtime builds are the Active State Member Table and the ROIs' PSEC. This table captures metadata about *active* PSEs such as the callstack of their allocation and size in bytes. The runtime generates a PSEC by enacting the FSA (§4.2.1) upon PSEs for each ROI.

Figure 4.4 shows the components and the processing flow of the runtime. The *Main Thread* runs the target program. The compiler-injected instrumentation and Pintool push requests into a *batch*. Once a batch fills, it is pushed into an ordered queue of filled batches, and the instrumentation calls begin filling a new batch. The *Master/Shadow Thread* schedules filled batches for processing by *Worker Threads*. Each processed batch is then added to a second ordered queue for

final processing. The results of processed batches are updates to the active state member table and PSEC. The batches are processed following a parallel pipeline:

Processing batches. This stage processes the instrumentation calls to build the ROIs' PSEC for all PSEs. It does so by implementing the FSA in Figure 4.1 on active PSEs. Once the batch has been processed, it is then queued to the next pipeline stage and the next batch can be processed.

Postprocessing batches. This stage adds contextual information to the ROIs' PSEC and connects metadata to PSEs. This includes the callstack, escaped pointers, source code information for PSEs (file and line), and accesses.

4.3 Evaluation

We now show the effectiveness of CARMOT in generating correct programming language abstraction recommendations with acceptable overhead, compared to a naive approach that lacks PSEC-specific optimizations. We use CARMOT on 15 benchmarks from the SPEC CPU 2017, NAS [53], and PARSEC (version 3.0) [49] benchmark suites. We include every benchmark from all of these suites that already use, or are well suited for, the abstractions that CARMOT currently supports. When evaluating the performance benefits of CARMOT we used the "reference", "class C", and "native" inputs, respectively. When evaluating the overhead of CARMOT we used the "test", "class A", and "simsmall" inputs. The difference in inputs for performance and overhead results reflects how we see CARMOT being used. We use the larger, production level inputs ("reference", "class C", and "native") to evaluate performance, because the results will be indicative of the actual speedup that programs developed with CARMOT can attain. However, we use the smaller inputs ("test", "class A", and "simsmall") to evaluate overhead. We believe that these results are conservative and are also more representative of what users will experience when determining the PSEC at development time.

4.3.1 Experimental Setup

Our evaluation platform is a dual socket server with two Intel Xeon Silver 4116 CPU running at 2.1GHz. Each processor has 12 cores with 2-way hyper-threading and 16.5 MB of last level cache. The cores are supported by 125 GiB of main memory at 2400 MHz. The OS is Red Hat Enterprise Linux 8.2 (kernel 4.18.0-193.6.3). CARMOT is built on top of LLVM 9.0.0 [23], Pin 3.13 [54], and NOELLE 9.3 [55]. The baseline for both performance and overhead evaluation we show is the sequential version of each benchmark, compiled with `clang -O3 -march=native`.

4.3.2 STATS Use Case

Here we show that CARMOT can be used to build the Input, Output, and State classes required by the STATS abstraction. In the benchmarks considered, we choose the ROI for PSEC to be the code region of the STATS state dependence. CARMOT is able to accurately generate the Input, Output, State classes required by the STATS abstraction, such that they match those we manually implemented in §3. Furthermore, CARMOT was able to identify some misclassifications of PSEs that we made when manually implementing the STATS abstraction for these benchmarks. While these misclassifications have no impact on correctness, they lead to extra unnecessary copies of variables. In this case, fixing the misclassification does not lead to a noticeable speedup. However, CARMOT's ability to outperform the manual and labor-intensive classification lends to its usefulness as a tool for abstractions that can be difficult for developers to use correctly.

Figure 4.5 shows CARMOT's overhead for classifying PSEs into STATS's Input, Output, and State classes. We can see that the CARMOT overhead is one order of magnitude lower than a naive approach, because the STATS abstraction does not require the tracking of all *Use-callstacks*, a costly operation, and because the PSEC-specific optimizations of CARMOT further reduce the overhead.

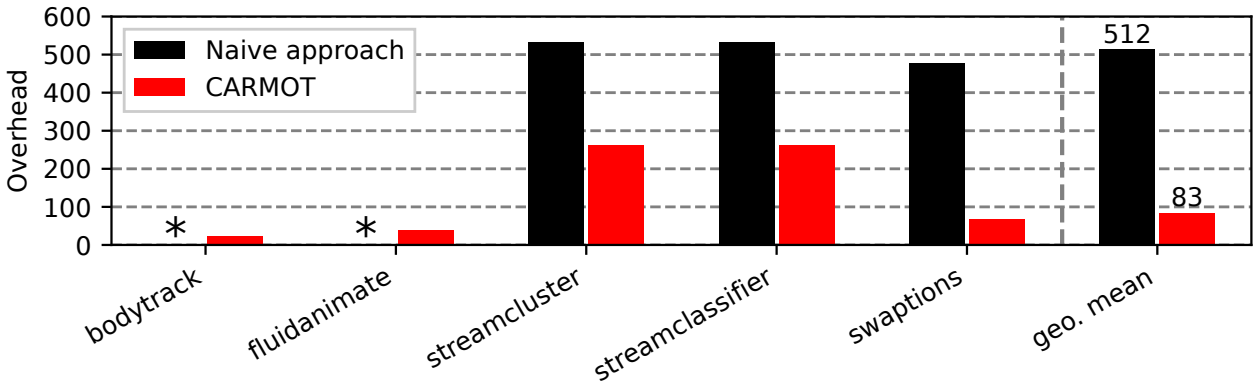


Figure 4.5: The CARMOT overhead to generate the Input-Output-State abstraction of STATS is one order of magnitude less than a naive approach.

4.3.3 OpenMP Use Case

Using PSEC, CARMOT is able to automatically generate *#pragma omp parallel for*, *#pragma omp critical*, *#pragma omp ordered*, and *#pragma omp task* annotations, and can be used by developers to verify the correctness and improve the performance of existing pragmas for a specific program execution. Many of the benchmarks we consider for this use case already use OpenMP pragmas. In this case, we choose as ROIs for PSEC the code regions of the already present OpenMP pragmas, and we verified that CARMOT's recommendations matched the original pragmas and our understanding of the parallelism in the benchmark. In cases where the benchmark is parallelized using pthreads (e.g., `swaptions` from PARSEC 3), we use as ROI the entry point function of such threads to build equivalent parallelism using CARMOT's recommended OpenMP pragmas. Furthermore, we use CARMOT to implement additional parallelization opportunities; for example, we add some OpenMP task parallelism to `mg` from NAS.

Figure 4.6 shows the speedup benefits of automatic CARMOT-generated pragmas (either verified pragmas or brand new ones) versus the original (manually extracted by the benchmarks' authors) parallelism (either through omp pragmas or pthread) for each benchmark we consider. This

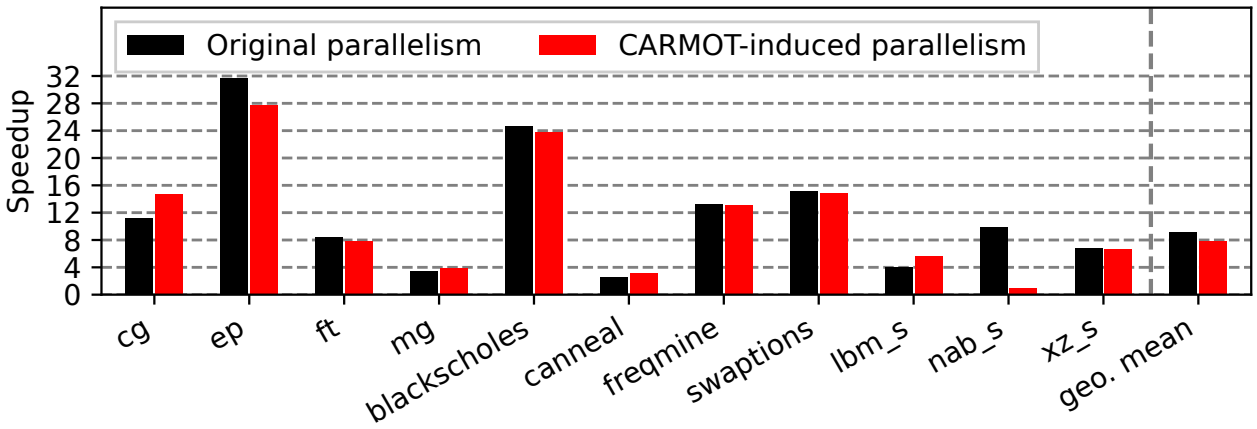


Figure 4.6: CARMOT-generated OpenMP pragmas achieve the same speedup of the original program parallelism manually implemented by a programmer. These experiments use the production-size inputs.

data shows that with CARMOT-generated pragmas, we are almost always able to achieve speedups that are as good as or better than pragmas implemented manually by a programmer. For benchmarks like `canneal` and `swaptions`, where the only original source of parallelism comes from pthreads, the new pragmas generated by CARMOT match the performance of the labor-intensive pthreads parallelism. The only exceptions are `ep` and `nab` for which CARMOT was unable to extract all parallelism potential. In both cases the main source of parallelism in these benchmarks comes from general OpenMP *#pragma omp parallel* sections that include synchronization mechanisms such as *#pragma omp barrier* or *#pragma omp master* that are abstractions currently not supported by CARMOT.

When designing new development tools, striking a balance between effectiveness and feasibility is paramount. The feasibility of CARMOT as a tool is measured by the computational overhead required to perform PSEC. Figure 4.7 shows the computational overhead of CARMOT when automatically generating OpenMP pragmas information. We compare the CARMOT overhead with a naive approach that does not employ any PSEC-specific optimization, but can still generate a

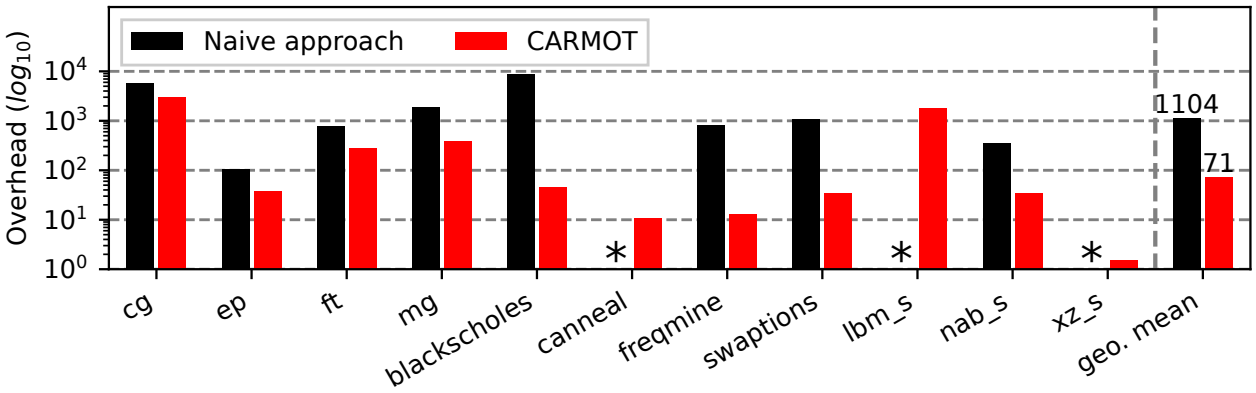


Figure 4.7: The CARMOT overhead to generate OpenMP pragma information is two orders of magnitude less than a naive approach.

correct PSEC. CARMOT outperforms the naive approach by lowering the overhead of performing PSEC by two orders of magnitude. In some cases the execution of the naive approach required an excessive amount of memory and did not complete, we mark the missing data with $*$.

To showcase the power of PSEC-specific optimizations, Figure 4.8 shows the impact of the optimizations described in §4.2.4. For the benchmarks where the naive approach finished successfully, we show in percentage the breakdown of the delta between the black and red bars of Figure 4.7 for every CARMOT optimization. The reduction of Pin instrumentation and the callgraph-based optimization, enabled by the complete callgraph of NOELLE, have the highest impact. Optimizations from 1) to 4) of §4.2.4 collaboratively enable each other to remove redundant instrumentation, for this reason we consider them together. Because they heavily rely on alias analysis, they have the highest impact in the more regular benchmarks from NAS.

4.3.4 Smart Pointers Use Case

Here we show the versatility of CARMOT on a use case unrelated to parallelization: identifying reference cycles in an ROI. In this use case we choose the ROI for PSEC to be the entire program (i.e., the entire *main()* function), since we are interested in any possible reference cycle in the

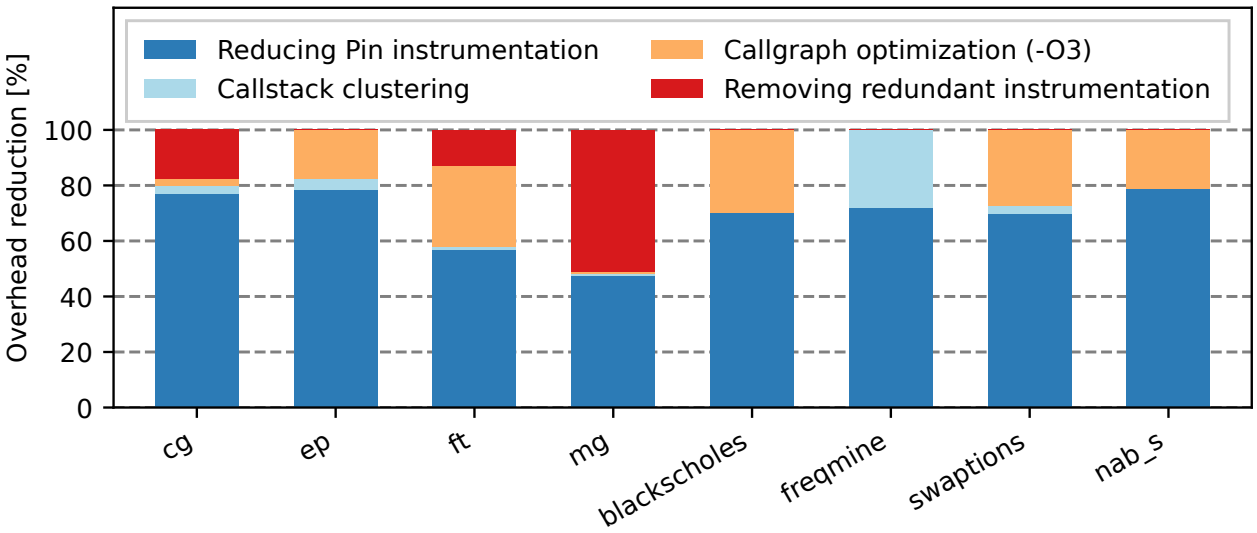


Figure 4.8: Overhead reduction of Figure 4.7 characterized per CARMOT optimization.

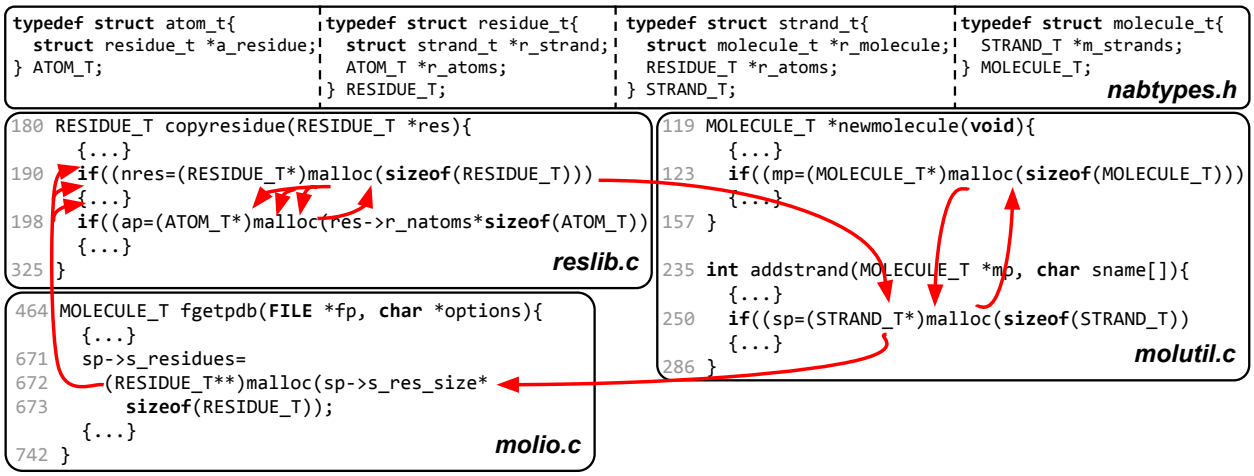


Figure 4.9: CARMOT-identified reference cycle across files, functions, and data structure in the `nab` benchmark.

program.

Figure 4.9 shows an example of a reference cycle that CARMOT identified in the `nab` benchmark of the SPEC CPU 2017 suite. This cycle spans across several different files, functions, and data structures and demonstrates the complexity of porting an existing application to use smart pointers correctly. We measure the benefit that utilizing smart pointers for this reference cycle

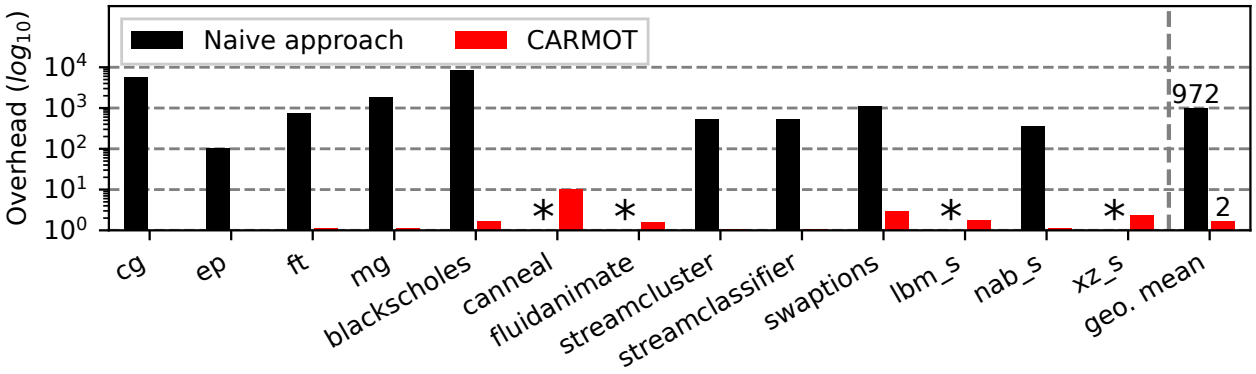


Figure 4.10: The CARMOT overhead for identifying reference cycles is two orders of magnitude less than a naive approach.

would generate for the benchmark. After correcting a naiveness in the original `nab` code, which over allocates memory, we measure the total bytes leaked by the application as 230,537 bytes. The total bytes leaked by the application that would have been realized by correctly porting this reference cycle to smart pointer is reduced to 127,633, a reduction of 44.6%.

Figure 4.10 shows the overhead of CARMOT when finding reference cycles. In this use case CARMOT needs to track only allocations of PSEs and the *Reachability Graph* of such allocations. For this reason, CARMOT's overhead is two orders of magnitude smaller than a naive approach that lacks PSEC-specific optimizations.

CHAPTER 5
RELATED WORK

5.1 STATS

STATS is related to prior work that either extracts TLP from programs or uses search to optimize program configurations (§5.1.1-5.1.2). Furthermore, because this dissertation characterizes the effects of the STATS parallelization scheme, it is also related with studies that characterize parallel workloads (§5.1.3).

5.1.1 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 [56]–[62] 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 cost 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 [16], [46], [47], [63]–[65]. 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 3.13 measures empirically this limit for some of them. One of these approaches generates compensation code [65], which is executed after the code involved in a dependence. While compensation code can avoid high inaccuracies, it does not preserve 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 a specific class of algorithms [66]–[69] 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 output quality.

Galois [70], [71] 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 [48] 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 non-determinism 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 [72]–[85] all rely on accurate data dependence analyses to identify code regions that can run safely in par-

allel. 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 [86]–[96]. 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 [97], [98]. However, they do not rely on algorithm-specific knowledge and are limited to simple data dependences of scalar values. Notably, ASC [99] speculates the entire computational state of a program. ASC performs multiple speculations of the entire program state, which result in multiple, parallel speculative executions of the program. The original program also executes at the same time, and performs checks of its own state against the predicted speculative states. If a check succeeds, the original program execution fast-forwards to the already executed speculative execution, which now stops being speculative. However, as of now this approach requires a functional simulator (which simulates the execution of x86 instructions using a transition function from one program state to the next) and has been successful only on a few, relatively simple benchmarks. STATS, instead, only needs to speculate the portion of program state that is related to a state dependence, and it does so by taking advantage of nonderminism and algorithm-specific knowledge encoded by a developer. Because the problem that STATS solves is simpler (compared to speculating the entire program state without any assistance), STATS is able to target more complex (nondeterministic) programs and run directly on commodity processors.

Manual approaches: In many multi-threaded programs (including those of PARSEC), TLP has been introduced manually using parallel programming APIs [100]–[102]. These programs

preserve all Read-After-Write actual dependences (including state dependences), which constrains TLP and overall program performance (as shown by the black lines of Figure 3.8). STATS goes beyond this limit.

5.1.2 Autotuning/Search-based Optimization

Considerable effort has gone into the general area of auto-tuning. A number of systems focus on tuning libraries in specific domains [103]–[109]. Others are designed as general auto-tuning frameworks [22], [110]–[113]. 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.

5.1.3 Parallel Workload Characterization

Most of the benchmarks [7] that we considered in §3 already include some TLP that was expressed manually by developers using parallel programming APIs like POSIX threads, OpenMP [100], and Intel TBB [101]. This TLP has been studied and characterized by prior work on multiple platforms [7], [49], [114]–[117]. The STATS compiler adds the parallelism related to state dependences to the original TLP. In this dissertation we characterized this additional parallelism both in isolation with the original TLP and when both sources of TLP are combined.

5.2 CARMOT

While CARMOT is the only tool capable of computing a complete and correct PSEC, there are other tools that enable programmers to better understand a program's behavior and how to improve it. Next, we compare CARMOT to these tools with respect to their ability to track PSEs, build aspects of the PSEC, and report back information to the user at the source code level. We categorize

these tools in three sets: memory analysis tools, parallelism discovery tools, and reference cycle discovery tools.

5.2.1 Memory Analysis

There exist many tools that investigate memory correctness such as memory leaks, double frees, and buffer over/underflow [18], [19], [52], [118]–[126] or memory bottlenecks [127]–[130]. Some of these tools report some source-code level information such as the callstack of the error site; however, none of them track any aspect of PSEC. The most notable tools that perform some tracking of PSEs are: AddressSanitizer [18], Valgrind [19], and the Pintool Pinatrace [52]. However, none of these tools track PSEs that are variables or are able to distinguish between different stack locations or globals. AddressSanitizer and Valgrind can detect memory leaks due to reference cycles that should have been garbage collected, but they cannot identify the actual cycles in the source code responsible for the leak.

5.2.2 Parallelism Discovery

Tools that identify parallelism [13]–[17], [47], [131]–[146] analyze the memory utilization of a program to identify potential parallelization opportunities using static [137], [146] and/or dynamic analysis [47], [132], [133], and profiling techniques. Their objective is orthogonal to CARMOT and its PSEC. Once potential parallel regions of a program are discovered, CARMOT can be used on those regions to understand exactly how they can be parallelized using the supported parallelism-related abstractions, verify the presence of actual parallelism, and improve it if possible.

5.2.3 Reference Cycle Discovery

To the best of our knowledge, only two approaches are able to aid programmers in finding reference counting cycles at the source code level: Xcode [147] and Distefano *et al.* [148]. Xcode works only for swift and objective-c but does not currently handle C++ smart pointers. Distefano *et al.* uses a static approach, which is limited by the accuracy of memory analysis that is known to be challenging for unmanaged languages. Neither of them can detect cycles formed in precompiled libraries.

CHAPTER 6
CONCLUSION AND FUTURE WORK

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. Dependences between instructions are the main obstacle that prevents the realization of TLP. So far, prior work has either satisfied or broken actual dependences. If actual dependences are satisfied, the program can only run sequentially and no TLP is generated. Conversely, if actual dependences are broken, there is no guarantee that the original semantics of the program and its output quality will be preserved. This dissertation proposes an alternative solution for nondeterministic programs: satisfying a subset of actual dependences, which we call state dependences, with auxiliary code. We implemented a system called STATS that takes advantage of state dependences. STATS is the first step in exploiting state dependences, and it demonstrates that it is possible to achieve large performance gains, energy savings, or output quality increases. STATS uses 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 of dependences that might yield important benefits.

Furthermore, we identified and characterized the main factors that can potentially block the performance obtained by STATS parallel binaries. Our analysis suggests that STATS can benefit from additional engineering efforts to reduce some of these factors and that the STATS execution model needs to evolve to remove the remaining performance roadblocks.

Finally, we realized that using the STATS programming language abstraction through the STATS interface to encode a state dependence and its related algorithmic-specific information

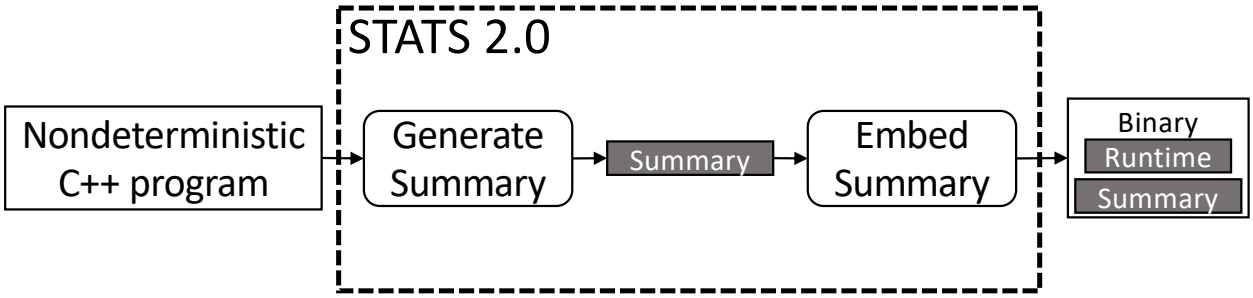


Figure 6.1: A new version of STATS (STATS 2.0) does not need to autotune on training inputs. It can use program summarization to detect the behavior of a program and tune the parallel execution.

can be challenging for developers in large code-bases. For this reason we built CARMOT, a tool that aids developers in using the STATS abstraction. We noticed that the information necessary to build an instance of the STATS abstraction is the same for several other abstractions such as many OpenMP pragmas and C++ smart pointers. This same fundamental knowledge is the PSEC of the ROI where the abstraction is applied to. We hope that CARMOT and its PSEC will help programmers to better understand their programs and to properly use the abstractions that are becoming increasingly prevalent and necessary in modern applications.

6.1 Opportunities for Future Research

6.1.1 An Improved Version of STATS with Program Summarization

The STATS autotuning phase can take a long time (e.g., for bodytrack, the STATS autotuning step takes about 3 days to find the best configuration). Also, as a program executes its behavior can change over time, because programs, including nondeterministic programs, have phases. An autotuning approach cannot capture these phases, because it treats a program as a black box, where the input is a configuration of the STATS-generated binary and the output is the chosen performance metric result (e.g., execution time). The configuration chosen by the autotuner is fixed for the whole program execution, which is sub-optimal because it cannot adapt to different program

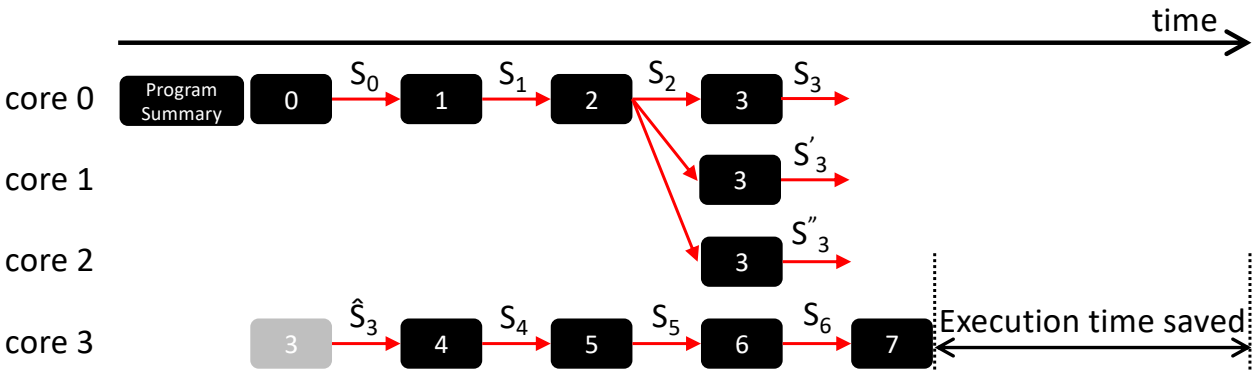


Figure 6.2: In a new STATS execution model a fast, summarized version of the original program is executed before the parallel, non-summarized version of the same program.

phases. Furthermore, the need for representative training inputs for autotuning is limiting, because they are not always available.

A solution to these issues is what we call *Program Summarization*. Every program, including nondeterministic programs, naturally trades off the accuracy of the computed output with the speed of the execution. Examples of these tradeoffs are: data type tradeoffs (e.g., a computation with float values rather than double is considerably faster, but less accurate), function tradeoffs (e.g., if a program needs to compute the square root of a number, there are many different functions available, some of them are slow but accurate, others are less accurate but faster), value tradeoffs (e.g., a loop in a program can execute multiple iterations until a convergence condition is satisfied, relaxing the convergence condition reduces the number of iterations, hence returning faster but less accurate results). Because of its Tradeoff Interface, STATS already has a mechanism to let the developer identify and tune these tradeoffs in a program. The objective of program summarization is to automatically tune these tradeoffs to maximize speed of execution at the expenses of output accuracy. The result is a much faster and approximate (hence "summarized") program.

A new version of STATS (shown in Figure 6.1) could automatically generate the summarized version of a program and embed it in the binary. Then, a new STATS execution model could exe-

cute the summarized version of the original program just before executing the STATS parallelized version of that program (Figure 6.2). The summarized version of the program would be used to detect program phases, which will then be used to tune the STATS parallel execution with a more dynamic runtime that can adapt to these phases. This removes the need for autotuning and training inputs, because we can directly use the "production input" (i.e., the input that the program will execute with) with the summarized program to quickly understand how the program behaves, and then use that knowledge to tune its STATS parallel execution that will be executed immediately after.

The challenge of program summarization lies in finding the sweet spot between a fast (summarized) program and a program that still preserves the main characteristics of the original program (e.g., its phases).

6.1.2 Improvements and Increased Abstraction Support for CARMOT

Although CARMOT can perform PSEC with reasonable overhead, we believe that there is still room for improvement. Specifically, CARMOT's runtime would benefit from a faster access to the PSEs that are being characterized into one of the four sets of PSEC (i.e., Input, Output, Cloneable, Transfer). A technique called Shadow Memory allows to do so by just computing an offset from the original PSE. Shadow Memory mirrors the original memory of a program (which contains its PSEs) at an offset that is not used by the program itself, but can be used by CARMOT's runtime. CARMOT could use Shadow Memory to quickly encode the set information for every PSE at byte granularity.

Another area of improvement for CARMOT would be increased support for more programming language abstractions such as C++ perfect forwarding, the OpenMP target pragma that enables interactions with accelerators (e.g., GPUs), and automatic detection of WARD regions [149]. We

believe that PSEC contains most of the information needed to support these abstraction, but we do not exclude the possibility that additional knowledge might need to be added to PSEC in order to provide full support for these abstractions.

REFERENCES

[1] E. A. Deiana, V. St-Amour, P. A. Dinda, N. Hardavellas, and S. Campanoni, "Unconventional parallelization of nondeterministic applications," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[2] E. A. Deiana and S. Campanoni, "Workload characterization of nondeterministic programs parallelized by stats," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2019.

[3] E. A. Deiana, B. Suchy, M. Wilkins, B. Homerding, T. McMichen, K. Dunajewski, P. Dinda, N. Hardavellas, and S. Campanoni, "Program state element characterization," in *Code Generation and Optimization (CGO)*, 2023.

[4] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, 1987.

[5] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006, ISBN: 0321486811.

[6] *LLVM Passes*, `https://llvm.org/docs/Passes.html`, Accessed: 2023-01-15.

[7] C. Bienia, "Benchmarking Modern Multiprocessors," Ph.D. dissertation, Princeton University, 2011.

[8] G. Bradski and A. Kaehler, *Learning OpenCV: Computer vision with the OpenCV library*. "O'Reilly Media, Inc.", 2008.

[9] S. D. Stoller, M. Carbin, S. V. Adve, K. Agrawal, G. E. Blelloch, Dan, Stanzione, K. A. Yelick, and M. A. Zaharia, "Future directions for parallel and distributed computing: Spx 2019 workshop report," in *NSF Workshop Reports*, 2019.

[10] *Clang Tidy*, `https://clang.llvm.org/extra/clang-tidy/`, Accessed: 2023-01-15.

[11] U. Bondhugula, J. Ramanujam, and P. Sadayappan, "PLuTo: A Practical and Fully Automatic Polyhedral Program Optimization System," *PLDI 2008 - 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008.

[12] H. Arabnejad, J. Bispo, J. G. Barbosa, and J. M. Cardoso, "An OpenMP based parallelization compiler for C applications," *Proceedings - 16th IEEE International Symposium on Parallel and Distributed Processing with Applications, 17th IEEE International Conference on Ubiquitous Computing and Communications, 8th IEEE International Conference on Big Data and Cloud Computing, 11t*, 2019.

[13] A. Raghesh, "A Framework for Automatic OpenMP Code Generation," *M. Tech thesis, Indian Institute of Technology, Madras, India*, 2011.

[14] Z. Li, R. Atre, Z. Huda, A. Jannesari, and F. Wolf, "Unveiling parallelization opportunities in sequential programs," *Journal of Systems and Software*, 2016.

[15] M. Norouzi, F. Wolf, and A. Jannesari, "Automatic construct selection and variable classification in OpenMP," *Proceedings of the International Conference on Supercomputing*, 2019.

[16] S. Misailovic, D. Kim, and M. Rinard, "Parallelizing Sequential Programs with Statistical Accuracy Tests," in *ACM Trans. Embed. Comput. Syst. (TECS)*, 2013.

[17] Z. Wang, G. Tournavitis, B. Franke, and M. F. P. O'boyle, "Integrating profile-driven parallelism detection and machine-learning-based mapping," *ACM Transactions on Architecture and Code Optimization*, 2014.

[18] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," *Proceedings of the 2012 USENIX Annual Technical Conference, USENIX ATC 2012*, 2012.

[19] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," *ACM SIGPLAN Notices*, 2007.

[20] *Standard ECMA-335 Common Language Infrastructure (CLI)*, 3rd, ECMA, Rue du Rhone 114 CH-1204 Geneva, 2005.

[21] M. Felleisen, R. B. Findler, M. Flatt, S. Krishnamurthi, E. Barzilay, J. McCarthy, and S. Tobin-Hochstadt, "The Racket Manifesto," in *Summit on Advances in Programming Languages (SNAPL)*, 2015.

[22] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "OpenTuner: An Extensible Framework for Program Autotuning," in *Parallel Architectures and Compilation Techniques (PACT)*, 2014.

[23] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Code Generation and Optimization (CGO)*, 2004.

[24] T. Schubert, A. Gkogkidis, T. Ball, and W. Burgard, "Automatic initialization for skeleton tracking in optical motion capture," in *2015 IEEE International Conference on Robotics and Automation (ICRA)*, 2015.

[25] J. Schmidt and M. Castrillón-Santana, "Automatic initialization for body tracking," *Proceedings of the Third International Conference on Computer Vision Theory and Applications*, 2008.

[26] A. Aristidou and J. Lasenby, "Real-time marker prediction and cor estimation in optical motion capture," *The Visual Computer*, 2013.

[27] E. De Aguiar, C. Theobalt, and H.-P. Seidel, "Automatic learning of articulated skeletons from 3d marker trajectories," in *Advances in Visual Computing: Second International Symposium (ISVC)*, 2006.

[28] A. Kirk, J. F. O'Brien, and D. A. Forsyth, "Skeletal parameter estimation from optical motion capture data," in *ACM SIGGRAPH 2004 Sketches*, 2004.

[29] M. Klous and S. Klous, "Marker-based reconstruction of the kinematics of a chain of segments: A new method that incorporates joint kinematic constraints," *Journal of Biomechanical Engineering*, 2010.

[30] J. Meyer, M. Kuderer, J. Müller, and W. Burgard, "Online marker labeling for fully automatic skeleton tracking in optical motion capture," in *2014 IEEE International Conference on Robotics and Automation (ICRA)*, 2014.

[31] M. Ringer and J. Lasenby, "A procedure for automatically estimating model parameters in optical motion capture," *Image and Vision Computing*, 2004.

[32] L. A. Schwarz, A. Mkhitaryan, D. Mateus, and N. Navab, "Human skeleton tracking from depth data using geodesic distances and optical flow," *Image Vision Comput.*, 2012.

[33] V. B. Zordan and N. C. Van Der Horst, "Mapping optical motion capture data to skeletal motion using a physical model," in *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, 2003.

[34] H. Kim, N. P. Johnson, J. W. Lee, S. A. Mahlke, and D. I. August, "Automatic speculative DOALL for clusters," in *Code Generation and Optimization (CGO)*, 2012.

[35] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard, "Managing Performance vs. Accuracy Trade-offs with Loop Perforation," in *European Conference on Foundations of Software Engineering (ESEC/FSE)*, 2011.

[36] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard, "Quality of Service Profiling," in *International Conference on Software Engineering (ICSE)*, 2010.

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

[38] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard, "Dynamic Knobs for Responsive Power-aware Computing," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[39] S. D. Casey, *How to Determine the Effectiveness of Hyper-Threading Technology with an Application*, `https://goo.gl/ycuL6E`, Accessed: 2018-01-14, 2011.

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

[41] S. Blagodurov, S. Zhuravlev, A. Fedorova, and A. Kamali, "A case for NUMA-aware contention management on multicore systems," in *Parallel Architectures and Compilation Techniques (PACT)*, 2010.

[42] B. Lepers, V. Quéma, and A. Fedorova, "Thread and Memory Placement on NUMA Systems: Asymmetry Matters.," in *USENIX Annual Technical Conference (USENIX ATC)*, 2015.

[43] G. Piccoli, H. N. Santos, R. E. Rodrigues, C. Pousa, E. Borin, and F. M. Quintão Pereira, "Compiler support for selective page migration in NUMA architectures," in *Parallel Architectures and Compilation Techniques (PACT)*, 2014.

[44] S. Srikanthan, S. Dwarkadas, and K. Shen, "Data Sharing or Resource Contention: Toward Performance Transparency on Multicore Systems," in *USENIX Annual Technical Conference (USENIX ATC)*, 2015.

[45] S. Srikanthan, S. Dwarkadas, and K. Shen, "Coherence stalls or latency tolerance: informed CPU scheduling for socket and core sharing," in *USENIX Annual Technical Conference (USENIX ATC)*, 2016.

[46] A. Udupa, K. Rajan, and W. Thies, "ALTER: Exploiting breakable dependences for parallelization," in *Programming Language Design and Implementation (PLDI)*, 2011.

[47] S. Campanoni, G. Holloway, G.-Y. Wei, and D. Brooks, "HELIX-UP: Relaxing Program Semantics to Unleash Parallelization," in *Code Generation and Optimization (CGO)*, 2015.

[48] 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.

[49] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[50] J. K. Hollingsworth, "Critical path profiling of message passing and shared-memory programs," *IEEE Transactions on Parallel and Distributed Systems*, 1998.

[51] R. Johnson, D. Pearson, and K. Pingali, "The program structure tree: Computing control regions in linear time," in *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, 1994.

[52] *Pinatrace*, `https://software.intel.com/sites/landingpage/pintool/docs/71313/Pin/html/`, Accessed: 2023-01-15.

[53] D. Bailey, E. Barszcz, B. J.T, B. D.S, C. R.L, D. D, F. R.A, P. Frederickson, L. T.A, R. Schreiber, H. Simon, V. Venkatakrishnan, and W. K, "The nas parallel benchmarks," *International Journal of High Performance Computing Applications*, 1991.

[54] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," *Acm sigplan notices*, 2005.

[55] A. Matni, E. A. Deiana, Y. Su, L. Gross, S. Ghosh, S. Apostolakis, Z. Xu, Z. Tan, I. Chaturvedi, D. I. August, and S. Campanoni, "NOELLE Offers Empowering LLvm Extensions," in *International Symposium on Code Generation and Optimization, 2022. CGO 2022.*, 2022.

[56] S. Campanoni, K. Brownell, S. Kanev, T. M. Jones, G.-Y. Wei, and D. Brooks, "HELIX-RC: An Architecture-compiler Co-design for Automatic Parallelization of Irregular Programs," in *International Symposium on Computer Architecuture (ISCA)*, 2014.

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

[58] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, N. Ranganathan, D. Burger, S. W. Keckler, R. G. McDonald, and C. R. Moore, "TRIPS: A polymorphous architecture for exploiting ILP, TLP, and DLP," in *Transactions on Architecture and Code Optimization (TACO)*, 2004.

[59] B. Robatmil, D. Li, H. Esmaeilzadeh, S. Govindan, A. Smith, A. Putnam, D. Burger, and S. W. Keckler, "How to Implement Effective Prediction and Forwarding for Fusable Dynamic Multicore Architectures," in *High-Performance Computer Architecture (HPCA)*, 2013.

[60] S. L. Scott, "Synchronization and Communication in the T3E Multiprocessor," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1996.

[61] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. Brown III, and A. Agarwal, "On-Chip Interconnection Architecture of the Tile Processor," in *International Symposium on Microarchitecture (MICRO)*, 2007.

[62] S. Borkar, R. Cohn, G. Cox, S. Gleason, T. Gross, H. T. Kung, M. Lam, B. Moore, C. Peterson, J. Pieper, L. Rankin, P. S. Tseng, J. Sutton, J. Urbanski, and J. Webb, "iWarp: An Integrated Solution to High-Speed Parallel Computing," in *International Conference on Supercomputing (ICS)*, 1988.

[63] L. Renganarayana, V. Srinivasan, R. Nair, and D. Prener, "Programming with relaxed synchronization," in *Relaxing synchronization for multicore and manycore scalability (RACES)*, 2012.

[64] R. Akram, M. M. U. Alam, and A. Muzahid, "Approximate Lock: Trading off Accuracy for Performance by Skipping Critical Sections," in *International Symposium on Software Reliability Engineering (ISSRE)*, 2016.

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

[66] K. Vora, S. C. Koduru, and R. Gupta, "ASPIRE: Exploiting asynchronous parallelism in iterative algorithms using a relaxed consistency based DSM," in *Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2014.

[67] T. M. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, "Project Adam: Building an Efficient and Scalable Deep Learning Training System.," in *Operating Systems Design and Implementation (OSDI)*, 2014.

[68] J. Meng, S. Chakradhar, and A. Raghunathan, "Best-effort parallel execution framework for recognition and mining applications," in *International Symposium on Parallel and Distributed Processing (IPDPS)*, 2009.

[69] J. Meng, A. Raghunathan, S. T. Chakradhar, and S. Byna, "Exploiting the forgiving nature of applications for scalable parallel execution," in *International Symposium on Parallel and Distributed Processing (IPDPS)*, 2010.

[70] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui, "The Tao of Parallelism in Algorithms," in *Programming Language Design and Implementation (PLDI)*, 2011.

[71] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew, "Optimistic Parallelism Requires Abstractions," in *Programming Language Design and Implementation (PLDI)*, 2007.

[72] S. Campanoni, T. Jones, G. Holloway, V. J. Reddi, G.-Y. Wei, and D. Brooks, "HELIX: Automatic Parallelization of Irregular Programs for Chip Multiprocessing," in *Code Generation and Optimization (CGO)*, 2012.

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

[74] J. Huang, A. Raman, T. B. Jablin, Y. Zhang, T.-H. Hung, and D. I. August, "Decoupled Software Pipelining Creates Parallelization Opportunities," in *Code Generation and Optimization (CGO)*, 2010.

[75] G. Ottoni, R. Rangan, A. Stoler, and D. I. August, "Automatic Thread Extraction with Decoupled Software Pipelining," in *International Symposium on Microarchitecture (MICRO)*, 2005.

[76] G. Tournavitis, Z. Wang, B. Franke, and M. F. 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.

[77] A. Hurson, J. T., L. M., and K. Lee, "Parallelization of DOALL and DOACROSS Loops - A Survey," in *Advances in Computers*, 1997.

[78] A. Aiken and A. Nicolau, "Perfect Pipelining: A New Loop Parallelization Technique," in *European Symposium on Programming (ESOP)*, 1988.

[79] C.-Z. Xu and V. Chaudhary, "Time Stamp Algorithms for Runtime Parallelization of DOACROSS Loops with Dynamic Dependences," in *Transactions on Parallel and Distributed Systems (TPDS)*, 2001.

[80] E. Raman, G. Ottoni, A. Raman, M. J. Bridges, and D. I. August, "Parallel-stage Decoupled Software Pipelining," in *Code Generation and Optimization (CGO)*, 2008.

[81] D.-K. Chen and P.-C. Yew, "On Effective Execution of Nonuniform DOACROSS Loops," in *Transactions on Parallel and Distributed Systems (TPDS)*, 1996.

[82] D.-K. Chen and P.-C. Yew, "Redundant Synchronization Elimination for DOACROSS Loops," in *Transactions on Parallel and Distributed Systems (TPDS)*, 1999.

[83] D. Liu, Z. Shao, M. Wang, M. Guo, and J. Xue, "Optimal Loop Parallelization for Maximizing Iteration-level Parallelism," in *Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, 2009.

[84] K. Ebcioglu and A. Nicolau, "A Global Resource-constrained Parallelization Technique," in *International Conference on Supercomputing (ICS)*, 1989.

[85] K. S. McKinley, "Evaluating Automatic Parallelization for Efficient Execution on Shared-memory Multiprocessors," in *International Conference on Supercomputing (ICS)*, 1994.

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

[87] L. Hammond, B. A. Hubbert, M. Siu, M. K. Prabhu, M. K. Chen, and K. Olukotun, "The Stanford Hydra CMP," in *International Symposium on Microarchitecture (MICRO)*, 2000.

[88] T. A. Johnson, R. Eigenmann, and T. N. Vijaykumar, "Speculative Thread Decomposition Through Empirical Optimization," in *Principles and Practice of Parallel Programming (PPoPP)*, 2007.

[89] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas, "POSH: A TLS Compiler That Exploits Program Structure," in *Principles and Practice of Parallel Programming (PPoPP)*, 2006.

[90] J. G. Steffan, C. Colohan, A. Zhai, and T. C. Mowry, "The STAMPede Approach to Thread-level Speculation," in *Transactions on Computer Systems (TOC)*, 2005.

[91] A. Zhai, J. G. Steffan, C. B. Colohan, and T. C. Mowry, "Compiler and Hardware Support for Reducing the Synchronization of Speculative Threads," in *Transactions on Architecture and Code Optimization (TACO)*, 2008.

[92] H. Zhong, M. Mehrara, S. A. Lieberman, and S. A. Mahlke, "Uncovering hidden loop level parallelism in sequential applications," in *High-Performance Computer Architecture (HPCA)*, 2008.

[93] A. Raman, H. Kim, T. R. Mason, T. B. Jablin, and D. I. August, "Speculative Parallelization Using Software Multi-threaded Transactions," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[94] L. Han, W. Liu, and J. M. Tuck, "Speculative Parallelization of Partial Reduction Variables," in *Code Generation and Optimization (CGO)*, 2010.

[95] C. Wang, Y. Wu, E. Borin, S. Hu, W. Liu, D. Sager, T.-f. Ngai, and J. Fang, "Dynamic Parallelization of Single-threaded Binary Programs Using Speculative Slicing," in *International Conference of Supercomputing (ICS)*, 2009.

[96] W. Ahn, S. Qi, M. Nicolaides, J. Torrellas, J.-W. Lee, X. Fang, S. Midkiff, and D. Wong, "BulkCompiler: high-performance sequential consistency through cooperative compiler and hardware support," in *International Symposium on Microarchitecture (MICRO)*, 2009.

[97] J. González and A. González, "The potential of data value speculation to boost ILP," in *International Conference on Supercomputing (ICS)*, 1998.

[98] C.-Y. Fu, M. D. Jennings, S. Y. Larin, and T. M. Conte, "Value speculation scheduling for high performance processors," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1998.

[99] A. Waterland, E. Angelino, R. P. Adams, J. Appavoo, and M. Seltzer, "Asc: Automatically scalable computation," *SIGARCH Comput. Archit. News*, 2014.

[100] L. Dagum and R. Menon, "OpenMP: An Industry-Standard API for Shared-Memory Programming," in *IEEE Comput. Sci. Eng.*, 1998.

[101] C. Pheatt, "Intel&Reg; Threading Building Blocks," in *J. Comput. Sci. Coll.*, 2008.

[102] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems," in *IEEE Des. Test*, 2010.

[103] A. Ali, L. Johnsson, and J. Subhlok, "Scheduling FFT Computation on SMP and Multicore Systems," in *International Conference on Supercomputing (ICS)*, 2007.

[104] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel, "Optimizing Matrix Multiply Using PHiPAC: A Portable, High-performance, ANSI C Coding Methodology," in *International Conference on Supercomputing (ICS)*, 1997.

[105] R. C. Whaley and J. J. Dongarra, "Automatically Tuned Linear Algebra Software," in *Supercomputing Conference (SC)*, 1998.

[106] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," in *International Symposium on Microarchitecture (MICRO)*, 2005.

[107] C. Kessler and W. Löwe, "Optimized Composition of Performance-aware Parallel Components," in *Concurr. Comput. : Pract. Exper.*, 2012.

[108] E.-J. Im and K. A. Yelick, "Optimizing Sparse Matrix Computations for Register Reuse in SPARSITY," in *International Conference on Computational Sciences (ICCS)*, 2001.

[109] Y. Voronenko, F. de Mesmay, and M. Püschel, "Computer Generation of General Size Linear Transform Libraries," in *Code Generation and Optimization (CGO)*, 2009.

[110] C. Ţăpuş, I.-H. Chung, and J. K. Hollingsworth, "Active Harmony: Towards Automated Performance Tuning," in *Supercomputing Conference (SC)*, 2002.

[111] J. Ansel, Y. L. Wong, C. Chan, M. Olszewski, A. Edelman, and S. Amarasinghe, "Language and Compiler Support for Auto-tuning Variable-accuracy Algorithms," in *Code Generation and Optimization (CGO)*, 2011.

[112] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe, "PetaBricks: A Language and Compiler for Algorithmic Choice," in *Programming Language Design and Implementation (PLDI)*, 2009.

[113] P. M. Phothilimthana, J. Ansel, J. Ragan-Kelley, and S. Amarasinghe, "Portable Performance on Heterogeneous Architectures," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[114] C. Bienia, S. Kumar, and K. Li, "Parsec vs. splash-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors," in *International Symposium on Workload Characterization (IISWC)*, 2008.

[115] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, 2008.

[116] C. Bienia and K. Li, "Characteristics of workloads using the pipeline programming model," in *Workshop on Emerging Applications and Many-core Architecture*, 2010.

[117] C. Bienia and K. Li, "Fidelity and scaling of the parsec benchmark inputs," in *International Symposium on Workload Characterization (IISWC)*, 2010.

[118] *Mpatrol*, `http://mpatrol.sourceforge.net/`, Accessed: 2023-01-15.

[119] *dmalloc*, `https://dmalloc.com/`, Accessed: 2023-01-15.

[120] *jemalloc*, `http://jemalloc.net/`, Accessed: 2023-01-15.

[121] *MemWatch*, `https://www.linkdata.se/sourcecode/memwatch/`, Accessed: 2023-01-15.

[122] *Mtrace*, `http://man7.org/linux/man-pages/man3/mtrace.3.html`, Accessed: 2023-01-15.

[123] *Electric Fence*, `https://linux.die.net/man/3/efence`, Accessed: 2023-01-15.

[124] D. Evans and D. Larochelle, "Improving security using extensible lightweight static analysis," *IEEE Software*, 2002.

[125] *Duma*, `https://www.linuxlinks.com/duma/`, Accessed: 2023-01-15.

[126] *Intel Inspector*, `https://software.intel.com/en-us/inspector`, Accessed: 2023-01-15.

[127] S. Miucin, C. Brady, and A. Fedorova, "End-to-End Memory Behavior Profiling with DINAMITE," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016.

[128] A. Pesterev, N. Zeldovich, and R. T. Morris, "Locating cache performance bottlenecks using data profiling," in *Proceedings of the 5th European Conference on Computer Systems*, 2010.

[129] R. Lachaize, B. Lepers, and V. Quéma, "Memprof: A memory profiler for numa multicore systems," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, 2012.

[130] C. McCurdy and J. Vetter, "Memphis: Finding and fixing numa-related performance problems on multi-core platforms," in *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, 2010.

[131] S. Garcia, D. Jeon, C. M. Louie, and M. B. Taylor, "Kremlin: Rethinking and rebooting gprof for the multicore age," *ACM SIGPLAN Notices*, 2011.

[132] S. Apostolakis, Z. Xu, Z. Tan, G. Chan, S. Campanoni, and D. I. August, "SCAF: A Speculation-Aware Collaborative Dependence Analysis Framework," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020.

[133] N. Murphy, T. Jones, R. Mullins, and S. Campanoni, "Performance implications of transient loop-carried data dependences in automatically parallelized loops," in *Compiler Construction (CC)*, 2016.

[134] C. Hammacher, K. Streit, S. Hack, and A. Zeller, "Profiling java programs for parallelism," in *2009 ICSE Workshop on Multicore Software Engineering*, 2009.

[135] A. Rountev, K. Van Valkenburgh, D. Yan, and P. Sadayappan, "Understanding parallelism-inhibiting dependences in sequential java programs," in *2010 IEEE International Conference on Software Maintenance*, 2010.

[136] F. Allen, M. Burke, R. Cytron, J. Ferrante, and W. Hsieh, "A framework for determining useful parallelism," in *Proceedings of the 2nd international conference on Supercomputing*, 1988.

[137] N. P. Johnson, J. Fix, S. R. Beard, T. Oh, T. B. Jablin, and D. I. August, "A collaborative dependence analysis framework," in *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2017.

[138] M. Kulkarni, M. Burtscher, R. Inkulu, K. Pingali, and C. Casçaval, "How much parallelism is there in irregular applications?" *ACM sigplan notices*, 2009.

[139] Y. He, C. E. Leiserson, and W. M. Leiserson, "The cilkview scalability analyzer," in *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, 2010.

[140] J. R. Larus, "Loop-level parallelism in numeric and symbolic programs," *IEEE Transactions on Parallel and Distributed Systems*, 1993.

[141] X. Zhang, A. Navabi, and S. Jagannathan, "Alchemist: A transparent dependence distance profiling infrastructure," in *2009 International Symposium on Code Generation and Optimization*, 2009.

[142] M. Kim, H. Kim, and C.-K. Luk, "Sd3: A scalable approach to dynamic data-dependence profiling," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010.

[143] P. Wu, A. Kejariwal, and C. Caşcaval, "Compiler-driven dependence profiling to guide program parallelization," in *International Workshop on Languages and Compilers for Parallel Computing*, 2008.

[144] C. von Praun, R. Bordawekar, and C. Cascaval, "Modeling optimistic concurrency using quantitative dependence analysis," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, 2008.

[145] K. Kennedy, K. S. Mckinley, and C.-W. Tseng, "Interactive parallel programming using the parascope editor," *IEEE Transactions on Parallel & Distributed Systems*, 1991.

[146] Y. Sui and J. Xue, "SVF: interprocedural static value-flow analysis in LLVM," in *Proceedings of the 25th international conference on compiler construction*, 2016.

[147] *Xcode*, `https://developer.apple.com/xcode/`, Accessed: 2023-01-15.

[148] D. S. Distefano, C. Calcagno, and D. Churchill, *Detecting and remedying memory leaks caused by object reference cycles*, US Patent 10,296,314, 2019.

[149] M. Wilkins, S. Westrick, V. Kandiah, A. Bernat, B. Suchy, E. A. Deiana, S. Campanoni, U. A. Acar, P. Dinda, and N. Hardavellas, "Warden: Specializing cache coherence for high-level parallel languages," in *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization*, 2023.

APPENDIX A
THE LLVM COMPILER OPTIMIZATION MEM2REG INTRODUCES AMBIGUITIES BETWEEN SOURCE CODE AND IR

The LLVM compiler optimization `mem2reg` promotes local variables of a function to virtual registers of the LLVM's IR. In more detail, the `clang` front-end maps each local variable of a function to a stack location in the IR. Therefore, each access to such source-code variable involves a load (when it is read) and a store (when it is written). The middle-end `mem2reg` pass maps (when possible) stack locations to a set of virtual registers in IR to avoid loads and stores related to it. This transformation not only optimizes the code, but it also unlocks many other analyses and transformations. In other words, without `mem2reg` little can be optimized in the middle-end.

Unfortunately, using `mem2reg` in the whole program makes PSEC impossible. This is because of two reasons: 1) `mem2reg` introduces ambiguities about which variable in the source code corresponds to a given virtual register, 2) it hides read operations of local variables because source code operations such as `varX=varY;` do not exist in SSA form. The source code in Listing A.1 and its corresponding LLVM IR in Listing A.2 showcase these issues. The IR in Listing A.2 has been generated by only applying the mem2reg optimization to the source code in Listing A.1.

Problem 1) can be observed by looking at the `printf()` statement. In the source code of Listing A.1 it is clear that variable `a` is being read and printed to standard output. However, the corresponding instruction in the IR of Listing A.2 reads and prints the phi-node `%.0`. The only way to infer which variable in the source code `%.0` corresponds to is by looking at the LLVM intrinsic `llvm.dbg.value()`. The semantics of `llvm.dbg.value()` predicates that the first argument is the new value assigned to the second argument, which in this case represents a

```
int main(int argc, char *argv[]){                    1
  int a = 42;                                        2
  int b = 43;                                        3
  if (a > argc){                                     4
    a = 44;                                          5
  }                                                  6
  b = a;                                             7
  printf("%d\n", a);                                 8
  return 0;                                          9
}                                                    10
```

Figure A.1: The semantics of operations at lines 7 and 8 must be preserved to compute a correct PSEC.

source code variable (wrapped as metadata). In our example the new value `%.0` is assigned to both `!273` (metadata for variable `a`) and `!274` (metadata for variable `b`) and then used in the `printf()` instruction. This creates an ambiguity about which variable (`a` or `b`) is actually being read and printed, which results in an incorrect PSEC.

Problem 2) manifests itself in the `b=a;` statement of Listing A.1. From the source code it is clear that variable `a` is read and variable `b` is written. In the corresponding IR of Listing A.2 the fact that `a` is read disappears because operations such as `varX=varY;` are not represented in SSA form. This also leads to an incorrect PSEC.

```
define i32 @main(i32, i8**) {
  call void @llvm.dbg.value(i32 %0, !270)
  call void @llvm.dbg.value(i8** %1, !272)
  call void @llvm.dbg.value(i32 42, !273)
  call void @llvm.dbg.value(i32 43, !274)
  %3 = icmp sgt i32 42, %0
  br i1 %3, label %4, label %5

4: ; preds = %2
  call void @llvm.dbg.value(i32 44, !273)
  br label %5

5: ; preds = %4, %2
  %.0 = phi i32 [ 44, %4 ], [ 42, %2 ]
  call void @llvm.dbg.value(i32 %.0, !273)
  call void @llvm.dbg.value(i32 %.0, !274)
  %6 = call i32 @printf(..., i32 %.0)
  ret i32 0
}

!270 = !DILocalVariable(name: "argc", ...)
!272 = !DILocalVariable(name: "argv", ...)
!273 = !DILocalVariable(name: "a", ...)
!274 = !DILocalVariable(name: "b", ...)
```

Figure A.2: Promoting local variables to virtual registers in SSA form introduces ambiguities that result in an incorrect PSEC.