UNIVERSITY OF CALIFORNIA
RIVERSIDE

Declarative Profiling for Parallel Systems

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Zachary Mitchell Benavides

September 2018

Dissertation Committee:

Dr. Rajiv Gupta, Chairperson
Dr. Nael Abu-Ghazaleh
Dr. Daniel Wong
Dr. Zhijia Zhao

Acknowledgments

First and foremost, I would like to thank my advisor, Professor Rajiv Gupta. He gave me a second chance, without which I would never have made it here. He believed in me even when I didn't believe in myself. His bottomless well of support and encouragement has been a an ever-present source of strength for me, for which I will be forever grateful.

I would like to thank Dr. Xiangyu Zhang for his invaluable assistance with my first paper. His insightful observations helped shape the profile format we designed for the better.

I would also like to thank my dissertation committee members: Dr. Nael Abu-Ghazaleh, Dr. Daniel Wong, and Dr. Zhijia Zhao for their valuable feedback, and remarkable flexibility with respect to scheduling.

I would like to express my sincere thanks to my lab-mates over the years: Keval, Vineet, Amlan, Farzad, Sai, Hongbo, Bruce, Arash, Bo and Gurneet. You guys were always there to lend an ear to my frustrations or crazy ideas. You're a constant source of inspiration.

I would also like to thank the professors under whom I had the privelege of being a TA: in particular Dr. Chinya Ravishankar and Dr. Stephano Lonardi. It was a pleasure to work with you and learn your philosophies on teaching.

Finally, I would like to thank the students who had me as a teaching assistant. I enjoyed every minute of sharing my passion for computer science with you. I hope I served you well.

Last, but not least, I would like to acknowledge the support I received from the National Science Foundation under grants CCF-1318103 and CCF-1524852, and the U.S.

Department of Education for the very generous GAANN fellowship.

To my family.

ABSTRACT OF THE DISSERTATION

Declarative Profiling for Parallel Systems

by

Zachary Mitchell Benavides

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, September 2018
Dr. Rajiv Gupta, Chairperson

The popularity of parallel systems for building high performance software only continues to rise. Programming these systems has always been a challenging task, and ensuring that they are performing optimally even more so. To assist programmers in this space, a wealth of research has been conducted into building profilers for these systems. Unsurprisingly, balancing the requirements of utility, accuracy, and overhead make this also a challenging task. While existing profilers do an admirable job of accomplishing their stated goals, they all suffer from a lack of flexibility. The toolbox of the parallel programmer is filled to the brim with finely crafted specialized tools, but hardly any general ones. Some require the use of a specific programming language or threading library. Others are closely coupled with the underlying hardware and assume the presence of specific monitoring support therein. Many are restricted to only one type of parallel system, such as shared memory multicore machines. To make matters worse, since these tools are all independent they have distinct interfaces, output formats, and requirements for their use. This makes performance analysis and debugging of parallel programs a needlessly frustrating task.

In this thesis, we propose and develop a new system for profiling parallel systems called Context Sensitive Parallel Execution Profiles (CSPs) which is vastly more flexible than existing options. CSPs adopt a declarative approach in which the developer uses our annotation language to specify code regions of interest, and our query language to specify quantities to measure in terms of those regions. CSPs do not require the use of a specific language or threading library. They use only widely available hardware features, making them mostly platform agnostic.

We first implement our system for shared memory multicore machines and show that it has low overhead, high accuracy, and can be used to diagnose and repair performance problems in real parallel programs. In a test using the Parsec benchmark suite, time overheads were typically less than 5%, and peak memory overheads were less than 46%. Measurements made using CSPs allowed us to optimize the execution of two of the programs by 36% and 17% respectively.

We then adapt our implementation to the distributed space, enabling the profiling of clusters of multicore machines. A fundamental problem in distributed profiling is that of timestamp synchronization, which involves the meaningful comparison of timestamps taken on different machines. We developed a new algorithm for timestamp synchronization which is up to 53.3% more accurate than existing algorithms. We further exhibit the flexibility of our system by extending it to compute a variant of causal profiles (a popular type of profile recently developed for shared memory systems) for distributed systems.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the landscape of modern computing, few concepts have achieved the ubiquity of parallelism. Its influence can be seen in devices as small as smart phones and as large as supercomputers. It has left its fingerprint on the design of components ranging from general purpose CPUs to specialized accelerators like GPUs. Its presence can be felt in the simplest imperative languages and the most complicated declarative languages. Its widespread adoption and time tested prevalence are a testament to its potential for improving performance.

Despite this ubiquity, realizing these performance improvements through parallel programming remains a frustrating and difficult task. Even after a programmer has gone through the painstaking process of constructing a parallel version and eliminating any concurrency bugs which affect correctness, they often find themselves in demoralizing possession of a parallel program which runs only marginally faster (and occasionally even slower!) than the sequential version with which they started. The doubt begins to creep slowly inward, smothering the hope of linear speedup. Is there a bug in the input partitioning scheme, causing some threads to be crushed under the weight of an uneven work distribution? Is there some shared resource for which the contention is high, causing threads to waste their time bickering like siblings at the dinner table? Or is it simply that the inherently sequential portion of the program is dominant, and our wrists are bound by the iron shackles of Amdahl's law?

What recourse does the beleaguered programmer have in such a dire situation? They can carefully examine their program line by line, mentally executing the code and methodically considering the performance implications of each component. But even for the most impressive intellect, this strategy would be tenable only for the simplest of programs. They need something more robust, more scalable, and less mentally exhausting than careful thought. So they reach into their digital toolbox and grab their profiler.

In the dark ages of sequential programming, the profiler of choice was GProf. It seemed almost a panacea. You presented it with your program and some input, and it returned to you an ordered list of the routines you should optimize, generated by measuring the percentage of execution time spent in each routine. Unfortunately, the oracular prowess of GProf was limited to sequential programs. With the introduction of parallelism, a key assumption of GProf, namely that the total execution time was the sum of the execution times of its constituent procedures, was shattered. And thus the search for a parallel profiler as effective as GProf was for sequential programs began. And thus began the search for a profiler as effective for parallel programs as GProf was for sequential programs.

That search continues today, for the path to such a profiler is strewn with complication. Many attempts have been made, but all have fallen short of the ideal primarily because of their inflexibility. Profilers based on the notion of normalized execution time generalize the GProf format by weighting regions based on the level of concurrency present when they are executed. [5, 14, 15, 25] They therefore seek to find regions of the program which will probably yield overall performance gains when optimized. These profilers suffer from an a priori decision about the types of code regions that the programmer will find interesting. If the statically selected type of region is too large (such as a function), then the profiler may not detect a problem at all, or if it does, that detection may be of little help because it can leave the programmer with a search space that is still too large to explore manually. On the other hand, if the statically selected type of region is too small (such as a basic block), then isolating the root cause of a detected performance problem can be complicated by the lack of contextual information. For example, a detected basic block may be a problem only when it is visited from one function but not another.

In contrast, profilers based on critical path analysis seek to find regions which will certainly yield some speedup when optimized. [10, 20, 22, 31] While these types of profilers succeed in finding regions which will yield some speedup if optimized, they provide no indication as to how much speedup can be realized overall. Additionally, they generally require the use of a specific communication library or programming language, which severely limits their flexibility.

Yet another class of profilers are based on hardware performance counters. [4, 38] These profilers diagnose performance problems by looking for their low-level symptoms, such as cache miss rates and branch predictor failures. Implemented alongside the hardware, these profilers have excellent overhead characteristics, but can be highly inflexible. Not only are the set of counters available different from machine to machine, an optimization performed for one machine might actually lead to slowdowns when executed on a different machine.

These different forms of inflexibility are a result of the myriad challenges that arise when designing a profiler for parallel programs. What would an ideal parallel profiler look like? Below we enumerate a profilers desired features.

First, it should be flexible. There are many ways of introducing parallelism into a program. One can use multiple threads in one process, multiple processes on one machine, multiple processes spread across multiple machines, or even a combination of the above. An ideal profiler should be able to handle all of these situations gracefully. It should not assume that a specific thread library is used, or be tied to the implementation details of a particular programming language.

Second, its overhead and intrusion should be minimal. In sequential programs, the overhead is the extra time spent to handle all of the profiling tasks. In parallel programs, there is also the intrusion to consider. Since threads interact with each other, time spent handling profiling tasks by one thread can affect the other threads with which it interacts. For instance, if profiling code is executed while a lock is held, then contention for that lock can increase. This type of overhead, which we refer to as intrusion, has the potential to drastically alter the performance characteristics of the application and should be minimized as far as possible. Aside from execution time, there is also overhead in terms of all the other program resources (memory, network requests, file handles, etc).

Third, it should be easy to use. Profilers can be difficult to use in many ways. The interface could be too complicated, making it difficult for the programmer to figure out how to use the profiler to measure what they want. The underlying model of program execution used by the profiler could be too arcane, causing programmers to be unsure when they can use the profiler, or what results they should expect when they do. A lack of flexibility, or an excess of overhead could also cause a profiler to be difficult to use if the profiler only works with one language or with programs that don't consume too much memory to begin with.

Finally, it should be extensible. A profiler intended for general purpose use should not be restricted in the types of quantities it measures. However, it is unrealistic to expect any single tool to be able to fulfill all conceivable roles. An ideal profiler would be amenable to extension and modification with low development cost.

In this thesis, we address these design goals with the development of context sensitive parallel execution profiles (CSPs). CSPs exceed the flexibility of existing profilers by taking a declarative approach: the programmer specifies at a high level what they are trying to measure, and our profiler does the measuring. Owing to their careful design, the overhead of CSPs is minimal; it typically falls within the normal variance of program execution time. Their declarative nature makes them particularly easy to use relative to their high flexibility. They are also readily extensible, which we demonstrate with an implementation for distributed systems.

1.1 Dissertation Overview

An overview of our system is given in Figure 1.1. The process begins with annotation, in which the source files comprising the program under test are annotated to indicate

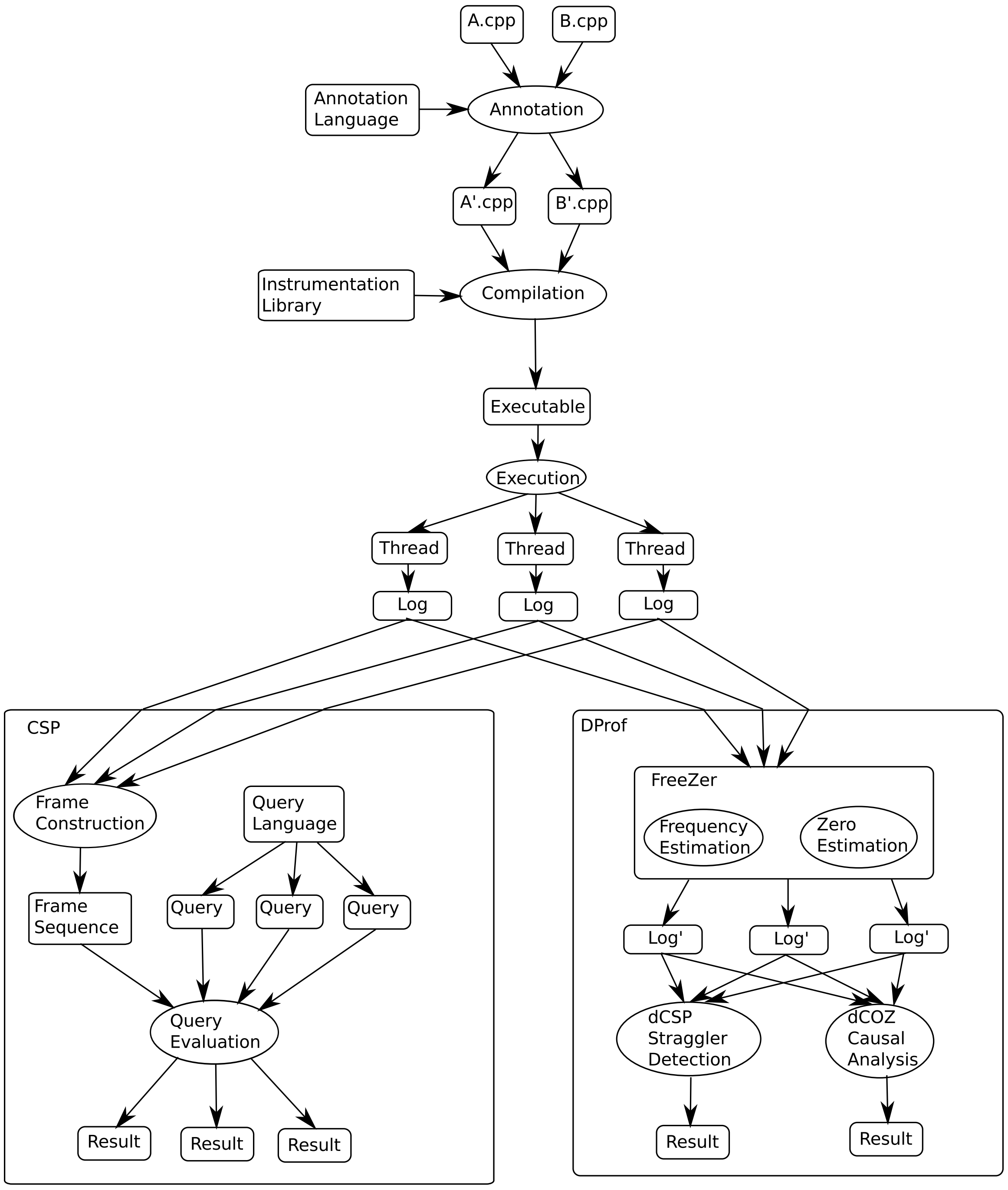


Figure 1.1: Overview of the declarative profiling architecture.

the regions of interest to the programmer. These annotated source files are then preprocessed and compiled, at which point they are linked with the CSP instrumentation library. When the resulting executable is run, it generates one log file for each thread in the program that contains timestamped relevant events from the execution. These log files are then passed to the frame constructor, which generates the primary profile representation. This sequence of frames is given as input to a query evaluator, along with the queries written by the user, and each of these is processed to produce a result. For distributed programs, the procedure on the lower right labelled DProf is taken instead. For these programs, the log files are processed once again in order to synchronize them before being passed to a higher level analysis (such as dCSP straggler detection, or dCOZ causal analysis). In the following sections, we will describe these processes in greater detail.

1.1.1 Context Sensitive Profiles on Shared Memory Machines

Two of the most prevalent forms of parallelism today are shared memory multicore machines and distributed systems. Typically, in order to meet the efficiency or ease of use requirements, profilers will focus on only one type of parallel program. This leads to a proliferation of specialized profilers that work only with one paradigm, language, or platform. One of the design goals of CSPs is flexibility; in particular they should work mostly unchanged with programs written for both shared memory and distributed systems.

The declarative nature of CSPs is key to accomplishing this goal. This is embodied in the interface to the programmer, which consists of the annotation language for specifying regions of interest, and the query language for describing quantities to measure in terms of those regions.

Our *annotation language* provides a rich set of simple constructs using which a programmer can identify regions of interest in their program. With these regions identified, instrumentation is inserted into the program before compilation in order to mark the entries and exits to and from these regions by each thread. During execution one log file of such events is generated locally to each thread, and stored to disk upon program termination. These log files are collected, and from them the CSP is constructed.

The *CSP format* consists of a sequence of what we call frames, which are regions of time during which no thread in the program transitioned between regions of interest as defined by the programmer through their annotations. This sequence of frames serves as a complete and exact record of the concurrent activities of the program's constituent threads. With this frame sequence in hand, the user can write and evaluate high level queries to measure quantities of interest.

The user constructs queries using our specialized *query language*. The essence of queries is to specify a set of frames in which the user is interested, and something to calculate given those frames. For instance, a query may specify all frames in which there is a thread holding a specific lock, and from those frames, determine which thread is in possession.

The combined interface of our annotation and query language, along with the careful implementation of our instrumentation library means that CSPs have very little runtime overhead: in most cases less than 5% overall. Additionally, the per thread memory overhead is modest as well (usually less than 1MB). (Mention something about accuracy here, or utility? The evaluations we did on the parsec benchmarks?) Thus CSPs are flexible, easy to use, and efficient.

1.1.2 Context Sensitive Profiles on Distributed Systems

When the limitations of a single machine stall progress, programmers often turn to distributed systems for support. Because of their importance and to showcase the flexibility and extensibility of CSPs, we implemented them in the distributed setting as well. This presented a unique set of challenges.

When two events occur on different machines, the timestamps with which they were captured will be generated by different clocks. These clocks were started at different times, and run at different rates. Therefore, direct comparison of these timestamps is meaningless. The problem of making meaningful comparisons between such timestamps is known as timestamps synchronization, and it is central to the construction of any profiler which supports distributed systems.

Though there are pre-existing algorithms for solving this problem [16, 33, 6, 35], all fall short in terms of efficiency and accuracy when faced with the types of targeted analysis that CSPs use. We therefore developed a new timestamp synchronization algorithm called FreeZer, which does not suffer these shortcomings. Previous algorithms work by using the timestamps to be converted, along with known causal relationships among them, to estimate a function which will convert from one clock base to another. In contrast, FreeZer overcomes the limitations of these algorithms by taking careful measurements offline which are independent of the timestamps that are being converted. By doing so, FreeZer is not only able to achieve higher overall synchronization accuracy, it is able to do so while maintaining strict bounds on the errors due to the synchronization process. This is a distinction that only one previous algorithm can claim, and FreeZer achieves the same thing up to 57× faster.

As seen in Figure 1.1, we used FreeZer as a critical component in our DProf work. Acting as a transparent conversion layer, we implemented a distributed version of our CSPs which required only minor changes to the query language. Using this we implemented a tool for the detection of straggler threads in distributed programs, a common problem faced in this domain. Additionally, using FreeZer and our annotation language, we developed a distributed version of the popular causal profiling [11] technique originally developed for shared memory systems only. Our experiments showed that not only were these tools able to correctly diagnose performance problems and accurately predict the results of optimizations, but they would have been unable to do so if a traditional synchronization algorithm had been used in lieu of FreeZer (exhibiting excess prediction errors ranging from 9% to 52%).

1.2 Dissertation Organization

The remainder of this dissertation is organized as follows. In chapter 2, we present the details of our annotation language and the precise description of our profile format. In chapter 3, we introduce our query language and illustrate how it can be easily used to diagnose and measure the severity of performance problems. In chapter 4, we dive into the implementation details of CSPs on shared memory multicore systems, and explore their accuracy and overhead. In chapter 5, we discuss the details of the timestamp conversion problem, and present the FreeZer algorithm. In chapter 6, we examine DProf, including dCSP and dDOZ and how they can be used for performance debugging of distributed parallel programs. In chapter 7, we conclude by giving a summary of our work and providing some directions for the future.

Chapter 2

Code Annotations and Context Sensitive Profiles

Every profiler must choose what code regions it will support (for instance, functions, statements, basic blocks, or loops). Existing profilers make this choice at design time. This inflexibility can cause frustration for the user of that profiler. If the choice of region is too large, then pinpointing the cause of problems can be challenging. If the choice of region is too small, then problems may go undetected due to the loss of context. The ideal region size depends on the specific analysis situation. To overcome this inflexibility, CSPs allow the programmer to choose the specific code regions relevant to their analysis. In this chapter, we introduce the mechanism by which this is enabled: our annotation language. We also show how the choice of code regions made using the annotation language parameterize our profile format.

2.1 Code Annotations

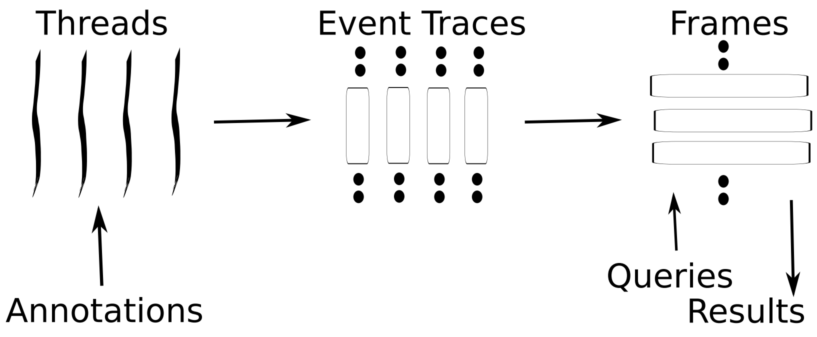


Figure 2.1: Overview

The steps of our approach are shown in Figure 2.1. Based upon a hypothesis for the cause of poor performance, the user introduces annotations into source code identifying code regions of interest. The annotations lead to instrumentation of the program that when executed produces timestamped traces for individual threads – thread local collection of event traces via lightweight instrumentation leads to *minimal perturbation* of program behavior and *low overhead*. The event traces are analyzed offline to generate a sequence of frames which describe what activities were performed by threads in parallel. By deferring frame construction to offline analysis, perturbation of program behavior is minimized. Finally, the user constructs queries that reveal how often and how long threads run in behavior states of interest, which reveals the absence or presence of hypothesized performance problem.

In this section we present the annotation framework available to the user. A set of easy to use annotations are supported that allow the user to mark code regions. The annotations provide the user with a great deal of flexibility via two features: alternate ways

Purpose	Annotation
ENTRY	`#Region` [NAME] [CONDITION]
EXIT	`#∼Region` [NAME] [CONDITION]
NAME	(RID CNST : [CVAR])
CONDITION	if (EXP)
RELATED	`#SubRegion` [NAME] [CONDITION] · · ·
NESTED	· · · `#∼SubRegion` [NAME] [CONDITION]
CONTEXT	`#Context` STMT ;

Figure 2.2: Summary of supported annotations.

of *naming* the region; and allowing *conditional* collection of region information.

Table 2.2 provides a list of supported annotations which consist of the following components:

– Marking region entry and exit. The annotations `#Region` and `#∼Region` mark the entry and exit of the region respectively. The corresponding `SubRegion` annotations are used to express related nested regions as will be described later.

– Naming regions. As the user may mark multiple regions of interest, names are assigned to them to distinguish their executions. The user provides a *static name* in form a constant (CNST). In addition, the user may also provide a *dynamic name* component in form an expression (EXP). This dynamic name is useful when the user wishes to distinguish executions of a given region into a finite number of categories according to their *execution context* (e.g., functions called, locks held, paths followed etc.). The context itself is captured in a variable by the `#Context` annotation (for instance, with `STMT` being CVAR = EXP;).

– Conditional regions. A user may be interested in only some of the executions of a marked region which can be selected at runtime based upon an associated condition (see CONDITION) defined in terms of the program's runtime state. The `#SubRegion` and `#∼SubRegion` annotations can be used to couple nested conditional regions, such that instances of the inner region are not captured unless instances of the outer region are captured as well.

Next we illustrate the use of above features through examples. A region can be a single-entry-single-exit or a single-entry-multiple-exit code region.

Static-name-only *regions.*

Let us consider the use of static names. It is often useful to analyze the relative execution times of a pair of regions. For example, Figure 2.3 shows two code fragments where region 1 has been introduced to capture the waiting time for a signal at a conditional wait (left) and time spent in acquiring a lock before entering the critical section (right). In Figure 2.4, the time spent waiting on a lock (region 1) is captured relative to the the time spent in the surrounding function (region 0). By introducing the surrounding region 0 in both cases, we can determine the time spent on waiting at the conditional relative to the execution time of the loop and time spent on acquiring the lock relative to the function's execution time.

Consider another example of *barrier synchronization* where it is useful to identify the presence of a *straggler thread* causing excessive waiting. Let us see how via appropriate region selection we can detect and find the cause of this behavior. By using the annotations shown on in Figure 2.5, we can determine the wait time for each thread at the barrier (region 1) as well as the total time spent in the loop (region 0). If it is found that all threads

```
void f() {
        //#Region(RID 0:)
        while(...) {
                ...
                //#Region(RID 1:)
                cond.wait();
                //#~Region
                ...
        }
        //#~Region
}
```

Figure 2.3: Annotations for measuring wait time relative to time spent in surrounding loop.

```
void f() {
        //#Region(RID 0:)
        ...
        //#Region(RID 1:)
        mutex.lock();
        //#~Region
        ...
        shared++;
        ...
        mutex.unlock();
        ...
        //#~Region
 }
```

Figure 2.4: Annotations for measuring the time spent waiting on a lock relative to the time spent in the entire closing function.

```
#Region(RID 0:)
for (...) {
        //Loop body
        ...
        #Region(RID 1:)
        barrier_wait();
        #~Region
}
#~Region
```

Figure 2.5: Annotations for identifying the time spent straggling relative to an entire loop execution.

```
for (...) {
        #Region(RID 0:)
        //Loop body
        ...
        #~Region
        #Region(RID 1:)
        barrier_wait();
        #~Region
}
```

Figure 2.6: Annotations for identifying the time spent straggling on a per-iteration basis.

except one thread wait for a significant duration at the barrier, then that one thread is the straggler. By comparing the execution time of the loop (region 0) with the time spent at the barrier (region 1) we can see if barrier causes significant performance degradation. Having detected the presence of a straggler, we can see if the same thread acts as a straggler or whether the straggler's identity varies. In the latter case, this behavior may be the result of variability in the amount of work performed by the loop body. This can be verified by using the modified annotation shown in Figure 2.6 where region 0 captures the time spent on the work performed during each loop iteration. If during an iteration, the thread identified as the straggler is also the one that spends the most time in region 0, then we know the cause is the code in region 0.

In the above examples because we only considered single-entry single-exit regions, we were able to assign static names upon entry. However, for single-entry multiple-exit code regions where we want to treat each exit as forming a different region, we must name the region on exit since the region id is known at the exit point. For example, consider a function with multiple return points. We can use region ids `0`, `1`, `2` etc. to distinguish different return points.

Context-sensitive dynamic-name *regions.*

The above examples illustrate regions with only static names. The user may want to collect additional execution `context` information to better understand the causes of observed timing behavior. In such situations, in addition to using a static name, a *dynamic name* is also assigned. Next we illustrate use of dynamic names in two scenarios. Let us consider the example of some code that acquires of a lock. The user may be interested in capturing the time spent in `acquire()` of various locks by each thread. As shown in Figure 2.7, this can be achieved by assigning a static name `0` to mark the code region and assigning a dynamic name using the *lock address* as the execution context. Thus, the time spent by a thread in acquiring locks can be divided among the different locks it acquires. Here the context (i.e., `&thislock`) was already available at the start of region 0 and thus it was directly referenced while creating the dynamic name. In general to ensure that the execution context is available at region entry or exit point, it may be necessary to first collect it explicitly at an appropriate execution point. In such a case we use `#Context` annotations for collection.

Consider the loop shown in Figure 2.8 which is an expanded version of the loop

```
class Lock {
        void acquire() {
                #Region(RID 0:this)
                ...
                #~Region
        }
}
```

Figure 2.7: Annotations for measuring lock acquisition times, distinguished by the object which is performing the acquisition.

in Figures 2.5 and 2.6. Further assume that we want to capture the function called (f() or g()) during the execution of each loop iteration because the user suspects that one of these functions is responsible for creating the straggler effect. As shown in the Figure 2.8, this can be achieved by specifying a static name in the annotation that marks the entry of the region as before and, in addition, using the context variable `fname` as the dynamic name in the annotation that marks the exit of the region.

The context is captured at the call sites of the functions via the two `#Context` annotations. As a result each execution of the loop body is assigned the name `0:1` or `0:2` depending upon where function f() or g() are called. Let us assume that we observe that a given thread acts as straggler when it calls f() but not g(), then we would know that we must optimize the code in f() to eliminate the straggler effect. Thus, dynamic names help the user to narrow and relate the cause of observed behavior to smaller code segments within marked regions.

```
for (...) {
        #Region(RID 0:)
        // Loop body
        ...
        if () {
                #Context fname = 1
                f() } else {
                #Context fname = 2
                g() }
                #~Region(RID :fname)
                #Region(RID 1:)
                barrier_wait();
                #~Region
        }
}
```

Figure 2.8: Dynamic names as execution context.

Conditional *regions.*

So far we have considered situations where all executions of an annotated region are captured and possibly categorized according to different contexts via different static names and/or via associated dynamic names. To handle situations in which we may not be interested in capturing all executions of an annotated region we support *conditional regions.* By specifying a condition as part of the annotation we can selectively capture executions of a region. This is yet another way of capturing context sensitive information. However, it is different from capturing context using names. This is because, conditional regions collect only relevant information corresponding to interesting contexts. Figure 2.9 illustrates the use of annotations for specifying conditional regions. The region corresponding to the outer loop indicates that the execution of this region is only captured if the region is being executed by the thread with id `1`. The second region in the inner loop uses the condition to sample the execution of its loop iterations – every fifth loop iteration is sampled. Further note that the inner region uses the `SubRegion` annotation. This couples its sampling to the outer region, i.e. it is only sampled when the outer region is being captured. Also note that here the context annotations are being used to create the variable `sample` needed to implement sampling.

Other Issues.

Dealing with unannotated exits. Note that in the examples so far, all exits from regions were marked by the user. This ensures the integrity of event traces generated,

```
while (...) {
        #Region(RID 0:) for Thread(TID = 1)
        ...
        #Context sample = 0
        while (...) {
                #Context sample++
                #SubRegion(RID 1:) if (sample % 5 == 0)
                if () {
                        #Context fname = 1
                        f()
                }
                else {
                        #Context fname = 2
                        g()
                }
                #~SubRegion(RID :fname)
        }
        #~Region
}
```

Figure 2.9: Conditional profiling of regions.

```
while (...) {
        #Region(RID 0:)
        ...
        if (...) break;
        ...
        #~Region
}
```

Figure 2.10: Exiting via unannotated break.

i.e. if a region entry event is captured, so is the corresponding region exit event. However, it is possible that the user may forget to mark an exit in which case the event trace would be incomplete. For example, Figure 2.10 shows a loop whose entire loop body is contained in region 0; however, during execution the region may be exited via the `break` statement that is not annotated. We deal with this problem by checking the integrity of generated traces.

Automating Instrumentation. For many types of analyses, automating the instrumentation phase is feasible using a tool like Clang to get access to the AST of the program. For instance, calls to barriers could be found automatically, and the barrier waits along with their associated loop bodies could have calls to the instrumentation library inserted directly around them.

2.2 Context Sensitive Profiles (CSP)

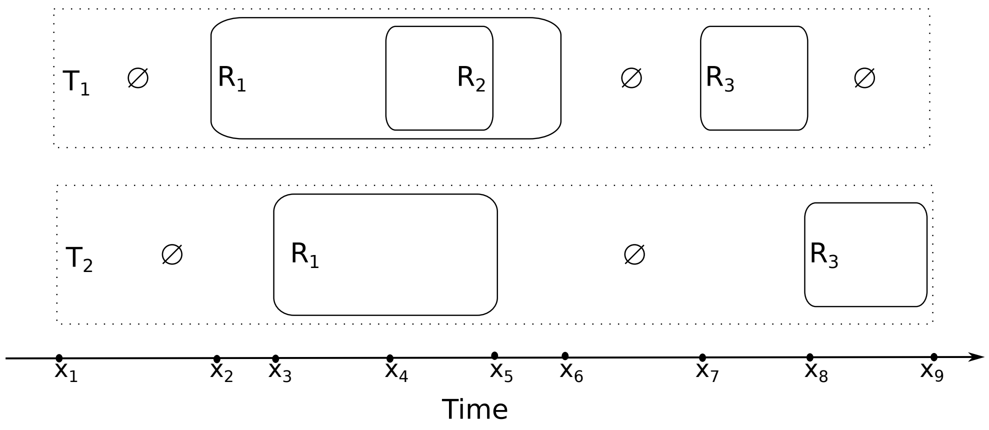
First we describe the thread local event trace that is generated when an annotated program is executed. Since event traces are thread local, they do not introduce any form of inter-thread synchronization, and thus minimally perturb program behavior. Moreover, the overhead of trace collection is low because it uses lightweight instrumentation. Second we present a novel CSP representation consisting of a series of frames that is derived offline. The local event trace of a thread tells us when the thread is executing a region of interest and when it is not. The frame sequence divides the application execution time into intervals where each frame captures the parallel behavior in terms of regions being executed by the threads in the interval.

The **Thread Local Event Trace** represents the execution history of a single thread as a series of events and the times at which they took place. The event trace of thread t that begins execution at time s_t, ends execution at time e_t, and along the way encounters region entry and exit events $e_1 \cdots e_n$ at times $x_1 \cdots x_n$ is denoted as follows:

$$[t@s_t \ \triangleright\ e_1@x_1 \ \triangleright\ e_2@x_2 \ \triangleright\ e_3@x_3 \ \triangleright\ \cdots \ \triangleright\ e_n@x_n \triangleright\,]@e_t$$

The types of events captured by the event trace are:

- Thread creation and termination. A thread trace begins and ends with the events `[tid` and `]` marking the creation of thread identified by `tid` and its termination respectively.
- Region entry and exit. Intervening events are either region entry or region exit that are of the form:



$$[t1@x_1 \ \triangleright \ (r1@x_2 \ \triangleright \ (@x_4 \ \triangleright \ r2)@x_5 \triangleright)@x_6 \ \triangleright \ (r3@x_7 \triangleright)@x_8 \triangleright]@x_9$$

$$[t2@x_1 \ \triangleright \ (r1@x_3 \triangleright)@x_5 \ \triangleright \ (r3@x_8 \triangleright)@x_9 \triangleright]@x_9$$

Figure 2.11: Per Thread Event Traces.

- Named only on entry → `(rid .... );`
- Named only on exit → `( .... rid)`; or
- Named on entry and exit → `(rid .... rid)`.

As an example, consider the event traces of threads T_1 and T_2 in Figure 2.11 where, during the execution shown, T_2 executes nested regions R_1 and R_2 and later R_3 while thread T_2 executes regions R_1 and R_3. The execution of a thread that does not execute a region of interest is named ϕ.

Since regions executed are either nested or disjoint, the integrity of the trace can be captured using the following grammar where $ThTRACE$ and $ReTRACE$ are the thread

$$
\begin{aligned}
\mathit{ThTRACE} \;&\rightarrow [\mathit{tid} \triangleright \mathit{ReTRACE} \triangleright\,] \\
&\mid\;\; [\mathit{tid} \triangleright\,] \\
\mathit{ReTRACE} \;&\rightarrow (\mathit{rid} \triangleright \mathit{ReTRACE} \triangleright\,)\; [\,\triangleright \mathit{ReTRACE}\,] \\
&\mid (\,\triangleright \mathit{ReTRACE} \triangleright \mathit{rid})\; [\,\triangleright \mathit{ReTRACE}\,] \\
&\mid (\mathit{rid} \triangleright\,) \mid (\,\triangleright \mathit{rid}) \mid (\mathit{rid} \triangleright \mathit{rid})
\end{aligned}
$$

Figure 2.12: Trace integrity grammar.

Annotation → Instrumentation
`#Region(RID x:y)` → `begin_code_region(x,y);`
`#~Region(RID x:y)` → `end_code_region(x,y);`
`#Region(RID x:y)if p` → `if p begin_code_region(x,y);`
`#~Region(RID x:y)if p` → `if p end_code_region(x,y);`
`#Region(RID x:y) if p ... #SubRegion(RID u:v) if q` → `if p begin_code_region(x,y); ... ... if p && q begin_code_region(u,v);`
`#Context stmt;` → `stmt;`

Table 2.1: Transforming annotations to instrumentation via calls to the profiling library.

and region trace respectively. For clarity we have omitted the timestamps from the trace. The grammar is used to test the integrity of the generated event trace and ensure that the user has not overlooked annotating any region exits.

Table 2.1 describes in detail how the annotations are transformed into library function calls. Notice that for sub-regions, the given condition (if any) is combined with that of its enclosing region. In the case where the predicate associated with the enclosing region may change before the sub-region is entered, the programmer can capture the initial value in a new variable using a context annotation, and use this new variable instead. This will guarantee that the predicate has the same value when the sub-region is entered. These transformations are implemented as a textual replacement phase in a stand-alone tool. Generated events are stored in a thread local std::vector initialized with enough space for one million events. The vector is written to disk upon thread termination. Following program execution the traces are analyzed offline to construct the CSP representation described next.

A **CSP** is represented in the form of a sequence of frames where each frame corresponds to the longest time interval over which the region being executed by each thread remains constant. Note that the region could be a region of interest or ϕ. A frame is represented as follows where the time interval that it represents begins at s (inclusive) and ends at e (exclusive) and each $S(tid_i)$ represents the state of thread tid_i in terms of the region(s) that it is executing.

$$[s,\ e)\ \rightarrow\ \{S(tid_1), S(tid_2), \ldots\ S(tid_n)\}$$

If thread tid is in an unnested region, then $S(tid)$ is given by:

$$S(tid) = \begin{cases} \phi & \text{if thread } tid \text{ is in an unmarked region} \\ rid & \text{if thread } tid \text{ is in a marked region } rid \end{cases}$$

On the other hand if tid is nested in n regions, then $S(tid)$ has following form:

$$S(tid) = rid_1\ \rhd\ rid_2\ \cdots\ \rhd\ rid_n$$

where rid_1 is outermost region and rid_n is the innermost.

~~$[x_1, x_2) \rightarrow \{S(T_1) = \phi, \quad S(T_2) = \phi\}$~~

$$[x_2, x_3) \rightarrow \{S(T_1) = R_1, \quad S(T_2) = \phi\}$$

$$[x_3, x_4) \rightarrow \{S(T_1) = R_1, \quad S(T_2) = R_1\}$$

$$[x_4, x_5) \rightarrow \{S(T_1) = R_1 \rhd R_2, S(T_2) = R_1\}$$

$$[x_5, x_6) \rightarrow \{S(T_1) = R_1, \quad S(T_2) = \phi\}$$

~~$[x_6, x_7) \rightarrow \{S(T_1) = \phi, \quad S(T_2) = \phi\}$~~

$$[x_7, x_8) \rightarrow \{S(T_1) = R_3, \quad S(T_2) = \phi\}$$

$$[x_8, x_9) \rightarrow \{S(T_1) = \phi, \quad S(T_2) = R_3\}$$

Figure 2.13: CSP - Frame Sequence.

Figure 2.13 shows the sequence of frames corresponding the event traces of Figure 2.11. As we can see, each frame indicates the regions being executed by the two threads. When an event causes the region of some thread to change, a new frame begins. We have crossed out two frames as in these frames none of the threads is executing an annotated region. Moreover, these frames can be inferred from other frames.

To construct the frame sequence, events are processed in the order of their occurrence one at a time – the ordering of events is made possible by the timestamps. With each event the current frame is updated to reflect the effect it has on the state of the relevant thread, producing the next frame in the sequence. By streaming events from log files and constructing frames one at a time, we construct and iterate over the frame sequence in constant space, enabling efficient trace analysis, even when traces are too large to fit in memory.

Behavior States. Since CSPs give the global picture of the execution, it is easy to characterize interesting behavior states in terms of the frames. For example, given the annotations in Figure 2.6, we would detect a straggler thread by searching for frames of the following form where region R_1 `(RID 0)` represents the loop body preceding the barrier and region R_2 `(RID 1)` represents the barrier itself:

$$[x_1, x_2) \quad \rightarrow \quad \{S(T_1) = \cdots = S(T_{n-1}) = R_2, S(T_n) = R_1\}$$

Note that threads $T_1 \cdots T_{n-1}$ are waiting at the barrier while thread T_n is executing the code preceding the barrier. The duration for which threads $T_1 \cdots T_{n-1}$ wait at the barrier for thread T_n is simply given by $x_2 - x_1$.

2.3 Data Centric Profiles

Many modern applications, such as iterative graph analytics and other forms of Big Data processing, are highly data-intensive in nature. In such applications, the nature of the input data can greatly impact performance. Therefore, it can be useful to correlate the execution time spent in code regions with the characteristics of data that are processed by them. To facilitate such analysis we support additional annotations shown in Figure 2.14. Collectively, these annotations allow a user to identify object properties, classify objects according to property values, and associate region executions with property values and

Purpose	Annotation
OBJECT DESCRIPTION	`#ObjectProperty` ID [PROPERTY] TYPE
OBJECT CLASSIFICATION	`#ObjectClass` ID TYPE (CONSTRAINT)
ASSOCIATE CURRENT REGION	`#Associate` TYPE OBJECT-INSTANCE

Figure 2.14: Annotations for data centric profiling.

classes of objects they process. The three kinds of annotations are as follows:

– ObjectProperty Annotation. Given an object type TYPE, this annotation defines an object property named ID along with an object method that returns the corresponding property value. Since an object type may posses multiple properties that may be of interest in different situations, multiple properties with different names can be identified by providing multiple annotations of this kind.

– ObjectClass Annotation. This annotation defines a class of objects named ID of the type TYPE that satisfy a constraint CONSTRAINT on its property value. If objects need to be classified into multiple classes, the programmer can provide multiple annotations.

– Associate Annotation. When a region is executed, we would like to associate its execution with a specific object class defined by the ObjectClass annotations. This association is achieved by introducing an Associate annotation in the region that specifies the object type TYPE and an object instance OBJECT-INSTANCE. The properties and classes of the specified object are captured at the point of association, and attached to the surrounding code region.

The use of the above annotations is illustrated in the context of the single source shortest path (SSSP) algorithm shown in Figure 2.15. For simplicity we have omitted the details of the convergence logic for this iterative algorithm. The vertices are divided into multiple batches and assigned to different threads for processing. The region with static id 0 captures the execution time of each thread. The region with static id 1 captures the execution time spent on processing a given vertex, and the region's dynamic id captures the vertex id (vid). Using the new annotations we identify interest in the vertex property

```
int batch = k/n
int begin = batch * tid
int end = tid == n -1 ? batch * (tid + 1) : k

#ObjectProperty  DEGREE [in_degree()] vertex
#ObjectClass  HIGHDEGREE vertex  (DEGREE >= 100)
#ObjectClass  LOWDEGREE  vertex  (DEGREE < 100)
while(not converged) {
        for(int j=begin; j<end; ++j) {
                #Region (RID 1:)
                #Associate vertex V[j]
                update_path(V[j])
                #Context vid = V[j]
                #~Region (RID 1: vid)
        }
        barrier();
}

void update_path(v: vertex) {
        path_value = v.value
        for(edge_iterator it = v.in_edges().begin();
                          it != v.in_edges().end();
                          ++it) {
                path_value = min(path_value, it->source().value + it->weight);
        }
        v.value = path_value
}
```

Figure 2.15: Example of data-centric profiling of SSSP algorithm.

DEGREE, which is the in-degree of vertex object. We further separate vertices into two classes, HIGHDEGREE and LOWDEGREE, according to the value of DEGREE property.

The CSP profile generated will not only have information about execution of these regions, but also details of object characteristics provided by data centric annotations. In particular, the profile of a region execution will include DEGREE information (i.e., value of `V[j].in_degree()`) and classification information (i.e., HIGHDEGREE or LOWDEGREE). Thus, we will be able to construct queries that will allow us to divide the total time spent on executing the specified region into two parts – time spent on high degree vertices and time spent on low degree vertices.

2.4 Related Work

Critical Path Analyses. Another technique for analyzing parallel programs is based on critical path, the longest path in the program dependence graph [42, 29, 23, 10, 21, 5, 20, 31]. In [23], the authors use hierarchical critical path analysis to build a tool for estimating the parallel speedup of each region in the program. [10] defines several new performance metrics based on critical path analysis with the intention of identifying performance problems, particularly load imbalance, in highly parallel systems. In [21] authors develop true zeroing which is a method for comparing performance metrics. By comparing the critical path metric with performance metrics obtained via Gprof and Quartz [5], the authors conclude that the critical path analysis is often the best guide for finding program bottlenecks. In [20, 31] algorithms for computing critical paths online, including the ability to report partial results, are presented. Critical path analysis identifies activities along the critical path where tuning efforts can be focused to improve performance. However, its drawback is that it provides an upper bound on the performance improvement, but gives no insight into how much actual improvement can be expected because optimizing code can switch the critical path. Thus, in [22], the authors use the *slack* metric to capture how much of an improvement can be expected. In [17], authors compare the notions of hierarchical critical path analysis and self-parallelism. They present a tool for measuring these quantities, and argue that self-parallelism is a good indicator of potential for real parallel speedup. Finally, Coz [11] provides virtual speedups which allows measurement of the benefit of optimizing a code region in terms of overall speedup. Thus the benefits can be estimated before effort into optimizing the code is expended. From the above discussion it is clear that even for critical path analysis many variations exist and hence our approach of providing a single versatile tool in which different metrics can be evaluated is beneficial to the end user. By viewing the program as a series of regions, we can also perform critical path analysis.

2.5 Summary

In this chapter, we presented our annotation language. We showed the different ways annotations can be used to declare arbitrary code regions, which define the possible locations of threads in the profile. We also introduced our data-centric annotations, which allow an even richer set of code regions which are dependent on the characteristics of the data being processed. Additionally, we presented our profile format, which is parameterized by the user defined code regions, and which allows the capture of context rich information that can be analyzed offline.

Chapter 3

Querying Context Sensitive Profiles

Many existing profilers are built to calculate only a single metric. In contrast, CSPs allow the programmer to perform many different analyses using the same profile. This flexibility is enabled via the CSP query language that we have developed, which we describe in this chapter. First we will describe the structure of the query language, and then we will show how queries can be used to diagnose and fix performance problems in different types of parallel programs.

3.1 Query Language

Once the CSP consisting of a sequence of frames has been generated, the user can construct a query to extract the subset of *pruned frames* representing interesting program behaviors. Each *pruned frame* contains the maximal part of the frame, called the *sub-frame*, that satisfies the query. Our query language is presented in Figure 3.1.

A ***FrQuery*** is constructed to express the forms of frames that satisfy properties

$$\underline{[\ Measures \mid Attributes\ ]\ FrQuery\ [\ Interval\ ]}$$

$$Measures \rightarrow Duration \mid MaxPar \mid Area$$

$$Attributes \rightarrow WhichThreads \mid WhatRegions$$

$$FrQuery \rightarrow \bar{[}\ \neg\ ]\ Quantifier\ ID\ Constraints\ :\ Predicate$$

$$Quantifier \rightarrow \forall \mid \exists \mid \exists_{Nat}$$

$$Var \rightarrow ID \mid Nat \qquad Nat \rightarrow 0|1|2|...$$

$$Constraints \rightarrow (\ =\ \mid\ \neq\ )\ Var\ \ [\ (\wedge \mid \vee)\ ID\ Constraints\ ]$$

$$Predicates \rightarrow \bar{[}\ \neg\ ]\ (Thread, Region)\ \ [\ (\wedge \mid \vee)\ Predicates\ ]$$

$$Thread \rightarrow Var \qquad Region \rightarrow Var$$

Figure 3.1: Query Language.

of interest in a time interval that may be specified. When no interval is specified, the entire execution is analyzed. A *FrQuery*, when evaluated, returns a subset of pruned frames from the profile that satisfy the query. The returned frames are pruned so that they contain the maximal sub-frame that makes the query true. For instance, consider a query that returns the set of frames in which there is at least one thread inside region 0. Each of those frames is pruned to contain *only* those threads which are actually in region 0.

Measures are a means of computing summary information for a set of frames. We support three forms of measures. The first measure, *Duration*, returns the sum of the durations of the subset of frames, i.e. it corresponds to the total elapsed time. The second measure, $MaxPar$, returns the maximum number of threads that were active among the given subset of frames. A thread is considered *active* if it is in some region of interest, i.e. it is not in the ϕ state. Therefore this measure corresponds to the degree of parallelism.

Query	Returns frames such that –
$(0,0)$	– thread *0* is in region *0*
$(0,0) \wedge (1,1)$	– thread *0* & *1* are in region *0* & *1*
$\forall t : (t,0)$	– *all* threads are in region 0
$\exists t : (t,0)$	– *some* thread is in region 0
$\exists_1 t : (t,0)$	– there is *exactly one* thread in region 0
$\forall r : \exists t : (t,r)$	– there is *some* thread in *every* region
$\forall t : \exists_1 r : (t,r)$	– *each* thread is in *outermost* region

Figure 3.2: Example queries and their meaning.

The last measure is $Area$, which represents the total work done by a subset of frames. $Area$ is computed by summing the areas of each of the individual frames, where the area of an individual frame is its $Duration$ times its $MaxPar$.

Attributes either return the set of threads ($WhichThreads$) or the regions involved ($WhatRegions$) in a subset of frames.

The basic construct in a query is a ***Predicate*** that asserts that some thread is inside some region. Using logical operators ($\wedge \mid \vee$) we can construct arbitrarily large predicates over multiple threads and regions. Examples of such queries are shown in Figure 3.2 – see the first two queries.

We permit quantification over both $Thread$s and $Region$s. We provide three quantifiers: $\forall$, $\exists$, and $\exists_{Nat}$, which we use as an exact existential quantifier. The quantifier $\exists_k$ states that there exist exactly k of some object (either $Thread$s or $Region$s) satisfying some property. The names bound by quantifiers can have equality constraints imposed upon them.

The quantifiers range over the active threads and regions of each frame. The third through fifth queries in Figure 3.2 are examples of using the three quantifiers. Finally, the last two complex queries in the figure employ two quantifiers, one over the threads and the other over the regions. Note that constraints on quantified variables must involve the variable captured by the quantifier. In the remainder of this chapter all the discussion assumes is based on static region names. However, in general, dynamic names can be used to further subdivide multiple executions of a static region into distinct groups corresponding to their dynamic names.

3.2 Using Queries for Iterative Debugging

Next we illustrate the usage of queries to identify opportunities for program restructuring to improve performance. We improve performance of two Parsec suite programs `blackscholes` and `cannel` by 36% and 17% respectively. In these case studies we use the largest available native inputs.

blackscholes assigns prices to each of a set of input options. The pricing of individual options is independent, so the benchmark is parallelized by dividing the set of options evenly among the available threads, and simply having each of those threads price its own subset of options. There is no synchronization among the threads that calculate the prices.

We begin our analysis by determining the total execution time of each thread in the program. To do this, we mark as a code region the body of every thread function. There are two such functions in blackscholes, `main` and `bs_thread`. To determine the time that each thread spends, we evaluate following query for each thread t: $Duration((t, \texttt{main}) \mid\mid (t, \texttt{bs\_thread}))$. We know that this will not over count, since the main thread does not itself execute the `bs_thread` function.

When we examine the results, we see that one thread has a run time nearly twice as long as the rest of the threads in the program. We can check which thread this is (main thread or one of the workers) by evaluating the following query.

$$\mathit{WhatRegions}((t, \texttt{main}) \mid\mid (t, \texttt{bs\_thread}))$$

The result of this is `main`, and so we examine the main thread further and observe that it performs three tasks: (1) Reading the options to be priced from an input file into an array; (2) Launching the threads to compute prices, and wait for them to finish; and (3) Writing the computed prices to a file. During tasks 1 and 3, only one thread is doing work, and that work consists primarily of disk accesses. We conclude that the program should be restructured to hide this latency. Before restructuring we would like to know by how much will the execution time be reduced. So we re-annotate the program, marking regions in which tasks 1 and 3 occur, and recompute the CSP. We estimate the upper bound on execution time reduction as:

$$\frac{\mathit{Duration}((t, \texttt{main})) - \{\mathit{Duration}((t, 1)) + \mathit{Duration}((t, 2))\}}{\mathit{Duration}((t, \texttt{main}))}$$

The result of this query is 48%, which tells us that hiding the latency is a profitable optimization. We modified the program by adding two more threads: one which produces values by reading them from disk, and one which consumes values by writing them to disk. In the middle sit the application threads, which transform options into prices. Application threads receive options from the producer, and give prices to the consumer. Lock-free queues are used as buffers between each of these stages. Implementing this change led to a 36% reduction in total execution time.

canneal attempts to find a minimal routing cost for a chip using multithreaded simulated annealing. As in the previous case study, we begin our analysis by marking each thread function as a code region to find the total time spent executing each thread. In addition, since this program has a call to `pthread_barrier_ wait`, we mark code region to determine how much time each thread spends waiting. Using the same queries as in previous case study, we find that the `main` thread again executes nearly twice as long as the other threads in the application. Furthermore, the initial time spent is again due to reading an input file and constructing the shared data structure for application threads. This time however, we cannot apply the same optimization that we did for blackscholes, because the data structure is being shared in its entirety by each of the other threads. Thus, we turn our attention to the remaining threads. We observe that these threads execute the same function, and the main thread simply waits for them while they do so. Thus, we can speed up the application by speeding up the function they run.

We examined the function named `annealer_thread::Run`. In this function, each thread repeatedly selects a pair of elements from the shared data structure and decides whether to swap them. If swapping those elements reduces the overall cost, then the swap is accepted, and if it increases the cost, then it is rejected with increasing probability as the computation proceeds. This probability is determined by the so-called “annealing temperature". The application threads periodically synchronize at a *barrier* before they change their temperature, so that the threads are always working with the same temperature. Removal of this barrier would have two effects. First, since the individual threads can have different temperatures, the final routing cost may be affected. Second, the total time taken by the program will likely be smaller, since waiting will be eliminated.

To judge whether eliminating this barrier would be beneficial, we need a rough estimate of the amount of time that we might save by applying the optimization. If the amount of time spent waiting is too small, then removing this barrier is unlikely to make a difference in the overall run time. We measure the total waiting time by evaluating the query: $Duration(\exists t : (t, Barrier))$ and we measure the waiting time of each thread with this query, but with t specialized to each thread ID. The total waiting time amounts to 26% of the overall execution time, and the waiting time per thread ranges from 11% to 15%. Thus, we surmise that removing this barrier might have a noticeable effect on the run time of the program. When we removed the barrier, the average runtime of the program was reduced by 17%, and the average routing cost computed changed by .003% which is a negligible change for a randomized algorithm.

3.3 Data-centric Queries

3.3.1 Analyzing via #ObjectProperty

Let us see how the DEGREE property in Figure 2.15 may be analyzed to facilitate performance debugging. Note that static partitioning assigns to each thread roughly equal number of vertices for processing. A region 1 execution captures the work done to process a single vertex assigned to the thread.

The first question we would like to answer is whether or not there is an imbalance among the individual thread execution times. For any thread t, the total time spent executing is given by the query $Duration(t, 0)$. On the other hand, the total amount of work done by all threads is given by the query $\exists t : Area(t, 0)$. If the total amount of work is well balanced among all threads, then we would expect the relative difference of these two queries to be small.

$$\frac{Duration(t, 0) - \frac{\exists t: Area(t,0)}{n}}{\frac{\exists t: Area(t,0)}{n}} \approx 0$$

If these differences are small then we conclude that workload distribution is balanced. When workload imbalance is observed, our goal is to reduce the deviation between $Duration(t, 0)$ and $\frac{Area(0)}{n}$ for all threads. Since each thread is assigned an equal number of vertices, a plausible explanation for high deviation is that each vertex represents different amount of work that is proportional to the number of its incoming edges when executing the update_path() function. Therefore we capture these times via region annotation with static id 1 along with the id of the processed vertex and its in-degree property value via the use of our object annotations.

To determine the extent to which high degree vertices contribute to the imbalanced execution time, we measure the amount of time that each thread spent processing vertices whose in-degrees lie within various ranges.

$$aggregate\_time(t, x, y] = Duration(t, 1, Filter(DEGREE \in (x, y]))$$

Here $(x, y]$ represents a degree range. In the above equation $Duration()$ takes an additional third parameter *Filter* which selects frames where degree object property lies between x and y. By normalizing this with respect to the total thread execution time for t as calculated above, we can see the effect of the high degree vertices on the execution time of each thread as follows.

$$\frac{aggregate\_time(t, x, y]}{Duration(t, 0)}$$

Additionally, we would like to determine if the time spent executing high degree vertices is balanced among the threads. In the balanced case, we expect each thread to spend approximately $avg(x, y] = Area(\exists t : t, 1, Filter(DEGREE, (x, y]))/n$ time processing vertices whose degree is in the target range. The actual time spent by thread t is given by the query $Duration(t, 1, Filter(DEGREE, (x, y]))$. As above, we can examine the relative difference to see the balance:

$$\frac{Duration(t, 1, Filter(DEGREE, (x, y])) - avg(x, y]}{avg(x, y]}$$

If this deviation is high, then we must aim to reduce it by evenly distributing the high degree vertices among the threads. Effectively, we should partition the workload based on the total in-degrees of the partitions. This can be done by maintaining a prefix sum over the degrees of vertices in a partition. So then the solution would be to pick a batch of vertices based upon a constant total for the in-degrees of that batch. Note that we may not necessarily minimize the differences in the number of high degree vertices processed by threads since load can be balanced by having one thread process only a few high degree vertices while another thread processes many low degree vertices.

Figure 3.3 shows measurements of the above queries on a collection of input graphs and algorithms. We used four popular graph algorithms taken from [39, 30]: PageRank (PR),

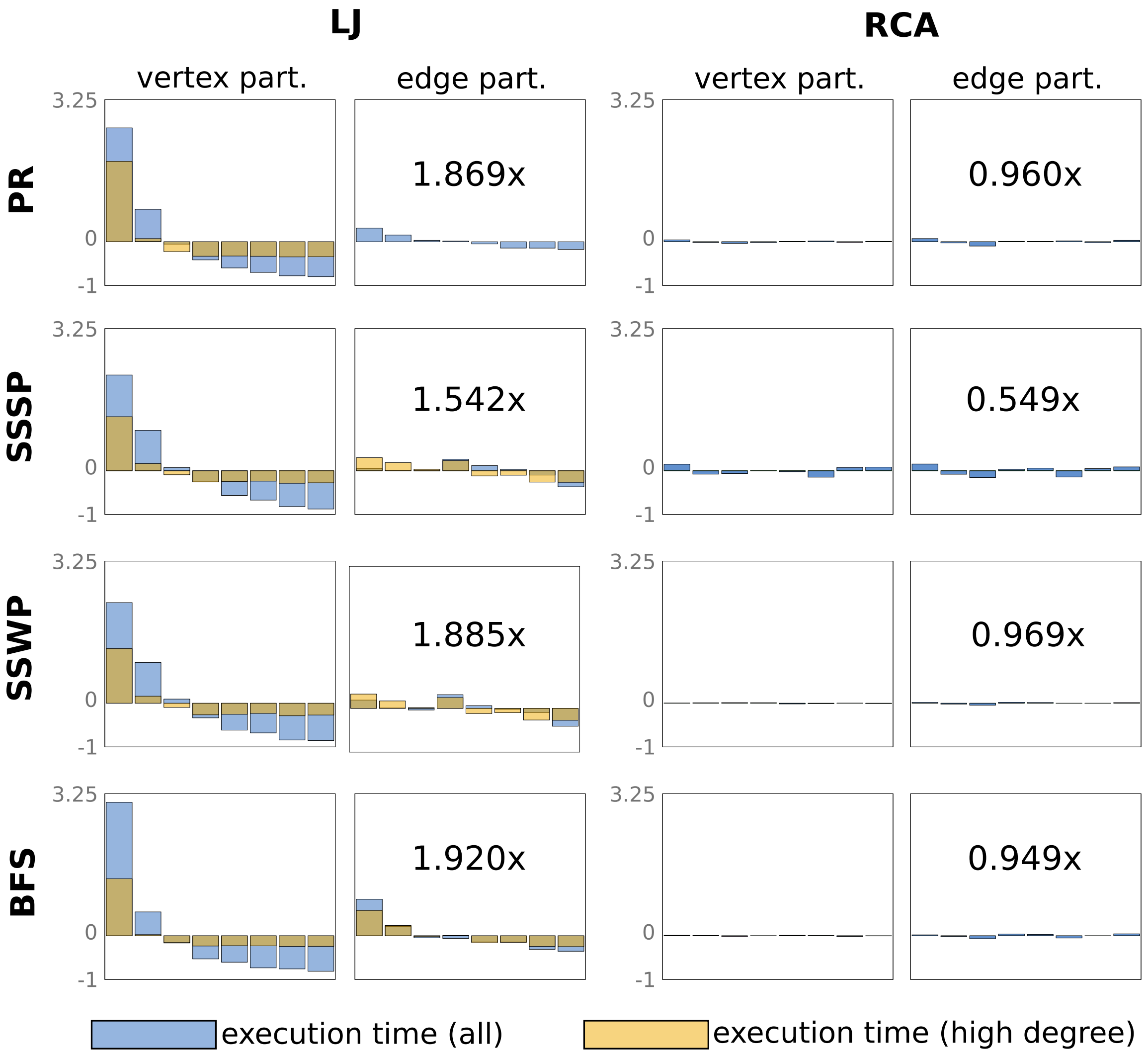


Figure 3.3: The effects of different partitioning strategies on the application work distribution: Blue bars represent the amount of work done by a thread (normalized to the average work). Yellow bars represent the amount of work done on high degree vertices (normalized to the average work on high degree vertices. Each row represents one algorithm, and each major column represents one graph. The minor columns correspond to the partitioning strategies (vertex partitioning on the left, edge partitioning on the right). The edge partitioning figures are annotated with the speedup relative to using vertex partitioning.

Single Source Shortest Path (SSSP), Single Source Widest Path (SSWP), and Breadth First Search (BFS); and two input graphs from SNAP repository [26]: soc-Livejournal1 (LJ) containing 68.9M edges and roadNet-CA (RCA) containing 2.7M edges. As shown in [40], LJ has a skewed degree distribution while RCA has a regular and sparse distribution. Experiments were run on a 2 socket, 8 core Intel Xeon E5607 (2.3GHz) and we used 100 as our threshold for high-degree vertices.

In Figure 3.3, bars that are in the positive range correspond to threads which spent more time executing than the average, while bars in the lower range correspond to threads which spent less than the average time executing. The blue bars show the normalized deviation from the average work; this is computed based on our first query and is expected to be close to 0 if work is well balanced. As we can see, for LJ the deviations are large for all graph algorithms (left most column of figure). This means, there is workload imbalance when we statically assign equal number of vertices among threads and hence, we further debug by analyzing the deviation for high degree vertices alone (using the second query). As shown by the yellow bars, the deviations are large which suggests that imbalance in high-degree vertices is high and is in-turn causing an imbalance across overall thread executions. We fix this issue by partitioning workload based on total in-degrees of vertices, as described earlier. As shown in Figure 3.3 (2nd column), the deviation gets substantially reduced which accelerates our overall processing by factors ranging from 1.54× to 1.92×. For RCA, we do not see much deviations with basic partitioning (values are close to 0); this is because RCA is regular and sparse with average degree between 3 to 6, and hence, the traditional vertex partitioning performs well.

```
udpate_path(v: vertex) {
  path_value = v.value
  for(edge_iterator it = v.in_edges().begin();
                    it != v.in_edges().end();
                    ++it) {
       if CHANGE[it->source()] {
           #Region (RID 2:)
           #Associate vertex V[j]
           path_value = min(path_value, it->source().value + it->weight);
           #~Region (RID 2:)
       }
  }
  if path_value != v.value {
       CHANGE[v.id] = true;
       v.value = path_value
  } else CHANGE[v.id] = false;
}
```

Figure 3.4: Data-centric profiling of SSSP algorithm for measuring redundancy.

3.3.2 Analyzing via #ObjectClass.

Next we illustrate how the object classes created in Figure 2.15, HIGHDEGREE and LOWDEGREE, can be helpful for debugging performance. If the collected information indicates that a significant fraction of time is spent on executing region 1 for high degree vertices we can proceed as follows to improve performance.

A developer may look for the presence of redundant operations in executions to identify opportunities for optimizing the code. The udpate_path() function uses a `min`

```
udpate_path(v: vertex) {
  for(edge_iterator it = v.out_edges().begin();
                    it != v.out_edges().end();
                    ++it)
      id = id->target().id;
      mutex[id].lock();
      it->target().value = min(it->target().value, v.value + it->weight);
      mutex[id].unlock();
}
```

Figure 3.5: Modifying SSSP algorithm to eliminate redundancy.

aggregation over incoming values of the vertex. For high degree vertices, this aggregation can be time consuming. Furthermore, dynamic values of vertices often change infrequently, and hence fetching and using those values (using `it->source().value`) often becomes a redundant operation. To verify this hypothesis, developers can further annotate udpate_path() as described in Figure 3.4. Effectively, the annotations capture region 2 only when the source vertex's value is changed. If the time spent in this region is small, this would indicate that a lot of redundant (i.e., wasteful) work is being done. In addition, #Associate qualifies the profiles with HIGHDEGREE and LOWDEGREE information. This enables the correlation of the redundancy information with the class of the vertices as follows.

$$Duration(t, 2, FilterClass(HIGHDEGREE))$$

$$Duration(t, 2, FilterClass(LOWDEGREE))$$

The *FilterClass* function selects frames whose associated objects belong to HIGHDEGREE or LOWDEGREE classes. As we can see, high in-degree vertices perform more redundant work, and eliminating it would lead to potential speedups. If this indicates that a significant fraction of time is spent on executing the region for high degree vertices, then we can be confident that the development effort of changing the algorithm will be worthwhile. We can modify update_path() as shown in Figure 3.5 to optimize the cost of processing high degree vertices. Effectively, vertices propagate values forward only when values change which completely eliminates redundant operations.

Figure 3.6 shows our results of debugging for redundant computations using the LJ input graph. Since SSSP, SSWP and BFS rely on selection functions like `min()` and `max()` which typically choose one of the values, eliminating redundancy that comes from reading all values (instead of just some of the values) can improve their performance. The first row of plots show the relative time spent on redundant computations based on the first DURATION query in this sub-section. As we can see, all three algorithms spend time in performing redundant computation by reading all unchanged values for `min()` and `max()` operation. Redundancy is higher for SSSP and SSWP and lower for BFS; this is because BFS performs fewer computations compared to SSSP and SSWP such that the entire graph is visited only once, and hence, most of the traversal is necessary. To eliminate redundant computations, we incorporated the optimization based on Figure 3.5 which improved the overall performance by 1.36× to 1.9×.

Note that the above redundancy removal optimization comes at a cost of using mutex locks to perform updates and hence, the next step is to understand the overheads induced by mutex locks to weigh the potential benefits that can be achieved by replacing them with lower-level atomic `min()`/`max()` operations. For this, we performed similar

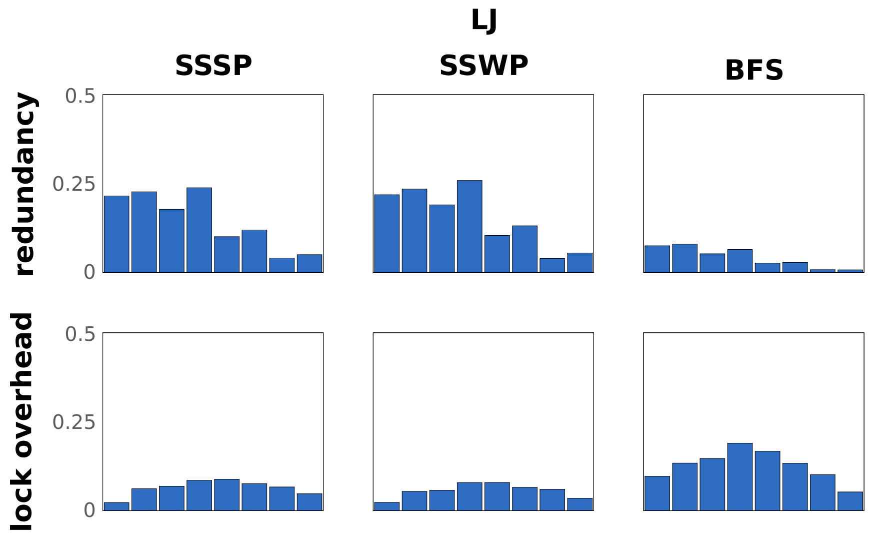


Figure 3.6: Redundancy measures and lock overhead times for the LiveJournal graph using push/pull based updates. In the first row: box heights show the amount of time spent performing redundant updates per thread, normalized to the thread execution time using the pull method. In the second row: box heights show the amount of time spent acquiring locks per thread, normalized to the thread execution time using the push method.

analysis using ObjectClass to capture the relative duration spent on lock acquisitions. The bottom row in Figure 3.6 shows the relative duration spent on lock acquisition. As we can see, the lock overheads for BFS are higher; this means, replacing locks with atomic operations (based on `compare-and-swap` instruction) can further accelerate BFS.

The trade off in using ObjectClass is that the correct threshold for high degree must be known before hand. If it is not known, directly annotating the degree information will help since during query analysis the user can try different ranges without rerunning the profiling. On the other hand, debugging is an iterative process. As in other case studies we analyze the program with a series of queries of decreasing granularity. The first queries determine whether a possible problem exists, and further refined queries provide details about possible causes and solutions.

3.4 Related Work

Our work is not the first to recognize the role that data plays in the diagnosis of performance problems.

In [12, 38] the authors present a Java based profiler for detecting lock contention based upon the free lunch metric for critical section pressure. It is defined as the ratio of the total time spent waiting for a lock, divided by the total time spent executing in the critical section associated with that lock. The data-centric techniques outlined in this section could be used to measure similar properties.

In [43], Yu et al. present a system for identifying performance problems in multi-threaded programs using execution traces. It examines complications caused by propagation of cost via functions calls and lock contention. One of their primary concerns is locating the causes of problems which may be located at any number of layers in a complex system. Our queries are well suited to handling this type of complexity since they make no decisions about the regions of interest ahead of time.

3.5 Summary

In this chapter, we presented our query language, which allows programmers to calculate a variety of interesting metrics over CSPs. We showed how our query language could be used to diagnose and fix performance problems in the Parsec benchmark suite, achieving 36% and 17% speedups on two of the benchmarks. We also showed how our data-centric queries could be used to accurately measure and optimize the load balancing in a parallel graph processing system. In the next chapter, we will examine the performance characteristics and accuracy of CSPs.

Chapter 4

Shared-Memory Platform: Implementation and Evaluation

In the previous chapters, we presented our annotation language, profile format, and query language, and showed how they can be used in concert to analyze the performance of parallel programs. In this chapter, we dive into the details of the implementation of CSPs on shared-memory platforms which make those analyses possible. We begin by discussing how to capture timestamps while meeting our goals of low overhead and intrusion. We then present experiments on real-world programs which assess the overhead of CSPs. Finally, we present experiments which test the accuracy of CSPs.

4.1 Capturing Timestamps

We generate timestamps using **RDTSCP** instruction available on modern x86 based architectures [1]. This instruction reads the value of the timestamp counter register,

Program	%Time
blackscholes	0.0000459
bodytrack	1.5
canneal	0.331
dedup	14.9
facesim	0.849
ferret	0.125
fluidanimate	0.0782
raytrace	0.0295
streamcluster	1.64
swaptions	0.00000353
vips	0.556
x264	0.676

Figure 4.1: Percentage of total execution time attributable to frames with duration less than 44,000 cycles.

which holds the number of cycles that have elapsed since the processor was last restarted. It ensures that all instructions that come before it have been executed before it reads the timestamp counter, and the values that are read are guaranteed to be monotonically increasing. **RDTSCP** instruction has two benefits. First, since it is a single instruction, it is much faster than a typical standard library time gathering function. Second, it measures time in terms of *cycles*. There is one issue however – since each core of the processor has a separate timestamp counter, and these timestamp counters are *not synchronized*, there can be inaccuracy in the measured frame durations. If this inaccuracy is too large, it can lead to observing a different event order, which will result in frames showing up that did not actually occur. We show that this inaccuracy is too small to have any meaningful impact on information collected and inferences made.

Using the approach proposed in [44], we measured the drift Δ between the timestamp counters of a pair of cores on the same socket and on different sockets of the machine used. Δ, computed as a range, was found to be $[0, 24]$ and $[0, 44]$ cycles respectively for intra-socket and inter-socket cases. Consider a frame that starts at time S on one core and finishes at time F on another core. Since the times F and S are determined by **RDTSCP** using different counters, the absolute error in the measured frame duration is $\Delta/(F - S)$. For this error to be less than 0.1% of the frame duration, and assuming worst case Δ of 44 cycles, the frame duration should be $>$ 44,000 cycles. That is, all frames with measured duration of $\leq$ 44,000 cycles can have $>$ 0.1% error. In Figure 4.1, for Parsec programs instrumented to detect waiting times at barriers and conditional variables, %Time is the percentage of execution time spent in frames smaller than 44,000 cycles. We see, excluding `dedup` where a large number of active threads in pipeline stages transition between regions, these frames represent less than 2% of the total execution time, i.e. frames representing over 98% of the execution have error $<$ 0.1% of their durations.

4.2 Overhead of Capturing CSPs

To measure the time and space overhead of CSPs, we performed a realistic analysis of twelve out of the thirteen benchmarks from the Parsec benchmark suite (all of the benchmarks which included pthreads parallelizations.) We created a histogram of all threads showing the their total time broken down into waiting time and running time. For code regions we chose waits on condition variables and barriers, as well as the entry function for each thread. For each thread t, we determined the total time by evaluating the query $Duration(t, entry_t)$, where $entry_t$ is the entry function for thread t. We determined the waiting time by evaluating the query $Duration(t, w)$, where w is the static region ID corresponding to the barrier and condition waits. This analysis is typical of a first step one

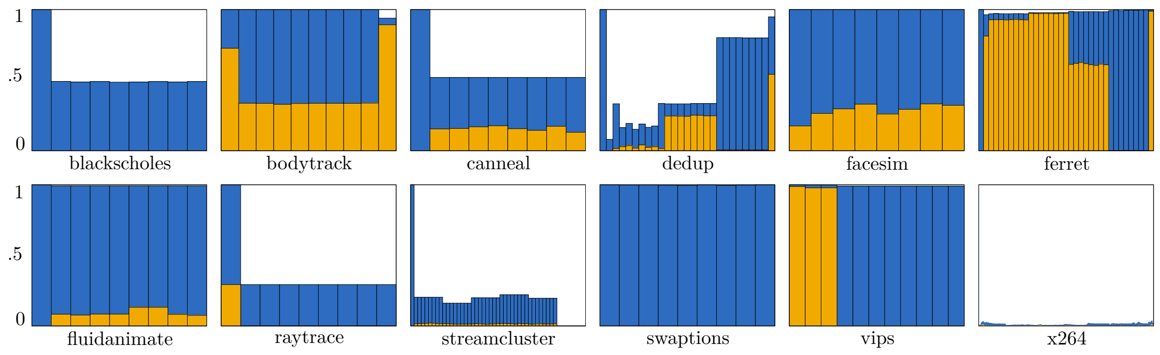


Figure 4.2: Execution time for each thread divided into wait time (gold) and total time (blue), normalized wrt the main thread.

might take in analyzing the runtime behavior of a parallel program, since it gives a rough idea of which threads are doing the most work and how efficiently that work is parallelized. Our experiments were conducted on a Dell Poweredge T410, having two 2.27GHz quad core Intel Xeon E5607 processors (no hyperthreading) for a total of 8 physical cores with 32GB of RAM. For all benchmarks, native inputs (the largest available) were used, and a thread count of 8 was used in the launch options.

In Figure 4.2 the blue regions correspond to total time, and gold regions correspond to waiting time. All measurements are normalized with respect to the total execution time of the main thread. About half of the benchmarks show very little gold, suggesting they are efficiently parallelized and require little or no waiting. The others have waiting times from 15% to 40%, suggesting that they are harder to efficiently parallelize. Of particular interest are streamcluster and x264, which employ dynamic parallelization, launching 49 and 1024 threads respectively. Neither exhibit large waiting times.

Figure 4.3 shows the time overhead. In five out of the twelve cases, the time overhead was within the variance of the unmodified runtime. In six of the remaining seven

Benchmark	**Min (+prof)**	**Max (+prof)**	**Avg (+prof)**
blackscholes	60.50 (-0.69)	62.61 (+0.61)	61.13 (-0.22)
bodytrack	73.27 (+0.21)	76.57 (-1.91)	74.07 (-0.11)
canneal	94.11 (+2.78)	100.67 (+4.05)	97.93 (+2.33)
dedup	12.22 (+0.87)	13.37 (+0.05)	12.78 (+0.45)
facesim	189.91 (+2.74)	199.28 (+2.93)	193.30 (+1.77)
ferret	76.54 (-0.18)	77.14 (+0.06)	76.86 (+0.03)
fluidanimate	95.42 (+11.26)	123.17 (+4.74)	110.61 (+1.95)
raytrace	103.91 (-0.15)	107.77 (-1.00)	105.80 (-0.51)
streamcluster	107.88 (+2.21)	111.08 (+0.41)	109.48 (+1.38)
swaptions	50.86 (-0.09)	51.21 (-0.04)	51.02 (-0.09)
vips	23.07 (+0.06)	26.41 (-1.12)	24.26 (-0.47)
x264	22.27 (+0.91)	29.35 (+2.82)	24.19 (+3.14)

Figure 4.3: Time overhead in seconds. Numbers in parentheses represent the overhead of profiling.

cases, the time overhead was less than 5%. The x264 benchmark has high overhead due to large number of threads launched. Since our instrumentation gathers thread local buffers that must be written to disk upon thread termination, a parallelization that launches a large number of threads with relatively little work causes a large serial overhead when those buffers are saved to disk. Even in this case, the runtime overhead is modest.

Figure 4.4 shows the space overhead. In nine out of twelve benchmarks, the space overhead is less than 10%. Two of the remaining three have overhead less than 25%. The highest overhead was exhibited by the bodytrack benchmark at 45.75%. These overheads are acceptable as experiments were conducted by holding all generated events in memory and only writing them out to disk at thread termination. If peak memory consumption becomes a problem, then we can periodically flush the thread local buffers to disk.

Finally, Figure 4.5 shows the size of the log files generated. Only one benchmark (facesim) had log files with an average size in excess of one MB. These small log files, as

Benchmark	**Min (+prof)**	**Max (+prof)**	**Avg (+prof)**
blackscholes	627048 (-12)	627088 (+1872)	627069 (+414)
bodytrack	33872 (+13764)	33948 (+16072)	33912 (+15516)
canneal	962436 (+3560)	968508 (-296)	964042 (+2902)
dedup	1642340 (+93244)	1760540 (+2210)	1717930 (+32310)
facesim	324576 (+29124)	329840 (+34948)	326874 (+30664)
ferret	117992 (+4920)	130232 (-2836)	121317 (+4443)
fluidanimate	693100 (+3256)	693304 (+5324)	693192 (+4482)
raytrace	1161224 (252)	1162460 (-270)	1161950 (-160)
streamcluster	113504 (+24984)	117896 (+23992)	114980 (+24977)
swaptions	6220 (-20)	8220 (-44)	7715 (-953)
vips	61432 (+7276)	65808 (+7564)	63171 (+7609)
x264	299528 (+6604)	303408 (+10912)	302154 (+8320)

Figure 4.4: Peak memory overhead in Kilobytes.

Benchmark	**Min (+prof)**	**Max (+prof)**	**Avg (+prof)**
blackscholes	627048 (-12)	627088 (+1872)	627069 (+414)
bodytrack	33872 (+13764)	33948 (+16072)	33912 (+15516)
canneal	962436 (+3560)	968508 (-296)	964042 (+2902)
dedup	1642340 (+93244)	1760540 (+2210)	1717930 (+32310)
facesim	324576 (+29124)	329840 (+34948)	326874 (+30664)
ferret	117992 (+4920)	130232 (-2836)	121317 (+4443)
fluidanimate	693100 (+3256)	693304 (+5324)	693192 (+4482)
raytrace	1161224 (+252)	1162460 (-270)	1161950 (-160)
streamcluster	113504 (+24984)	117896 (+23992)	114980 (+24977)
swaptions	6220 (-20)	8220 (-44)	7715 (-953)
vips	61432 (+7276)	65808 (+7564)	63171 (+7609)
x264	299528 (+6604)	303408 (+10912)	302154 (+8320)

Figure 4.5: Log file sizes in Kilobytes.

< 1000	< 10000	< 100000	< 1000000
800857	247680	38	1

Table 4.1: Sizes of empty regions, in cycles.

well as the very small execution time overhead and modest memory overhead are strong evidence of the efficiency of CSPs for the analysis of real world, long running multi-threaded programs.

In addition to the above, we also performed an experiment designed to directly quantify the impact of the profiling code on the size of the measured frames. We did this by repeatedly measuring the size of an empty region. We use the size of this region as an estimate of the profiling overhead for a single frame. We used the same experimental setup as before, and measured the size of an empty region 2^{20} times. The breakdown of the sizes of these regions is given in the following table:

The figures listed in this table do not overlap. In other words, the second column does not include all regions included in the first column. 76.38% of the regions that were measured had values less than 1000, and 99.99% had values that were less than 10000. The 39 outlier regions were due to two factors: first, expansion of the thread local buffer that stores the events, and second, context switching by the operating system.

To put these figures in context, we examined the frames that we captured while running the Parsec benchmarks in terms of these numbers. Specifically, we looked at the contribution of frames less than these sizes to the overall waiting time. In contrast to the total time overhead figure above, this gives a more accurate picture of the frames of interest are affected (in this case, those in which some thread is waiting). The results are summarized

benchmark	% < 1000	% < 10000
blackscholes	N/A	N/A
bodytrack	0.012	0.275
canneal	0.0063	0.0397
dedup	1.518	3.055
facesim	0.0002	0.236
ferret	0.000	0.006
fluidanimate	0.0056	0.0537
raytrace	0.0005	0.0080
streamcluster	0.9411	6.1688
swaptions	N/A	N/A
vips	0.0164	0.3157
x264	0.3326	1.4250

Table 4.2: Ratio of time spent in small frames to overall waiting time in Parsec benchmarks.

in table 4.2.

The first column lists the percentage of overall waiting time contributed by frames whose duration was less than 1000. The second column does the same for frames whose values were less than 10000. The highest overhead was exhibited by the streamcluster benchmark at just over 6%. This is because very little waiting occurred in this benchmark, and so the frames in which waiting occurred were small relative to the size of the instrumentation overhead. This case, in which small regions need to be measured, is where our tool (and indeed any tool which uses instrumentation) will have the least accurate results. The impact of this is mitigated by the tendency to focus on large regions during performance debugging, since those are typically the places where optimizations will have the largest impact.

4.3 Accuracy of Captured Behaviors

If profiling changes the behavior of the program too drastically, then the corresponding measurements will not be useful. To test the accuracy of our profiler, we studied three microbenchmarks with predictable behavior: one to study contention, another to study software pipelining, and a final one to study straggler threads. We introduce the queries needed to understand performance behaviors of these common parallel programming patterns. The control over their operating parameters allows us to effectively study CSP's accuracy.

4.3.1 Accuracy of Contention Measurement

Contention for shared resources is one of the primary factors limiting achievable parallelism. A common programming pattern when dealing with shared resources is to use locks to provide mutual exclusion. However, waiting on locks to shared resources can lead to significant performance degradation. Using CSPs, one can easily measure the contention for shared resources by marking the associated lock acquisition as a code region, and then issuing any number of interesting queries. For example, one might be interested in the cumulative waiting time of each thread on the lock. If we represent the lock acquisition code region as L, then the above measure for a thread t is given by the query $Duration((t, L))$. If one is interested in not only which threads experience long wait times, but also which threads cause those threads to wait, then this can be easily accomplished as well. If we assume that the critical section itself is marked as code region C, then the length of time which thread t waited on lock L due to thread t' is given by the query $Duration((t, L) \wedge (t', C))$.

This analysis also generalizes to other synchronization primitives such as semaphores and reader/writer locks.

This microbenchmark tests the ability of our profiler to accurately measure contention among threads. It consists of a number of threads which attempt to acquire a shared global mutex, simulate some work while holding it, and then release the mutex. We simulate the work by each thread incrementing a local variable some number of times (32K in our experiments). Each thread calls this function 128 times, and a barrier is established at the start of the function to ensure that all iterations are performed in lock step. The work simulation is performed identically to the straggler case, and for this benchmark we used a work degree of 32768 (2^{15}) and an iteration count of 128. We measure the contention as the cumulative amount of time spent waiting on the mutex. For instance, if there are two threads each doing W work, then the ideal measure of the contention would be W, because the second thread has to wait for the first to complete its work before it can acquire the mutex and begin. In general, for T threads, the **ideal measurement** of contention in terms of work W is as follows:

$$(T-1)W + (T-2)W + ... + W = W \cdot \sum_{T=0}^{T-1} T = W \cdot \frac{(T)(T-1)}{2}.$$

If M is the code region representing the acquisition of the global mutex, then the query to measure the degree of contention as described above is given by $Area(\exists t : (t, M))$. We ran this benchmark for between 1 and 16 threads, and plotted the results in Figure 4.6. In this figure, the x-axis marks the number of threads, and the y-axis marks the cumulative

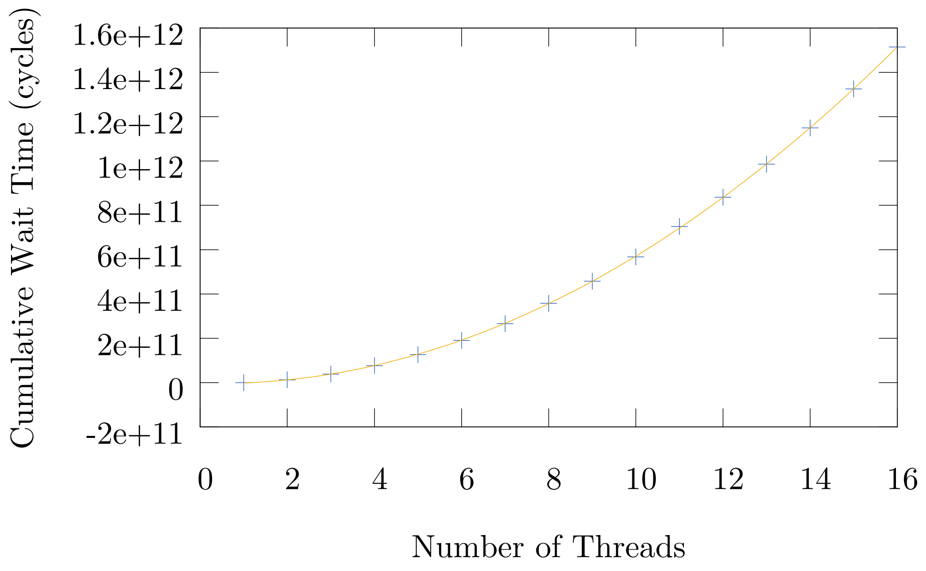


Figure 4.6: Contention variation.

time spent in acquiring the mutex. The crosses represent measured values and the curve represents the best quadratic fit. As Figure 4.6 shows, measured results conform quite closely to the expected quadratic distribution.

To illustrate the use of this contention query, we computed the total waiting time among application threads of the `bodytrack` benchmark from Parsec suite – the program was run with 16 threads and waiting was measured by annotating calls to `pthread_cond_wait`, `pthread_barrier_wait`, and `pthread_mutex_lock`. The contention was found to be quite high. In an attempt to alleviate this high contention, we reduced the number of threads by half, and measured the contention again. We found that the waiting time was reduced to 31.5% of original which translated into 2% reduction in overall execution time.

4.3.2 Accuracy of Pipeline Behavior

Software pipelining is commonly used for parallelization. In a typical implementation of a software pipeline, each pipeline stage is represented by a thread pool and a function which implements the work of that stage. Data is passed from one stage to the next, and threads in a pool cooperate to transform that data in some way and pass it along to the next stage. Synchronization between stages occurs during this hand off of data. Since each stage is data-dependent on the one before it, the overall performance of the pipeline is limited by the slowest stage. Analyzing these types of applications using traditional tools can be challenging since most of those see threads as individual actors instead of cooperating entities. CSP queries make this type of aggregation very convenient, as shown below.

We characterize the performance of a software pipeline by measuring the amount of time that each stage is caused to wait due to every other stage. For a pipeline with n stages, this can be summarized in what we call a *waiting matrix*:

$$W_{m,n} = \begin{pmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,n} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,1} & w_{m,2} & \cdots & w_{m,n} \end{pmatrix}$$

in which the entry $w_{i,j}$ denotes the amount of time stage i spent waiting for stage j. These quantities can be calculated in a straightforward manner using CSPs. Let $threads(k)$ be the threads that implement stage k of the pipeline (i.e., $threads(1)$ are the threads belonging to the pool of the first stage). Furthermore, let B be the pipeline stage barrier for stage i, and let P be the work region for stage j. The query which calculates this quantity has two parts. The first part specifies that stage i has completed. Another way to say this is that all of the threads which belong to stage i are waiting at the pipeline barrier. This query can be written as:

$$Duration(\forall t \in threads(i) : (t, B))$$

The second part specifies that pipeline stage j has not completed. Another way of saying this is that there is at least one thread from stage j which is still executing that stage region. This query can be written as:

$$Duration(\exists t \in threads(j) : (t, P))$$

The final query is just a conjunction of these two queries, or:

$$Duration(\forall t \in threads(i) : (t, B) \wedge \ \forall t' \in threads(j) : (t', P))$$

In this microbenchmark, we set up a pipeline with 3 stages. Each stage does some amount of work, which is split among a set of threads. The amount of work, the number of threads per stage, and the distribution of the work among the threads of the pipeline stage is configurable via command line arguments. We simulate the passing of data between the stages of the pipeline by having each of the stages synchronize with each other. That is, a stage cannot proceed to the next iteration until all other stages have completed.

In experiments related to this microbenchmark, we use our profiler to create a waiting matrix as defined above, with the entries normalized with respect to the stage length execution time. We allocate three threads to the first stage, three threads to the second stage, and two threads to the third stage.

– **Balanced Stages.** For our first experiment, we gave each thread in each stage equal amount of work (2^{30} increments). For any waiting matrix, we expect the diagonal entries to be exactly zero since it is not possible for a pipeline stage to be waiting for itself. For the other entries of this matrix, we expect the values to be close to zero, since the work is balanced across stages. The waiting matrix (W), and the expected waiting matrix ($\mathcal{W}$) for this experiment are shown below. We observe the measured values are close to expected values.

$$\mathcal{W} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad W = \begin{pmatrix} 0 & 0 & 6.41 \times 10^{-2} \\ 8.96 \times 10^{-3} & 0 & 7.3 \times 10^{-2} \\ 0 & 0 & 0 \end{pmatrix}$$

– **Single Stage Imbalance.** For our second experiment, we removed the workload for the threads in the third stage of the pipeline. Since the threads in this stage now have no work, we expect the third row of the resulting matrix to have values very close to one. The resulting waiting matrix is given below. Once again we can see the measured values are very close to expected values.

$$\mathcal{W} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 0 \end{pmatrix} \quad W = \begin{pmatrix} 0 & 8.57 \times 10^{-4} & 0 \\ 0 & 0 & 0 \\ 9.99 \times 10^{-1} & 1 & 0 \end{pmatrix}$$

– **Double Stage Imbalance.** For our final experiment, we removed the workload for the threads in the second stage as well. We now expect to see very high values for $W_{2,1}$ and $W_{3,1}$, but very low values for the other entries in these rows. As in previous cases, the resulting waiting matrix is shown below and conforms closely with our expectations.

$$\mathcal{W} = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix} \qquad W = \begin{pmatrix} 0 & 0 & 0 \\ 9.98 \times 10^{-1} & 0 & 0 \\ 9.99 \times 10^{-1} & 0 & 0 \end{pmatrix}$$

As an example of performance tuning using a waiting matrix, let us consider the `ferret` program from Parsec suite that uses a 6 stage pipeline. The middle four stages have an equal number of threads. We captured the waiting matrix when running the default version using 4 threads each for the middle stages. The waiting matrix indicates that only the `rank stage` experiences no waiting. Therefore we increased the number of threads of this stage to 8 threads which yielded 1.73× speedup and thus a better performing configuration.

4.3.3 Accuracy of Straggler Degrees

Next we show how we can use the frame query language to detect and identify *straggler threads*. Let L and B denote the regions corresponding to some loop and a barrier at the end of each iteration of that loop respectively. The presence of a straggler is indicated by frames where one thread is in region L while all other threads are in region B. The performance impact of the straggler can be characterized by computing the ratio of $StaggerDuration$ and $LoopDuration$ which represent the time all other threads spend waiting at the barrier for the straggler thread and the total time spent on executing the entire loop respectively. We denote this as *StragglerDegree* whose value falls between 0 and 1 with closer to one being worse:

$$\text{LoopDuration} = \mathit{Duration}(\exists t : (t, L))$$

$$\text{StragglerDuration} = \mathit{Duration}(\exists_1 t : (t, L) \wedge \forall t' \neq t : (t', B))$$

$$\text{StragglerDegree} = \frac{\text{StragglerDuration}}{\text{LoopDuration}}$$

There can be multiple causes for the presence of a straggler. (1) A thread may be assigned relatively too much work. (2) The code representing the work preceding the barrier can exhibit a great deal of variability in its execution time and thus different threads manifest as stragglers during different executions of the barrier. To determine the actual scenario we can identify the *StragglerDegree* for each thread t using concretization below:

$$\text{StragglerDuration}_t = \mathit{Duration}((t, L) \wedge \forall t' \neq t : (t', B))$$

$$\text{StragglerDegree}_t = \frac{\text{StragglerDuration}_t}{\text{LoopDuration}}$$

This microbenchmark is structured around a single function which executes a loop for a predetermined number of iterations. The body of this loop contains a call to a function which simulates work by incrementing a local variable w times. Each iteration of the loop is guarded at the beginning and the end by a barrier. This ensures that no thread is executing the work region if there is some other thread in a different iteration of the loop. This function is executed concurrently by 8 threads. In each experiment we mark the work function and loop barriers as code regions, spawn 8 threads, and perform the straggler analysis as above.

Tid	StragglerDegree
1	0.124939
2	0.123676
3	0.123717
4	0.123647
5	0.126137
6	0.124745
7	0.124485
8	0.125010

Figure 4.7: Small Workload Even Distribution.

Tid	StragglerDegree
1	0.000172
2	0.000191
3	0.000139
4	0.000151
5	0.000164
6	0.000162
7	0.000179
8	0.000168

Figure 4.8: Even Work Distribution.

– **Small Workload Evenly Distributed** In this experiment we give each of the threads an equal but very small amount of work. We expect each thread to be able to finish its work before the next thread is released from the barrier and begins its work.

$$\text{StragglerDegree} = \frac{w}{8w} \approx \frac{1}{8}$$

Since we expect no bias in the order in which the threads are released from the barrier, we expect the threads to have approximately the above degree. The results of this experiment are shown in Figure 4.7. The above table confirms that measured straggler degrees for all threads are close to the expected value.

Tid	StragglerDegree
1	0.497517
2	0.000000
3	0.000000
4	0.000000
5	0.000000
6	0.000000
7	0.000000
8	0.000000

Figure 4.9: Uneven Distribution.

– **Even Work Distribution.** In this experiment we increase the amount of work given to each thread so that it is large relative to the amount of time it takes to exit the barrier. This should lead to a situation where threads do almost all their work concurrently. Assuming there is no scheduling bias, then all of the threads should arrive at the end barrier at about the same time. Whichever thread arrives last will incur a straggler penalty equivalent to however long it remains as the only thread in the loop. We expect the straggler score for this last thread to be:

$$\text{StragglerDegree} = x/w \approx 0$$

Furthermore, as the work size increases, we expect this degree to approach zero. We expect no bias amongst the threads as to which arrives last on any given iteration, and so we expect to see every thread with a straggler degree close to zero. The measured results of this experiment, shown in Figure 4.8, are as expected.

– **Uneven Work Distribution.** In our final experiment, we induce a straggler problem by doubling the work that the last thread performs. In this case, we expect all but the last thread to arrive at the barrier with about w work still left for the last thread to perform. We expect the straggler degree to be:

$$\text{StragglerDegree} = w/2w \approx 1/2$$

This time, since the work is not evenly distributed, we expect to see the same thread as the straggler every time. By extension, we expect one thread to have a straggler degree of about one half, and the rest to have a score of zero. The measured results of this experiment, shown in Figure 4.9, are close to expected results.

As an example of using this query, let us examine the `swaptions` benchmark from Parsec. In an older version of Parsec, this benchmark was arranged so that work was not ideally distributed amongst the worker threads [34]. When the native input set containing 128 work items is split amongst 13 threads, 12 of the threads are given 9 items, and the 13th is given 20 items. This leads to one of the threads having a distinctly high straggler degree as evaluated using the above query. When we amend the work distribution so that 11 threads are given 10 items and 2 threads are given 9 items, we realize a speedup of 1.45×.

4.4 Related Work

There are tools that make use of sampling and hardware performance counters to provide a highly scalable way to analyze concurrent programs [4, 36, 18]. There are a few challenges with this approach. The first is that of relating the performance statistics (such as cache misses) to source level entities. The second is in interpreting the fine grained measurements to locate and explain performance problems.

HPCToolkit [4] uses hardware sampling techniques to collect information about many low level events, and then attributes those events to program entities such as loops and procedures. It also provides an interface that allows these metrics to be combined into new metrics for interactive analysis. The main benefit of this approach is that it is highly scalable, and supports detailed hardware level measurements. [28] provides similar measurements in a data-centric manner. Tmon [24] is a tool for measuring program bottlenecks induced by various forms of excessive waiting. It primarily constructs four things. The first is a *waiting graph*, which indicates how much time each thread spent waiting for every other thread. The second is a histogram of the *waiting queue lengths* for each synchronization variable, which is used to determine synchronization variables which are the cause of excessive waiting. The third is a similar histogram for the *ready queue*, which gives a measure of the contention for the CPU. The final thing they measure is a histogram of the *number of wakeups* associated with each condition variable, which when combined with the number of total waits on that condition variable, gives a measure of what they call "semi-busy-waiting". The main benefit of hardware performance counters based approaches is their scalability. Since hardware performance counters are (as their name suggests) implemented in hardware, their cost is effectively zero. This strength also begets one of their biggest weaknesses: inflexibility.

Other works have considered software approaches for limiting the cost of performance profiling. Log^2 [13] system reduces the cost of logging via a sophisticated filtering system which is configured and adapted to discard events that are unlikely to be useful. We limit logging of events by: allowing user to select regions; and conditional logging.

4.5 Summary

In this chapter, we presented the implementation of CSPs on shared memory machines. By making use of the very common timestamp counter feature in modern x86 machines, CSPs are able to capture profiling information with very little overhead: typically less than 5% in execution time and 46% in peak memory consumption. Additionally, using the timestamp counter imparts zero inter-thread communication overhead since there is a timestamp counter local to each core of the machine. Using carefully crafted synthetic microbenchmarks, we showed that the information computed from CSPs is highly accurate across a diverse set of circumstances. In conclusion, CSPs are both highly efficient and highly accurate.

Chapter 5

Distributed Platform: The FreeZer Timestamp Conversion Algorithm

A flexible profiler should not be restricted to a single threading library or platform. Thus far we have presented an implementation of CSPs only for shared-memory multicore machines. In the next two chapters, we show how to generalize CSPs so that they can work with distributed systems as well.

Timestamps play a fundamental role in the construction of CSPs. Without them, we would not be able to determine frame boundaries or even order events in the system. A distributed system poses a challenge for systems that use timestamps, since they have multiple clocks. Two approaches are possible: either use a global clock (expensive) or use local clocks and deal with the inconsistencies that involves. The algorithms for dealing with these inconsistencies are known as *timestamp synchronization* algorithms. In this chapter, we present the timestamp conversion problem in detail, show that existing algorithms are inadequate for the construction of CSPs, and present our new algorithm called FreeZer for timestamp synchronization.

5.1 Timestamp Conversion Problem

Performance analysis of distributed programs involves capturing relevant program events along with their timing information to deduce underlying performance bottlenecks. While timing information can be easily gathered across events within a single machine, absence of a global clock in any distributed setting makes it challenging to accurately capture the timing information for events spanning across multiple machines.

To facilitate comparison across event timings that are captured locally on different machines, we first convert them to a common timeline and then directly compare these converted times. We first formally define the timestamp conversion process, and then discuss the challenges involved in accurately converting event timestamps.

Distributed Timing Model Let $N = \{n_0, n_1, ..., n_k\}$ be the set of nodes in a distributed system with k compute nodes and let $C = \{c_0, c_1, ..., c_k\}$ be the set of clocks such that $\forall c_i \in C, c_i$ is the clock for n_i. With absence of a global clock, the clocks in C are not synchronized with each other and operate at different frequencies. Today's multicore machines have a separate clock associated with each core, hence, each core becomes a separate node in our distributed system. However, this model can be used for distributed settings where all cores within a socket or all cores within a machine rely on single synchronized clock, by simply modeling each socket or each machine as a node in the distributed system.

Let $TS = \{ts_0, ts_1, ..., ts_k\}$ be the captured timestamps such that $\forall ts_i \in TS, ts_i$ is captured on c_i in units of clock cycles. Let $rt(ts_i)$ be an abstract oracle that maps ts_i to the real time. Since the captured timestamps are relative to their respective local epochs, $\forall ts_i,\ ts_j \in TS, ts_i$ may or may not be equal to ts_j even though $rt(ts_i) = rt(ts_j)$. Timestamp conversion is the process to convert all the local timestamps to a common base so that they become directly comparable. In particular, a function $conv(ts_i)$ that converts any given timestamp to a common base must achieve the following $\forall\ ts_i,\ ts_j \in TS$:

$$\begin{gathered} rt(ts_i) \oplus rt(ts_j) \leftrightarrow conv(ts_i) \oplus conv(ts_j) \\ |rt(ts_i) - rt(ts_j)| = |rt(conv(ts_i)) - rt(conv(ts_j))| \end{gathered} \tag{5.1}$$

where $\oplus \in \{=, \neq, >, <, \geq, \leq\}$. The first relation ensures that the ordering of timestamps becomes comparable after conversion while the second relation ensures that the timing information is preserved across the converted timestamps. For simplicity, we assume that our common base is provided by n_0, i.e., $\forall ts_i \in TS \setminus \{t_0\}, conv(ts_i)$ converts the ts_i in terms of c_0 such that $conv(ts_i)$ is directly comparable to ts_0. To convert ts_i in terms of c_0, we need two measures:

1. Frequencies of c_i and c_0, referred to as $fr(c_i)$ and $fr(c_0)$ respectively.

2. Special timestamps, z_i and z_0 that are captured on c_i and c_0 respectively at *the same moment of real time*. We call these special timestamps as *zeros*.

With the above measures, ts_i can be converted in terms of c_0 using $conv(ts_i)$ defined as:

$$conv(ts_i) = (ts_i - z_i) \times \frac{fr(c_0)}{fr(c_i)} + z_0 \tag{5.2}$$

Problem The main challenge is to compute $fr(c_i)$ and z_i measures with high precision to ensure that the relations in Eq. 5.1 are maintained correctly. While accurate estimates can be computed when low-precision frequency and zero measures are adequate, the difficulty increases quickly with rising precision requirement.

Our Approach Since in practice we cannot fully control the accuracy of frequency and zero measures to a precision high enough that guarantees correct comparison of timestamps, we tightly bound the inaccuracies in these estimates so that the timestamp conversion can provide strong guarantees for converted values. This means, $fr(c_i)$ and z_i are now in interval domains with strong lower and upper bounds and $conv(ts_i)$ converts ts_i from time domain to interval domain to preserve strong bounds over the converted timestamp value, hence capturing conversion inaccuracies.

5.2 FreeZer: Bounded Frequency & Zero Estimation

In this section, we analyze the inaccuracies involved in estimating frequency and zero measures, and develop solutions to tightly bound the estimates for strong profiling guarantees.

Algorithm 1 Computing absolute frequency of c_i

```
1: $ts_1 \leftarrow$ TIMESTAMP( )    // e.g., using x86 TSC
2: SLEEP($t$)
3: $ts_2 \leftarrow$ TIMESTAMP( )
4: $f_i(t) \leftarrow (ts_2 - ts_1)/t$
```

5.2.1 Estimating Relative Frequencies

A straightforward way to compute frequency estimates is to individually determine the absolute frequency at which each clock is operating by counting the number of cycles elapsed for a predetermined amount of time, as shown in Algorithm 1. Here counting cycles can be achieved using local x86 timestamp counters (TSC) – the use of the invariant TSC, widely available on modern x86 Intel and AMD systems, provides a constant frequency time source even in presence of dynamic frequency scaling. While this solution may suffice for capturing less accurate frequency estimates, computing highly accurate frequencies requires precisely measuring a given amount of time. Unfortunately, exact measurements are impossible due to the inherent non-determinism in today's computing systems. For instance, the `sleep` commands only guarantee suspension for *at least* (*and not exactly*) the amount of time provided by the user. Furthermore, there is no guarantee of the closeness of the amount of time that is actually slept to the amount of time that is requested. Although increasing t to a high enough value in Algorithm 1 can arbitrarily mitigate these inaccuracies, determining a sufficient value of t for some user defined tolerance is infeasible for reasons already discussed.

Since we want to compare the timestamps of distributed events (i.e., timestamps captured across on clocks), we aim to compute a relative measure between multiple clocks. Comparing frequencies of multiple clocks can be achieved by directly computing *relative frequencies* between those clocks. Furthermore, the computation for relative frequencies can be carefully controlled to provide stronger guarantees which become useful to capture the deterministic error bound. By carefully computing relative instead of absolute frequencies, we can not only arbitrarily mitigate the sources of inaccuracy, but we can do so in a way that bounds the accuracy of our estimate.

Similar to computing absolute frequencies, a simple way to compute relative frequencies across a pair of clocks is to measure the number of cycles spent for the same amount of real time t on those clocks (see Algorithm 2). In this case, $f_{j \to i}(t)$ (computed on line 5) is the relative frequency to convert durations captured on c_j to the units of c_i. We can capture the inaccuracies introduced by the non-deterministic factors as follows:

$$f_{j \to i}(t) = \frac{fr(c_i) \times (t + a)}{fr(c_j) \times (t + b)} \tag{5.3}$$

where $fr(c_i)$ is the frequency of clock i, and variables a and b represent the inaccuracies introduced by the sleep calls (i.e., a call to `sleep(t)` will introduce a delay of $t+a$ seconds for some positive value of a). We can carefully eliminate the effect of inaccuracies by taking the limit as t tends to ∞ as described next. Let A and B be positive constants such that $a < A$ and $b < B$. Consider continuous functions $f^{+}_{j \to i}$ and $f^{-}_{j \to i}$ as defined below:

Algorithm 2 Computing relative frequency of c_i and c_j using their absolute frequencies

```
c_i:
1: $ts_1 \leftarrow$ TIMESTAMP( )
2: SLEEP($t$)
3: $ts_2 \leftarrow$ TIMESTAMP( )
4: $diff_i \leftarrow ts_2 - ts_1$

c_j:
1: $ts_1 \leftarrow$ TIMESTAMP( )
2: SLEEP($t$)
3: $ts_2 \leftarrow$ TIMESTAMP( )
4: $diff_j \leftarrow ts_2 - ts_1$

5: $f_{j \to i}(t) \leftarrow diff_i \; / \; diff_j$
```

$$f^{+}_{j\to i}(t) = \frac{fr(c_i) \times (t+A)}{fr(c_j)(t)} \quad \Bigg| \quad f^{-}_{j\to i}(t) = \frac{fr(c_i)(t)}{fr(c_j) \times (t+B)}$$

By L'Hopital's rule, we have:

$$\lim_{x\to\infty} f^{+}_{j\to i}(x) = \lim_{x\to\infty} f^{-}_{j\to i}(x) = \frac{fr(c_i)}{fr(c_j)}$$

Since $f^{-}_{j\to i}(t) \leq f_{j\to i}(t) \leq f^{+}_{j\to i}(t)$, we can conclude that:

$$\lim_{t\to\infty} f_{j\to i}(t) = \frac{fr(c_i)}{fr(c_j)}$$

Hence, the impact of inaccuracies when calculating relative frequencies can be eliminated by increasing t to a very high value (as is the case with absolute frequencies). While exact relative frequencies cannot be practically achieved, we can determine the rate at which relative frequencies become accurate as t increases. The error in a frequency estimate can be computed based on Eq. 5.3 as:

$$\text{Error} = \frac{fr(c_i) \times (t+a)}{fr(c_j) \times (t+b)} - \frac{fr(c_i)}{fr(c_j)}$$

To achieve frequency estimates that are accurate within a tolerance T, we have:

$$\frac{fr(c_i) \times (t+a)}{fr(c_j) \times (t+b)} - \frac{fr(c_i)}{fr(c_j)} < T$$
$$\implies t > \frac{1}{T}(\frac{fr(c_i)}{fr(c_j)}(a-b)) - b$$

In other words, the length of time that is required to achieve a desired accuracy grows linearly with desired accuracy.

Algorithm 3 Bounding relative frequency of c_i and c_j

by introducing causal dependencies

```
                 c_i                                          c_j

 1: $ts_1 \leftarrow$ TIMESTAMP( )              1: RECEIVE_FROM($c_i$)
 2: SEND_TO($c_j$)                              2: $ts_1 \leftarrow$ TIMESTAMP( )
 3: RECEIVE_FROM($c_j$)                         3: SLEEP($t$)
 4: $ts_2 \leftarrow$ TIMESTAMP( )              4: $ts_2 \leftarrow$ TIMESTAMP( )
 5: $diff_i \leftarrow ts_2 - ts_1$             5: SEND_TO($c_i$)
 6:                                             6: $diff_j \leftarrow ts_2 - ts_1$

                 7: $f^U_{j \to i}(t) \leftarrow diff_i \ / \ diff_j$

                                                8: $ts_1 \leftarrow$ TIMESTAMP( )
 8: RECEIVE_FROM($c_j$)                         9: SEND_TO($c_i$)
 9: $ts_1 \leftarrow$ TIMESTAMP( )             10: RECEIVE_FROM($c_i$)
10: SLEEP($t$)                                 11: $ts_2 \leftarrow$ TIMESTAMP( )
11: $ts_2 \leftarrow$ TIMESTAMP( )             12: $diff_j \leftarrow ts_2 - ts_1$
12: SEND_TO($c_j$)                             13:
13: $diff_i \leftarrow ts_2 - ts_1$

                 14: $f^L_{j \to i}(t) \leftarrow diff_i \ / \ diff_j$
```

Bounding Relative Frequencies Now we show how the error in computed frequency estimates can be bounded. To determine the accuracy of computed frequency estimates, we bound the relative frequency to intervals that contain the true relative frequency. The challenge is to make these intervals as small as possible so that the residual inaccuracies do not meaningfully impact the profiling results.

The key insight is that instead of trying to measure the same duration on each clock, we intentionally measure durations which differ in a specific way. In particular, an upper bound can be computed by measuring a larger interval on the first clock compared to that on the second clock, and a lower bound can be computed by reversing this procedure, i.e., by measuring a smaller interval on the first clock compared to that on the second clock. Hence, we can achieve lower bound $f^L_{j\to i}(t)$ and upper bound $f^U_{j\to i}(t)$ defined as follows:

$$f^L_{j\to i}(t) = \frac{fr(c_i)\times t}{fr(c_j)\times(t+\epsilon)} \quad \Bigg| \quad f^U_{j\to i}(t) = \frac{fr(c_i)\times(t+\epsilon)}{fr(c_j)\times t}$$

$$f^L_{j\to i}(t) < f_{j\to i}(t) < f^U_{j\to i}(t)$$

where ϵ is an arbitrary variation to increase time intervals ($\epsilon > 0$). While any positive ϵ value can be used to compute $f^L_{j\to i}(t)$ and $f^U_{j\to i}(t)$, smaller ϵ values produce tighter bounds.

The key challenge is to control the timing to ensure that ϵ gets correctly applied to the desired clock's measurement. Hence, we don't rely on measuring an increased real time using `sleep` due to the inherent non-determinism, but we instead achieve this using appropriate synchronization primitives. In order to ensure the epsilon is applied to the correct machine, we introduce a causal dependency between the appropriate clock measurements using communication primitives (i.e. send/recv). Algorithm 3 shows how we compute bounds $f^L_{j\to i}(t)$ and $f^U_{j\to i}(t)$.

With $f_{j\to i}(t)$ bounded by $f^L_{j\to i}(t)$ and $f^U_{j\to i}(t)$, the error in $f_{j\to i}(t)$ is bounded by $f^U_{j\to i}(t) - f^L_{j\to i}(t)$. If we were interested in minimizing the maximum error, we could take the midpoint of this interval as a point estimate. However, directly using the midpoint will lose the information about the direction of the error. Hence, instead of using the midpoint value, we directly use the captured bounds ($f^L_{j\to i}(t)$ and $f^U_{j\to i}(t)$) while processing runtime profiles to provide strong end-to-end guarantees over profiling results.

5.2.2 Estimating Zeros

To analyze timestamps measured via different clocks, we also need to compute comparable timestamps across those clocks that capture the same moment in real time. These comparable timestamps will be used to shift all the remaining timestamps so that the resulting time scale starts from zero. Hence, we refer to these pairs of comparable timestamps as *zeros*.

Algorithm 4 Computing zero for c_i and c_j using causal dependencies

```
c_i                                    c_j

1: RECEIVE_FROM($c_j$)  <──┐           1: $ts_1 \leftarrow$ TIMESTAMP( )
2: $ts_2 \leftarrow$ TIMESTAMP( )  └──  2: SEND_TO($c_i$)
3: SEND_TO($c_j$)  ──────────────────> 3: RECEIVE_FROM($c_i$)
                                       4: $ts_3 \leftarrow$ TIMESTAMP( )

5: $zero_{j\to i} \leftarrow <(ts_1, ts_3), ts_2>$
```

Algorithm 4 shows how we can compute zeros across a pair of clocks. We do so by trapping a timestamp from one clock between a pair of timestamps taken on the other clock. To achieve this, we introduce causal dependencies that ensure that ts_2 on c_i is captured between ts_1 and ts_3 on c_j. Hence, $zero_{j \to i} = <(ts_1, ts_3), ts_2>$ effectively means:

$$rt(ts_1) < rt(ts_2) < rt(ts_3)$$

where $rt(ts_i)$ maps ts_i to the moment in real time when it was captured. While the zero can be approximated by picking the midpoint so that it becomes $<(ts_1 + ts_3)/2, ts_2>$, similar to relative frequency bounds, we explicitly maintain the bounds for zeros so that they can be directly used along with relative frequency bounds to provide strong guarantees over profiling results. The lower bound of the zero (i.e., ts_1) is matched with that of relative frequency (i.e., $f^L_{j \to i}(t)$) to achieve a lower bound of the converted timestamp, while the upper bound of zero (i.e., ts_3) is matched with that of relative frequency (i.e., $f^U_{j \to i}(t)$) to achieve an upper bound of the converted timestamp. With both bounded frequencies and zeros, strong conversion bounds can be provided for all timestamps taken in the system.

5.3 Evaluation of Timestamp Synchronization

In this section we evaluate the accuracy and cost of our timestamp synchronization method and compare it with the accuracy and costs of existing methods. The experiments were carried out on a heterogeneous 8-node cluster with a total of 76 cores operating at 800-2,261 MHz and 8-32 GB main memory per node. Each node runs 64-bit Ubuntu 14.04 kernel.

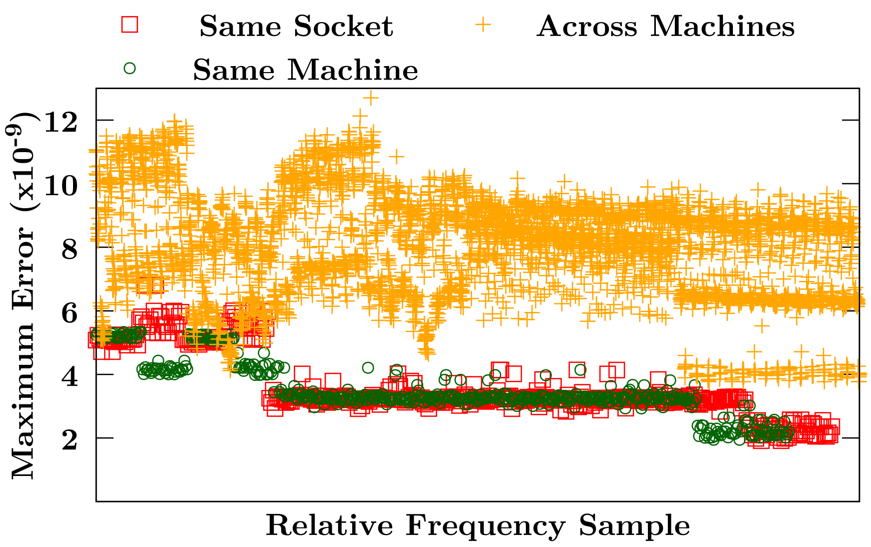


Figure 5.1: Distribution of maximum error in relative frequency measurements.

Maximum Error - Relative Frequency and Zeros In our first experiment, we sought to get an idea for how the error in the relative frequency is affected by the relative location of the clocks in the cluster. To do so, we calculated the relative frequencies of each of the clocks in our cluster. Measurements were performed for each pair of cores (there are 76 cores spread across 8 machines). We measured the frequencies over a duration of 24 hours in order to minimize the effects of variance in communication costs. Figure 5.1 shows the maximum error in terms of the difference between the bounds for our relative frequency measurements. Since the actual relative frequency can be anywhere within the computed bounds, we measure the worst case error as the size of the range defined by those bounds. As we can see, the maximum error is clustered in two regions: (across machines) the causal dependencies incur a round trip network communication that increases the error; and on (same machine) the errors for different cores are lower as there is no network overhead.

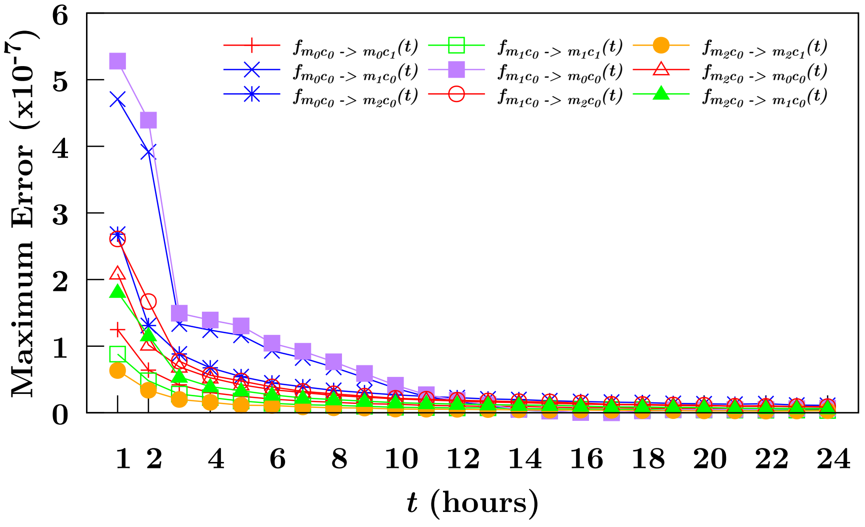


Figure 5.2: Increase in relative frequency accuracy with t. Subscript m_ic_j indicates core j on machine i.

While frequency error for cores across different sockets have higher error in some samples compared to error for those within the socket, the inter-socket communication does not impact the relative frequency measurements as much.

We validated our claim that length of time required to achieve a desired accuracy grows linearly with desired accuracy by measuring the maximum error of a relative frequency as difference of its bounds, i.e., $error_{j\to i} = f^{+}_{j\to i}(t) - f^{-}_{j\to i}(t)$ for varying t. Figure 5.2 shows that the error drops rapidly and after 12 hours the accuracy increase is extremely small.

Our next experiment measures the distribution of the error sizes among the zero values calculated over the clocks in our cluster. We calculated the zero between each pair of clocks in our system, and took the maximum error as the difference between the upper and lower bounds in the measurement. Figure 5.3 summarizes the results of this experiment. The plurality of the zero values had an error around 0.7 million cycles, and none of the

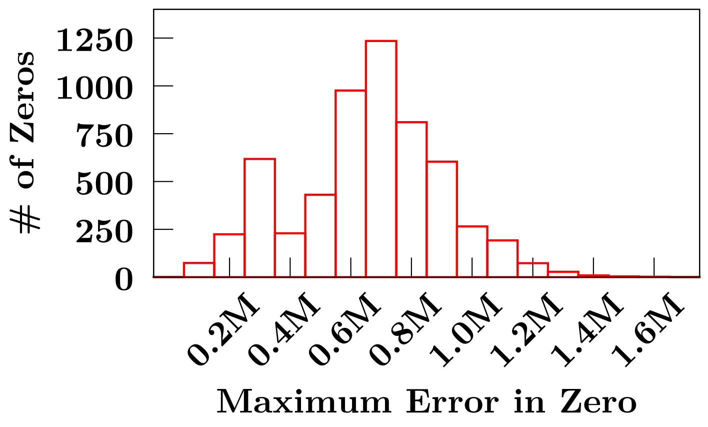


Figure 5.3: Distribution of maximum errors in zero measures.

measurements had zero values greater than 1.6 million cycles. For the gigahertz clocks present on the machines in our system, the error in the zero measurement is in the millisecond range.

Algorithm 5 Algorithm for timestamp inversions

```
Process 1
1: $ts_1 \leftarrow$ TIMESTAMP( )
2: SLEEP($d$)
3: BARRIER( )

Process 2
1: BARRIER( )
2: $ts_2 \leftarrow$ TIMESTAMP( )
```

Inversions due to error If the error in frequency and zeros is larger than the difference between two causally ordered events in the system, then an *inversion* will occur. Our next experiment investigates how far apart these causally related events needed to be before we stopped witnessing inversions. We used the algorithm from Algorithm 5 to generate pairs of timestamps which are causally ordered, then we converted the first timestamp to the base

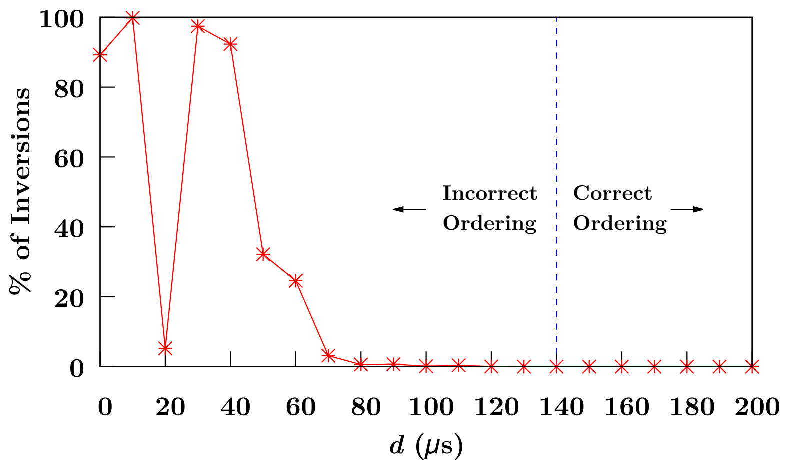


Figure 5.4: Timestamp inversions in Algorithm 5 indicating potential $rt(ts_2) \ngtr rt(ts_1)$ as d increases.

of the second timestamp and compared them to see if there was an inversion. By inserting a sleep between the taking of the first timestamp and the waiting at the barrier, we can separate the timestamps by an arbitrary amount of time. To mitigate the effects of variance in the sleep function we averaged each of our measurements across 1000 trials. Figure 5.4 shows that the percentage of inversions drops drastically at around the 50 microsecond delay mark and disappeared completely when the delay exceeded 140 microseconds. Thus, if the events in a program are separated by more than about 140 microseconds in real time, we expect the ordering to be correctly inferred.

We further study the impact of error in our measured frequency and zero in Algorithm 5 by computing the number of inversions with respect to the communication delay. While the `barrier()` ensures that ts_2 is captured after ts_1 in real time, we separate the two timestamps further apart by adding a delay (d in μs) on line 2. Figure 5.4 shows that the percentage of inversions across 1000 runs starts dropping sharply after 50 μs and fur-

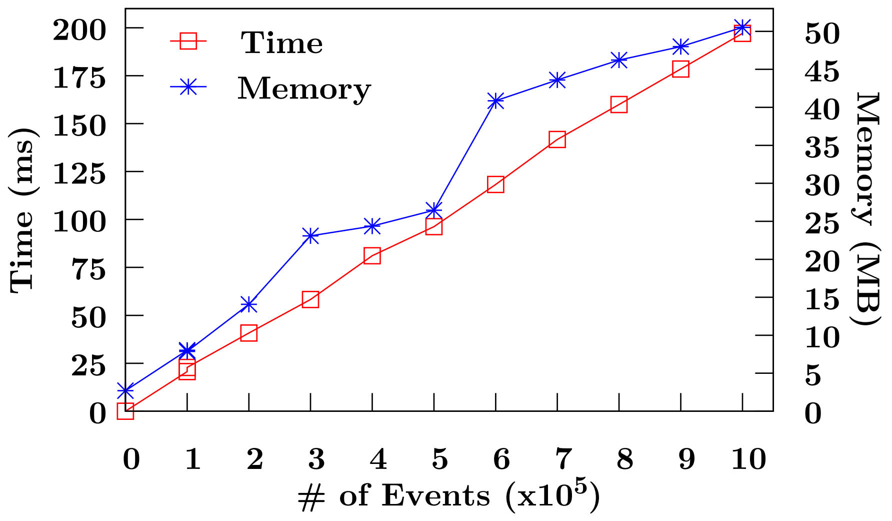


Figure 5.5: Time & memory consumption to capture events.

thermore, there are no inversions after 140 μs. This is a narrow enough window where very few (inherently concurrent) events fall that remain unordered in our profiles.

Cost of capturing events The timing information is captured using the x86 timestamp counters whose relative frequency and zero bounds are captured by FreeZer in the previous step. Querying these counters is a relatively inexpensive operation since it does not need any synchronization across different cores and can be executed in user space. This helps to keep the overall profiling lightweight and cause minimal perturbation to the program. We measured the amount of time that it took to generate and save one hundred thousand to one million events. The results in Figure 5.5 show that capturing 1M events requires only 197 ms while consuming less than 51MB of main memory.

Algorithm	MIN	AVERAGE	MAX
FreeZer	0.000	0.000	0.000
Convex hull [16]	36.855	40.647	49.490
Linear regression [16]	0.075	3.142	30.790

Table 5.1: Accuracy comparison in terms of % *fast sends*.

5.4 Comparison with Other Algorithms

Accuracy comparison Next we compare FreeZer with two other timestamp synchronization methods, *linear regression* and *convex hull* algorithms from [16]. The linear regression algorithm forms a set of points from pairs of timestamps taken from message transmission events during the execution. A linear regression is performed on this set, and the resulting trend line is used as the conversion function estimate. In the convex hull algorithm, the same set is formed as above, and then further split into two subsets based on the message transmission direction. The conversion function is estimated as the line passing through the middle of the corridor formed by the convex hulls of these subsets.

Since most algorithms do not provide explicit error bounds, we must take care to define what is meant by accuracy. We use a popular measure of accuracy for timestamp synchronization algorithms called *fast sends*. A *fast send* is a send which is closer to its corresponding receive than the minimum network transmission time. To measure the number of fast sends we used two trace files of timestamps generated by running a program on a pair of machines that repeatedly exchanges messages in both directions. Each message exchange consists of a send and a receive and causes two events, one before the send and another after the receive, to be generated. Since the intervals reported by the bounded algorithms are not

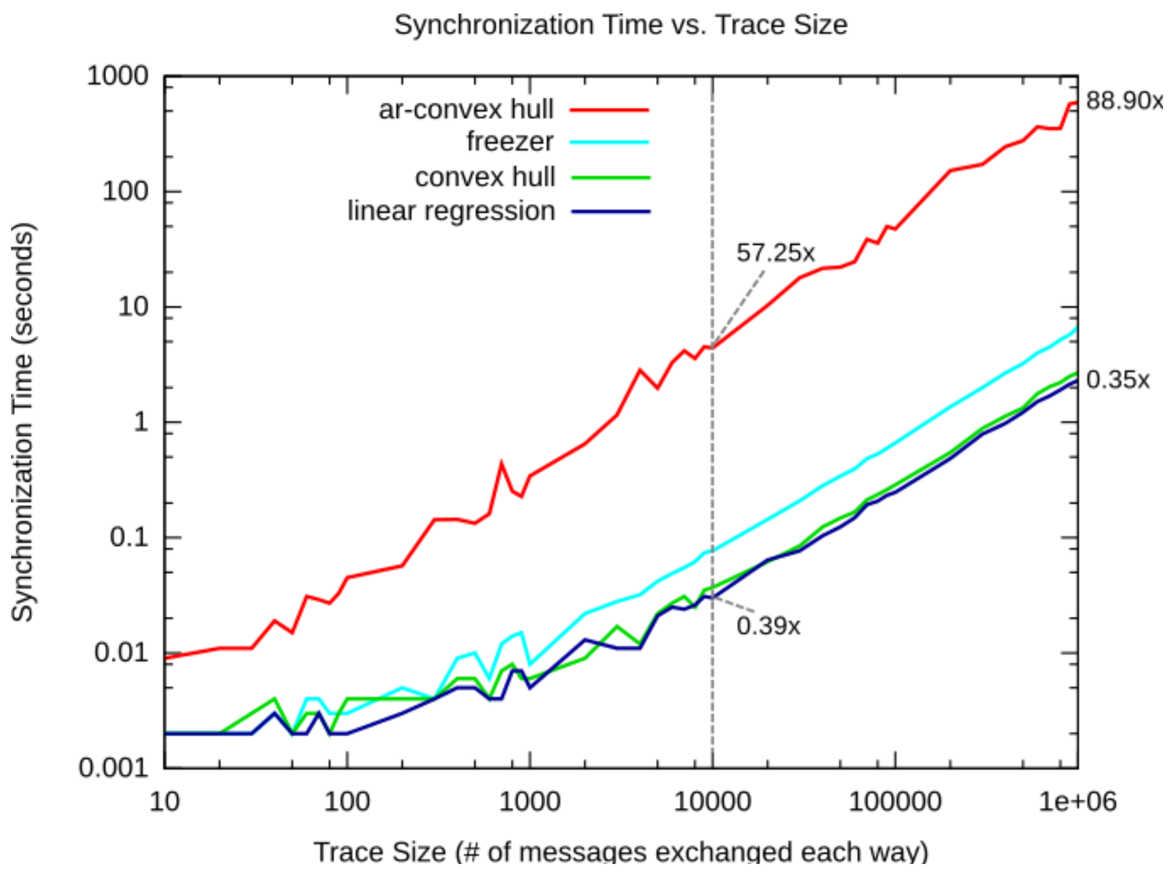


Figure 5.6: Comparison of the synchronization times for different algorithms – note that both axes are on a log-scale; Annotations show speedup relative to FreeZer.

directly comparable to the point estimates of the unbounded algorithms, we use the interval midpoints in the following comparison. The results gathered over 100 trials, presented in Tables 5.1, show that FreeZer performs favorably relative to the other two algorithms. This is due primarily to the fact that FreeZer performs its synchronization offline and independent of the traces being converted.

Efficiency comparison The primary factor affecting the *synchronization time* is the number of events in the traces. Therefore in Figure 5.6 we compare the synchronization times of different algorithms with increasing trace size. Traces were generated in the same way as for accuracy measurements.

First let us consider the efficiency of *linear regression* and *convex hull* algorithms from [16] who efficiency is nearly the same. FreeZer is only slightly slower than the linear regression algorithm – at 10k messages 0.39× and at 1M messages 0.35×. The small performance gap between FreeZer and these algorithms is due to FreeZer's use of arbitrary precision integer arithmetic to ensure that floating point inaccuracy and finite integer precision do not affect the validity of the bounds. While the two algorithms are slightly more efficient than FreeZer, they do not provide error bounds.

The only previous algorithm that reports error bounds is the *accuracy reporting convex hull* (ar-convex hull) algorithm of Poirier et al. [33]. This works similarly to the regular convex hull algorithm above, but it derives bounds on the conversion error for individual timestamps by considering the set of all possible lines in the corridor of the convex hulls. We observe that the ar-convex hull algorithm is the slowest. For instance, with 10k messages exchanged in each direction, the speedup of FreeZer over ar-convex hull is 57.25×. The expense of the ar-convex hull algorithm stems from its formulation. Every bound for each timestamp is found by solving a linear programming problem whose size grows with the number of timestamps being converted. However, FreeZer finds each bound for each timestamp by simply using the appropriately bounded relative frequency and zero. Since the calculation of the relative frequency and zero is independent of the timestamps being converted, it can be done offline.

There are two additional algorithms. The *two-point algorithm* [6] first forms the same set of points as the linear regression algorithm. It then uses this set to construct a set of "equivalences", which are estimates of points that lie on the true conversion line. The conversion function estimate is taken as the line connected the two "best equivalences" as defined by a set of rules intended to minimize error. The *controlled logical clock* [35] first runs one of the above algorithms, and then retroactively adjust timestamps which violate pre-determined causality rules. We chose not to further evaluate these two algorithms for the following reasons. The controlled logical clock algorithm is a wrapper algorithm which first requires a pre-synchronization step; it is not a standalone algorithm. The two-point algorithm requires a minimum number of events to operate, and this threshold is not met by the traces encountered in our later experiments with real applications.

Thus our results show that FreeZer not only exhibits high efficiency, it also provides error bounds.

5.5 Summary

In this chapter, we presented FreeZer, our new algorithm for timestamp synchronization. FreeZer works by calculating relative frequencies and zeros using carefully controlled communications between nodes in the system. The FreeZer algorithm is independent of the timestamps being converted, so it can be done offline, minimizing perturbation of the program under test. FreeZer not only provides error bounds on its conversion, but it does so up to 57x faster than the only other bounded algorithm available. Furthermore, FreeZer beats other existing algorithms in the commonly accepted synthetic measure of fast sends. In the next chapter, we show how FreeZer can be used as the basis for DProf, our implementation of CSPs on distributed systems. We will also show that FreeZer is more accurate than other algorithms when used to make actual predictions about performance improvements resulting from potential optimizations.

Chapter 6

The DProf Distributed Profiler

In the previous chapter, we presented FreeZer, our new algorithm for timestamp synchronization. In this chapter, we show how we use FreeZer as a basis for DProf, a platform for the construction of distributed profilers. First, we present the overall structure of DProf. Then, we present dCSP, our implementation of CSPs for clusters of shared-memory multicore machines. Additionally, we use DProf to construct dCOZ, a distributed version of the popular causal profiling technique for shared-memory multicore machines. Finally, we validate the accuracy of FreeZer by using dCSP and dCOZ in combination to detect performance problems and predict the impact of optimizations on those problems.

6.1 Putting It All Together: DProf and its Uses

Figure 6.1 shows how DProf uses the bounded frequency and zero estimates generated by FreeZer to perform distributed profiling. DProf performs the following steps: first, FreeZer collects bounded frequency and zero estimates based on techniques just discussed.

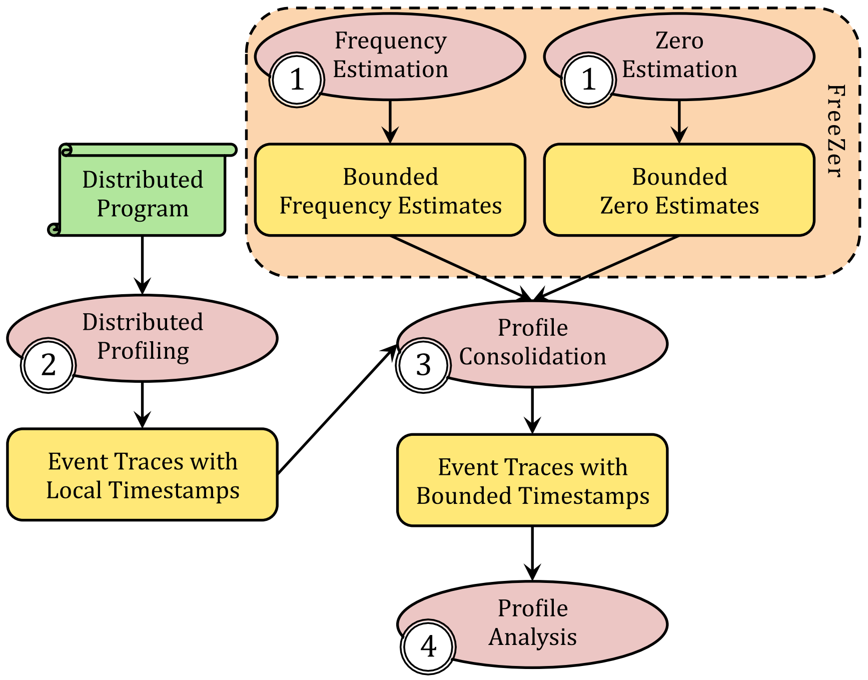


Figure 6.1: DProf workflow.

Then, DProf profiles the distributed execution using local x86 timestamp counters to generate process local profiles. These local profiles are then consolidated using the bounded frequency and zero estimates to generate a unified profile with global event ordering and timing information. The consolidated profile is then queried and analyzed based on the kind of profiling being performed by the end user. Next, we discuss each of the steps in detail.

Step 1: Computing Relative Frequency & Zero Bounds

FreeZer computes relative frequency and zero bounds between pairs of clocks. To compare all the timestamps in the system a base core is chosen, and all the relative frequencies and zeros involving that base core are computed. Then during the profile consolidation phase, the timestamps are all converted to the units of the base core. While causal dependencies across cores may require network operations, it is important to ensure that they don't introduce unpredictable delays that impact the overall accuracy. For this, DProf performs send-receive operations using UDP over unix sockets instead of using a higher level network programming layer like MPI.

Step 2: Capturing Distributed Event Profiles

DProf profiles the distributed execution by capturing events and timing information about *marked code regions* that are of interest for profiling (we support annotations introduced in CSP [8]). DProf provides an API to mark entry and exit points for code regions that need to be profiled. During execution, these entry and exit points translate into events whose timing information must be captured. As mentioned earlier, the timing information is captured using the x86 timestamp counters whose relative frequency and zero bounds are captured by FreeZer.

DProf also enables enriching profiles by dynamically capturing context sensitive information that is necessary for performing different types of analyses. For example, straggler detection (case study presented in next section) needs information about distributed barriers; hence, the entry and exit events for distributed barriers are annotated with information about the barrier, like its unique identifier and number of expected threads. Such context sensitive enrichment of profiles enables our profiler to be generalized across various profile analyses.

Note that the collected distributed profiles have event and timing information that is local to individual processes in the distributed system. Next, we discuss how they are consolidated so that profile information across processes can be collectively used to perform a global analysis.

Step 3: Consolidating Distributed Event Profiles

Consolidation of event profiles mainly includes converting the captured timestamps across all processes into a globally consistent time scale. DProf picks one of the core's clock as a reference and converts all the timestamps from remaining clocks to that in terms of this reference clock. Since FreeZer captures relative frequency and zero information in terms of bounds, the timestamps are converted into interval of timestamps, i.e., the profiling information is transformed from point domain to interval domain.

Consider a timestamp k captured with $core_j$'s clock c_j which needs to be converted in terms of the reference clock c_i of $core_i$. Assuming the zero for $core_j$ with respect to $core_i$ to be $zero_{j \to i} = <(z_j^L, z_j^U), z_i>$, k is converted to $<k^L$, $k^U>$ as:

$$k^L = (ts_1 - z_i) \times f^L_{j \to i}(t) + z_j^L$$

$$k^U = (ts_1 - z_i) \times f^U_{j \to i}(t) + z_j^U$$

Converting all the timestamps in this manner with respect to the same reference core makes them globally consistent and enables distributed profile analysis.

Step 4: Analyzing Profiles

The consolidated profiles represent directly comparable event and timing information along with context sensitive information that is dynamically captured during runtime. Hence, different kinds of analyses can be performed over these profiles to gain useful insights about the program, ranging from simply determining how much time was spent in different regions of the program, to understanding complicated relationships like expected impact of speeding up a code region on the application's performance.

— ***Strong Guarantees.*** Since the timing information in consolidated profiles are in terms of bounded intervals, the profiling strategy can leverage these intervals to provide strongly bounded results. This requires carefully maintaining the interval bounds throughout the analysis by designing interval-aware strategies like interval arithmetic and translation of time intervals into domain-specific intervals based on the kind of profiling that is being performed. For example, in next section we will carefully transform the time intervals to strongly bounded straggler scores and speedup ranges to provide domain-specific guaranteed results.

— ***Eliminating Inverted Orderings.*** With bounded time intervals, ordering of events occurring nearly at the same time can sometimes become difficult if the intervals are overlapping. In such cases, causality ordering across such events can be used to enforce event ordering in the consolidated profiles and hence, eliminate inverted orderings. For instance, a synchronous send-receive operation with end of send occurring at $<t_{sd}^L, t_{sd}^U>$ and end of receive occurring at $<t_{rv}^L, t_{rv}^U>$ has the causality ordering $t_{sd}^L < t_{rv}^L$ and $t_{sd}^U < t_{rv}^U$. This causality ordering can be explicitly incorporated in the profile to eliminate inversion between the end of send and the end of receive events. However, it is important to note that such enhancement of consolidated profiles must be done without adjusting the timing information in bounded intervals in order to maintain strong guarantees.

— ***Extensibility.*** Different analyses can be performed by viewing the overall profiles in different ways. In the remainder of this section, we demonstrate this by developing distributed versions of two recent shared memory performance analysis techniques: *Context Sensitive Profiling* (dCSP) and *Causal Profiling* (dCOZ). These can be effectively used in tandem for performance debugging – dCSP identifies performance bottlenecks and dCOZ estimates how much application performance can be expected to improve if the identified performance bottleneck is removed.

6.2 Context Sensitive Profiling: dCSP

Context Sensitive Profiling (CSP) [8] allows analysis of execution times for different code regions of interest in shared memory parallel programs. The overall execution time is divided into a sequence of time intervals called *frames*, during which no process transitions between code regions. The sequence of frames is then queried to expose different performance insights like bottlenecks at synchronization points, workload imbalance, and many others. Using DProf, we develop CSP to analyze distributed programs by representing our distributed profiles as sequence of frames.

Aggregated Event View Constructing the frame sequence using DProf profiles requires an ordering across events. Since inter-process events that occur nearly at the same time

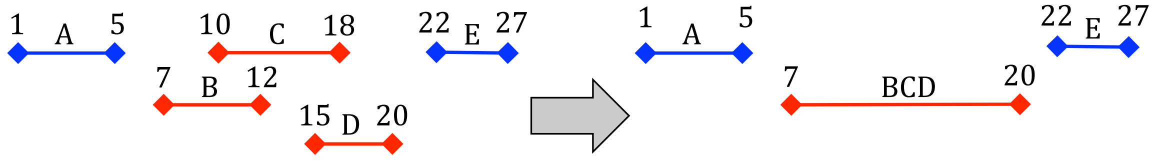


Figure 6.2: Aggregating overlapping events B, C and D into a single aggregated event BCD.

can have overlapping time intervals, ordering events results in a simple DAG structure that captures multiple possible execution orders. Typical context sensitive analysis like straggler detection requires computation for each different execution path; however, enumerating all possible execution paths can become infeasible due to path explosion.

To address the above issue, we develop an *Aggregated Event View* that merges overlapping events into combined events so that the resulting sequence of events have total ordering. Availability of such a total ordering enables computation of straggler scores with a linear pass over the entire distributed profile. Since the aggregated event represents multiple underlying events, its timing interval is computed as the largest interval range spanning across all the underlying events' intervals. The interval lower (upper) bound of the aggregated event is the minimum (maximum) of its underlying events. As illustrated in Figure 6.2, the overlapping events B, C and D are merged together to a single aggregated event BCD that spans from 7 to 20 so that it fully incorporates the three intervals.

6.3 Causal Profiling: dCOZ

Causal profiling [11] is a technique similar to logical zeroing [22] that determines the potential impact of optimizing a selected code region on the overall performance of the program. Such kind of profiling enables users to understand different bottlenecks in

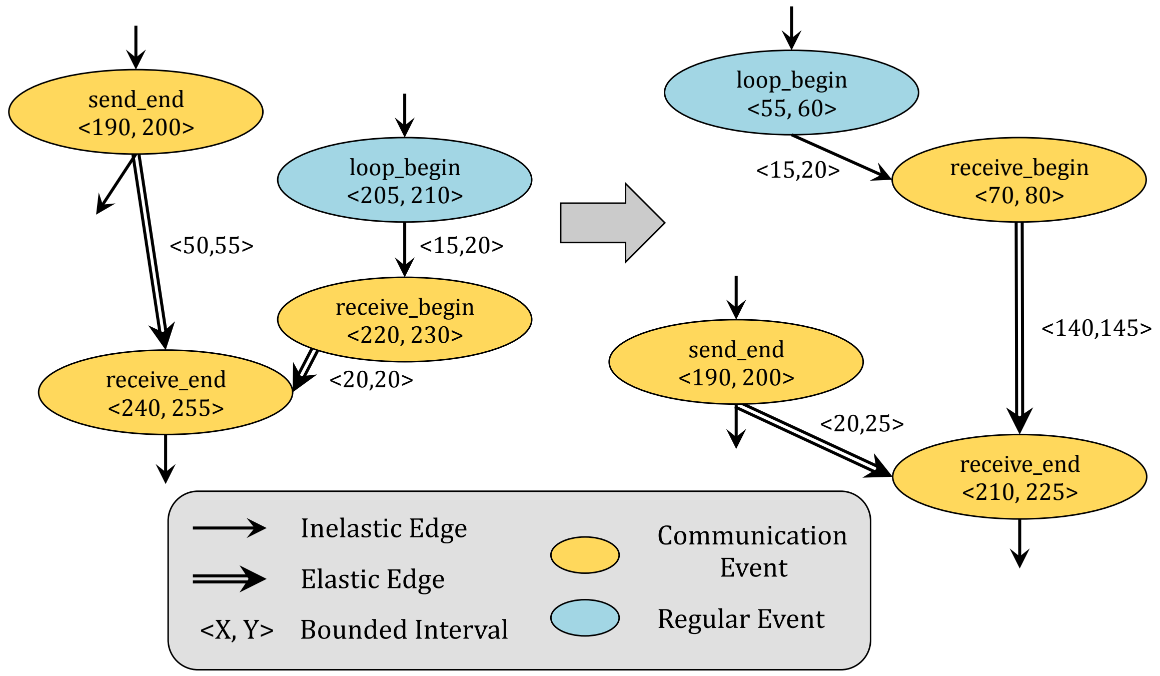


Figure 6.3: Adjusting the time weighted DAG on left to the one on the right causes change in weights of elastic edges.

concurrent programs and to prioritize various optimizations to be incorporated in those programs. While causal profiling systems like COZ [11] enable causal analysis over parallel programs, we develop causal profiling for distributed applications using DProf.

Time Weighted DAG View To perform causal analysis on the captured events, we need to carefully propagate the impact of optimizing a code region to the remaining execution events. Such propagation ensures that the causal dependencies remain consistent while the timing information for events change.

We design a Time Weighted DAG to represent the event and timing information captured by profiling. Vertices in the graph represent captured events while edges connect events that are sequentially executed by the same thread, and inter-thread events that are directly related by causality. The edges are weighted with timestamp intervals that represent the difference between intervals of events represented by source and destination vertices. For COZ analysis, we categorize the edges into two types: *elastic* and *inelastic* edges. An *elastic* edge is one whose weights can be adjusted during COZ analysis, i.e., the duration between their source and destination events is allowed to grow or shrink during impact propagation. We set all the communication edges to be elastic so that it enables us to carefully adjust the timestamps of communication events while simultaneously ensuring that causal dependencies are never violated. Inelastic edges, on the other hand, are those whose weights remain same throughout the analysis. We set all the non-communication based edges as inelastic. Figure 6.3 shows two time weighted DAGs with elastic and inelastic edges. We can transform the left time weighted DAG to the one on the right by adjusting the timing information; note that the inelastic edge weights remain the same while elastic edges grow (e.g., between `receive_begin` and `receive_end`) and shrink (e.g., between `send_end` and `receive_end`) based on the adjustments.

Algorithm 6 shows the overall algorithm to perform causal analysis using time weighted DAG. Given a code region r and the amount of reduction, lines 5-18 process the events belonging to r by directly reducing their execution times. Events that occur after r are collected in `global_list` to be processed in lines 20-29. While processing each event in the `global_list`, the amount of reduction to be performed can be limited by causal dependencies that need to be maintained during reduction. For example, while adjusting an end of receive event, the analysis should ensure that the timestamps do not precede end of send events to avoid inversions (as shown in Figure 6.3). Similarly, any exit event

Algorithm 6 Causal Analysis using Time Weighted DAG.

```
1: $r$: code region whose impact is being analyzed
2: $reduction$: percentage of time reduction for $r$
3: $region\_list$: $\{v | source(edge(v))$ is an entry event for $r\}$
4: $global\_list$: $\varnothing$
5: while $region\_list \neq \varnothing$ do
6:     $v \leftarrow first(region\_list)$
7:     $region\_list \leftarrow region\_list \setminus \{v\}$
8:     if $v$ is not processed then
9:         $cut \leftarrow weight(v) \times (reduction/100)$
10:        $weight(v) \leftarrow weight(v) - cut$
11:        if $v$ is an exit event for $r$ then
12:            $pair \leftarrow <dest(edge(v)), cut>$
13:            $global\_list \leftarrow global\_list \cup \{pair\}$
14:        else
15:            $region\_list \leftarrow region\_list \cup \{dest(edge(v))\}$
16:        end if
17:    end if
18: end while
19:
20: while $global\_list \neq \varnothing$ do
21:    $<v, cut> \leftarrow first(global\_list)$
22:    $global\_list \leftarrow global\_list \setminus \{<v, cut>\}$
23:    $new\_cut \leftarrow adjust(v)$
24:    if $new\_cut > 0$ then
25:        $weight(v) \leftarrow weight(v) - new\_cut$
26:        $pair \leftarrow <dest(edge(v)), new\_cut>$
27:        $global\_list \leftarrow global\_list \cup \{pair\}$
28:    end if
29: end while
30:
31: $new\_time \leftarrow time(program\_exit\_event)$
32: $speedup \leftarrow (original\_time - new\_time)/original\_time \times 100$
```

from a given barrier should not precede the latest entry event to the same barrier. Such dependencies are taken care by `adjust()` (line 23) which analyzes the causal dependencies of the given event and its predecessor events, its elastic edges and its inelastic edges to determine the amount of reduction that can be safely performed. This makes the reduction value dynamic, which gets attached to the future events to be adjusted (lines 26-27). The work-list based reduction algorithm terminates when there is no remaining reduction to be performed on any event.

6.4 Performance Debugging with dCSP & dCOZ

We present how dCOZ and dCSP can be used for performance debugging of distributed programs. In particular, we show how dCSP can be first used to identify a performance bottleneck, and then how dCOZ can be used to estimate the expected performance improvement resulting from removal of the identified bottleneck. We also demonstrate the importance of error bounds by showing that the inferences drawn by dCSP and dCOZ deteriorate significantly if instead of using FreeZer we use either the *linear regression* or *convex hull* algorithms.

To showcase the strength of FreeZer, we choose three popular distributed programs which have varying degrees of performance bottlenecks: Connected Components [45] (CC), PageRank [32], and K-Means [2]. All three programs iterate over data-elements (graph vertices/edges for CC and PageRank, points for K-Means) to compute results; as an example, the overall structure of K-Means is shown in Algorithm 7. We use [37] for K-Means and LiveJournal [27] for PageRank and CC to generate important events for performance

Algorithm 7 Distributed K-Means. Region 0 (blue) is the body region, and region 1 (red) is the barrier region for the dCSP and dCOZ analyses.

```
 1: $objects \leftarrow$ read_objects( )
 2: Send_Receive($objects$)
 3: $clusters \leftarrow$ init_clusters($random\_centroids$)
 4: All_Reduce($clusters$)
 5: **do**
        #Region(0)
 6:     **for** $o \in$ objects **do**
 7:         $c \leftarrow$ compute_closest_cluster($o$, $clusters$)
 8:         $o.cluster \leftarrow c$
 9:         $c.objects \leftarrow c.objects \cup \{o\}$
10:     **end for**
        # Region(0)
        #Region(1)
11:     All_Reduce($clusters$)
        # Region(1)
12:     $new\_centroids \leftarrow$ compute_new_centroids($clusters$)
13:     All_Reduce($new\_centroids$)
14:     $clusters \leftarrow$ init_clusters($new\_centroids$)
15:     $change \leftarrow$ compute_change($objects$)
16:     All_Reduce($change$)
17: **while** $change > threshold$
```

analysis. These three distributed programs reflect modern real-world distributed programs that also extract multicore performance via threading on each machine; in each iteration, all machines execute their share of the workload in parallel using multiple threads and then they synchronize at a barrier, exchange information, and move to the next iteration.

We consider *stragglers* as performance bottlenecks in our distributed programs. Straggler processes arise when all processes but one finish their assigned task and are left waiting at the barrier while the single remaining process slowly finishes its task. Stragglers are a common reason for slowdown in distributed programs since they can be caused due to several reasons including load imbalance, hardware performance imbalance, network delays, etc. We used 5 machines to perform this case study and the presence of a straggler process was ensured via workload imbalance, i.e., one of the processes was assigned significantly more work compared to other processes. On each machine the workload was executed in parallel by 8 threads. The impact of stragglers is highest in CC and lowest in K-Means which allows us to perform sensitivity analysis of Freezer's strong guarantees.

Algorithm 7 shows the code region annotations used to generate events for detecting stragglers. The blue region is where primary work is performed, and the red region is the immediately following barrier synchronization point. These regions allow us to detect stragglers by searching for dCSP frames in which all but one process/thread are in the red region, while there is exactly one thread/process in the blue region. For concrete analysis, we define *straggler score* (SS) of a thread t to be the percentage of time that the thread was in the blue region while all other threads were in the red region.

We perform our analysis in three steps. First, we use dCSP to calculate the straggler

	CC	PageRank	K-Means
FreeZer	47.19 - 47.21 %	37.17 - 37.19 %	18.9 - 19.9 %
Linear Reg.	23.46 %	18.48 %	8.7 %
Convex Hull	11.75 %	9.73 %	6.9 %

Table 6.1: Straggler scores for the straggler process. Higher percentage means more severe straggling. The execution times for the programs were: CC – 1799.173 seconds; PageRank – 844.251 seconds; and K-Means – 13.7 seconds.

scores for each of the threads in the system. Second, we use the straggler scores calculated by dCSP as input to dCOZ, and calculate the expected speedup when the straggler identified by dCSP is sped up by the value of its straggler score. And finally, we eliminate the straggler by removing workload imbalance and measure the *actual* execution time to compute the overall *prediction error* under each of the three timestamp synchronization schemes: FreeZer, Convex Hull, and Linear Regression.

Step 1. dCSP: Straggler Detection. With one process chosen to be the straggler, we used dCSP to measure the straggler score of the chosen process. The results of this step are shown in Table 6.1. Since Freezer provides bounded timestamp information, we calculate bounds for straggler scores by propagating the timestamp bounds throughout the straggler analysis. As we can see, Freezer's bounds for straggler scores are tight, i.e., at most one percent, which is due to our strongly bounded synchronization technique. This shows that FreeZer not only provides tight bounds in theory, but also in practice. It is interesting to see that linear regression (Linear Reg.) also provides positive straggler scores, however they are not even close to FreeZer's bounds. Furthermore, convex hull (Convex Hull) provides even smaller straggler scores, incorrectly diminishing the imbalance issue among threads.

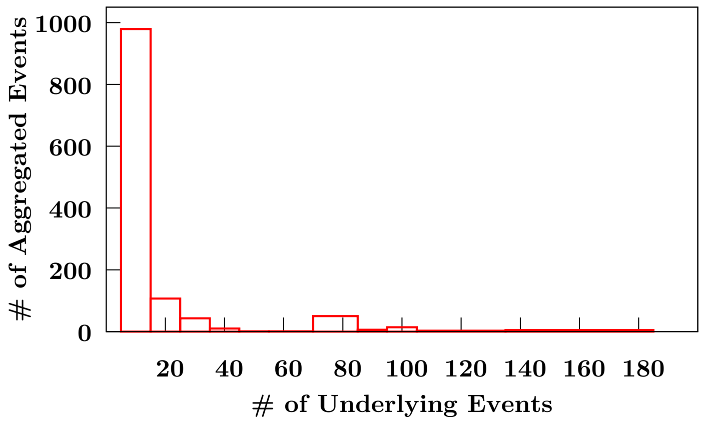


Figure 6.4: Size of aggregated events.

It is interesting to note that the coarse grained view provided by aggregated events does not impact our profiling results such that the straggler threads go undetected. This is because the aggregated view mostly summarizes small number of events, leaving out majority of events from the original profile. Figure 6.4 shows the distribution of aggregated event sizes in terms of number of underlying events. As we can see, the size distribution drops sharply; with 1,219 aggregated events, over 89% have at most 20 underlying events and less than 1% have over 100 underlying events.

Step 2. dCOZ: Causal Analysis. Next we use dCOZ to answer the question: if the straggler had finished on time, what reduction in execution time would be observed? Our dCOZ profiler takes two inputs: the code region that will be sped up to eliminate delay caused by straggler, and the amount of speedup to apply to that region. It estimates the expected overall speedup of the program. We again examine the primary work region (blue region in Algorithm 7), except that instead of using a static region identifier of 0, we use

	CC	PageRank	K-Means
FreeZer	44.37-45.30 %	32.94-33.72 %	18.1-18.4 %
Linear Regression	21.60 %	15.53 %	7.7 %
LR w/ FreeZer SS	23.46 %	18.48 %	8.1 %
Convex Hull	8.44 %	6.79 %	5.3 %
CH w/ FreeZer SS	11.75 %	9.72 %	7.6 %

Table 6.2: Predicted overall speedups via causal profiling given a speedup of the straggler process equal to its straggler score as computed in Step 1. LR w/ Freezer SS and CH w/ Freezer SS indicate Linear Regression and Convex Hull synchronization when using straggler scores calculated by FreeZer.

a dynamic region identifier corresponding to the rank of the process in the MPI global communicator. This allows us to select the right edge in time weighted DAG view whose loop body corresponds to the process that we've elected as the straggler.

The results of dCOZ are shown in Table 6.2. When we calculated the causal profiles using the *linear regression* and *convex hull* synchronizations, we did so using not only the straggler scores corresponding to those methods, but also using the straggler score as calculated with the FreeZer synchronization; this eliminates the errors coming from step 1 and clearly shows errors induced by *linear regression* and *convex hull* in dCOZ alone. As we can again see, FreeZer provides strongly bounded results; in fact, even with smaller execution times for K-Means, FreeZer's bounds are tight. While *Linear Regression* and *Convex Hull* indicate some performance improvement upon removal of stragglers, are again far from FreeZer's results. Even when using FreeZer's straggler scores, *Linear Regression* and *Convex Hull* provide low numbers, again incorrectly estimating that little performance improvement can be achieved.

	CC	PageRank	K-Means
FreeZer	4.28-5.88 %	4.46-5.57 %	1.8-1.8 %
Linear Regression	34.9 %	20.35 %	10.5 %
LR w/ FreeZer SS	31.7 %	16.14 %	10.5 %
Convex Hull	57.54 %	32.8 %	14.0 %
CH w/ FreeZer SS	51.85 %	28.63 %	11.4 %

Table 6.3: Percentage error in prediction.

Step 3. Prediction Error. Next we show the strength of FreeZer over linear regression and convex hull methods by comparing our dCOZ predictions (from Table 6.2) with execution times when the straggler is eliminated, i.e., using balanced workload across all threads. The execution times with/without straggler are: 1799.2s/1045.6s for CC, 844.3/592.6s for PageRank, and 13.7/11.4s for K-Means, and the error in our dCOZ prediction is shown in Table 6.3. The error values for FreeZer is an order of magnitude lower compared to convex hull and linear regression methods; this is because FreeZer captures and retains tight bounds throughout the analysis instead of approximating the estimates, as done by other techniques. Even when using FreeZer's straggler scores for linear regression and convex hull, their error values remain high which indicates the low accuracy of these methods in dCOZ alone.

It is interesting to note that Linear Regression and Convex Hull underestimate straggler scores in all cases; this is because our straggler score is defined based on *all but one* thread being inside the barrier and synchronization inaccuracies offsetting relative timestamps end up causing *multiple* threads to be observed as being outside the barrier.

6.5 Summary

We developed DProf, a profiler for distributed programs that provides results with strong guarantees. DProf relies on bounded frequency and zero measures that tightly capture the inaccuracies in the timing information. Using DProf, we developed two performance analysis techniques, dCSP and dCOZ, to demonstrate its versatility and effectiveness in developing tools for distributed performance debugging.

Chapter 7

Conclusion and Future Work

7.1 Contributions

In this thesis, we presented Context Sensitive Parallel Execution Profiles (CSPs), which are a novel, flexible method for understanding and debugging performance problems in parallel programs. Our contributions can be divided into three key components: interface, shared memory implementation, and distributed memory implementation.

7.1.1 User Interface

The CSP interface consists of three parts: the annotation language, the frame representation, and the query language. The annotation language allows users to identify arbitrary regions of interest in their program. With features like dynamic names, conditional regions, and object properties, runtime behavior can be captured with a wealth of context sensitivity. The frame representation models the execution of the program in terms of the regions defined with the annotation language. A single frame gives a vertical slice of the program execution that shows you what each thread was doing during that slice of time, in terms of the code regions each thread was executing. Thus, frames capture the concurrency context of the program behavior. The CSP itself is a sequence of frames, and due to the simple nature of frames, the overall CSP capturing and construction costs are very low. Finally there is the query language, using which programmers can extract information from the CSP using an intuitive first-order logic inspired language.

7.1.2 Shared Memory Profiling

Owing to its careful design, our implementation of CSPs on shared memory systems is very efficient. The time overhead of capturing CSPs is typically less than 5%, and memory overhead typically less than 25%. We use the commonly available x86 timestamp counter register to capture timestamps with no system call overhead or inter-thread communication/synchronization. Microbenchmarks show that our approach is able to accurately capture fine-grained timing information, and case studies show that CSPs are effective at diagnosing and fixing performance problems in real applications. Among the programs in the Parsec benchmark suite, we were able to realize optimizations of 36% and 17%.

7.1.3 Distributed System Profiling

In addition to implementing CSPs for shared memory multi-core machines, we adapted and implemented them for distributed systems as well. In the course of doing so, we developed a new algorithm (FreeZer) for the timestamp synchronization problem, which is more accurate for the targeted analysis style of CSPs than existing options. Additionally, we exhibited the flexibility of CSPs by building a distributed causal profiler on top of our annotation language. Based upon FreeZer we built the DProf distributed profiler and used it to develop dCOZ and dCSP analysis tools. We showed how dCSP can be first used to identify a performance bottleneck, and then how dCOZ can be used to estimate the expected performance improvement resulting from removal of the identified bottleneck.

7.2 Future Work

While the contributions of this thesis provide a definite step in the right direction, there still remains much to be done in the field of performance analysis for parallel programs. In this section, we discuss two avenues that would be interesting to explore as natural extensions of the work presented herein.

7.2.1 Online Performance Monitoring

In general, there are two strategies that can be adopted when measuring the performance of a program. The first (and what we have chosen to do for this thesis) is to collect relevant information during execution and then perform analysis offline. The advantage of this approach is that it helps to minimize the intrusion on the program under test, which if too large could render the analysis results meaningless. This approach allows the most freedom in designing the profile format and accompanying analyses, because we need not worry about the analysis cost.

The offline approach does have its fair share of downsides however. First, it is wholly inappropriate for programs which do not terminate, such as servers or long-lived programs with multiple phases of processing. Simply put, if the program never terminates then the analysis never happens.

In contrast, an online approach performs the analysis during the execution of the program, turning the analysis program into a monitor. This is very well suited to programs like servers which are not expected to terminate, but which nonetheless can have a great impact on overall system performance.

There are a number of reasons CSPs would be a natural fit for building online performance monitoring tools. First, the algorithm for constructing the profile itself, which is a sequence of frames, can be structured as a multi-way fold over the event streams. This means that each frame can be calculated from only the previous frame and the next event to be generated. In principle, this could form the basis for a monitoring tool with constant memory overhead and amortized constant time overhead. Additionally, the query language examines only one frame at a time, so it is a natural fit for a streaming processing structure as well.

DProf would also form a strong basis for constructing CSP based monitoring tools for distributed systems. Because the FreeZer calculations are completely independent of the timestamps being converted, FreeZer could be adapted for online use with only minimal changes. Since existing timestamp synchronization algorithms all make use of the timestamps being synchronized, they would be incapable of being used in an online setting.

The challenges to solve to make this a reality revolve primarily around the question of intrusion. In the offline setup, there is no additional inter-thread communication overhead. But in the online case, there would have to be some communication of the locally generated events. Determining the best time to do this in order to perturb the program minimally would be an interesting problem to solve. Inspiration could be drawn from research in parallel garbage collection, as that topic shares some of the same problem constraints.

7.2.2 CSPs in a Virtual Environment

When we explored the extensibility of CSPs, we did so by porting them to distributed systems composed of individual shared memory parallel machines. While this is a common and effective technique for building distributed systems, it is becoming popular to use virtualized resources to do the same. Virtualization presents some interesting challenges in terms of profiling. In our work, a major contributor to the overall efficiency is the use of the timestamp counter register on the machine.

If this register is virtualized, it raises questions about its ability to be used as a measure of wall clock time. For instance, when the virtual machine is paused, is the counting of the virtual register paused as well? Does the virtualized register share the same guarantees as the physical register vis a vis monotonicity and constancy of rate? Both of these properties are critical to the accuracy of profiles built on top of timestamps generated using this register.

On the other hand, if this register is not virtualized, there are still other concerns that arise. What if the virtual machine running our program is migrated from one physical machine to another during execution? If we blindly use the timestamp counter register from the new machine, we will experience a jump discontinuity, potentially backwards in time. Moreover, the frequency of the new machine may not be the same as the frequency of the old machine, causing the duration of intervals to be incomparable as well.

Furthermore, there is the question of whether the physical/virtual barrier should be visible in the profile format. Certainly some performance problems could arise as a result of the underlying virtualization. In these cases having this information available would aid the programmer in their debugging task. But for most cases this information is probably not relevant, especially if the programmer is unable to control the virtualization subsystem (because it's being provided by a third party for instance). Handling this will be a trade off amongst the goals we defined in the introduction, and must be carefully considered.

Though this is a challenging direction, the potential impact is significant. In the world of virtualization execution time translates directly to money, so a good profiler would quite literally pay for itself.

Bibliography

[1] Intel 64 and ia-32 architectures software developer's manual, volume 2: Instruction set reference, a-z. http://www.intel.com/content/dam/www/public-/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf. Accessed 22-July-2016.

[2] Parallel k-means data clustering. `http://www.ece.northwestern.edu/~wkliao/Kmeans/index.html`. 2013.

[3] Parallel k-means data clustering. http://www.ece.northwestern.edu/ wkliao/Kmeans/index.html, 2013.

[4] Laksono Adhianto, Sinchan Banerjee, Mike Fagan, Mark Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan R Tallent. Hpctoolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010.

[5] Thomas E Anderson and Edward D Lazowska. *Quartz: A tool for tuning parallel program performance*, volume 18. ACM, 1990.

[6] Paul Ashton. Algorithms for off-line clock synchronization. 1995.

[7] Daniel Becker. *Timestamp Synchronization of Concurrent Events*, volume 4. Forschungszentrum Jülich, 2010.

[8] Zachary Benavides, Rajiv Gupta, and Xiangyu Zhang. Annotation guided collection of context-sensitive parallel execution profiles. In *The 17th International Conference on Runtime Verification*, 2017.

[9] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81. ACM, 2008.

[10] David Böhme, Felix Wolf, Bronis R de Supinski, Martin Schulz, and Markus Geimer. Scalable critical-path based performance analysis. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 1330–1340. IEEE, 2012.

[11] Charlie Curtsinger and Emery D Berger. Coz: finding code that counts with causal profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 184–197. ACM, 2015.

[12] Florian David, Gael Thomas, Julia Lawall, and Gilles Muller. Continuously measuring critical section pressure with the free-lunch profiler. In *ACM SIGPLAN Notices*, volume 49, pages 291–307. ACM, 2014.

[13] Rui Ding, Hucheng Zhou, Jian-Guang Lou, Hongyu Zhang, Qingwei Lin, Qiang Fu, Dongmei Zhang, and Tao Xie. Log2: A cost-aware logging mechanism for performance diagnosis. In *USENIX Annual Technical Conference*, pages 139–150, 2015.

[14] Kristof Du Bois, Stijn Eyerman, Jennifer B Sartor, and Lieven Eeckhout. Criticality stacks: Identifying critical threads in parallel programs using synchronization behavior. *ACM SIGARCH Computer Architecture News*, 41(3):511–522, 2013.

[15] Kristof Du Bois, Jennifer B Sartor, Stijn Eyerman, and Lieven Eeckhout. Bottle graphs: visualizing scalability bottlenecks in multi-threaded applications. In *ACM SIGPLAN Notices*, volume 48, pages 355–372. ACM, 2013.

[16] Andrzej Duda, Gilbert Harrus, Yoram Haddad, and Guy Bernard. Estimating global time in distributed systems. In *ICDCS*, volume 87, pages 299–306, 1987.

[17] Saturnino Garcia, Donghwan Jeon, Christopher M Louie, and Michael Bedford Taylor. Kremlin: rethinking and rebooting gprof for the multicore age. In *ACM SIGPLAN Notices*, volume 46, pages 458–469. ACM, 2011.

[18] Markus Geimer, Felix Wolf, Brian JN Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. The scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, 2010.

[19] Susan L Graham, Peter B Kessler, and Marshall K Mckusick. Gprof: A call graph execution profiler. In *ACM Sigplan Notices*, volume 17, pages 120–126. ACM, 1982.

[20] Jeffrey K Hollingsworth. An online computation of critical path profiling. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, pages 11–20. ACM, 1996.

[21] Jeffrey K Hollingsworth and Barton P Miller. Parallel program performance metrics: a comprison and validation. In *Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, pages 4–13. IEEE Computer Society Press, 1992.

[22] Jeffrey K Hollingsworth and Barton P Miller. Slack: a new performance metric for parallel programs. *University of Maryland and University of Wisconsin-Madison, Tech. Rep*, 1994.

[23] Donghwan Jeon, Saturnino Garcia, Chris Louie, and Michael Bedford Taylor. Kismet: parallel speedup estimates for serial programs. In *ACM SIGPLAN Notices*, volume 46, pages 519–536. ACM, 2011.

[24] Minwen Ji, Edward W Felten, and Kai Li. Performance measurements for multithreaded programs. In *ACM SIGMETRICS Performance Evaluation Review*, volume 26, pages 161–170. ACM, 1998.

[25] Melanie Kambadur, Kui Tang, and Martha A Kim. Parashares: Finding the important basic blocks in multithreaded programs. In *European Conference on Parallel Processing*, pages 75–86. Springer, 2014.

[26] Jure Leskovec and Andrej Krevl. {SNAP Datasets}:{Stanford} large network dataset collection. 2015.

[27] Jure Leskovec, Kevin J Lang, Anirban Dasgupta, and Michael W Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009.

[28] Xu Liu and John Mellor-Crummey. Pinpointing data locality problems using data-centric analysis. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 171–180. IEEE Computer Society, 2011.

[29] Barton P Miller, Morgan Clark, Jeff Hollingsworth, Steven Kierstead, S-S Lim, and Timothy Torzewski. Ips-2: The second generation of a parallel program measurement system. *IEEE Transactions on parallel and distributed systems*, 1(2):206–217, 1990.

[30] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 456–471. ACM, 2013.

[31] Yoshihiro Oyama, Kenjiro Taura, and Akinori Yonezawa. Online computation of critical paths for multithreaded languages. In *International Parallel and Distributed Processing Symposium*, pages 301–313. Springer, 2000.

[32] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[33] Benjamin Poirier, Robert Roy, and Michel Dagenais. Accurate offline synchronization of distributed traces using kernel-level events. *ACM SIGOPS Operating Systems Review*, 44(3):75–87, 2010.

[34] Kishore Kumar Pusukuri, Rajiv Gupta, and Laxmi N Bhuyan. Thread reinforcer: Dynamically determining number of threads via os level monitoring. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pages 116–125. IEEE, 2011.

[35] Rolf Rabenseifner. The controlled logical clock-a global time for trace based software monitoring of parallel applications in workstation clusters. In *PDP*, pages 477–484, 1997.

[36] Sameer Shende, Allen D Malony, Janice Cuny, Peter Beckman, Steve Karmesin, and Kathleen Lindlan. Portable profiling and tracing for parallel, scientific applications using c++. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, pages 134–145. ACM, 1998.

[37] Allan Stisen, Henrik Blunck, Sourav Bhattacharya, Thor Siiger Prentow, Mikkel Baun Kjærgaard, Anind Dey, Tobias Sonne, and Mads Møller Jensen. Smart devices are different: Assessing and mitigatingmobile sensing heterogeneities for activity recognition. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, pages 127–140. ACM, 2015.

[38] Nathan R Tallent, John M Mellor-Crummey, and Allan Porterfield. Analyzing lock contention in multithreaded applications. In *ACM Sigplan Notices*, volume 45, pages 269–280. ACM, 2010.

[39] Keval Vora, Rajiv Gupta, and Guoqing Xu. Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations. *ACM SIGOPS Operating Systems Review*, 51(2):237–251, 2017.

[40] Keval Vora, Sai Charan Koduru, and Rajiv Gupta. Aspire: exploiting asynchronous parallelism in iterative algorithms using a relaxed consistency based dsm. In *ACM SIGPLAN Notices*, volume 49, pages 861–878. ACM, 2014.

[41] Keval Vora, Chen Tian, Rajiv Gupta, and Ziang Hu. Coral: Confined recovery in distributed asynchronous graph processing. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 223–236. ACM, 2017.

[42] C-Q Yang and Barton P Miller. Critical path analysis for the execution of parallel and distributed programs. In *Distributed Computing Systems, 1988., 8th International Conference on*, pages 366–373. IEEE, 1988.

[43] Xiao Yu, Shi Han, Dongmei Zhang, and Tao Xie. Comprehending performance from real-world execution traces: A device-driver case. In *ACM SIGPLAN Notices*, volume 49, pages 193–206. ACM, 2014.

[44] Xiang Yuan, Chenggang Wu, Zhenjiang Wang, Jianjun Li, Pen-Chung Yew, Jeff Huang, Xiaobing Feng, Yanyan Lan, Yunji Chen, and Yong Guan. Recbulc: reproducing concurrency bugs using local clocks. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 824–834. IEEE Press, 2015.

[45] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A computation-centric distributed graph processing system. In *OSDI*, pages 301–316, 2016.