

Statement of Originality

I certify that the content of this thesis represents my own work, except where otherwise acknowledged.

Reena Nair

Copyright

The copyright of this thesis rests with the author. Unless otherwise indicated, its contents are licensed under a Creative Commons Attribution-Non Commercial 4.0 International Licence (CC BY-NC).

Under this licence, you may copy and redistribute the material in any medium or format. You may also create and distribute modified versions of the work. This is on the condition that: you credit the author and do not use it, or any derivative works, for a commercial purpose.

When reusing or sharing this work, ensure you make the licence terms clear to others by naming the licence and linking to the licence text. Where a work has been adapted, you should indicate that the work has been changed and describe those changes.

Please seek permission from the copyright holder for uses of this work that are not included in this licence or permitted under UK Copyright Law.

Imperial College London
Department of Computing

Automatic Bottleneck Detection and Scaling in Virtual Time

Reena Nair

Supervised by: Dr. Tony Field

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London, September
2020

Abstract

This thesis addresses two aspects of parallel software performance engineering. The first concerns the automatic detection of bottlenecks in parallel Linux applications. We present GAPP, a profiler which identifies serialization bottlenecks by tracing context-switch events in the OS kernel. GAPP is extremely lightweight, requires no program instrumentation and works for a variety of serialization bottlenecks. GAPP has been evaluated using the Parsec3.0 benchmark suite and two large open-source projects: MySQL and Nektar++. The experiments show that GAPP is able to detect a wide range of performance issues, for example arising from synchronization primitives, busy-wait loops, memory operations, thread imbalance and resource contention. It is also able to pinpoint the exact lines of code in the application that correspond to the bottleneck.

The second relates to bottleneck optimization. What performance gain can be expected if the bottleneck code, or any hardware resource associated with that code, were to be optimised? To support this type of ‘what-if’ experimentation, we present VIRBS, which simulates the effect of optimizing a particular section of code, by pausing other threads, whenever that code is executed – a form of virtual-time scaling, where real-time and virtual-time progress at different rates. The idea is similar to that used in the Coz framework [CB15]. Whilst Coz is sample-based, VIRBS works by statically instrumenting scaled functions to insert delays precisely whenever the bottleneck code is executed, and this can substantially reduce the time taken to conduct a ‘what-if’ experiment.

In developing VIRBS, the objective has been to build and evaluate a prototype implementation that exhibits Coz-like capabilities, but with reduced overall cost. Experiments with VIRBS has exposed two weaknesses in Coz’s present implementation, one of which relates to inter-thread synchronization, and which is shared by the prototype implementation of VIRBS. These are described in detail and some potential solutions presented as part of future-work.

Contents

List of Tables	7
List of Figures	8
1 Introduction	14
1.1 Motivation	14
1.2 Contributions	16
1.3 Publications	17
2 Background Study and Related Work	18
2.1 Tracing Application Behaviour	19
2.1.1 Source code instrumentation	19
2.1.2 Binary instrumentation	19
2.1.3 Analysing call graphs / stack traces	21
2.1.4 Sampling	22
2.1.5 Tracing kernel events	22
2.2 Related Work - Bottleneck Detection	24

2.3	Related Work - Performance Prediction	34
2.3.1	Performance models	35
2.3.2	Execution driven approaches that simulate the passage of time	35
2.3.3	Graphical representation of execution	38
3	GAPP - A Generic Automatic Parallel Profiler	40
3.1	Thread Criticality - A Metric for Ranking Serialization Bottlenecks	44
3.2	Calculating <i>CMetric</i>	46
3.3	A Software Solution for Thread Criticality	49
3.3.1	Monitoring active threads	49
3.3.2	<i>CMetric</i> calculation of individual time-slices	50
3.4	Bottleneck Detection from Thread Criticality	51
3.4.1	Locating lines of code that contribute to the bottleneck	54
3.4.2	User-space probe	54
3.5	Using GAPP	58
4	GAPP - Evaluation	60
4.1	Experimental Setup	61
4.2	Multithreaded Applications	61
4.2.1	Parsec-3.0 benchmark suite	61
4.2.2	MySQL	77
4.3	Parallel MPI Applications	79

4.4	Sensitivity to Parameters	82
4.5	Customizing GAPP	85
4.5.1	Shared library call paths	85
4.5.2	Choosing between processes and threads	85
4.6	Limitations	86
4.6.1	Spin locks	86
5	VIRBS: VIRtual-time Bottleneck Scaling	88
5.1	Coz - Finding Code That Counts	90
5.2	Time Dilation and Virtual Clocks	91
5.3	Performance Prediction using Virtual-time Execution	94
5.3.1	Monitoring function duration	94
5.3.2	Notifying scaling events	95
5.3.3	Updating virtual execution state	97
5.3.4	Worked examples	97
5.3.5	Implementation details - VIRBS	102
5.4	Evaluating Virtual-time Execution	104
5.4.1	Threads with synchronization primitives	109
6	Conclusion and Future Work	116
6.1	GAPP	116
6.1.1	Discussion and Future Work	117

6.2 VIRBS 119

6.2.1 Discussion and Future Work 119

6.3 Summary 121

Bibliography **122**

List of Tables

3.1	eBPF Maps for calculating <i>CMetric</i>	49
3.2	Command line options of GAPP	59
4.1	Critical functions identified with overhead (O/H), execution time in seconds (T), Number and percentage of critical time-slices (CTS), memory usage in MB (M) and post-processing time in seconds (PPT).	62

List of Figures

2.1	Example of static binary instrumentation - instructions at the function entry point are replaced by a call to the instrumented code	20
2.2	Overview of how uprobes/kprobes work	23
2.3	User annotation to label potential bottlenecks [JSMP12]	25
2.4	Criticality stack and total program speedups from accelerating the identified critical and non-critical threads [DESE13]	29
2.5	Example BottleGraph [DBSEE13]	30
3.1	Example code fragment	41
3.2	Queuing network for Figure 3.1	42
3.3	GAPP profile for example code	43
3.4	Switching intervals and their <i>CMetric</i>	45
3.5	The BCC - eBPF framework	48
3.6	Post-processing and sample profile	56
4.1	Critical path - <i>Blackscholes</i>	64
4.2	<i>Blackscholes</i> - <i>CMetric</i> (64 Threads)	64

4.3	Kernel call paths - <i>Blackscholes</i> (64 threads)	65
4.4	<i>Blackscholes</i> - <i>CMetric</i> (32 Threads)	65
4.5	Top critical path - <i>Bodytrack</i> (Profile including the parent thread)	67
4.6	Top critical function - <i>Bodytrack</i> (Profiling worker threads alone)	67
4.7	General logic of <i>Bodytrack</i>	68
4.8	GAPP profile - <i>Dedup</i> with thread allocation- 1-20-20-20-1	70
4.9	Pipeline stages - <i>Ferret</i>	71
4.10	Top critical path, functions and lines of code - <i>Ferret</i>	72
4.11	CMetric for different thread allocations - <i>Ferret</i>	73
4.12	GAPP Profile - <i>Streamcluster</i> with 64 threads	74
4.13	Top kernel stack traces for critical path1 - <i>Streamcluster</i> with 64 threads	76
4.14	Critical paths - <i>MySQL 5.7</i>	78
4.15	CMetric of individual processes before and after combining partitions - <i>Nektar++</i>	79
4.16	Critical functions - <i>Nektar++</i> with MPICH	80
4.17	Bottleneck functions in <i>Nektar++</i> before and after optimization	81
4.18	Cmetric of individual processes before and after optimization - <i>Nektar++</i>	81
4.19	<i>Perf</i> profile - Unoptimized version of <i>Nektar++</i>	82
4.20	<i>Perf</i> profile - Optimized version of <i>Nektar++</i>	82
5.1	Change in resource access upon scaling [BF11]	89
5.2	Coz's result for <i>Bodytrack</i>	91
5.3	Virtual-time execution - an example	92

5.4	Annotated virtual-time execution	98
5.5	Virtual-time execution with concurrent scaling	100
5.6	<i>Swaptions</i> - logic	105
5.7	Scaling injected function - <i>Swaptions</i>	106
5.8	Scaling HJM_SimPath_Forward_Blocking() - <i>Swaptions</i>	107
5.9	Threads with synchronization primitives - Sample program	108
5.10	Scaling f1() in Figure 5.9	110
5.11	Scaling f1() - actual vs prediction	111
5.12	Coz results - scaling f1()	111
5.13	Scaling f2() of Figure-5.9	112
5.14	Scaling f2() - actual vs prediction	113
5.15	Coz results - scaling f2()	113
5.16	Scaling f2() - modified logic	114

Acknowledgements

I would like to thank the following people, without whom I would not have been able to complete this research

First, I would like to express my sincere gratitude to the Commonwealth Scholarship Commission, UK, for choosing me to be part of this incredible network of innovators and leaders. Further, I would like to thank my supervisor Tony, whose insight and knowledge into the subject steered me through this research. Your enthusiasm, support, encouragement, and patience helped me keep the spark alive throughout the project. Special thanks to Amani, Jeremy and Chris who were always willing to assist in all ways they could during this project. Above all, my biggest thanks to my husband, Hiran, who was the sole inspiration to apply for PhD, I would not have reached here without you, and to our parents for always standing by us.

Dedication

To my husband Hiran, and my little boys Govind and Nandan, without whom this thesis would have completed a bit earlier ☺. Thanks for keeping this journey sparkling and live.

'And, when you want something, all the universe conspires in helping you to achieve it.'

Paulo Coelho, The Alchemist

Chapter 1

Introduction

Modern applications demand increased processing power to achieve high performance. However, the maximum clock speed achievable on modern CPUs has been limited due to the increase in leakage current, as transistors have shrunk in size. This has shifted the focus from building high speed, single core processors to multi-core processors and many existing applications are now being adapted, and new ones are being developed, to exploit this multi-core parallelism. Together with architectural improvements, several parallel programming frameworks have been developed to aid this shift. This in turn has led to many performance issues specifically arising from the execution, and coordination, of multiple tasks executing in parallel.

1.1 Motivation

Traditional profilers, which report execution hot-spots, are useful in analysing some aspects of parallel application performance, but can miss some important performance issues, especially when tasks contend for shared resources. Even if contention happens for a short time, they can serialize the execution, thereby slowing down the application. However, such serializing bottleneck functions will be ranked low by traditional profilers, as long as their contribution to the total execution time remains low. To address this, several performance analysis tools have been developed specifically aimed at detecting performance *bottlenecks*, which in the context

of parallel application code, refers to fragments of code that impede the progress of individual tasks, for example, due to resource contention. However, existing tools/approaches exhibit at least one of the following limitations:

1. Their domain is limited to applications that use a specific threading library.
2. They identify the critical thread in the application, without providing information about the bottleneck code.
3. They require capturing and post-processing large amount of application trace data.

Having identified one or more bottlenecks in a parallel application, the next question is whether it is worth trying to fix them. Because bottlenecks are associated with resource contention, it is well known (e.g. from queuing theory) that small improvements in the performance of the bottleneck code can bring substantial improvement in overall performance. On the flip side, improvements to one bottleneck can yield very modest overall improvement should the bottleneck simply switch from one source to another. The question is: how do we know whether it is worth trying to fix a bottleneck without actually implementing a fix?

An abstract performance model, e.g. a queuing network is one way of investigating the likely payoffs. However, most developers are not well versed in performance modelling etc. This has prompted the development of new tools designed to investigate potential performance benefits, but without having to build and solve a complex model or to re-architect or optimise the code, which may be time consuming and expensive. Two examples are VEX [BF11] and Coz [CB15]; both use execution-driven approaches to explore the effect of hypothetical changes to the program, when executed on a specific hardware platform. VEX estimates performance benefits of scaling individual methods on single-core machines, by controlling thread execution using a user-level wrapper scheduler. Coz estimates performance benefits of scaling individual lines of code by using the concept of time-dilation. However, since Coz follows a sample based approach, experiments need to be repeated many times to achieve an acceptable level of prediction variance.

In this thesis, we explore ways to overcome the limitations exhibited by existing parallel profilers and prediction frameworks such as VEX and Coz.

1.2 Contributions

This thesis is split into two parts. The first part investigates how operating system kernel events can be harnessed to identify different types of bottlenecks in parallel applications, and in doing so, how to balance the trade off between the overhead in gathering and processing large amount of trace data and providing sufficient information about the bottleneck code to the user. The second part explores combining the concept of time-dilation and virtual clocks to virtually scale bottleneck functions by altering the thread schedule, and estimating its effect on application performance. The key challenge is in simulating the correct thread schedule when the bottleneck functions are actually scaled, keeping the overall experiment cost low, while producing accurate results.

These investigations have resulted in two frameworks aimed at pinpointing bottlenecks and then subsequently evaluating the likely effect of reducing the demand on the bottleneck resource, through virtual-time scaling. The key idea is to evaluate the potential benefits of fixing a bottleneck without having to modify the code. Specifically, the thesis contributions are the following:

- We present GAPP, a Generic Automatic Parallel Profiler, which automatically identifies arbitrary serializing bottlenecks in parallel Linux applications based on a weighted *Criticality Metric*, or *CMetric* (Chapter 3). The tool requires no binary instrumentation, no user annotations and executes with extremely low overhead. It works on the principle of reduced parallelism by tracing kernel events, and hence can identify a wide range of bottleneck-related performance issues, for example arising from synchronization primitives, busy-wait loops, memory operations, thread imbalance and resource contention. GAPP has been implemented using eBPF, a framework for fast and secure kernel tracing.

- We evaluate the effectiveness of GAPP in identifying bottlenecks using benchmarks taken from the Parsec 3.0 benchmark suite, and two large open source projects, MySQL and Nektar++ (Chapter 4). These experiments show that GAPP is able to reproduce results from previous work and also identify new, unreported, bottlenecks. Performance experiments show that the overhead on program execution time is extremely small (average circa 4% and maximum circa 13% for the applications considered) and that it is effective in identifying different types of serialization bottlenecks.
- We present VIRBS, a tool which emulates the effect of scaling a previously identified bottleneck function using the concept of time dilation and virtual clocks (Chapter 5). VIRBS is a prototype implementation that is capable of reproducing the same behaviour as the Coz framework [CB15], but without the need to perform the often very many replications required to reduce variance. VIRBS has been tested on data parallel multithreaded applications without inter-thread synchronization and was able to correctly predict performance improvement for an injected function and the dominant function in the *Swaptions* benchmark from the Parsec benchmark suite. Experiments had to be conducted only once for every scaling factor chosen, and the results remained consistent across multiple repeated experiments. Experiments with VIRBS have exposed limitations with the general idea of time-dilation, as adopted by Coz (Section 5.3.4 and Section 5.4.1). We discuss some possible solutions to these in Section 6.2.1, although their implementation goes beyond the scope of this thesis.

1.3 Publications

The work on GAPP has led to the following publication.

- Nair, R., & Field, T. (2020, April). GAPP: A Fast Profiler for Detecting Serialization Bottlenecks in Parallel Linux Applications. In Proceedings of the ACM/SPEC International Conference on Performance Engineering (pp. 257-264).

Chapter 2

Background Study and Related Work

Processor architecture has evolved from traditional single-core designs to many-core, in order to harness the power of parallelism. However, computation on many-core machines instigated performance issues due to contention for shared resources; hardware and software. Computational cost no more remained the only criterion in determining application bottlenecks. The need for better CPU utilization in the presence of shared resources became important. Hence, traditional sample based profilers such as *GProf* became less handy in identifying performance issues/optimization opportunities in parallel programs. While identifying and mitigating software resource contention due to locks and similar synchronisation mechanisms has been emphasized, hardware resource contention due to memory and I/O access patterns also pointed towards possible performance improvement opportunities. Nevertheless, detecting the root cause of performance anomalies/bottlenecks required the execution to be traced while recording events of interest such as synchronisation primitives and I/O access, for further investigation [Thi06].

Extensive research has been conducted in devising techniques for bottleneck identification and evaluating their impact on multithreaded applications. This chapter attempts to discuss and analyse these strategies, comparing one to another. Before discussing existing solutions, we outline the major techniques widely used by profilers/performance measurement tools to record program behaviour.

2.1 Tracing Application Behaviour

Several techniques exist for tracing application behaviour. They primarily differ based on whether the application needs to be modified/re-compiled and whether it works in the user/kernel space. These characteristics have wide effect on the overhead induced, as well as the granularity and the amount of data retrieved. The following section discusses the key features of different tracing mechanisms used by existing performance analysis tools.

2.1.1 Source code instrumentation

Source code instrumentation involves injecting user code into the application source. For example, code can be inserted at start and end of a function to record its execution time. Such approaches require access to the entire source code and needs the application to be re-compiled to gather data. This limits the scope of this methodology, often restricting it to applications written in a particular language or using a particular library. Likewise, it is not possible to analyse behaviour of shared libraries, whose source code is not available. However, source code instrumentation allows the application to communicate language/domain specific details to the analysis tool, making it easier to map performance data to the program execution state. Since applications are instrumented statically, the overhead induced is proportional to the execution time of the injected code.

2.1.2 Binary instrumentation

Binary instrumentation achieves the same effect as static instrumentation by inserting additional code into the binary of an application to observe or alter its behaviour [LTCS10]. There are two approaches to instrument the binary; statically before execution or dynamically during execution of the application.

a. Unmodified Function Entry Point	b. Instrumented Function Entry Point
-----	-----
0000000000400c83 <main>:	0000000000400c4f <main>:
push %rbp	jmpq 700dd3 <threadFun_dyninst+0x342>
mov %rsp,%rbp	sub \$0x60,%esp
sub \$0x60,%rsp	mov %edi,-0x54(%rbp)
mov %edi,-0x54(%rbp)	mov %rsi,-0x60(%rbp)

Figure 2.1: Example of static binary instrumentation - instructions at the function entry point are replaced by a call to the instrumented code

Static binary instrumentation

Static binary instrumentation inserts extra code at the assembly level, modifying the executable permanently. This adds several restrictions, one of which is that the instrumentation persists throughout the execution of the application. Another limitation is that static binary instrumentation requires shared libraries to be instrumented separately (if to be included in the analysis scope), and the application should be re-configured to use the modified library. Static binary instrumentation can be performed by tools such as *PEBIL*, *DynInst* etc., or by a compiler pass [SDD⁺17]. Figure 2.1 shows an example of static binary instrumentation using *DynInst*, where the first two instructions in the function entry point are replaced by a call to an injected function. As with static source code instrumentation, the overhead due to instrumentation is limited to the execution time of the injected code.

Dynamic binary instrumentation

Dynamic Binary Instrumentation (DBI) tools perform instrumentation at run time, which avoids re-compilation or re-linking of the application under test [BH00]. *PIN* [LCM⁺05], *Valgrind* [NS07], *DynamoRIO* [10] and *DynInst* [BM11] are some examples of DBI frameworks. They are widely used to build tools that observe system events such as memory references, instruction execution, control flow etc., and higher level abstractions such as procedure invocation, thread creation, system call invocation [GS15] etc., to name a few.

PIN and *Valgrind* uses dynamic binary recompilation to inject code into the applications ad-

dress space. In dynamic binary recompilation, tools attach itself to the client process and recompile each instruction. During the recompilation process, instructions for interest (for example, all memory write operations) are instrumented and written back in the target instruction set format. *DynInst* and *PEBIL* instruments the program using the concept of *patch based instrumentation*, where points of interest in the original code are overwritten with branches to the instrumented code. *DynamoRIO* performs *DBI* by forcing the dynamic loader to load a shared library into the application's address space. The shared library re-defines the routines that need to be traced.

DBI permits addition and removal of instrumentation code during program execution. However, *DBI* incurs extra overhead in reverse-engineering the binary to insert the instrumentation code, and this increases the execution time of the application [KBH⁺14] [GSdS⁺11]. This makes static binary instrumentation more efficient in terms of instrumentation overhead [LTCS10]. Still, *DBI* has been widely used to build profiling tools especially to trace and profile memory and cache usages. One of the challenges faced by *DBI* frameworks is that instructions, semantics and calling conventions can vary across different architectures and hence tools built using these frameworks will have to address these issues.

2.1.3 Analysing call graphs / stack traces

Several performance analysis strategies have been developed to analyse the behaviour of the application and identify performance issues from call graphs [ACGS04], stack traces [HDG⁺12] and core dumps [AAF⁺10]. They are particularly suitable for post-production systems where modification of the binary or recompilation of the application is not desirable. Industrial applications generate huge amount of data as execution logs, and hence it is not easy for individuals to dig into this information and arrive at conclusions. In such cases where the data is often unstructured, techniques such as data mining, expert systems and graph algorithms have been used to analyse and correlate them.

2.1.4 Sampling

Sampling profilers such as *QProf*, *Oprofile* and *Gprof* samples and records program state at regular intervals without modifying the application. Such profilers generate execution profile by correlating recorded samples to source code using debug information linked to the application's executable. Even though sampling has reduced overhead, the precision in capturing the runtime behaviour highly depends on the sampling frequency, as events between samples can be missed [GSdS⁺11].

2.1.5 Tracing kernel events

Tracing involves collecting execution data which can be later used for profiling and debugging. The Linux kernel provides several tracing mechanisms which can be used to trace and probe a live kernel without recompiling it. Kernel events thus collected can be used to analyse program behaviour. This section briefly explains existing kernel tracing mechanisms in Linux.

Kprobes and kretprobes

Kprobes can be used to dynamically trap into any kernel instruction and collect debugging and performance information at that instant. *Kprobes* are typically implemented as kernel modules, whose *init* function registers them and *exit* function removes them. The *register_kprobe()* function specifies the address to which the probe is to be attached and the handler to be invoked when the probe is hit.

Registering a *kprobe* replaces the first byte of the probed instruction with a *breakpoint* instruction (e.g., *int3* on i386 and x86_64). When a CPU hits the *breakpoint* instruction, a trap occurs, the CPU's registers are saved, and control passes to the probe handler (Figure 2.2). Normally *kprobes* are attached to entry points of kernel functions. *Kretprobes* are triggered when kernel functions return, and allows access to function return values. These probes can provide access to a wide variety of information related to the program being executed, such as the user/kernel

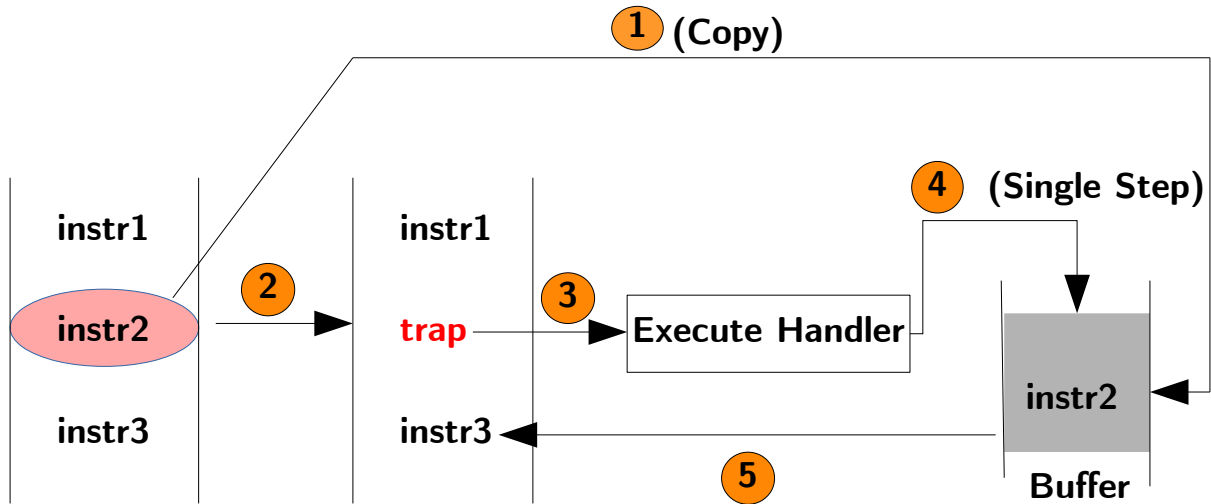


Figure 2.2: Overview of how uprobes/kprobes work

stack, the *task struct*, hardware performance counters etc.

Uprobes and uretprobes

Uprobes are the equivalent of *kprobes*, but are used to trace user-space code. User-provided handler functions are executed in kernel mode in the context of the process being traced. Much like *kretprobes*, *uretprobes* are fired when user-space functions return [CF19, Chapter 4].

Tracepoints

Tracepoints are static markers in the kernel source code, that act as hooks to which a user defined probe function can be attached [Des]. Just as *kprobes*, they record data at specific locations in the kernel which can be used for later analysis. *Tracepoints* are compiled into the kernel and enabled when required. Hence unless activated, tracepoints induce only minimum overhead.

Since *tracepoints* are statically defined, their binary interface remains unchanged across different kernel versions. However, when using *kprobes* to attach to kernel functions, the function signature should be known to the probe function. Hence, a *kprobe* defined for a particular

kernel version may not work for a different kernel version. *kprobes* and *uprobes* modify the assembly code at run time and hence incurs greater overhead when compared to *tracepoints*. It may be noted that *tracepoints* are limited in number and scope, and hence, need not be defined for a particular event of interest. An example for a *tracepoint* is the `sched_switch` *tracepoint*, `trace_sched_switch(preempt, prev, next)`, defined in `kernel/sched/core.c`. User-defined functions can be attached to this *tracepoint* to collect information regarding the tasks involved in a context-switch.

User statically defined tracepoints (USDT)

Similar to *tracepoints*, *USDT* provides static tracepoints (added before compilation) for applications in the user-space. They allow tracing user space code with low-overhead, and are suitable for applications running in production environment [Fle18]. Several popular frameworks such as *Node.js*, *Java (HotSpot VM and OpenJDK)*, *PostgreSQL* etc. have embedded *USDTs*, which can be enabled during configuration.

Rather than writing kernel modules to attach on traceable hooks in the kernel, several frameworks can be used to attach handler functions to *kprobes*, *uprobes*, *tracepoints* etc. *Dtrace*, *Perf*, *SystemTap*, *LTTng* and *eBPF* are examples of such frameworks available in Linux.

2.2 Related Work - Bottleneck Detection

Computational bottlenecks in data parallel applications can be identified by generic profilers such as *GProf*. However, bottlenecks can also be caused due to contention for resources, hardware or software, and they may not be identified by generic software profilers. This section summarizes the different approaches proposed in literature for identifying bottlenecks triggered by resource contention in parallel Linux applications. The focus is on the method used to collect trace data and the type of bottlenecks identified by each approach.

The existing approaches discussed in this section follow one or more of the program tracing

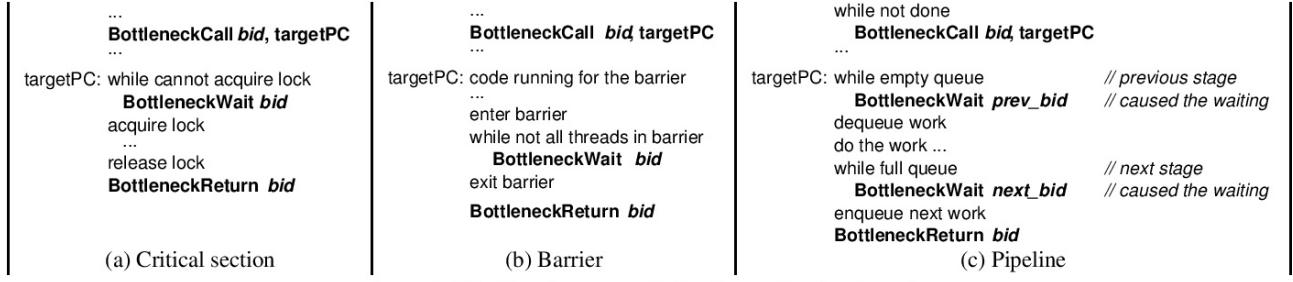


Figure 4. Modifications to code implementing bottlenecks.

Figure 2.3: User annotation to label potential bottlenecks [JSMP12]

strategies discussed in Section 2.1. The strategies are broadly categorized based on the type of bottlenecks identified by each. Section 2.2-a analyses approaches that focus on bottlenecks introduced by locks and threading library primitives. As the underlying hardware architecture becomes complex, bottlenecks can also arise due to contention for hardware resources such as memory and I/O. Such bottlenecks can be architecture dependent, or even compiler dependent. Section 2.2-b discusses approaches that leverage thread execution pattern rather than resource access mechanisms to identify bottlenecks.

a. Synchronization bottlenecks

Access to shared software resources in a multithreaded application is controlled by synchronisation mechanisms. This section analyses the different strategies proposed to identify performance issues triggered by critical sections and synchronization primitives.

Bottleneck Identification and Scheduling (BIS) [JSMP12] proposes a hardware-software framework to rank bottlenecks by the number of *wait* cycles aggregated across all waiting threads. Wait cycles are measured by delineating potential bottlenecks with special instructions, *BottleneckCall* and *BottleneckReturn*, and replacing the code that waits with a special instruction, *BottleneckWait*. These instructions should be injected either by the programmer, compiler or shared library. Wait cycles of each bottleneck is stored in a table, *Bottleneck Table* (BT). Thread waiting cycles for entries are updated every cycle by the number of waiters for that bottleneck. Performance is improved by routing bottlenecks with waiting cycles greater than a threshold, to fast cores on an x86 cycle-level simulator of an Asymmetric Chip Multi-Processor.

Free-Lunch profiler [DTLM14] is a Java lock profiler that identifies locks that impedes the performance of threads most in a multithreaded application. The impact of locks on thread progress is represented in terms of a metric, *Critical Section Pressure (CSP)*, represented as the ratio of the time spent by the threads in acquiring the lock to the cumulative run time of those threads. A higher CSP value represents longer waiting time. The lock acquire time is measured by instrumenting lock acquisition methods in *JVM*. The profiler shows that lock contention alone is not enough to assess the impact of a lock on thread's progress.

Parallel Execution Profiles [BGZ16] generates execution profile for each thread in a multithreaded application, arranged as a series of time stamped frames. Each frame represents what activities were performed by threads in parallel. It uses user-annotations to mark regions of interest, such as conditional wait calls, lock acquire and release calls, or a barrier wait. These annotations guide the profiler in collecting per thread timestamped event traces. The execution profile generated provides a global picture of threads executing the same region of code. Overlapping timestamped frames indicate parallel execution, and regions with no overlap indicate possible optimization opportunity. The frames are analysed offline to identify issues such as lock contention and straggler threads (threads that cause excessive waiting at a barrier).

SyncPerf [LZM⁺17] categorizes synchronization related performance issues based on lock contention rate and its frequency of acquisition. While prior work concentrates on highly contended locks with high acquisition rate, *Syncperf* establishes that even with reduced contention, large critical sections, and locks with high rate of acquisition provide potential optimization opportunities. *Syncperf* proposes two tools: a) a *detection tool* that reports call sites and synchronization variables with potential performance issues, and b) a *diagnosis tool* that helps identify root cause of the anomalous behaviour by monitoring memory accesses inside specific critical regions. The *detection tool* gathers information by intercepting *Pthread* library synchronization primitives with a pre-loaded library, and the *diagnosis tool* is built using the binary instrumentation tool, *Pin*. *Syncperf's* data collection strategy is similar to the one proposed by [TMCP10] (discussed in the section 2.2); however, *Syncperf* incurs much lower overhead by optimizing call site information capture.

The strategies discussed above identify execution regions where a shared resource is acquired and released by instrumenting the source code, binary or threading library. Such instrumentation mechanisms allow execution traces to be easily mapped to the source code. Information is generated only at instrumentation points thereby limiting the amount of data collected, which makes post-processing faster and easier. However, they can only pinpoint bottlenecks that arise due to a particular class of synchronization mechanism and are also bound to a particular language or library which restricts their use.

As applications become complex with many cooperating components, bottlenecks need not arise in all executions. The role of workload is crucial in the performance of multithreaded applications. *SyncProf* [YP16], *FOREPOST* [LNGP17] and *GA-Prof* [SLPG15] not only identify bottlenecks, but also classify or generate workloads that can trigger performance issues.

SyncProf [YP16] detects synchronization bottlenecks, identifies their root cause and suggests optimization strategies. The performance impact of individual critical sections are summarized in terms of metrics such as *All-path Wait Time*, *Critical Path Wait Time* and *All-path Lock Time*. These metrics are computed by building a graph that represents wait relationship between critical sections. The same graph is analyzed to heuristically suggest optimization strategies for the bottlenecks detected. Critical sections are traced by instrumenting the *Pthreads* library synchronization APIs in the application using the *Pin* instrumentation tool. To mitigate the non-deterministic behaviour of multithreaded applications, *SyncProf* dynamically analyzes multiple executions of the program for different inputs and selects inputs where the workload increases execution time while reducing CPU utilization.

FOREPOST [LNGP17] classifies workload as *good/bad* based on its execution time. Test cases which result in increased execution time are labeled as *good*. The application is instrumented to generate execution profiles, which are fed to a feedback-directed learning system, that marks test cases as *good/bad*. Bottlenecks are identified as method calls that seriously affect the performance of the *Application Under Test*. Significance of a method is computed using *Independent Component Analysis* where each method is represented in terms of its resource consumption, which is a normalized weighted sum of the number of times the method is invoked, total elapsed

time of the method and number of methods invoked from it. GA_Prof [SLPG15] follows a similar approach except that the test cases are generated using Genetic Algorithms.

b. Thread behaviour

A further category of tools identify bottlenecks based on thread parallelism using sampling, kernel tracing, thread library instrumentation, or a combination of these methods. This section summarizes strategies that identify bottlenecks by analyzing thread behaviour.

[TMC09] quantifies reduced parallelism in a multithreaded application by attaching an *idleness metric* to each call path. It maintains n_w (number of working processor cores) and $n_{\bar{w}}$ (number of idle processor cores), by monitoring when a core acquires/finishes a unit of useful work. This is done by modifying the run-time systems of multithreaded programming models. The *idleness metric* is measured using a sampling mechanism, that attributes $n_{\bar{w}}/n_w$ to a working thread on each sample. The sampler also obtains the call path by unwinding the call stack at each sample point. The approach has been implemented and evaluated on applications using the *Cilk* threading model.

[TMCP10] from the same authors extends the idea of *idleness metric* to analyse lock contention in multithreaded applications. They show that *idleness metric* derived from lock contention should be apportioned precisely to the lock holder to identify the root cause behind contention. To achieve this, the authors use lightweight instrumentation of synchronization primitives, and associates the *idleness metric* of a waiting thread to the lock, rather than to the thread. When the lock holder releases the lock, the entire *idleness metric* accumulated on the lock is attributed to the lock holder.

WAIT [AAF10] proposes a sample-based strategy to identify root cause of idle time in modern enterprise class Java applications. Each sample collects stack frames, Java thread states and processor utilization. Information thus collected is analysed using pattern matching and decision trees to create a hierarchical categorization of wait state of threads across time. The wait state provides information at a higher abstraction level about its cause, for example whether it

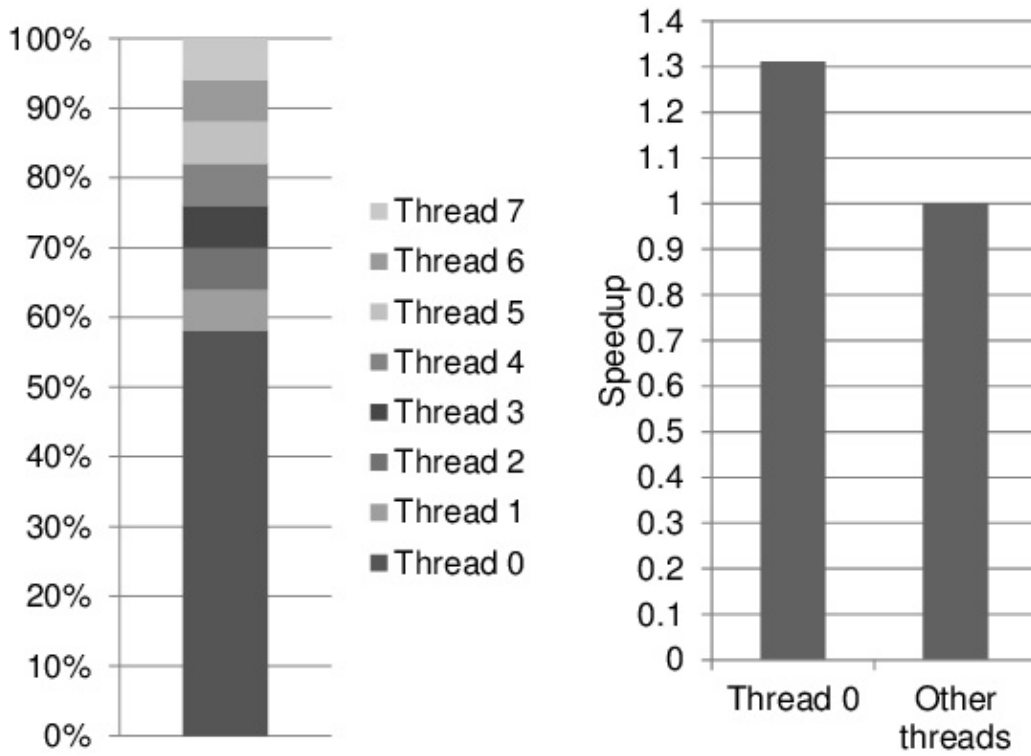


Figure 2.4: Criticality stack and total program speedups from accelerating the identified critical and non-critical threads [DESE13]

is in a *runnable* state spinning for a lock, or in a *waiting* state blocked on disk I/O or lock, or in a *sleeping* state. The application behaviour is summarized in terms of processor utilization, runnable/waiting threads and wait categories averaged across the collection period. Processor utilization is recorded using *vmstat* and Java thread states are monitored from "javacore" files.

Criticality Stacks [DESE13], proposes a hardware-software approach to identify most critical threads, those whose performance determines the performance of the program as a whole. The *criticality* of a thread is calculated in terms of its execution time and number of waiting threads. The execution time is divided into intervals, each which begins when a thread changes state from active to inactive or vice-versa. For an interval t , if r out of n threads are running, t/r is added to the *criticality counter* of the r active threads. A hardware component keeps track of the *criticality counter* of each thread which is updated on receiving a signal from the core. A thread is considered to be inactive if it is either scheduled out or if it is spinning. Whether a thread is spinning or not, is tracked by instrumenting the threading library primitives. The total execution time is split into individual thread's criticality component, and stacked one over

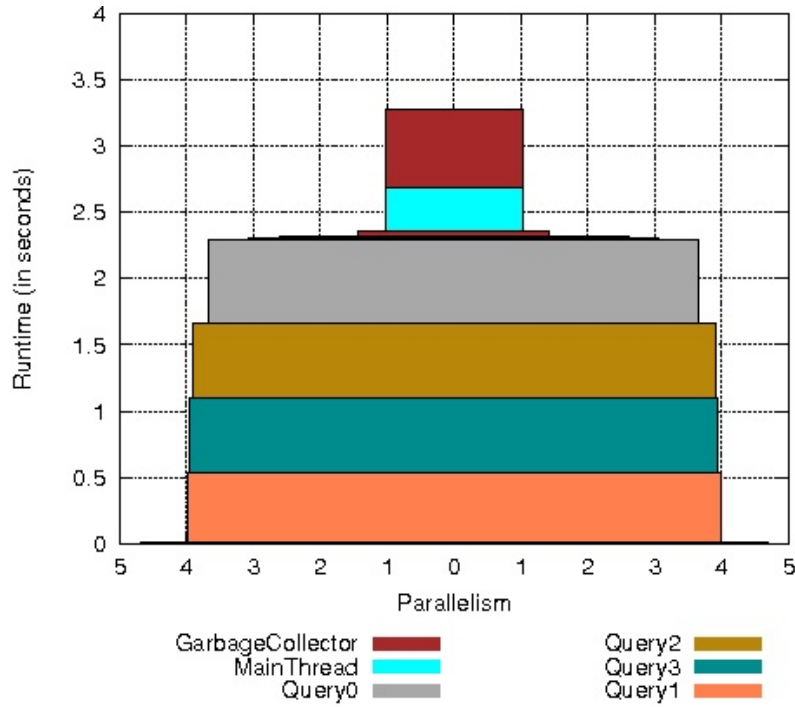


Figure 2.5: Example BottleGraph [DBSEE13]

the other to build a *criticality stack* (Figure 2.4). The correctness of the metric is validated by demonstrating that scaling the frequency of the most critical thread achieves maximum performance improvement.

BottleGraphs [DBSEE13] presents a tool to visualize performance of multithreaded applications in terms of thread parallelism and execution time. It generates a graph that consists of boxes stacked one over the other, one box per thread, where the height of the box represents the thread’s contribution to total execution time and the width represents its parallelism (Figure 2.5). As in *Criticality Stacks* [DESE13], this helps the developer in identifying critical threads in an application. *BottleGraphs* utilizes Linux kernel modules to gather information about active threads. These modules keep track of the number of active threads and are triggered when threads are scheduled in or out. Both *Criticality Stacks* and *BottleGraphs* do not identify the code fragment which triggers the actual bottleneck.

ParaShares [KTK14] scores and ranks all basic blocks in the application based on their share of parallel execution. It tracks how many times each basic block executes at different thread counts and ranks them based on each block’s share to the program’s total execution time. Each block of code is weighed by the run time parallelism exhibited during each execution.

Consequently, blocks of code that exhibit reduced parallelism will be assigned greater weight when compared to blocks that exhibit higher parallelism. This information is collected using an extended version of LLVM compiler, *Harmony* [KTK12], which instruments the application at thread entry, exit and thread library calls.

TaskProf [YN17] identifies parallelism bottlenecks by calculating the critical work (longest chain of tasks executed serially) at each spawn site of a task parallel application. Spawn sites that perform large fraction of the critical work are identified as potential bottlenecks. The profiled application is statically instrumented to use a modified task parallelism library, with which spawn sites are traced. Computation performed by tasks is measured using hardware performance counters during execution. Data thus collected, which represents work performed at spawn site, is used to build a DPST (*Dynamic Program Structure Tree*) that captures the series-parallel relationship between tasks. The total and critical work at each spawn site, is computed by performing a bottom up traversal of the DPST. *TaskProf* also implements a causal profiler that estimates the improvement in parallelism when a region of critical work is optimized. This is further discussed in Section 2.3.

wPerf [ZGMW18] represents the wait relationship between threads as a *wait-for-graph*, where an edge from A to B represents that A is waiting for B . The edge-weight, which is proportional to wait time, represents the local impact of the waiting event. *wPerf* consists of two components; a *recorder* that records time-stamped, scheduling, IRQ, I/O and wake-up information using kernel modules and call stacks using the *perf* tool, and an *analyzer* that post processes the recorded information, builds the *wait-for-graph*, and analyzes it to identify bottlenecks. The bottleneck identification algorithm searches for *Strongly Connected Components* with no outgoing edges in the graph. Such knots are potential optimization opportunities, which can be further analyzed by the user.

[IUNO19] proposes building a *thread dependency graph* to detect *layered bottlenecks* in software applications with multiple service layers (for example, web applications). Each node in the graph represents a group of threads (the node is annotated with thread count) performing the same task and a directed edge from node a to node b indicates that threads in a are waiting

for services from *b*. Paths having the largest number of threads are identified as potential bottlenecks. The graph is built from stack traces collected by sampling all threads (runnable and blocked) in the application.

Solutions discussed in this section follow one of the two strategies: *a.* identify bottlenecks based on thread parallelism, or *b.* build and analyze graphs that represent wait relationships among threads. While *Criticality Stacks* and *BottleGraphs* are capable of identifying the most critical thread, they do not provide information regarding the code section that causes the bottleneck. In order to extend the concept of thread parallelism, so that the critical code fragment can be pinpointed, either sampling with call stack retrieval [TMC09] or instrumentation of the threading frameworks [YN17] is required. As discussed before, instrumentation will limit the scope of the strategy to a specific threading library. When sampling is used to trace program behaviour, the proximity of the trace to the actual behaviour highly depends on the sampling frequency. Retrieving call stacks at every sample can be expensive in terms of the overhead.

The strategy followed to determine active threads can also affect the metric values used to determine parallelism in the framework. For example, [TMC09] samples a core to determine if a thread is active or not. This can produce wrong results when the number of threads is greater than the number of cores, or when there are other application executing in parallel. This is because threads can be in the *runnable* state, waiting for the CPU if the machine is over-subscribed.

Techniques that build graphs to analyze performance issues write entire trace data to files which are post processed to build the graph. They put forth two main challenges: a) size of the trace file, and b) post-processing time, which both depend on the amount of data and the distinct events monitored.

c. Critical call paths

Yet another category of approaches identify bottlenecks by analyzing call path profiles. Such strategies usually do not require the application to be modified and hence are commonly tar-

geted for enterprise class applications or applications in a production environment. They gather execution information through core dumps, execution logs, stack traces or kernel modules.

BOTTLENECKS [ACGS04] presents a method to find expensive call sequences from call-tree profiles of one or more execution of a system. It also identifies call sequences that are significantly expensive in one call-tree profile when compared to others. To achieve this, *BOTTLENECKS* implements two interfaces; a *profile interface* that provides a set of primitives to generate summaries from call-tree profiles and a *user interface*, which provides a set of commands with which the summaries can be queried or labeled. The interfaces have been designed such that they can be applied on any profile that associates a metric to a call path.

[JHHF09] analyses execution logs of load tests to identify scenarios that deviate from a baseline performance derived from previous runs. Event sequences extracted from execution logs of uncommon scenarios, together with a comparison between corresponding events of the previous runs will help the developer identify potential reasons behind the performance deviation. The approach claims to be useful in studying how changing system configurations like caching, affect the event sequences and thereby the overall performance of the application.

StackMine [HDG⁺12] uses data mining and clustering techniques to identify call traces that cause performance issues in post-production systems. It identifies performance issues by extracting relevant events and call stacks from the traces, mining them for call stack patterns and clustering the patterns based on a similarity model so that each cluster points to a particular performance bug. *StackMine* has been successfully used for digging out performance bugs from millions of stack traces collected from *Windows 7* deployment-sites.

d. Hardware performance counters

Every processor provides a set of hardware performance counters that count events such as cache misses, branch mis-predictions, pipeline stalls etc.

ADP [YLB⁺12] proposes a machine learning based automated system that identifies performance issues in multithreaded applications from hardware performance events. Functions in

the application are mapped to specific pathological classes such as *heavy/light branch mis-prediction*, *inefficient access on array* etc., with the help of a decision tree. The decision tree classifies functions based on hardware events collected during execution. However, this strategy may not be able to categorize bottlenecks that arise due to contention for software resources such as locks.

Section 2.2 discussed the principles and strategies followed by existing performance analysis/bottleneck detection tools. The emergence of new techniques to analyse program behaviour have enabled performance engineers in developing analysis tools with rich features. Chapter 3 explains how the state of the art tracing techniques have motivated the design of a light weight, generic parallel profiler, GAPP, which leverages thread parallelism to identify bottleneck code sections in an application, without instrumenting it.

2.3 Related Work - Performance Prediction

Detecting performance issues in advance of them occurring is always desirable, as such issues may require the system design to be changed. Once a bottleneck is identified, fixing it might include several strategies such as refactoring the code, rewriting the algorithm, changing synchronization strategy or even choosing a different communication mechanism. If the bottleneck is the result of hardware-software interaction, fixing it might even involve changing the hardware design. In such cases, developers will not be able to quantify the effect on performance, unless and until these alternatives are tested with.

As an aid for developers in such situations, several performance prediction strategies have been proposed in literature. These include performance models, simulation, and analyzing graphs built from execution traces. This section discusses different performance prediction strategies published in literature.

2.3.1 Performance models

A popular approach in software performance analysis is to build performance models to investigate optimization opportunities in an application. They abstract a system's behavioural details into mathematical or empirical models.

Several analytical models have been proposed to predict performance of a software system for different hardware configurations [Moo14][ASE17], for a different workload [BHD⁺00] or for a faster code [YM88][HM94]. [BJ18] models the time between synchronization events to explore how different architectural/application designs affect the ordering of these events and its effect on overall performance. In general, the application to be investigated is profiled to gather its characteristics and the model is applied to estimate the performance of the application for different application/architectural features.

In spite of its wide acceptance among performance engineers, complex synergies between different system layers viz., hardware, operating system, system libraries, run-time environment, middle-ware and application layer of multi-core and many-core systems makes it difficult to build accurate models that replicate the real scenario [MBDH99][NS07][KBH⁺14].

Rather than focusing on analytical models that abstracts away a system's behaviour into mathematical models, the focus of this study are system level, execution-driven, real-time approaches that predict the performance impact when application components such as logic, synchronisation pattern, I/O mechanism etc. are modified. Hence strategies that are purely analytical are not discussed further.

2.3.2 Execution driven approaches that simulate the passage of time

Several of the prediction strategies use the concept of virtual-time, where the system/application progress is synchronized with an imaginary clock that follows a virtual time line [Jef85]. Such simulation frameworks may implement time virtualization at different levels, for example, in the operating system, in *VMs*, in schedulers or in the application level. This section discusses

different approaches that implement performance prediction through time virtualization.

Virtual machines and time-dilation

Time dilation was first proposed by [AS90] as a method to emulate and quantify the effect of increased communication speed on the performance of an application, without redesigning the communication hardware. Here, the duration of the computation part is dilated/increased by a user specified *dilation factor* to create the effect that the communication speed has increased in relation to the computation speed. The ratio of normal execution time to the normalized *dilated* execution time provides the speedup achievable with a faster communication channel.

[GYM⁺05] implements *time dilation* on virtual machines to make physical devices appear relatively faster to applications and operating systems. In virtual machines, the notion of time to the guest operating systems is exposed by the *Virtual Machine Monitor*. In this approach, frequency of timer interrupts delivered to the virtual machine is controlled, so that, time in the guest operating system elapses at a different rate relative to the time of physical devices. The ratio between rate of change in time of the physical world and guest operating system is specified in terms of a *Time Dilation Factor* (TDF). However, with this approach, all physical devices are scaled uniformly.

[LST15] proposes the concept of *adaptive time dilation*, where the dilation factor remains constant during the time when a *VM* is active, but can be changed to a different value (greater or less than one) during the time when a *VM* is suspended, so that the next active time slice of the *VM* will execute using the modified *TDF*. Time dilation is controlled at the *hypervisor* level so that *VMs* can run unmodified operating systems. The framework is used to simulate large scale heterogeneous distributed systems, where each system is represented as a *VM*.

TimeKeeper [LNC14] puts forth a kernel modification that allows multiple Linux Containers to progress at different rates specified by their *TDFs*. It uses a custom kernel with modified *task struct* and system calls (*gettimeofday()*, *sleep()*, and *poll()*). It also includes APIs to freeze or unfreeze a process. Since Linux Containers share the same kernel, this approach makes it easier

to conduct experiments with multiple processes progressing at multiple rates. Each process is allocated a *time-slice* based on its *TDF* and are scheduled based on a FIFO scheduling policy. The progress of each process is maintained in terms of its virtual-time.

In all these studies, *time dilation* is used to make a computational or communication entity appear faster or slower when compared to other components in the system. While we can use *time dilation* to control the execution speed of a thread/processes running on a *VM*, it imposes a limitation on the granularity where this concept can be applied; functions in a process/threads cannot be scaled using virtual machine strategies.

Simulating passage of time at functional level

VEX [BF11] proposes a strategy to accelerate/decelerate (scale) code on a per-method basis, specified by a *scaling factor*. It implements a user level profiling scheduler, as a wrapper around the operating system scheduler, to impose a thread schedule that mimics the scaled execution. VEX uses the concept of virtual-time to determine the time allotted to each thread by the profiling scheduler. For example, if the default time-slice of the profiling scheduler is t units of time, with a *scaling factor* of 2, the virtual-time of the scaled method will advance only by $\frac{t}{2}$. Since scaled thread is lagging behind others in terms of the virtual-time, the profiling scheduler will choose this thread during the next scheduling opportunity. Consequently, threads executing scaled methods with a *scaling factor* > 1 will be allotted longer execution time-slices. The virtual-time of methods performing I/O is calculated using a prediction mechanism that works on the basis of recent I/O activity.

[BF12] integrates performance models to the VEX framework. It allows threads to invoke methods described by performance models and accounts for the time spent by such methods using the time scaling scheduler described before.

[FCW18] extends the concept of virtual time to embed performance models in mock objects used in unit testing frameworks. Since models can produce performance estimates faster when compared to a similar real execution, unit testing frameworks using models will have better

turn around time. At the same time, this can cause impedance mismatch between the real execution and performance models, which needs to be addressed for correct simulation.

In fact, [BF12] and [FCW18] are examples of execution-driven simulation models that combine executing code with performance models, where part of the application that needs special treatment is handed over to models that deliver the required results.

Curtsinger and Berger have proposed a causal profiler, Coz [CB15], that predicts the performance of an application when a chosen line of code is accelerated. It does so by pausing other threads for a time interval proportional to the speedup chosen and updating the overall execution time by subtracting the total wait time inserted for all peer threads. The effect of this optimization is calculated by measuring the rate at which pre-defined points of interest, named as *progress-points* are executed. Performance data are gathered using sampling and *progress points* are specified by statically instrumenting the source code. As the analysis and prediction is based on real execution of the software, the correlation between different abstract layers of the execution environment will be automatically taken care of. Its basic principle relies on sampling, however for handling suspended threads, it intercepts and instruments POSIX thread blocking functions.

Note that, VEX and Coz simulates the effect of scaling a function/line of code by altering thread schedule, which also alters the resource usage pattern during execution.

2.3.3 Graphical representation of execution

A different class of performance prediction strategies are built on the concept of *Critical Path Analysis* (CPA) [YM88], which identifies the execution path that took the longest time to execute. CPA is performed by building a *program activity graph* (PAG) with nodes representing events of interest, edges the ordering of events, and edge weights the time between events.

Optimizing a procedure along a critical path can shift the critical path to a different route. [HM94] proposes a metric called *slack*, which quantifies the performance improvement possible

on a procedure on the critical path, without actually altering it. *Slack* is calculated by finding the difference in lengths of the arcs terminating at a merge node in a PAG.

TaskProf [YN17], discussed in Section 2.2 generates a causal profile that estimates the improvement in parallelism when code regions that execute critical work are optimized. *TaskProf* builds a tree structure, *DPST* (Dynamic Program Structure Tree), which captures the serial-parallel relationship between tasks. The bottleneck identification part discussed in Section 2.2 identifies spawn sites that perform critical work. To estimate a critical spawn site's effect on parallelism, they need to be annotated by the user and re-run on *TaskProf*. The post-execution analysis tool computes the asymptotic parallelism and critical work at each spawn site from the *DPST* and also generates a causal profile that quantifies the impact of optimizing a region of code.

Prediction methodologies that rely on performance models are used to study the effect of different hardware configurations. Those which use virtual machines simulate the effect of a faster communication technology or a faster component in a distributed heterogeneous system. A few use faster processor cores to speed up critical threads in an application [DESE13]. These approaches cannot apply time-scaling on individual functions in an application. Program activity graphs built from execution profiles can be used to predict the effect of scaling individual functions. However, scaling functions can also alter hardware resource access pattern, which will not be captured by prediction mechanisms that use program activity graphs.

Chapter 5 propose an execution driven prediction framework that combines time dilation and virtual-time, to emulate the behaviour a multithreaded application would exhibit, if individual functions are scaled.

Chapter 3

GAPP - A Generic Automatic Parallel Profiler

A key challenge for multi-core program developers is identifying and fixing performance problems and this is exacerbated in multi-core systems because of competition for shared resources. Such resources may be physical, e.g. a hardware accelerator which can only be accessed by one thread of computation at a time, or logical, e.g. a lock protecting access to parts of, or the entirety of, a shared data structure. The speed-up achievable in such cases is thus limited by the inherent serialization that occurs, as governed by Amdahl’s law.

The resource(s) that experience the greatest contention are called the *bottleneck resources*, or just *bottlenecks*, akin to the terminology used in queuing theory. The key insight from queuing theory is that performance can *only* be improved significantly by “fixing the bottleneck”, i.e. by reducing the load placed on the resource, e.g. by accessing it less often or holding it for less time, on average. To illustrate the problem, consider the program listed in Figure 3.1 (only the key lines of code are detailed). Here, `n` threads execute a function `fun1()` followed by a function `fun2()`, which has a code segment protected by a shared lock. Each thread executes for t units of time in `fun1()` and $\frac{t}{4}$ units of time in `fun2()` respectively.

Suppose we want to speed up the application. A generic profiler will suggest targeting `fun1()`, as this dominates each thread’s CPU time. For example, *GProf* [GKM82] reports that approx-

```
28 void fun1(...) {  
    ... //Execute for t units of time  
46 }  
  
48 void fun2(...) {  
52     pthread_mutex_lock(&mutex);  
    ... //Execute for t/4 units of time  
65     pthread_mutex_unlock(&mutex);  
    }  
  
69 void *threadFun(...) {  
    ...  
    fun1();  
    fun2();  
    ...  
81 }  
  
103 int main() {  
    ...  
    for (int i=0, i<n, i++)  
        pthread_create(..,threadFun,..);  
  
    for (int i=0, i<n, i++)  
        pthread_join(...);  
    ...  
}
```

Figure 3.1: Example code fragment

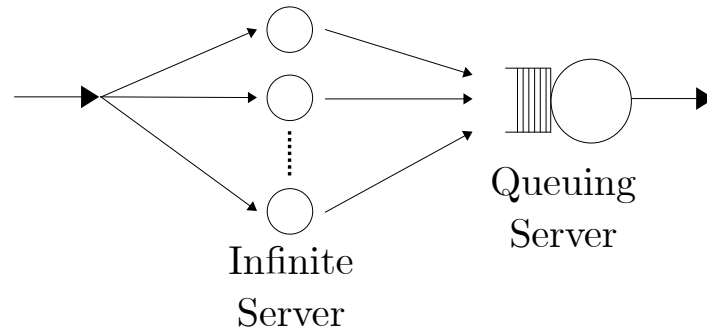


Figure 3.2: Queuing network for Figure 3.1

imately 58% of the time is spent in `fun1()` in the above program as shown below.

%	cumulative	self		self	total	
time	seconds	seconds	calls	ms/call	ms/call	name
58.21	1.08	1.08	8	135.35	135.35	fun1
42.04	1.86	0.78	8	97.75	97.75	fun2

If we were to speedup `fun1()` by a factor of two, say, then the average execution time would improve by circa 17%. However, if the code segment guarded by the lock in `fun2()` was speedup by a factor of two, then that would improve the program execution time by approximately 33%. This shows that we can achieve almost double the improvement in performance with a fraction of the effort if we focus on the critical code segment rather than on `fun1()`.

The reason for this is clear when we think of this as a queuing network, as shown in Figure 3.2. The infinite server models `fun1()` (assuming each thread is pinned to a separate core) and thus represents N independent delay nodes, one per job, so the average number of visits each job makes to one of those nodes is $1/N$. In contrast, each job visits the queuing node, which models the lock, exactly one at a time. With the above parameterization and with N jobs in the system, the bottleneck device is actually the lock, as it has the highest *service demand*, which is the product of the average number of visits to a resource and the average service time at that resource.

```

*****
Critical Call Paths, functions and Lines of Code:
*****

Critical Path 1 -- CMetric, Frequency
-----

fun2 ()[example_time]
  <---threadFun ()[example_time]
  <---start_thread ()[libpthread-2.27.so] --762859993, 77.

Functions and lines + Frequency
-----
fun2 -- 236
  example_time.c:55 -- 128
  example_time.c:57 -- 95
  example_time.c:58 -- 5

*****

Top Critical Functions and lines of code with frequency
-----

fun2 -- 236
  example_time.c:55 -- 128
  example_time.c:57 -- 95
  example_time.c:58 -- 5

```

Figure 3.3: GAPP profile for example code

Because the lock is the bottleneck device, the only way to improve performance is to reduce the demand imposed on it and this can only be achieved by speeding up the critical section in `fun2()`.

The above principle applies to any shared resource, hardware or software, with synchronized access. Such resources are characterized by having a queue of service requests waiting for access. Intuitively, the node that is active when the majority of the service requests are queued is most likely the bottleneck in the system. We show that this symptom of waiting service requests can be tapped to identify bottlenecks that arise due to a variety of reasons. Moreover, when there are multiple bottlenecks in a system, we can rank them based on the service demand.

This chapter presents a new bottleneck detection tool, called GAPP (Generic Automatic Parallel Profiler), which automatically pinpoints the line(s) of code that represent serialization bottlenecks in a multithreaded Linux application. In keeping with prior work on bottleneck

detection, the idea is to detect the critical code sections that have a tendency to serialize execution, i.e. which prevent other threads from running concurrently with them. The bottlenecks are ranked based on a *Criticality Metric* [DESE13], that takes into account the duration of execution and degree of parallelism during each time-slice. Figure 3.3 shows the profile generated by GAPP for the example code in Figure 3.1. As we can see, GAPP not only identifies the function that serializes the execution, but also the lines of code in the function that causes this behaviour.

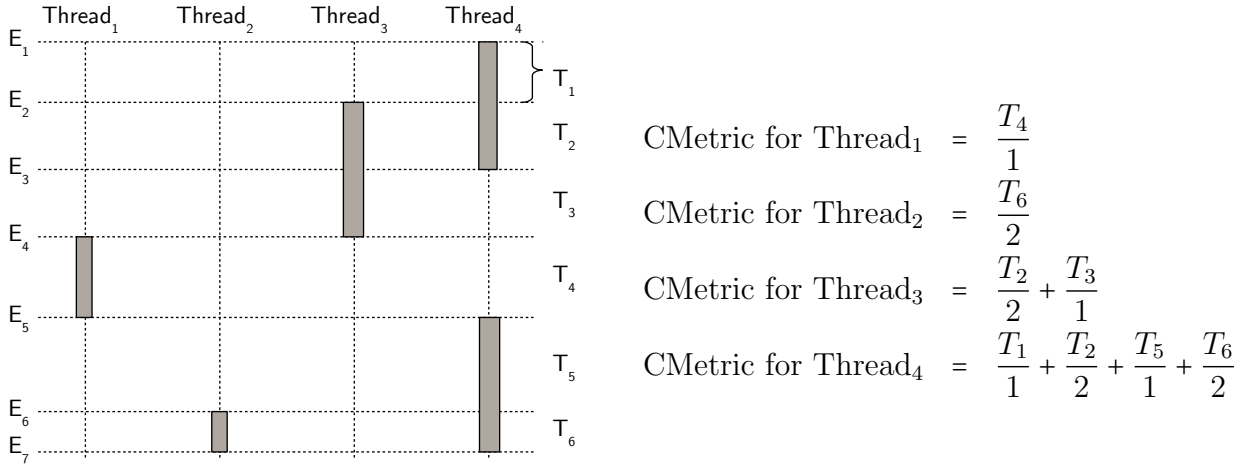
In contrast to some other bottleneck detection systems which work by instrumenting specific languages, libraries or synchronization primitives [YN17, YP16, LZM⁺17, TMCP10, DTLM14], GAPP uses lightweight instrumentation at the kernel level. Furthermore, all bottleneck analysis in GAPP is performed at run time, which avoids the need to generate, and subsequently process, potentially expensive trace files. Indeed, a key objective in GAPP’s design has been to minimize overhead, both during program execution and in post processing.

This chapter discusses the concept governing GAPP, its design and implementation.

3.1 Thread Criticality - A Metric for Ranking Serialization Bottlenecks

Serialization bottlenecks in parallel applications are characterised by execution regions with reduced parallelism. [DESE13] has proposed a *Criticality Metric* (hereafter referred to as the *CMetric*), as a measure to identify critical threads in a multithreaded application. The *CMetric* is calculated taking into account the duration and number of active threads during that duration. For an execution duration of t , its *CMetric* is defined as t/n , where n represents the number of active threads during time, t .

The *CMetric* of a time-slice reflects how critical the time slice is in determining the performance of a parallel application. This is because it takes into account both the duration and number of active threads. The advantage is that it distinguishes threads that execute with low parallelism

Figure 3.4: Switching intervals and their *CMetric*

from those which execute with high parallelism for the same duration, and also threads that execute for short duration with low parallelism from those that execute for long duration with a similar low degree of parallelism.

The *CMetric* is calculated from the duration of the thread's time-slices and the degree of parallelism exhibited whilst each is executing. The execution time-slice of a thread starts when it is switched in and ends when it is switched out. To compute the *CMetric*, we further divide time slices into *switching intervals* demarcated by instants when *any* thread in the application changes state from the active to inactive state, or vice versa, as it is only this action that can change the degree of parallelism. Figure 3.4 shows an example trace of a multithreaded application with four threads. The $E_i, 1 \leq i \leq 7$ are context switching events and the $T_i, 1 \leq i \leq 6$ are the time intervals between consecutive events, E_i and E_{i+1} . Thus, for example, *Thread₃*'s time slice spans two switching intervals, T_2 and T_3 .

The *CMetric* for every active thread is updated at the end of each switching interval. For example, in the interval T_2 in Figure 3.4, there are two active threads, and hence the *CMetric* for the overall execution is $\frac{T_2}{2}$, which is added to the individual *CMetric* of both *Thread₃* and *Thread₄*. The *CMetric* values for all four threads are thus computed as given in Figure 3.4.

3.2 Calculating *CMetric*

Calculating the *CMetric* involves tracking the following events:

- The instant when threads become active or inactive.
- The duration of switching intervals.
- The duration of time-slices.
- The number of active threads at any instant.

This can be done by instrumenting those threading library primitives that alter thread states, for example, primitives that spawn new threads, acquire/release locks, wait on barriers/conditional variables etc. However, this limits the scope of this method to applications that use specific threading libraries. Instead, frameworks that rely on the Linux tracing subsystem track kernel events that alter thread states. There are two advantages to this approach; a. such tracing frameworks work independently of the threading libraries and b. they can capture events triggered not only by threading library primitives, but also by I/O or any operating system events.

Kernel tracing with eBPF and *BCC*

A limitation in using the kernel tracing mechanism is that it generates large amount of data, as every kernel event is traced and reported, irrespective of whether it is of interest or not. Rather than forwarding the entire data from traced events to the user space, it is more desirable to have a filtering mechanism in the kernel so that only relevant kernel events are passed to the user space. However, kernel events occur very frequently, thus the filtering mechanism should be fast enough to avoid performance issues in the probe functions.

eBPF, which stands for *extended Berkeley Packet Filter*, is a framework that implements fast filters for kernel event tracers. It uses an in-kernel sand-boxed virtual machine that allows user-provided programs to be run fast and secure inside the kernel [Fle18][SD16]. Moreover, it is

part of the mainline kernel and does not require any third party modules. User-provided eBPF programs that define the filters for kernel events can be attached to *kprobes*, *tracepoints*, *uprobes*, *USDTs* or *perf-events* so that the attached programs are executed when the corresponding event is triggered. Since allowing user-provided code to run in the kernel can pose security risks, eBPF uses a *verifier* that performs a number of checks to ensure the safety of the program before it is loaded.

User-programs attached to kernel events are represented in BPF bytecode (in-kernel VM's instruction set), a set of instructions that are very close to the x86-64 instruction set. The bytecode generated in the user space is loaded into the kernel, verified by the in-kernel verifier, and JIT compiled to the target instruction set for faster performance. User programs can be written as BPF instructions and transformed into BPF bytecode using the kernel's `bpf_asm` assembler, or can be written in a restricted form of C and compiled to BPF bytecode using LLVM's eBPF backend support, or can be written using wrapper functions provided by BPF Compiler Collection (BCC) toolkit [Fle17]. The *BCC* toolkit makes it easier to write eBPF programs as it automatically compiles them to the virtual machine's bytecode using the LLVM backend.

BPF Compiler Collection (*BCC*)

A BCC program has two parts, a front end written in Python/Lua and a kernel instrumentation part written in restricted C. The C part holds the probe functions that trace kernel events and filter the data. It is compiled to BPF bytecode and attached to kernel hooks using the `bpf()` system call. Kernel events are traced by attaching user programs to *tracepoint probes* or *kprobes* at interesting points in the kernel. For example, to trace context-switch events, we can attach a probe function to the `sched_switch` tracepoint, which will be fired each time the scheduler switches execution between tasks.

To support data sharing among eBPF programs, and also between the user and kernel space, eBPF defines a set of kernel data structures called *maps*. They store data as key-value pairs and can be used to maintain state across different executions of probe functions. Maps are

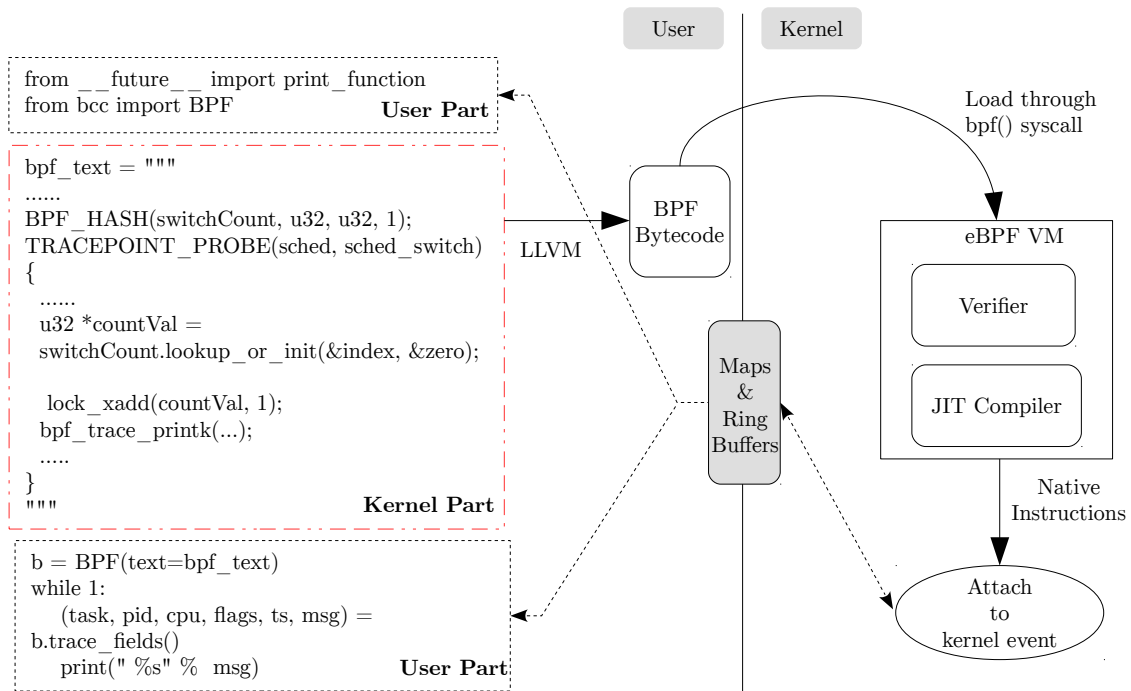


Figure 3.5: The BCC - eBPF framework

implemented in kernel memory with access rights from user space. On a multi-core processor, multiple instances of the same probe function can be running simultaneously on different cores. Each *map*, however, has a single instance across all the cores, access to which is protected by the kernel Read-Copy Update (RCU) mechanism. Maps can be shared across all cores, or can be *per-cpu*, where each core will have a local copy. These maps are readable from user space programs. Data can also be passed from kernel to user space through a shared ring buffer. Figure 3.5 depicts how a *BCC* program written in *Python* is attached to kernel events.

The JIT compiled in-kernel virtual machine allows eBPF programs to execute with minimum overhead. It also facilitates retrieving user/kernel stack traces from within probe functions. The *BCC* toolchain supports high-level languages such as C, Python, Lua etc to develop front ends for eBPF programs. GAPP uses the Python front-end provided by *BCC* to write, compile and load eBPF programs. eBPF programs are automatically unloaded when the user process that loaded it exits [SK].

The following sections explain how the *CMetric* of a thread is calculated using *BCC* and eBPF.

3.3 A Software Solution for Thread Criticality

In order to compute the *CMetric* for application threads, GAPP attaches probe functions to different kernel *tracepoints* and sets up the following eBPF maps:

Map name	Description
<code>tsp</code>	Global scalar - stores timestamp of the most recent switching event
<code>cm_hash</code>	Global hash map - stores the <i>CMetric</i> of each thread
<code>global_cm</code>	Global scalar - cumulative sum of <i>CMetrics</i> across all switching events
<code>local_cm</code>	Local scalar - records value of <code>global_cm</code> when a thread switches in
<code>thread_count</code>	Global scalar - keeps track of the no. of active threads at any time
<code>total_count</code>	Global scalar - stores the total number of threads in the application
<code>thread_list</code>	Global hash - list of thread ids of the application; for each thread, stores 0 if thread is inactive and 1 if active
<code>t_switch</code>	Local scalar - stores the timestamp of the most recent switching event
<code>global_wt_tc</code>	Global scalar - cumulative sum of (switching interval * <code>thread_count</code>)
<code>local_wt_tc</code>	Local scalar - records value of <code>global_wt_tc</code> when a thread switches in
<code>in_TSP</code>	Per-CPU map that stores the timestamp when an application thread is switched in

Table 3.1: eBPF Maps for calculating *CMetric*

3.3.1 Monitoring active threads

As explained in section 3.1, calculating the *CMetric* involves identifying time slices and switching intervals that constitute them. Each switching interval is demarcated by the instance when a thread becomes active or inactive, and this can be monitored by tracing context-switch events of the participating threads. To achieve this, an eBPF probe function is attached to the `sched_switch` tracepoint (defined in `sched/core.c` of the kernel code) exported by the Linux tracing subsystem. This probe function, which is triggered on each context switch, stores the timestamp of the switching event in the eBPF map `tsp` (see Table 3.1), the instance that cor-

responds to E_i in Figure 3.4. The number of active threads, `thread_count`, is updated when an active thread becomes inactive or vice-versa. The threads are marked active/inactive in `thread_list`, based on its current state; threads in the `TASK_RUNNING` state are considered to be active and those in other states are considered inactive. As threads change state during context-switches, the `thread_count` can be monitored by the `sched_switch` tracepoint probe.

The probe functions need to be executed only if the threads involved belong to the application. The identifiers of the application threads are extracted by attaching additional probes to `task_rename` and `task_newtask` tracepoints. `task_rename` probes are triggered for each new process created and `task_newtask` for a new thread. As new tasks are created, `total_count` is incremented. Likewise, a probe attached to `sched_process_exit` tracepoint decrements `total_count`, when a thread/process exits. Hence at any time, `total_count` represents the total number of tasks in the application.

3.3.2 *CMetric* calculation of individual time-slices

Calculating the *CMetric* requires recording time-slice duration along with the number of active threads. A time-slice is demarcated by the instances when a thread is switched in and switched out, and this can be recorded in the `sched_switch` tracepoint probe. However, `thread_count` can fluctuate throughout a thread's time-slice, each change triggered by another thread being switched in or out. Since such events are associated with threads on a different core, it is not possible for a thread to monitor all context-switches that happen during its execution time-slice. Nevertheless, to calculate the *CMetric* of a time-slice, we just need the cumulative sum of the *CMetric* of all switching intervals during that time-slice. To achieve this, at every context-switch event that involves at least one of the application threads, `global_cm` (Table 3.1) is updated from the `sched_switch tracepoint` probe thus:

```
global_cm += (t - t_switch)/thread_count
```

where `t` is the timestamp of the current switching event and `t_switch` is the timestamp of the previous switching event. Notice that `(t - t_switch)` corresponds to the length of the latest switching interval (T_i for some i in Figure 3.4).

If the thread being switched in belongs to the application, then we store the value of `global_cm` to prepare for the next update to `cm_hash`, viz. `local_cm = global_cm`.

When an application thread is switched out, the `cm_hash` entry for that thread (identifier `prev_pid`) is then updated thus:

```
cm_hash[prev_pid] += global_cm - local_cm
```

Note that if n_i is the number of active application threads in the i^{th} switching interval and the current time-slice spans the switching intervals between j and k inclusive, then the right-hand side above is equivalent to $\sum_{i=j}^k \frac{T_i}{n_i}$, as required.

3.4 Bottleneck Detection from Thread Criticality

A possible way to identify critical time-slices is to record and rank all time-slices based on their *CMetric*. Time-slices can be grouped according to the call paths retrieved at the end of a time-slice. The *CMetric* of similar call paths can then be combined to evaluate their cumulative effect on the overall *criticality* of the application.

However, the profiler can add substantial overhead to the application by extracting call paths for every time-slice. Moreover, time-slices that execute with higher degree of parallelism are unlikely to be representatives of serialization bottlenecks and hence can be omitted. Keeping these points in mind, time-slices and their *CMetric* need to be recorded only if the number of active threads during a time-slice, N , is less than some threshold value, N_{min} , say. However, the number of threads can fluctuate during a time-slice, and hence we need to calculate the average number of threads during a time-slice, weighted by the duration of each switching interval.

The weighted average of the number of active threads in a time-slice is calculated similar to *CMetric*, with the help of eBPF maps, `global_wt_tc` (global weighted thread count), `local_wt_tc` (local weighted thread count) and `in_TSP`. On every switching event E_i , $(T_{i-1} * n_{i-1})$ is added to `global_wt_tc`, where n_{i-1} represents the number of active threads during

switching interval T_{i-1} (see Figure 3.4). Besides, whenever application threads are switched in, they store the switched-in time stamp to `in_TSP` as well a snapshot of `global_wt_tc` to `local_wt_tc`. Both `in_TSP` and `local_wt_tc` are per-cpu local scalars.

Consequently, if a thread is switched out at timestamp `out_TSP`, then the weighted average thread count (`thread_wt_avg`) during its time-slice can be obtained thus:

$$\text{thread_wt_avg} = \frac{\text{global_wt_tc} - \text{local_wt_tc}}{\text{out_TSP} - \text{in_TSP}}$$

To summarize, the `sched_switch` tracepoint probe performs the following actions if either or both threads involved in the context-switch belongs to the application:

- Mark the thread as active and increment `thread_count` if switched-in from an inactive state.
- Mark the thread as inactive and decrement `thread_count` if switched-out to an inactive state.
- Calculate the *CMetric* for every *switching interval*.
- Calculate the weighted average thread count and retrieve the user stack trace if `thread_wt_avg < Nmin`, when a thread is switched out.
- Convey critical stack traces to the user space by writing them to the ring buffer.

The detailed algorithm of the `sched_switch` probe function is provided in Algorithm 1.

When a waiting thread is woken up, it will eventually be switched in, but there may be a delay between the wake-up and the context switch. This can happen especially when the test machine is heavily loaded. During this time the thread is runnable, so it should be marked as active in `thread_list` as soon as it is woken up. We therefore also trace wake-up events by attaching a probe function to the `sched_wakeup` tracepoint. This increments `thread_count` if the thread being woken up belongs to the application. In effect, `thread_count` is managed together by the tracepoint probes attached to `sched_switch` and `sched_wakeup`.

Algorithm 1 sched_switch tracepoint probe

Input: prev_pid, next_pid, prev_state, next_state

```

1: if (prev_pid  $\notin$  thread_list)  $\wedge$  (next_pid  $\notin$  thread_list) then
2:   return
3: end if
4: oldTimeStamp = tsp[0]
5: newTimeStamp = currentTime()
6: tsp[0] = newTimeStamp
7: switchInterval = newTimeStamp - oldTimeStamp
8: prev_threadCount = thread_count[0]
9: intervalCMetric =  $\frac{\text{switchInterval}}{\text{prev\_threadCount}}$ 
10: global_cm += intervalCMetric
11: wt_threadCount = switchInterval * prev_threadCount
12: global_wt_tc += wt_threadCount
13: if prev_pid  $\in$  thread_list then
14:   if prev_state  $\neq$  TASK_RUNNING then
15:     thread_count -= 1
16:     thread_list[prev_pid] = 0 {Update thread status as inactive}
17:   else
18:     thread_list[prev_pid] = 1
19:   end if
20: timeSliceCM = global_cm - local_cm
21: cm_hash[prev_pid] += timeSliceCM
22: timeSlice = newTimeStamp - in_TSP
23: thread_wt_avg =  $\frac{\text{global\_wt\_tc} - \text{local\_wt\_tc}}{\text{timeSlice}}$ 
24: if thread_wt_avg <  $N_{min}$  then
25:   Get user stack and write to ring buffer the stackId, threadId, timeSliceCM and whether
   thread_count of last switching interval was less than  $N_{min}$ .
26: end if
27: end if
28: if next_pid  $\in$  thread_list then
29:   if thread_list[prev_pid] == 0 then
30:     thread_count += 1 {Increment thread count if switched-in from inactive state}
     thread_list[prev_pid] = 1
31:   end if
32: local_cm = global_cm
33: in_TSP = newTimeStamp
34: local_wt_tc = global_wt_tc
35: end if

```

3.4.1 Locating lines of code that contribute to the bottleneck

The scheme to detect bottlenecks from stack traces retrieved at the end of critical time slices works well if threads happen to be executing bottleneck code when a context switch happens. However, in many cases bottleneck code can complete its execution before the next context-switch event, i.e. before the next stack trace retrieval opportunity. In such cases, stack traces taken at context switches alone are often not enough to pinpoint the root cause of a bottleneck.

This problem can be addressed with an additional lightweight periodic sampling probe, with period Δt , to record the instruction pointer and pass it to user space if the absolute number of active threads, as given by `thread_count`, is less than N_{min} . Again, the probe first checks that the executing thread belongs to the application. If these conditions are met, then the thread id and instruction pointer are written to the same eBPF circular buffer referred to in Section 3.2 (Kernel tracing with eBPF and *BCC*). The sampling probe executes with a default period of $\Delta t = 3\text{ms}$. This was chosen as it was around half the average time-slice duration, which will result in around two additional sample probe points for every time-slice, on average.

3.4.2 User-space probe

Data retrieved by the kernel probes are written to a ring buffer, maintained by the Linux tracing subsystem, and shared between the kernel and user space. A user space probe that runs in parallel with the kernel probes, polls, retrieves and processes data from the ring buffer. This probe is implemented using the Python front end provided by the *BCC* tool kit. *BCC* provides a tool chain for writing, compiling and attaching eBPF programs as well as exports a set of APIs for reading and processing data from the kernel.

Processing data from the kernel

The ring buffer contains data written by the `sched_switch` tracepoint probe and the sampling probe described in the previous sections. Listed below are the data fields transferred from the

kernel probes to the user through the ring buffer:

- `sched_switch` tracepoint probe
 1. Thread ID
 2. *CMetric* of the timeslice
 3. User Stacktrace ID
 4. A flag to indicate if the last switching interval exhibited reduced parallelism (the reason is explained in Section 3.4.2 - Critical time-slices with no samples).
- sampling probe
 1. Thread ID
 2. Instruction pointer at the sampled instance.

Instruction pointers from the sampling probe are queued in a hash map indexed by the thread id. When data from the `sched_switch tracepoint` probe is received for the same thread id, the information accumulated is used to populate three locally-managed hash maps, indexed by a unique id, `ts_id`, associated with the time-slice. These hash maps contain the *CMetric*, the call path corresponding to the stack trace, and a list of addresses from the sampling probe (these addresses are the candidates for bottleneck lines of code).

If a stack trace is *not* triggered at the end of a time-slice, then a special entry is written to the circular buffer, along with the current thread id, which instructs the user probe to reject any instruction pointers from sample probes associated with that thread.

Post processing phase

When the program terminates, the user probe enters a post-processing phase which seeks to determine the instruction pointers that have the highest collective *CMetric*. This is achieved by combining the *CMetric* and instruction pointers of identical call paths. If *i* and *j* are two

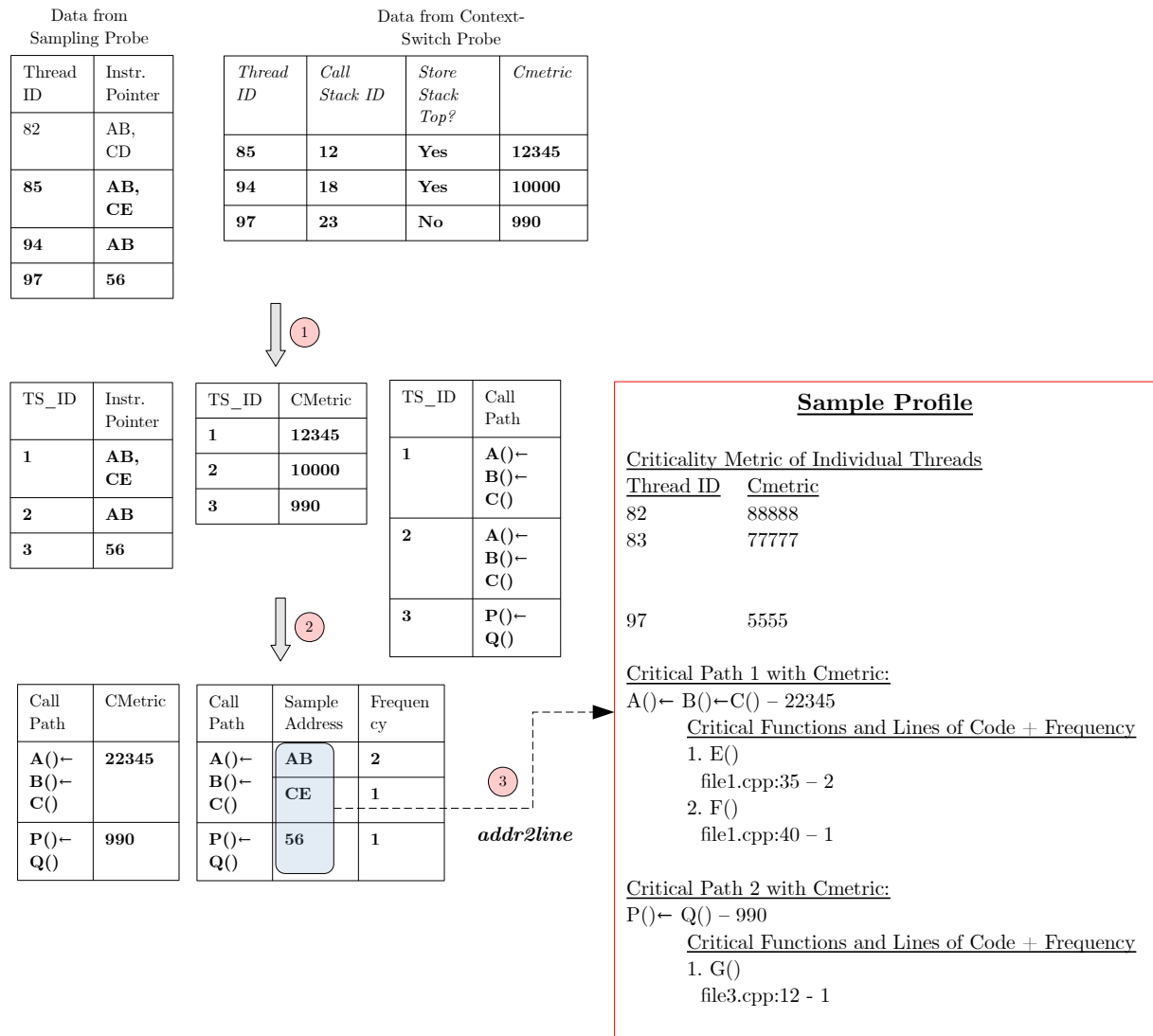


Figure 3.6: Post-processing and sample profile

`ts_id` indices with identical call paths, then the corresponding entries in the address lists and *CMetric* map are *merged* by:

1. summing the *CMetric* values at indices `i` and `j` and
2. combining the addresses at indices `i` and `j` to generate a frequency table for those addresses.

The merge process generates two hash maps, each indexed by a unique call path; one with the accumulated *CMetric* for the call path and the other with the list of sampled addresses for the same call path. The entries with the top N total *CMetrics* are then taken as the bottlenecks. The steps involved in the post processing phase are illustrated in Figure 3.6.

The reason why the top N entries are chosen instead of the top one alone is because one call path could be a subset of another, as would occur if a context-switch happened during execution of one function that indirectly calls another that contains a bottleneck.

Finally, the addresses in the sample list are mapped to function names and the lines of code associated with them by calling the Linux `addr2line` utility. The final profile is presented as a frequency table of functions and lines of code for each critical path, as illustrated in Figure 3.6. The profile also lists the *CMetric* for individual threads/processes in the application.

Critical time-slices with no samples

Sometimes, a time-slice may be identified as critical, but the sampler may fail to gather instruction pointer(s) that define the location of the bottleneck. This can happen due to two reasons: either the time-slice was too short, so that the sampler missed it, or the sample belonged to a shared library or kernel code and hence could not be mapped to the source code. In the absence of any samples, the next best thing is to add the address at top of the stack, i.e., the return address of the caller, to the list of samples. In summary, the stack top address is attached to the samples, if *a.* the sample count is zero and *b.* the active thread count is less than or equal

to N_{min} , when the thread is switched out. Such samples are labelled as being from stack top to help the user interpret results correctly.

Optimizing the performance of user-space probe

The user space probe runs in parallel with the application and hence consumes computing power that can otherwise be used by the application. Hence it is imperative to minimise its computation and thereby the overhead induced on the application being traced. The major operations performed by the user space probe are:

1. retrieving call path from the stack trace and
2. mapping instruction pointers to source code.

Since stack traces are retrieved only from potential bottlenecks, critical call paths are likely to be repeated. This also implies that there is a greater probability for addresses in the stack trace to be identical. Hence addresses and their symbols are stored in a look-up table to speedup address to symbol mapping.

3.5 Using GAPP

GAPP provides a set of command line options to customize the information gathered during tracing. To trace an application, GAPP should be initiated before starting the traced application using the command:

```
python GAPP.py -x <path to executable file>
```

The path to the traced executable file is mandatory, and is specified using the `-x` option. When executed without any other options, GAPP uses the default values set for them. Table 3.2 summarizes the different command line options available with GAPP and their default values. The effect of varying these options are discussed in Sections 4.4 and 4.5.

Option	Description	Default value
<code>-x</code>	Path to the executable file to be profiled	
<code>-t</code>	Threshold value to trigger stack trace	$\frac{total_threads}{2}$
<code>-f</code>	Sampling frequency	333 Hz (3 ms)
<code>-d</code>	Maximum stack depth in stack traces	10
<code>-b</code>	Size of ring buffer in pages	64
<code>--thread_only</code>	Trace threads, omitting processes	False
<code>--process_only</code>	Trace processes, omitting threads	False
<code>--trace_lib</code>	Include stack traces from dynamic shared libraries	False
<code>--kernel_stack</code>	Retrieve kernel stack traces of blocking critical paths	False

Table 3.2: Command line options of GAPP

Chapter 4

GAPP - Evaluation

The objective of this chapter is to evaluate GAPP's ability to identify serialization bottlenecks in parallel Linux applications. A number of experiments were designed with a wide variety of applications, for example, data parallel, task parallel, MPI applications etc. In each experiment, results were analyzed to identify the cause and possible fixes for the bottleneck. The steps for fixing the bottleneck were straightforward in some cases, whereas it involved further analysis in some others. These are explained in detail for each case. The effect of fixing the bottlenecks was quantified by comparing the execution time of the optimized application with its original, unmodified version.

This chapter describes in detail the experiments performed to:

1. Evaluate GAPP's ability to identify serialization bottlenecks in various parallel Linux applications
2. Quantify the overhead of the tool for various applications tested
3. Explore manual fixes to bottlenecks and
4. Explore the sensitivity of the tool to various configuration parameters.

4.1 Experimental Setup

Experiments were performed on a server machine with four AMD Opteron 6282SE eight-core/16-thread CPUs (with a total capability to run 64 threads in parallel) and 128GB of RAM. The system runs on Linux 4.15 with the `bcc` toolkit installed. The applications were compiled with `gcc` version 7.3.0 at the `-O3` optimization level.

The compiler options `-g` and `-fno-omit-frame-pointer` were enabled to aid effective call stack retrieval. Recent versions of `gcc` generate position-independent executable by default. To map addresses to source code using `addr2line`, this behaviour needs to be overridden through compiler and linker options `-fno-pie` and `-no-pie` respectively.

GAPP was evaluated using 11 applications from the Parsec-3.0 benchmark suite and two large open source projects: *MySQL* and *Nektar++* (a spectral/ hp element framework for solving Partial Differential Equations). All applications except *Nektar++* are multithreaded; *Nektar++* is a multiprocess application that uses the Message Passing Interface (MPI) for inter-process communication. The applications were written in either in C/C++, but used different threading primitives. Table 4.1 summarizes the results from the applications evaluated.

4.2 Multithreaded Applications

4.2.1 Parsec-3.0 benchmark suite

The applications from the Parsec 3.0 benchmark suite have been extensively used in various performance studies. It covers a diverse set of application domains such as computer vision, media processing, computational finance, content-based search, data compression etc. The algorithms implemented in this benchmark exhibit high computational demand, and represents state of the art techniques used in the respective areas [BKSL08].

Out of the 11 applications evaluated, two, namely *Dedup* and *Ferret*, are task-parallel and the remaining are data-parallel. In the data-parallel benchmarks every thread executes the same

Application	Critical Functions identified by GAPP	O/H	T (s)	CTS	M (MB)	PPT (s)
Blackscholes	CNDF()	< 1%	29.4	470 (2%)	109	0.02
Bodytrack	OutputBMP, RecvCmd	5%	21.3	6823 (0.5%)	112	0.4
Canneal	netlist_elem::swap_cost	2.2 %	62	267 (0.06%)	112	0.1
Dedup	deflate_slow	12%	13	362544 (40%)	372	3.3
Facesim	Update_Position_Based_State_Helper	4.4%	59	334 (0.004%)	118	0.8
Ferret	dist_L2_float	3%	30	42127 (51%)	132	1.2
Fluidanimate	parsec_barrier_wait	2%	37	11512 (1%)	112	0.3
Freqmine	FPArray_scan2_DB	2%	34	11721 (13%)	110	0.6
Streamcluster	parsec_barrier_wait, dist	5.6%	201	2246172 (10.6%)	784	2.5
Swaptions	HJM_SimPath_Forward_Blocking	1%	8.6	43 (0.07%)	111	0.2
Vips	imb_LabQ2Lab	4%	14	10460 (3.2%)	118	2.3
MySQL	fil_flush, sync_array_reserve_cell	< 1%	60	825 (0%)	194	3
Nektar++	dgemv_	9%	37	189990 (16%)	446	2.1

Table 4.1: Critical functions identified with overhead (O/H), execution time in seconds (T), Number and percentage of critical time-slices (CTS), memory usage in MB (M) and post-processing time in seconds (PPT).

code. For the task-parallel benchmarks, execution is divided into several stages/phases with different logic in each stage. Sometimes a particular phase is being referred to as a bottleneck, meaning that the bottleneck code is associated with that phase. All applications were executed with 64 threads.

GAPP was able to identify not only previously established bottlenecks, but also several that have not been reported previously.

We compare the results with those from previous studies and show that GAPP is able to detect bottlenecks that arise from a variety of sources, such as synchronization primitives, workload imbalance, busy-wait loops and memory operations. Moreover, it is able to direct the developer to the block of code that is causing the bottleneck in both data parallel and task parallel applications.

Blackscholes

Blackscholes is a data parallel financial application that solves the *Black-Scholes* partial differential equation to calculate the prices for a portfolio. It does not use any locks or barriers and hence there is no dependency between threads. Normally, a parallel application without any user-space shared resources should not have any serialization bottlenecks, except, maybe some straggler thread due to load imbalance.

Figure 4.1 shows the critical path and the number of critical time slices identified by GAPP. The profile shows that approximately 2.7% of the context switches were critical, which means those time slices exhibited reduced parallelism. Moreover, threads exhibited significant variance in their *CMetric* (Figure 4.2). However, the critical paths or the critical lines of code did not provide any useful information from which the bottleneck could be identified. Repeating the experiment enabling kernel stack traces of critical time slices (with the `--kernel_stack` option of GAPP) revealed that threads were blocked on a read-write semaphore accessed while handling page-faults (Figure 4.3).

In a multithreaded application, the address space is shared among threads, and hence, concur-

```

bs_thread()[blackscholes]
<---start_thread ()[libpthread-2.27.so] --301893617, 592.

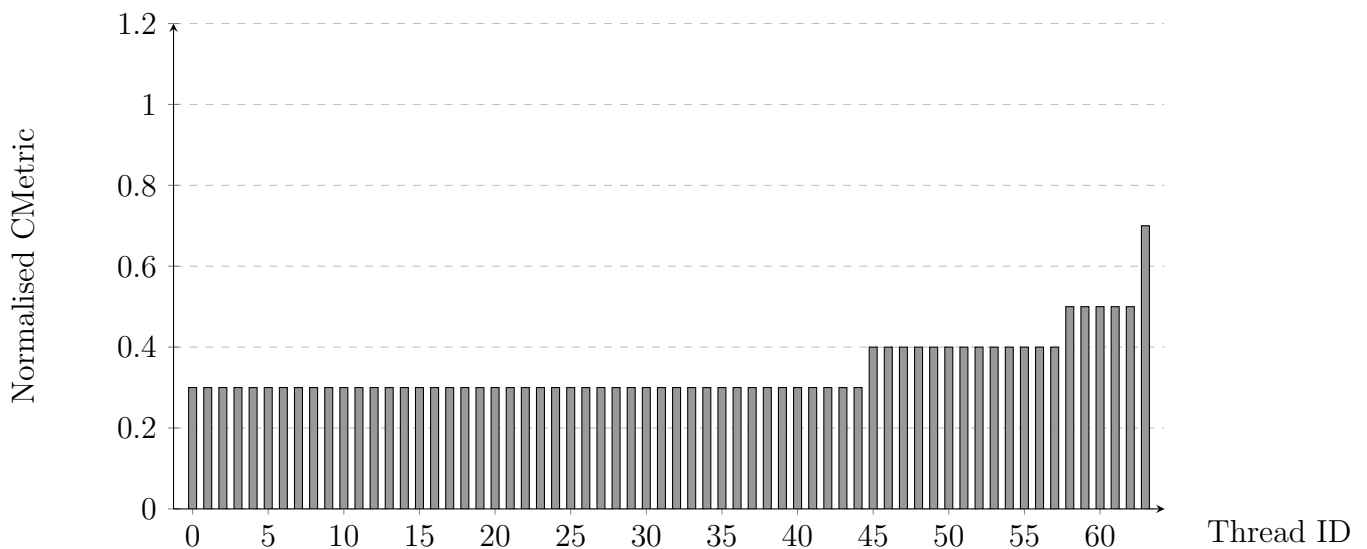
Functions and lines + Frequency
-----

bs_thread(void*) -- 386
    blackscholes.m4.cpp:381 (StackTop) -- 261
    blackscholes.m4.cpp:377 -- 65
    blackscholes.m4.cpp:377 (StackTop) -- 35

CNDF(float) -- 241
    blackscholes.m4.cpp:185 -- 42
    blackscholes.m4.cpp:216 -- 25
    blackscholes.m4.cpp:196 -- 18

*****
Total switches: 22241 Critical switches: 593
*****

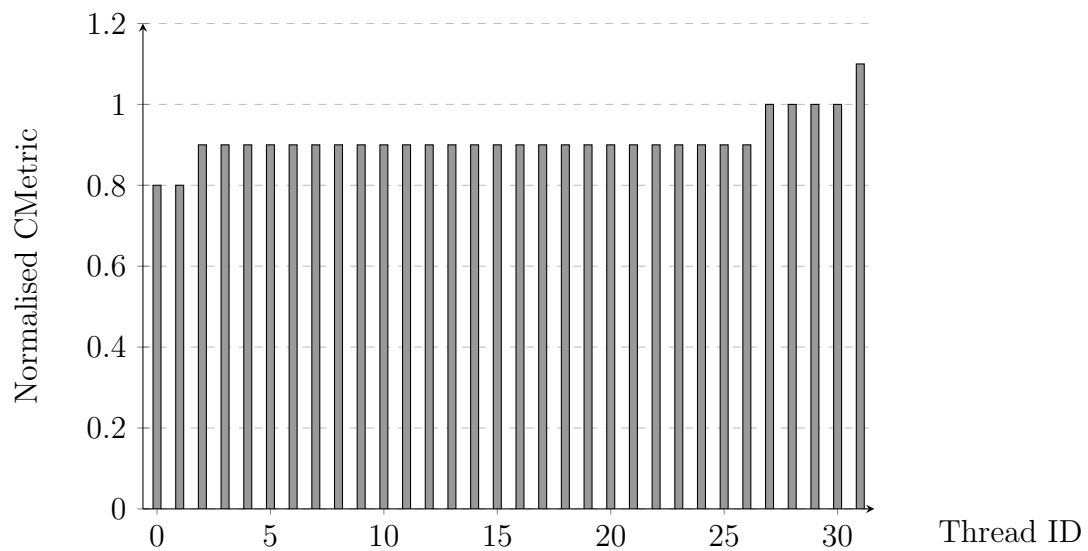
```

Figure 4.1: Critical path - *Blackscholes*Figure 4.2: *Blackscholes* - *CMetric* (64 Threads)

```

Kernel Call Paths
-----
__schedule
<---schedule
<---rwsem_down_read_failed
<---call_rwsem_down_read_failed
<---down_read
<---__do_page_fault
<---do_page_fault
<---page_fault -- 164

```

Figure 4.3: Kernel call paths - *Blackscholes* (64 threads)Figure 4.4: *Blackscholes* - *CMetric* (32 Threads)

rent address space operations such as page faults, memory related system calls etc. need to be synchronized by the kernel virtual memory subsystem. Linux achieves this using a single, per process read-write lock, which limits the scalability of virtual memory intensive applications [ZC18]. The same problem is being observed here, where threads block while trying to access the shared virtual address space.

An easy fix for this issue is to reduce the number of concurrent memory operations by decreasing the number of threads in the application. However, the performance improvement achievable in that case depends on the severity of the contention, when compared to the computational load in the application. Executing *Blackscholes* with 32 threads marginally increased the execution time from 29.2 secs to 30.2 secs (by circa 3%). However, the number of critical time slices reduced from circa 2.7% to nearly 0.3% of the total context switches. Moreover, while the percentage difference between the lowest and maximum *CMetric* with 64 threads was around 133%, the same for 32 threads dropped to nearly 37.5%. This shows that executing the same input with 32 threads resulted in a more uniform workload, without much effect on the application's performance, showing *Blackscholes*'s limited scalability with 64 threads.

A question that may arise is whether this bottleneck can be revealed by existing tools. Since *Blackscholes* does not have any shared user-space resources, this bottleneck will not be revealed by tools that instrument user-space synchronization primitives. Since the bottleneck is not computational, sampling based profilers are not useful either. A possible class of tools that may identify similar bottlenecks are those which trace wait events in the kernel and retrieve kernel stack traces.

Bodytrack

Bodytrack is a vision application that tracks the 3D pose of a marker-less human body through image sequences captured with multiple cameras . The application follows a producer-consumer paradigm where the worker threads process video frames as per the commands from the parent thread.

Critical Path 1:

```
-----
void FlexDownSample2<unsigned char>()[bodytrack]
<---TrackingModel::OutputBMP()[bodytrack]
<---mainPthreads()[bodytrack]
<---main()[bodytrack]
<---__libc_start_main()[libc-2.27.so] --2875482321.
```

Functions and lines + Frequency

```
-----

bool FlexFilterRowV() -- 284
    FlexFilter.h:63 -- 133
    FlexFilter.h:62 -- 127
    FlexFilter.h:58 -- 19

bool FlexFilterColumnV() -- 241
    FlexFilter.h:88 -- 115
    FlexFilter.h:89 -- 92
    FlexFilter.h:84 -- 33
```

Figure 4.5: Top critical path - *Bodytrack* (Profile including the parent thread)

Top Critical Functions and lines of code with frequency

```
-----

threads::WorkerGroup::RecvCmd() -- 4639
    WorkerGroup.cpp:101 (StackTop) -- 4639

ParticleFilterPthread<TrackingModel>::Exec() -- 429
    TicketDispenser.h:98 (StackTop) -- 425
    ParticleFilterPthread.h:136 (StackTop) -- 2
    ParticleFilter.h:132 (StackTop) -- 2

TrackingModelPthread::Exec() -- 342
    TicketDispenser.h:99 (StackTop) -- 213
    TicketDispenser.h:101 (StackTop) -- 128
    TicketDispenser.h:98 (StackTop) -- 1
```

Figure 4.6: Top critical function - *Bodytrack* (Profiling worker threads alone)

With *Bodytrack*, GAPP identified the `OutputBMP` function executed by the parent as the top critical function (Figure 4.5). When the worker threads alone were profiled (with the `--threads_only` command line option), GAPP identified the `RecvCmd` function as the top critical function (Figure 4.6). In `RecvCmd`, the worker threads wait on a conditional variable until they receive commands from the parent thread.

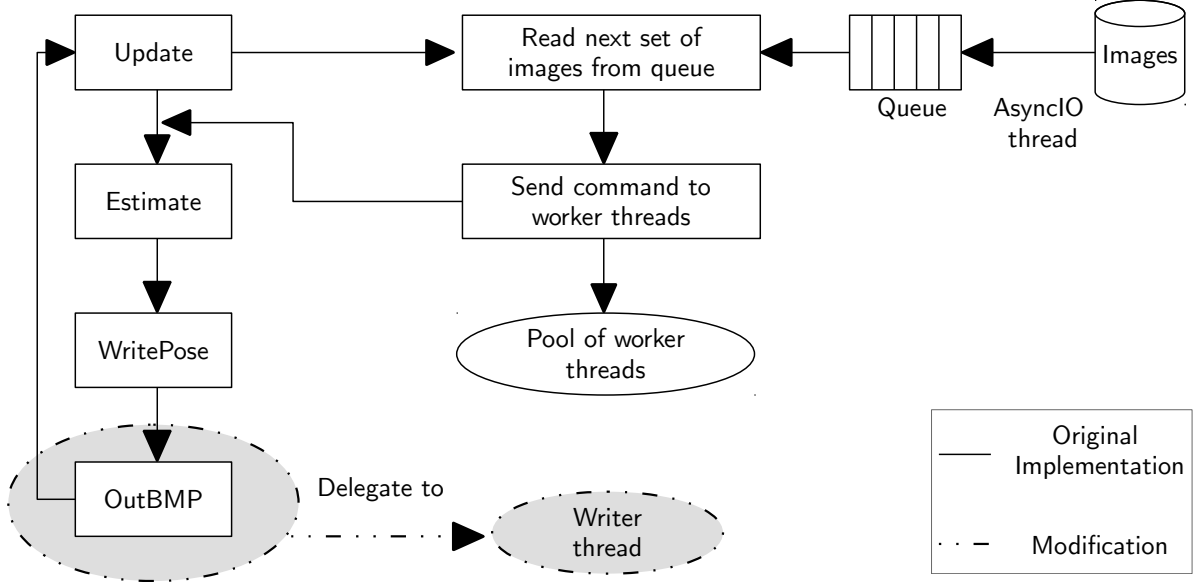


Figure 4.7: General logic of *Bodytrack*

The `OutputBMP` function invoked by the parent thread generates the camera image and saves it in BMP format. While the images are processed by `OutputBMP`, the worker threads are waiting on the conditional variable in the `RecvCmd` function. Profiling the application after commenting out `OutputBMP` showed approximately 45% reduction in the number of samples from `RecvCmd`. This confirms that the `OutputBMP` function is indeed the bottleneck.

Commands can be passed to worker threads at a faster rate by speeding up the `OutputBMP` function. As an alternative, we offloaded the `OutputBMP` function from the parent thread to a new thread called *writerThread*. Consequently, the computational load on the parent was reduced and it started sending commands to the worker threads at a faster pace, thereby reducing the worker’s waiting time. This modification improved the performance of *Bodytrack* by circa 22%. Figure 4.7 outlines the logic of *Bodytrack*, before and after optimization. This bottleneck has not been reported in the literature. However, tools that trace synchronization primitives might be able to identify this.

Dedup

Dedup is a task parallel application having five pipeline stages, with a queue connecting each stage to the next one. The stages are: *Fragment*, *FragmentRefine*, *Deduplicate*, *Compress* and *Reorder*. The first and last stages perform I/O with a single thread. The rest parallelize the task among a pool of worker threads. By default, *Dedup* is configured to run with equal number of threads in the intermediate stages. For our experiments, threads were allocated as 1-20-20-20-1 (a total of sixty-two threads in the application).

With the initial thread allocation of 1-20-20-20-1, the *Compress* and *Reorder* phases displayed higher *CMetric* when compared to the other phases. The sequential phase, *Reorder*, which writes the compressed chunks to a file, is known to be a bottleneck [DSDMT+17]. Since modifying the *Reorder* serial phase would require altering the application design, we focused on the bottleneck in the parallel stage, *Compress*.

The *Compress* stage's higher *CMetric* can be a result of higher computational load or lower parallelism in it. If the bottleneck is caused due to computational load, then increasing the parallelism in that stage will improve the application's performance. Hence, we moved threads from the *FragmentRefine* and *Deduplicate* phases to the *Compress* phase, and ran the experiment with 1-16-16-28-1 threads in the respective phases (threads were moved from other phases so that the total number of threads remain the same). However, increasing parallelism in the *Compress* phase increased the execution time, indicating possible contention.

The top critical path identified by GAPP pointed to the `deflate_slow()` function in the *zlib* compression library (Figure 4.8), invoked from the *Compress* phase. A possible reason for its higher *CMetric* could be some blocking operation in the compression logic. However, the *Compress* phase did not use any synchronization primitives, and hence, we re-profiled *Dedup* with GAPP, enabling kernel stack traces. The kernel stack traces revealed that the `deflate_slow()` function exhibited large number of page-faults that resulted in threads being blocked on a per-process semaphore used to serialize concurrent memory mapping operations and page-faults, as explained in Section 4.2.1-*Blackscholes*. As this indicates a scalability issue,

Critical Path 1 -- CMetric, Frequency

```
-----
deflate_slow ()dedup
<---deflate ()dedup
<---compress ()dedup
<---Compress ()dedup
<---start_thread ()libpthread-2.27.so --1771156682, 212225.
```

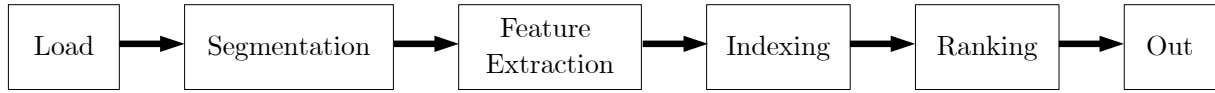
Functions and lines Frequency

```
-----
deflate_slow -- 211135
    deflate.c:1650 (StackTop) -- 179558
    deflate.c:1580 (StackTop) -- 25779
    deflate.c:1635 (StackTop) -- 1554
```

Kernel Call Paths with frequency

```
-----
__schedule
<---schedule
<---rwsem_down_read_failed
<---call_rwsem_down_read_failed
<---down_read
<---__do_page_fault
<---do_page_fault
<---page_fault -- 212066
```

Figure 4.8: GAPP profile - *Dedup* with thread allocation- 1-20-20-20-1

Figure 4.9: Pipeline stages - *Ferret*

we reduced the number of threads in the *Compress* phase from 20 in the initial configuration to 15, and re-ran the program with the thread configuration: 1-20-20-15-1. This in turn improved the execution time by approximately 14%. There might be other thread allocations that provide better performance improvement. However, the focus of this experiment is to show that GAPP reports real bottlenecks that provide positive performance improvement and guides the user to the reason behind the bottleneck, even without heavy instrumentation. Moreover, some of the bottlenecks identified are platform-specific and hence bottleneck detection tools that instrument application or threading libraries may not identify them.

Ferret

Ferret is a task parallel application that implements content-based similarity search on images, audio, 3D shapes etc, and is implemented using six pipeline stages. The first and last phases, which perform I/O, are serial phases. The middle four stages implement query image segmentation, feature extraction, indexing and ranking respectively. Figure 4.9 shows the different pipeline stages in *Ferret*.

As in the case of *Dedup*, *Ferret*, by default, is configured to run with equal number of threads in its parallel phases. *Ferret* was profiled while executing with 15 threads in each of the parallel phases, making a total of 62 threads for the whole application. The top critical functions identified by GAPP reveals that they were invoked from the `emd()` (Earth Mover's Distance) function, which finds the pair-wise distance between query image and candidate images, and also forms the core of the *ranking* phase (Figure 4.10).

Ferret exhibited substantial imbalance in *CMetric* among the threads (see Figure 4.11). The threads with the highest *CMetric* belonged to the *ranking* phase, followed by threads in the

Critical Path 1:

```

-----
    emd ()
<---sdist_emd ()
<---raw_query ()
<---cass_table_query ()
<---t_rank ()
<---start_thread ()

Functions and lines + Frequency
-----
isOptimal -- 41314
    emd.c:422 -- 20813
    emd.c:423 -- 10760
    emd.c:420 -- 6657
findBasicVariables -- 41301
    emd.c:350 -- 7366
    emd.c:353 -- 6713
    emd.c:383 -- 5827

```

Figure 4.10: Top critical path, functions and lines of code - *Ferret*

indexing, *segmentation* and *feature extraction* phases respectively (threads in the plot are arranged in the order of the phases in the pipeline).

Reallocating threads from phases with lower *CMetric* to phases with higher *CMetric* until a uniform *CMetric* was obtained, improved the run-time of the application by approximately 50% (thread allocation: 2-1-18-39). Figure 4.11 shows how the *CMetric* gradually changes as threads are re-allocated. This bottleneck was also identified by Coz, which optimized the application by reallocating threads as 20-1-22-21 among the parallel phases. However, this thread configuration only provided approximately 23% speedup on the test bed. Even though Coz identifies the bottleneck, it doesn't have a mechanism to visualise the load imbalance among the phases. The workload imbalance among threads in **Dedup** and **Ferret** was reported in [LZM⁺17], which also suggests an optimal thread assignment for the two applications, on a machine with 32 threads.

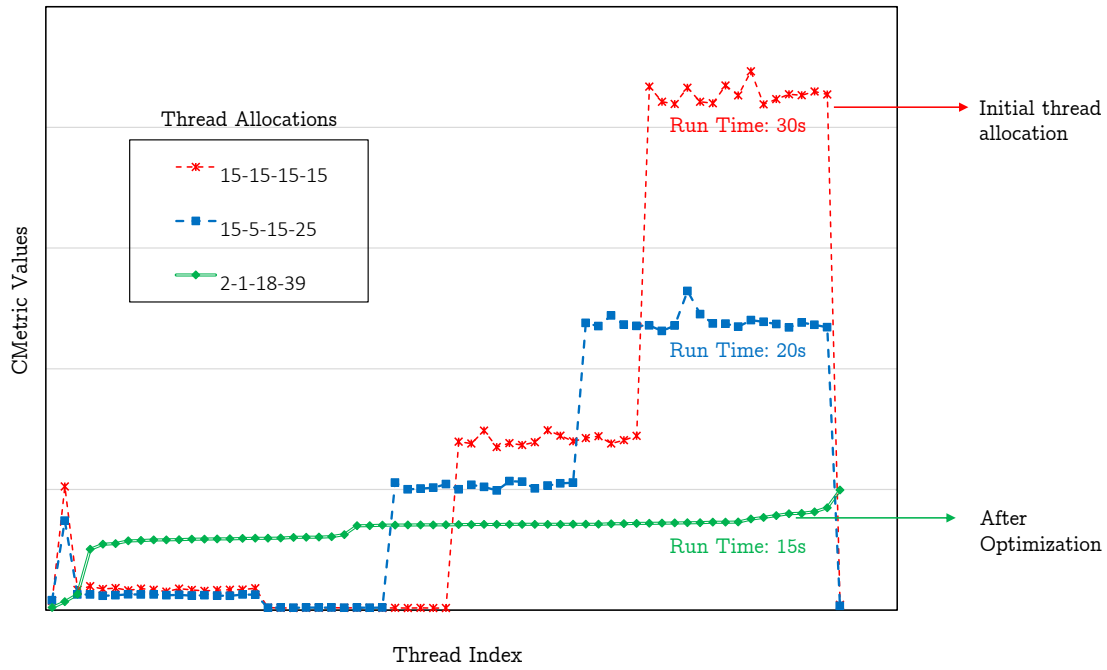


Figure 4.11: CMetric for different thread allocations - *Ferret*

Streamcluster

Streamcluster is a data parallel application that implements a clustering algorithm which measures the quality of clustering based on the sum of squared distances between points. It runs in six phases, with n threads in each phase, separated by a barrier. GAPP identified the `parsec_barrier_wait` and `dist` functions as the top bottlenecks in *Streamcluster* (Figure 4.12). Both these bottlenecks were reported previously in Coz [CB15] and Parashares [KTK14] respectively. It can be observed that the `parsec_barrier_wait` function was not explicitly listed under the critical functions of the most critical path, *Critical Path1* (Figure 4.12). Instead, the critical lines of code under *Critical Path 1* were addresses at the stack top, i.e., return addresses, as evident from the *StackTop* label associated with them. This happens when critical stack traces do not have any samples associated with them in which case the return address is added to the list of samples (see Section 3.4.2 - Critical time-slices with no samples). The lines of code, `streamcluster.cpp:1106`, `1042` and `1120`, marked with *StackTop*, were preceded by calls to `parsec_barrier_wait`. This shows that the bottleneck was associated with `parsec_barrier_wait`.

Critical Path 1:

```
pgain()streamcluster
<---pkmedian()streamcluster
<---localSearchSub()streamcluster
<---start_thread ()libpthread-2.27.so
```

Functions and lines with Frequency

```
pgain(long, Points*, double, long*, int, parsec_barrier_t*) -- 262917
    streamcluster.cpp:1106 (StackTop) -- 50717
    streamcluster.cpp:1042 (StackTop) -- 35328
    streamcluster.cpp:1120 (StackTop) -- 31023

dist(Point, Point, int) -- 96329
    streamcluster.cpp:653 -- 72967
    streamcluster.cpp:653 (StackTop) -- 15714
    streamcluster.cpp:652 (StackTop) -- 6753
```

Top Critical Functions and lines of code with frequency

```
pgain(long, Points*, double, long*, int, parsec_barrier_t*) -- 262923
    streamcluster.cpp:1106 (StackTop) -- 50717
    streamcluster.cpp:1042 (StackTop) -- 35328
    streamcluster.cpp:1120 (StackTop) -- 31023

dist(Point, Point, int) -- 97549
    streamcluster.cpp:653 -- 73589
    streamcluster.cpp:653 (StackTop) -- 16306
    streamcluster.cpp:652 (StackTop) -- 6756

pspeedy(Points*, float, long*, int, parsec_barrier_t*) -- 1203
    streamcluster.cpp:762 (StackTop) -- 556
    streamcluster.cpp:815 (StackTop) -- 280
    streamcluster.cpp:828 (StackTop) -- 116
```

Figure 4.12: GAPP Profile - *Streamcluster* with 64 threads

`parsec_barrier_wait()` implements a custom barrier that spins for a certain number of iterations before blocking the thread. Replacing it by the default *Pthread* barrier as suggested in Coz [CB15] improved the run time by approximately 42%.

The second critical function under the most critical path, `dist()`, computes the Euclidean distance squared between two points using floating point operations. [YLB⁺12] suggests an optimization for the `dist()` function using loop unrolling, pre-fetching and SIMD instructions. However, this modification did not improve the performance of *Streamcluster* on the test bed, as the function was automatically optimized by the compiler. Still, the GAPP profile indicates that `dist()` executed with reduced parallelism.

As the `dist()` function did not use any synchronization primitives, but still exhibited reduced parallelism, we re-profiled *Streamcluster* enabling kernel stack trace retrieval (Figure 4.13). The majority of the kernel stack traces pointed to `futex_wait` calls; this is no surprise, as the majority of critical samples were from `parsec_barrier_wait()`. This reveals that threads became blocked after spinning inside the custom barrier, which justifies why replacing the custom barrier with the default *Pthread* barrier improved run time. The remaining kernel stack traces showed that threads blocked due to page faults (only the first two page-fault stack traces are shown in Figure 4.13). These page-faults either triggered conditional re-scheduling, which is mostly caused by code that loops for long in the kernel ([Cor14]), or resulted in NUMA migration, both of which caused threads to block.

The test bed has four sockets, with eight physical cores per socket, i.e, 32 physical cores in the system. Each core is capable of running 2 threads, thereby supporting 64 threads concurrently. The 64 virtual cores are arranged into 8 NUMA nodes (verified using `numactl -hardware`). NUMA migrations can be triggered due to thread contention, when threads are trying to access the same memory. Since the majority of page-faults were triggered from the same line of code, there is a strong possibility that memory contention triggered the page-faults. If so, reducing parallelism should reduce the contention. Re-running the modified *Streamcluster* (the version where the custom barrier was replaced with a *Pthread* barrier) with 32 threads improved the run time by approximately 54%, cf. 64 threads.

Critical Path 1:

```
pgain()streamcluster
<---pkmedian()streamcluster
<---localSearchSub()streamcluster
<---start_thread ()libpthread-2.27.so
```

Kernel Paths with Frequency

```
__schedule
<---schedule
<---futex_wait_queue_me
<---futex_wait
<---do_futex
<---sys_futex
<---do_syscall_64
<---entry_SYSCALL_64_after_hwframe -- 605530
```

```
__schedule
<---_cond_resched
<---__do_page_fault
<---do_page_fault
<---page_fault -- 518
```

```
__schedule
<---_cond_resched
<---wait_for_completion
<---stop_two_cpus
<---migrate_swap
<---task_numa_migrate
<---numa_migrate_preferred
<---task_numa_fault
<---__handle_mm_fault
<---handle_mm_fault
<---__do_page_fault
<---do_page_fault
<---page_fault -- 345
```

Figure 4.13: Top kernel stack traces for critical path1 - *Streamcluster* with 64 threads

When executed with 32 threads, *Streamcluster* displayed comparable execution time with the custom barrier and the *Pthread* barrier. However, in both cases, the number of blocking paths due to page-faults reduced by around 45%. This indicates that the bottleneck in the custom barrier, `parsec_barrier_wait()`, when executed with 64 threads, was a repercussion of the bottleneck due to page-faults in the `dist()` function. A tool that leverages synchronization primitives will identify the bottleneck in the *barrier*, however, it may not identify the real cause of this bottleneck, which was memory contention. These experiments substantiate that GAPP's generic approach helps to identify a wide variety of bottlenecks.

4.2.2 MySQL

MySQL 5.7 was profiled with GAPP while executing the `OLTP_Read_Write` workload from the *Sysbench* benchmark. The top critical samples were from `pfs_os_file_flush_func()`, which flushes the write buffers of a given file to disk. The call path shows that this function was invoked by *InnoDB*, the transactional storage engine for *MySQL* (Figure 4.14a.). Threads being blocked here means frequent disk I/O operations are happening. The disk I/O operations can be optimized by increasing the buffer pool size. By increasing the buffer pool size to 90GB (70% of the total system memory), as suggested in [mysb], the transaction rate (measured in transactions per second) improved by circa 19% and the average latency reduced by approximately 16%.

The second most critical samples were from a spin-wait loop in `sync_array_reserve_cell()` invoked from `rw_lock_s_lock_spin()`, as shown in Figure 4.14b.

Mutexes and read-write locks in *MySQL* are designed such that the threads poll for them for a short period of time before they block. This is because these locks are reserved only for a short period of time and hence blocking threads on every access increases overhead.

The duration of the spin-wait loop is determined by the number of `PAUSE` instructions in a loop, which is calculated based on a random number between 0 and `INNODB_SPIN_WAIT_DELAY` (a configurable constant). The default value of this constant is 6. A lower delay in spin locks

a. Critical Path 1:

```

fil_flush()[mysqld]
<--log_write_up_to()
<--trx_commit_complete_for_mysql()
<--innobase_commit()
<--ha_commit_low()
<--TC_LOG_DUMMY::commit()
<--ha_commit_trans()
<--trans_commit()
<--mysql_execute_command()
<--Prepared_statement::execute()

```

Functions and lines + frequency

```

-----
pfs_os_file_flush_func -- 1462
  os0file.ic:507 (StackTop) -- 1462

```

b. Critical Path 2:

```

sync_array_reserve_cell()
<--rw_lock_s_lock_spin()
<--pfs_rw_lock_s_lock_func()
<--row_search_mvcc()
<--ha_innobase::index_read()
<--handler::ha_index_read_idx_map()
<--join_read_const_table()
<--JOIN::extract_func_dep..dt_tables()
<--JOIN::make_join_plan()
<--JOIN::optimize()

```

Functions and lines + frequency

```

-----
sync_array_reserve_cell() -- 469
  sync0arr.cc:389 (StackTop) -- 469

```

Figure 4.14: Critical paths - *MySQL 5.7*

can cause frequent cache invalidation when the atomic variable in the spin loop is accessed repeatedly by individual cores. This problem is exacerbated when the number of cores is large. [mysa] advises to increase the `INNODB_SPIN_WAIT_DELAY` constant to improve performance. Setting the constant to 30 cumulatively improved the transactions rate by approximately 34% and reduced the average latency by circa 25%. Moreover, the number of cache misses decreased by approximately 10.5%, when compared with the default delay variable value. This shows better cache performance with increased spin-wait delay. Interestingly, optimising the spin-wait delay *without* first optimising the buffer size made negligible difference to the overall performance. This emphasises the importance of ranking the bottlenecks by criticality and tackling each in turn.

The bottlenecks detected in *MySQL* were fixed by tuning configuration parameters, rather than re-factoring the source code. Existing *MySQL* performance tuning tools are designed to identify similar optimization opportunities. Nonetheless, the experiments show that GAPP, which is a generic tool, is capable of identifying similar architecture-dependent bottlenecks.

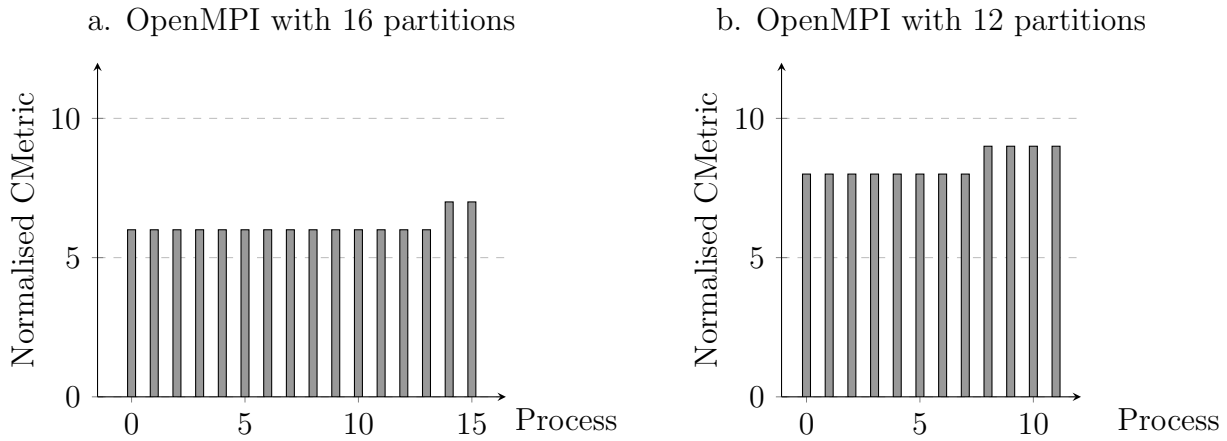


Figure 4.15: CMetric of individual processes before and after combining partitions - *Nektar++*

4.3 Parallel MPI Applications

Nektar++

GAPP's effectiveness on parallel MPI applications was evaluated with the **Nektar++** framework. **Nektar++** is a software framework designed for the development of high-performance scalable solvers for partial differential equations using the spectral/hp element method [CMC⁺15].

The **Incompressible Navier-Stokes solver** (IncNSS) provided with the framework, which simulates the flow of a viscous fluid on a surface, was chosen as the target MPI application. The solver was configured to run with 16 processes on a cylindrical surface. The solver divides a complex problem into several small elements represented as an unstructured mesh, partitions the mesh, and assigns each partition to an individual process.

The initial profile generated by GAPP for the IncNSS exhibited uniform *CMetric* for all processes (Figure 4.15a) and all critical samples were from the initialization and finalization phases of the application. We used the *top* command to compare the computational load of the MPI processes, and all 16 processes in the solver exhibited 100% CPU usage. In an optimal configuration, processes can exhibit 100% CPU usage when there is uniform load and almost zero waiting time associated with MPI communication. To verify this, we fabricated load imbalance among the processes by combining the last 5 partitions to a single partition and ran the solver with 12 processes, instead of 16. However, the individual processes exhibited uniform

Top Critical Functions and lines of code with frequency

```

dgemv_ () [libblas.so.3.8.0] -- 781

double Vmath::Dot2<double>() [libLibUtilities-g.so.5.0.0b] -- 170

gather_double_add () [libMultiRegions-g.so.5.0.0b] -- 100

dgemm_ () [libblas.so.3.8.0] -- 96

MPIDU_Sock_wait () [libmpi.so.12.1.1] -- 56

```

Figure 4.16: Critical functions - *Nektar++* with MPICH

CMetric (Figure 4.15b) and displayed 100% CPU usage with the *top* command.

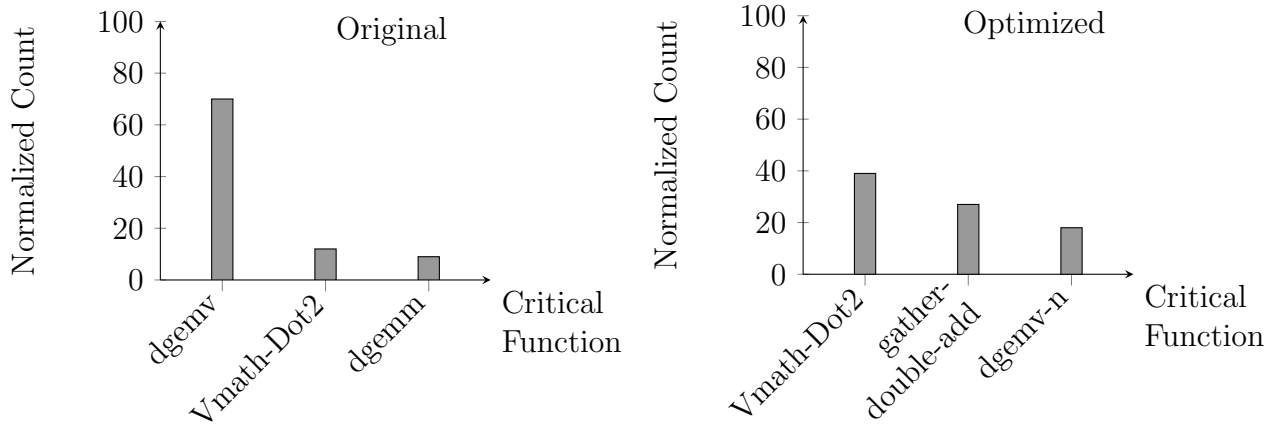
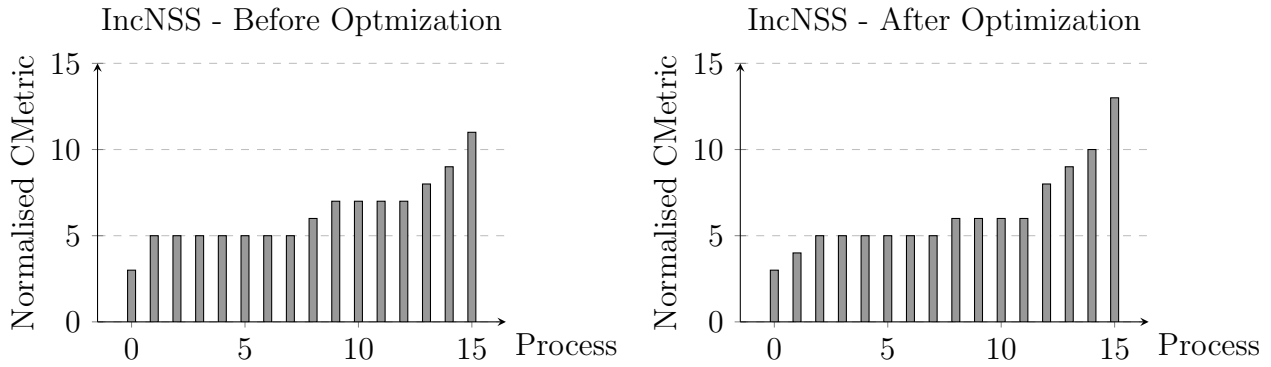
Message passing in *Nektar++* is implemented with *OpenMPI*, and the MPI processes are by default configured to run in aggressive mode. In this mode, rather than blocking waiting for messages, processes spin in a busy wait loop. This explains why processes exhibited uniform *CMetric* in the initial profile generated by GAPP. However, such behaviour masks any load imbalance among the processes as they are never idle.

OpenMPI can be configured to run in the *degraded* performance mode by enabling the `mpi_yield_when_idle` command line option, which yields processes, rather than busy-wait on MPI library calls. However, this had no effect on the CPU usage of the solver with the combined partitions, and the processes exhibited uniform *CMetric* as in Figure 4.15b. Similar issues have been reported with some versions of *OpenMPI*¹.

We recompiled *Nektar++* with *MPICH*, a different implementation of MPI standard. *MPICH* was built with the `--with-device=ch3:sock` option, which disables busy waiting. Consequently, process with the combined partition displayed a higher *CMetric*, indicating higher load when compared to others.

The profile of *IncNSS* recompiled with *MPICH* displayed `dgemv_()` from `libblas.so` and `Vmath::Dot2()` from `libLibUtilities-g.so`, both invoked from `solvePressure()`, as the top bottleneck functions (Figure 4.16). `dgemv_()`, is a matrix multiplication routine exported by the

¹<https://www.mail-archive.com/devel@lists.open-mpi.org/msg20407.html>

Figure 4.17: Bottleneck functions in *Nektar++* before and after optimizationFigure 4.18: CMetric of individual processes before and after optimization - *Nektar++*

BLAS library. By default, *Nektar++* is compiled with the system *BLAS* libraries. Recompiling *Nektar++* with *OpenBLAS*, an optimized version of the *BLAS* library improved the execution time of the application by approximately 27% and moved the bottleneck from `dgemv_()` to `Vmath::Dot2()` function (Figure 4.17). However, this modification did not improve the load imbalance among processes as observed in Figure 4.18. Note that Figure 4.16 does not list any critical lines of code, as the samples are from the shared libraries and hence could not be mapped to the source code of the executable.

With a structured mesh for a cuboid surface, uniformly partitioned among 8 processes, processes exhibited negligible variation in *CMetric*. This indicates that non-uniform partitioning caused the variation in *CMetric* among the processes.

The original and optimized versions of *Nektar++* was also profiled with the Linux *Perf* tool.

CPU Utilization	Shared Object	Symbol
42.66%	libblas.so.3.8.0	dgemv_
4.44%	libLibUtilities-g.so.5.0.0b	Vmath::Svtvp<double>
4.12%	libblas.so.3.8.0	dgemm_
2.51%	libLibUtilities-g.so.5.0.0b	Vmath::Dot2<double>
2.51%	libLibUtilities-g.so.5.0.0b	Vmath::Assmb<double>

Figure 4.19: *Perf* profile - Unoptimized version of *Nektar++*

CPU Utilization	Shared Object	Symbol
4.37%	libopenblas_bulldozerp-r0.3.8.dev.so	dgemv_n
1.76%	libLibUtilities-g.so.5.0.0b	Vmath::Assmb<double>
1.64%	libLibUtilities-g.so.5.0.0b	Vmath::Svtvp<double>
1.38%	libLibUtilities-g.so.5.0.0b	Vmath::Dot2<double>
0.77%	libMultiRegions-g.so.5.0.0b	gather_double_add

Figure 4.20: *Perf* profile - Optimized version of *Nektar++*

For the original version, *Perf* identified `dgemv_()`, the same function identified by GAPP, as the dominant function, contributing most (circa 43%) to the total execution time (Figure 4.19). This is an example where the dominant function in an application happened to be a serialization bottleneck executing with reduced parallelism. However, for the optimized version, the most expensive functions identified by *Perf* belonged to the *Pthreads* library, possibly invoked from *OpenBLAS* because *Nektar++* as such does not use any threads. Moreover, the critical function identified by GAPP, `Vmath::Dot2()`, was reported by *Perf* as contributing only around 1.38% to the total execution time (Figure 4.20). These experiments show that routines with lesser CPU utilization can also cause serialization bottlenecks and hence GAPP becomes a better tool when compared to generic profilers in identifying bottlenecks in such cases.

4.4 Sensitivity to Parameters

Threshold - N_{min}

For the experiments reported above, the threshold N_{min} was set to $\frac{n}{2}$, where n is the total number of threads in the application. With a lower threshold value of $\frac{n}{4}$, the number of critical

stack traces decreased. However, the critical functions and lines of code remained the same for the applications evaluated. A lower N_{min} will correctly pinpoint more ‘extreme’ bottlenecks, but is likely to miss situations where there is more modest loss of parallelism.

As with all of GAPP’s parameters, the user is free to experiment with other settings simply by modifying the command line parameter, $-t$, which stands for threshold. For example, $-t\ 8$ implies an N_{min} value of 8.

Sampling period

The sampling probe was set up with a period of $\Delta t = 3\text{ms}$ for the above experiments. This was chosen as it was around half the average time-slice duration, which will result in around two additional samples for every time-slice, on average. Reducing Δt to 1 ms did not produce any observable change in the bottleneck locations identified, but increased the frequency of bottleneck functions and lines of code, making them conspicuous. This change was more noticeable with applications that generated a limited number of stack traces. With $\Delta t = 1\text{ ms}$, there was only a marginal increase in the average overhead; from approximately 4% to 5%.

With a lower sampling rate of 10 msec, even though the critical functions and lines of code remained the same, sometimes the order of bottleneck functions changed as there were fewer samples from smaller functions. Even though the experiments with a lower sampling rate reported all the bottleneck functions, they can be missed, especially if bottleneck functions have very low latency.

Size of the ring buffer

The default size of the ring buffer is set as 8 pages by eBPF. With a sampling period $\Delta t = 3\text{ms}$, three applications, viz. *Dedup*, *Facesim* and *Streamcluster* caused the ring buffer to overflow. To prevent overflow, the buffer size had to be increased to 256, 16 and 4096 pages respectively, for each of these applications. This increased the memory usage of GAPP to 372 MB, 118 MB and 2.8 GB respectively in these cases. However, it was observed that even with lost samples,

the results remained the same as those generated with the increased buffer sizes. This is because samples are generated only from bottlenecks, and the more frequent/longer the bottleneck, the more samples are generated. To limit the memory usage, we have set the buffer size to 64 pages, which resulted in a maximum memory overhead of 782MB, for *Streamcluster*. However, this parameter, which should be a power of 2, is also configurable by the user through the command line option, *-b*.

Performance overhead

The advantage of GAPP when compared to existing parallel profilers is its low overhead and its ability to extend the scope to arbitrary libraries (the libraries should be compiled to enable stack traces) without any instrumentation, or significant overhead in execution time or memory footprint. For the 13 applications evaluated, the maximum performance overhead incurred was approximately 13% (see Table 4.1), which is substantially less when compared to the maximum overhead of similar parallel profilers (*ParaShares* [KTK14] (44%), *COZ* [CB15] (65%), *TaskProf* [YN17] (56%) and *SyncPerf* [LZM⁺17] (19%)). However, it is important to note that profilers such as Coz inserts delays as part of its profiling mechanism and this also gets added to the tool's overhead. Hence it is not reasonable to make a one to one comparison on the overhead of many of these tools. Nevertheless, the highlight is that GAPP was able to identify optimization opportunities detected by many of these tools, with less overhead and without instrumenting the application.

The overhead induced by a profiler on the application is important, as it can alter the application's normal behaviour. To minimize perturbations, we collect call paths only at the end of a context-switch, that too only if the entire time-slice exhibited reduced parallelism. Likewise, instruction pointers from samples are collected only if they too exhibit reduced parallelism.

The overhead induced depended on the number of critical time slices, the depth of the stack traces and the number of distinct stack traces. GAPP caches address-to-symbol mapping, and hence the mapping time will be less when stack traces are identical. However, the call stack needs to be traversed for every critical time-slice, and hence the more the number of critical

time-slices, the more will be the overhead. The memory overhead (see Table 4.1) was found to be proportional to the number of samples and the number of critical time-slices.

4.5 Customizing GAPP

4.5.1 Shared library call paths

It was observed that call paths retrieved on a context-switch sometimes belonged to dynamic shared libraries, for example, functions exported by the standard ‘C’ library or threading libraries. Dynamic shared libraries are usually built without frame pointers for performance reasons. This makes it impossible to retrieve call paths from routines exported by them during a context switch. Hence, GAPP by default, ignores call paths that do not belong to the application’s address space.

However, some applications, for example *Nektar++*, implements much of its logic in dynamic shared libraries bundled in the framework; hence, ignoring them can produce wrong results in this case. For example, all functions listed in Figure 4.16 were invoked from different dynamic shared libraries. Omitting them would have listed other functions as more critical. Therefore, we have added an option, `--trace_lib`, to include call paths from dynamic shared libraries.

4.5.2 Choosing between processes and threads

Often we may selectively want to omit traces from the parent thread in the case of multi-threaded applications, or from threads in a parallel applications that uses a combination of processes and threads. For example, in *Nektar++*, we wanted to omit threads created by the *BLAS* library and focus only on the MPI processes spawned by the application. To cater such requirements, GAPP allows the user to choose between processes and threads using command line options, `--process_only` and `--threads_only`. By default, both processes and threads are traced by GAPP.

4.6 Limitations

GAPP identifies serialization bottlenecks from code sections where at least half the threads/processes in the application are inactive. While this enables GAPP to identify a wide variety of bottlenecks, applications can still have bottlenecks without exhibiting reduced parallelism. For example, locks or synchronization primitives with low contention may not be identified by GAPP. Even though these are not strict serialization bottlenecks, it has been proven that using the proper primitive in such cases can improve the performance [LZM⁺17] and hence is a possible optimization opportunity.

4.6.1 Spin locks

With spin locks, threads trying to acquire the lock spins or busy-waits until the lock is available, rather than blocking on it. Hence code segments protected by spin locks will not exhibit reduced parallelism, even though they serialize the execution. Since GAPP identifies bottlenecks based on reduced parallelism, code segments protected by spin locks will not be identified by it. The common approach followed by tools to identify bottlenecks triggered by spin locks is to instrument the lock primitives. However, sometimes applications implement spin-locks using custom-built primitives, and tools that instrument specific lock primitives will not work in such cases.

Spin locks are meant to be held for a minimum amount of time so that threads do not contend for it for long, wasting CPU cycles. When the contention is high, it is better to block threads until the lock becomes available. Hence, some spin locks are designed to yield the thread after spinning for more than a few cycles. Then, contention for the spin lock will result in reduced parallelism as threads eventually block. Such cases will be identified by GAPP, irrespective of the primitives used for implementation (e.g. `parsec_barrier_wait()` in *Streamcluster* and spin loop in *MySQL*).

The default behaviour of `gcc`, of generating `position-independent executables`, needs to be overridden for the `addr2line` utility to work properly. This can be overcome by finding a

way to map the offset provided by the `sym()` primitive of `bcc` to the source code.

Also it may be noted that, adding eBPF probes to the kernel and removing probes from it require root privileges and hence GAPP requires root privileges for its operation.

Chapter 5

VIRBS: VIRtual-time Bottleneck Scaling

Optimizing the performance of a software application involves identifying the bottlenecks and fixing them. Chapter 3 and 4 describe how GAPP identifies bottlenecks using a lightweight kernel tracing mechanism. Fixing these bottlenecks may involve experimenting with different strategies such as rewriting the code, changing thread allocation, architectural features or hardware resources. In Chapter 4, we manually investigated possible ways of fixing the bottlenecks, applied the fixes and quantified their effect by measuring application performance. Manually fixing bottlenecks to estimate the benefits is expensive, especially if fixing them has little effect on performance. In this chapter, we explore ways of estimating performance benefits by fixing the bottlenecks, without actually modifying the code.

Prediction approaches such as performance models and virtual machines have been used to explore different hardware configurations or faster communication technologies (see Section 2.3.1 and Section 2.3.2). Some focus on the effect of speeding up critical threads in the application [DESE13]. Approaches that scale processes or hardware components on a virtual machine, or those that scale specific threads, are not able to apply scaling selectively on chosen locations where a scaled component is used. The requirement here is a framework that can be combined with GAPP to predict the effect of scaling bottleneck functions in the application. A function

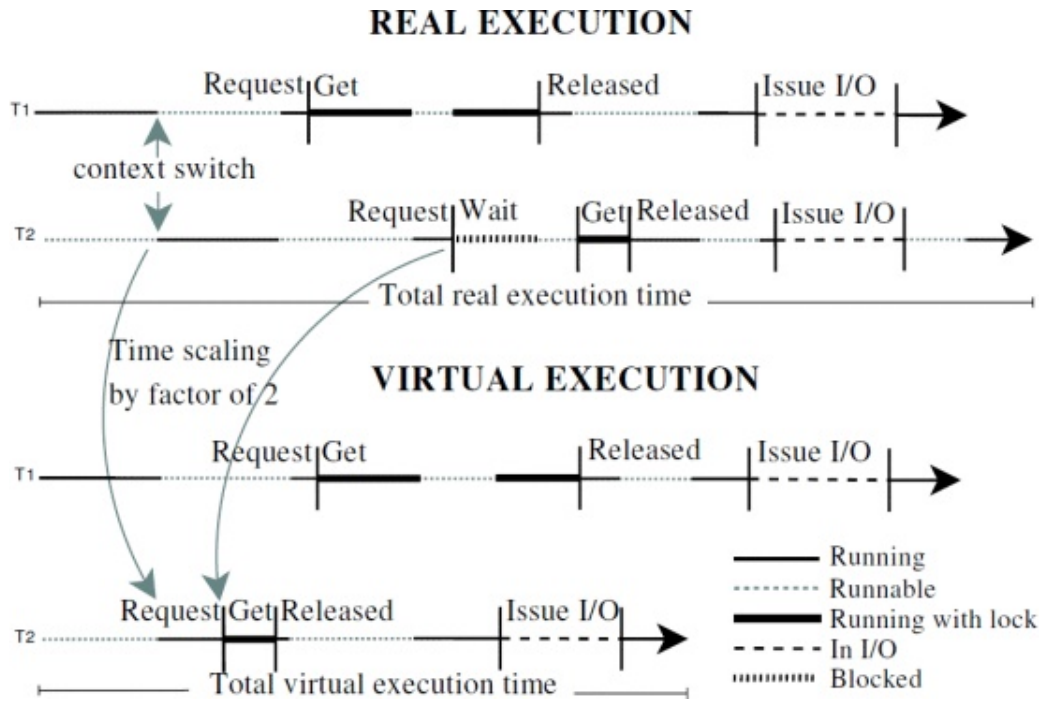


Figure 5.1: Change in resource access upon scaling [BF11]

can be a bottleneck for a variety of reasons; for example, the algorithm it implements, the synchronization primitives used, or hardware resources shared by multiple threads. A general strategy for predicting the impact of fixing bottlenecks should be able to do so, irrespective of the reason behind the bottleneck. The challenge here is that scaling needs to be applied to specific parts of the program, rather than everywhere a particular component is used, and these code segments may be executing on any core, any time.

The common approach followed by function level prediction mechanisms is to build program activity graphs from execution profiles [HM94] [YN17]. However, such offline mechanisms do not alter the thread schedule or resource usage, and hence, the effect of such changes may not be properly reflected. For example, consider the example situation illustrated in VEX [BF11], reproduced in Figure 5.1. It can be observed that scaling thread T_2 not only changes the lock acquisition pattern, but also the contention for I/O which existed in the real execution. Likewise, architectural features can also affect the performance of parallel applications and these details are abstracted away in prediction methods that use program activity graphs. VEX simulates the desired behaviour by imposing the required thread schedule using a user-level wrapper scheduler.

A more recent system, Coz [CB15], is able to predict the effect of scaling specific lines of code in a multithreaded application. It emulates the effect of speeding up a line of code by pausing other concurrent executions. This alters the thread schedule, and hence, the resource usage pattern. Therefore, Coz is a good fit as a tool for predicting the effect of scaling specific code segments. However, since Coz is sample-based, there can be substantial variation in predictions from one run to the next. It is therefore necessary to run some programs many hundreds of times in order to produce acceptable narrow confidence intervals in the results.

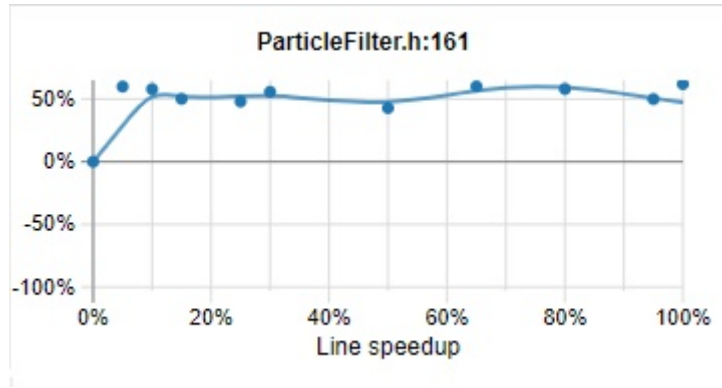
Coz virtually speeds up chosen lines of code by combining the concept of *time dilation* [AS90] with sampling. This chapter analyzes the issues that arise when implementing virtual speedup using time dilation, and proposes a strategy, VIRBS, for building a Coz-like engine, but with much reduced prediction variance using similar concepts used to build GAPP . The basic principle is to scale individual methods by controlling and synchronizing the execution of threads on a virtual timeline using *time dilation*. The prediction framework thus built, can be combined with GAPP to quantify the effect of scaling bottleneck functions.

Before explaining VIRBS in detail, the following section discusses the strategy followed by Coz to speedup chosen lines of code.

5.1 Coz - Finding Code That Counts

Coz [CB15] proposes a novel method to predict the effect of speeding up different lines of code, on the performance of a multithreaded application.

To virtually speed up individual lines of code, Coz pauses other threads for a time proportional to their execution time. The lines are chosen randomly using a sampling mechanism. The number of samples derived from a chosen line provides an approximation of its execution time and an estimate of the delay to be introduced on other threads. The difference between the simulation time and original running time plus total delay inserted provides an estimate of the speedup achievable for each scaled line of code. Figure 5.2 shows an example where we

Figure 5.2: Coz’s result for *Bodytrack*

were able to reproduce the result published by Coz for the *Bodytrack* application in the *Parsec* benchmark suite.

Coz chooses the lines of code to be optimized using sampling, and the delay inserted on peer threads depends on the sample count. It can therefore take very long execution time and/or may require repeated experiments in order to target bottlenecks, and for subsequent scaling predictions to converge.

The forthcoming sections in this chapter tries to answer the following questions:

1. Can Coz’s logic be modified to apply performance prediction at the method level, rather than on specific lines of code?
2. Can we combine virtual clocks with time dilation to obtain accurate and consistent performance prediction with reduced overall experimentation time?

Our objective is to build a prototype implementation that answers these questions. The following section discusses the design, implementation and evaluation of this framework.

5.2 Time Dilation and Virtual Clocks

Similar to Coz, VIRBS is built on the concept of *time dilation* [AS90], where the effect of speeding up a function is simulated by simultaneously pausing peer threads. We use the term

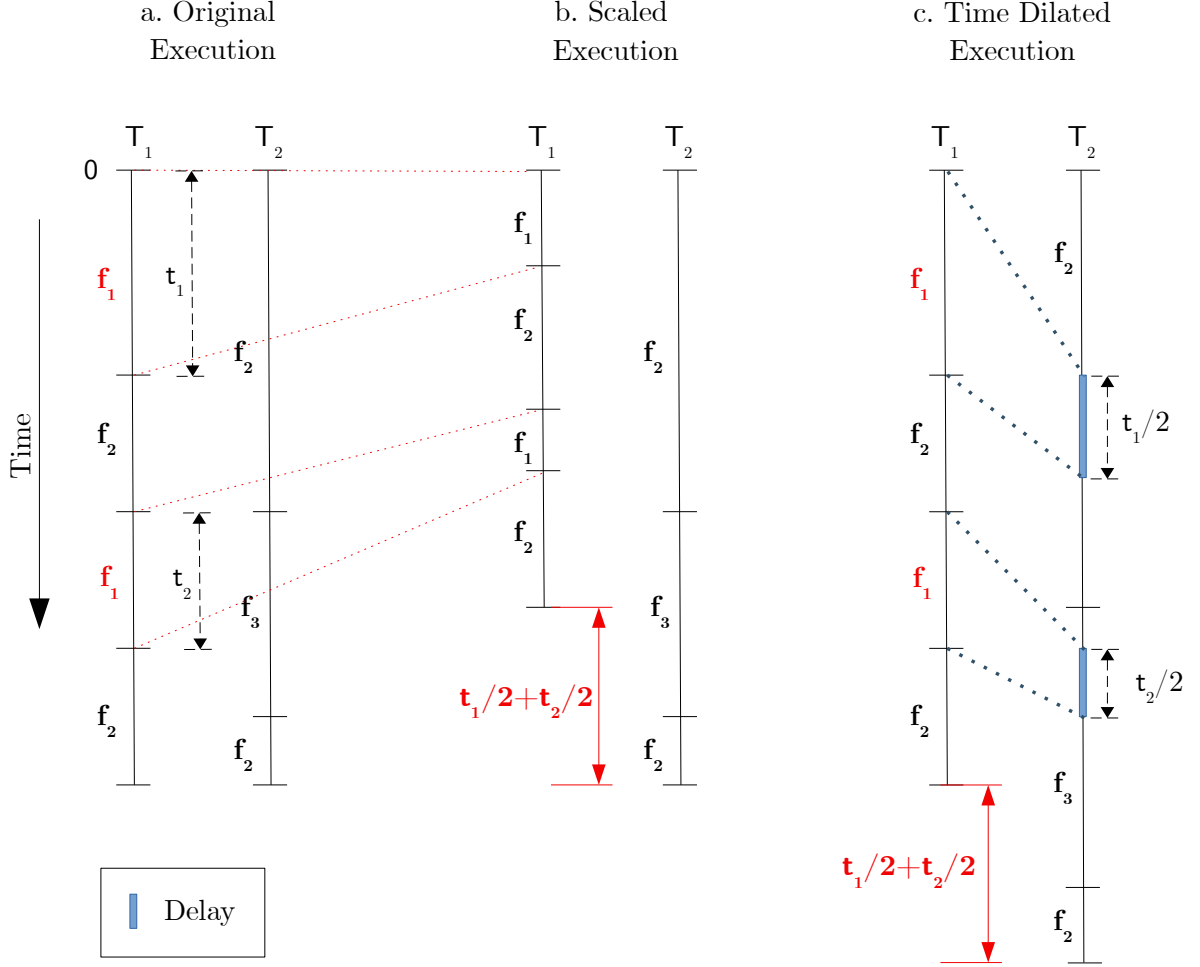


Figure 5.3: Virtual-time execution - an example

scaling factor to represent the proportion by which function duration is scaled. For example, a *scaling factor* of 2 implies that the function duration is scaled to 50% of its original time. In general, if t is the original function duration, and s is the *scaling factor*, the scaled function duration will be $\frac{t}{s}$. The simulated execution where threads are paused is hereafter called as virtual-time execution.

Consider an example scenario where two parallel threads, T_1 and T_2 , are executing functions, f_1 , f_2 and f_3 , as shown in Figure 5.3a. Figure 5.3b, the scaled execution, shows the execution sequence that would occur if function f_1 were optimized so as to reduce its execution time by a factor of 2. In order to mimic the speedup of f_1 , T_2 is paused for a time proportional to the function duration and *scaling factor*, for each of f_1 's invocations (Figure 5.3c). The delays

inserted in T_2 eventually stretches its execution time, simulating the same effect as the scaled execution in Figure 5.3b. It can be observed that in the virtual-time execution, T_1 finishes $(\frac{t_1}{2} + \frac{t_2}{2})$ units of time before T_2 , as in the scaled execution.

Ideally, peer threads should be paused simultaneously when scaling happens. However, in a virtual-time execution, delays are inserted after the scaled function exits because the function duration is not known until it finishes. Nevertheless, this maintains the relative difference in execution time between the threads as in the scaled execution (Figure 5.3c).

To implement virtual-time execution, every thread maintains two clocks: a real-time clock (RT) and a virtual-time clock (VT). Real time always advances forward as per the wall time, while virtual-time can advance or retreat in response to events that trigger virtual-time changes. When a function is being scaled, its virtual-time clock advances at a rate inversely proportional to its *scaling factor*. For example, in Figure 5.3c, at time t_1 , thread T_1 's RT will be t_1 , whereas its VT will be $\frac{t_1}{2}$, as f_1 is being scaled by a factor of 2 (Figure 5.4 illustrates the same example annotated with RT and VT). A thread's VT remains unchanged while it is paused in response to scaling events, whereas its RT advances forward. Hence in Figure 5.3c, at real-time $(t_1 + \frac{t_1}{2})$, both thread T_1 and T_2 will have the same virtual-time, t_1 . However, T_1 will be ahead of T_2 , as T_1 was executing for the whole duration of $(t_1 + \frac{t_1}{2})$, whereas T_2 was paused for $\frac{t_1}{2}$ units of time. In summary, the simulation tries to synchronize the executions of all threads across the virtual timeline by pausing threads whenever a function is scaled.

Effectively, the problem becomes one of simulating the execution of multiple parallel threads relative to a single global notion of virtual-time. As such, the issues faced are similar to those in a Parallel Discrete Event Simulation (*PDES*) [Fuj15]. A thread represents a sequential simulation that interacts with others by exchanging messages. An event in the system might represent scaling a function or a synchronization event, and this happens at irregular, discrete points in time. The state of a sequential execution (thread), represented by its RT and VT , is modified in response to events.

The execution of the threads remains synchronized along the virtual timeline as long as scaling is not applied. A thread's virtual-time changes in response to a local scaling event and this causes

threads in the system to be at different points in virtual-time. To synchronize their execution across the virtual timeline, timestamped messages, which contain information regarding pause duration, are sent from the scaled thread to others in the system.

Coz uses sampling to calculate the duration of scaled lines of code, whereas VIRBS tracks function duration by statically instrumenting function entry/exit points. Unlike Coz which puts threads to sleep to delay its execution, VIRBS pauses threads by forcing them to execute a delay loop. This avoids the extra overhead associated with context switches when threads are put to sleep. Subsequent sections illustrate how virtual clocks are incorporated with *time dilation* to build VIRBS, a prototype performance prediction framework that estimates the effect of scaling individual functions.

5.3 Performance Prediction using Virtual-time Execution

5.3.1 Monitoring function duration

The fundamental task in virtual-time execution is monitoring function duration and determining the pause duration for peer threads. Coz achieves this using sampling, where threads are paused proportional to the number of samples. The underlying principle is that more samples will be retrieved from lines of code that execute longer or more frequently, and hence, the delay will be proportional to its execution time. Function duration can also be estimated based on the number of CPU cycles gathered from hardware performance counters, as in VEX [BF12]. Both these approaches work well with CPU bound functions; we revisit this in Section 6.2.1.

VIRBS calculates function duration by statically instrumenting the entry and exit points of scaled functions using the *DynInst* [BH00] instrumentation framework. The function duration is calculated as the wall clock time between the entry and exit points. This approach will add the time spent by threads in the run queue, waiting for a resource or performing I/O, to the function duration. Nevertheless, since the duration without scaling (*scaling factor* of 1) is also

calculated the same way, the overhead added due to system load will be nullified. Moreover, this approach will consider the time threads spent blocking, and hence the effect of scaling the blocked time can also be investigated. A threads' real-time and virtual-time clocks, *RT* and *VT*, are updated based on the execution duration and its *scaling factor*. The disadvantage of this approach is that the application needs to be instrumented and this extra code will increase the overhead of the prediction framework.

5.3.2 Notifying scaling events

Ideally, in *time dilation*, peer threads should be paused concurrently with the scaled function. However, since function duration cannot be determined until a function exits, peer threads are notified about the scaling event only when a scaled function completes. Notification to pause peer threads are delivered via Linux *signals*, which are inter-process communication mechanisms used to inform a process or group of processes about the occurrence of an event. Each signal in Linux has an associated number. Signals from 1 to 31 are called *standard signals* and those from 32 to 63 are called *real-time signals*. If a process has a pending *standard signal*, subsequent signals of the same type are discarded and not queued. Conversely, *real-time signals* of the same type are queued. In virtual-time execution, since a thread can receive signals about scaling events from multiple threads, and since all scaling events need to be processed, we use *real-time signals* to notify the end of a scaled function. Each signal representing a scaling event, from here on named the *pause signal*, is labelled with the *RT*, *scaled duration* and *scaling factor* of the signal sender.

Pause signal handling

On receiving a *pause signal*, threads pause themselves for a duration proportional to that specified in the signal. This is achieved by executing a delay loop that freezes its execution for that time. As this needs to be done in response to the *pause signal*, a signal handler with the required logic can be registered to be invoked when signals are delivered.

Although straightforward in principle, there are several complications in implementing this idea. Signal handlers are defined in the user space and are included in the user code segment. However, signal delivery is implemented in the kernel and involves setting up user space stack to execute the signal handler and switching between kernel and user modes [BC06, Chapter 11].

Signals can be delivered any time to a process and hence the recipient's state when the signal is delivered is unpredictable. This requires that operations performed by the handler do not interfere with the signalled application. Moreover, handlers should use only *async signal safe* [Ker16] functions (functions that can be safely invoked from signal handlers so that there are no chances of deadlocks).

Apart from these functional aspects, a signal handler can interfere with the virtual-time execution logic. If the *pause signal* is delivered in the middle of a function, the signal processing time will be added to the function duration, as the function duration is calculated as the absolute time between its entry and exit points. Moreover, setting up the user stack frame and switching between kernel and user mode during signal handling will also add overhead to the execution, especially if signals are delivered frequently. This goes against one of the primary goals of this study to make the prediction framework lightweight. Moreover, a miscalculation in execution duration can make the prediction inaccurate as all clocks are updated based on it.

Linux supports using file descriptors to accept signals targeted at a process using the *signalfd()* system call. When using *signalfd()*, signal handlers are disabled for the specific signal. Instead, signal information is queued on the file descriptor configured, which can be accessed later. The signal is consumed as each entry is read. In VIRBS, a separate signal descriptor is defined for each thread so that signals directed to it can be accessed separately. The signal descriptor is accessed at every function exit point to check for any pending *pause signals*. This will prevent signal processing time from being added to the function duration.

Each *pause signal* acts as an asynchronous heartbeat signal that synchronizes the execution of all threads to the same virtual-time. Since *pause signals* are handled at a later time and since there can be multiple pending *pause signals*, the receiver thread handles them in the order of the signal real-time (*RT*).

5.3.3 Updating virtual execution state

A thread's virtual execution state is represented by its RT and VT and is updated by functions statically injected at the following checkpoints, hereafter called *vte-safepoints*:

1. Entry and exit of thread start function.
2. Entry and exit of scaled functions.
3. Exit point of every other function.

The virtual execution state is updated at the *vte-safepoints*, rather than instantaneously in response to signals, so that the function duration are not altered by the overhead of the injected routines. The injected routines are defined in a shared library that is statically attached into the application's address space by the *DynInst* instrumentation framework. The primary task of these routines are to monitor the execution duration, update the thread's RT and VT , send/receive *pause signals* and pause itself for the required duration.

The following section describes how the injected routines implement virtual-time execution assuming no inter-thread synchronisation. Scenarios involving inter-thread synchronisation are discussed in Section 5.4.1.

5.3.4 Worked examples

Consider again the execution scenario depicted in Figure 5.3, where only one thread is being scaled. Figure 5.4 illustrates the virtual-time execution of the same example with *pause signals*, delays, and real and virtual clocks, for a *scaling factor* of 2. In this example, function f_1 is being scaled and hence, threads send *pause signals* to every other thread each time f_1 finishes. *Pause signals* are labelled with the RT , *scaled duration* and *scaling factor* respectively of the sender. For example, the *pause signal* sent by thread T_1 , (14, 4, 2), denotes that the signal is sent at RT 14, and T_1 was being scaled for 4 units of time, by a factor of 2. This signal, which

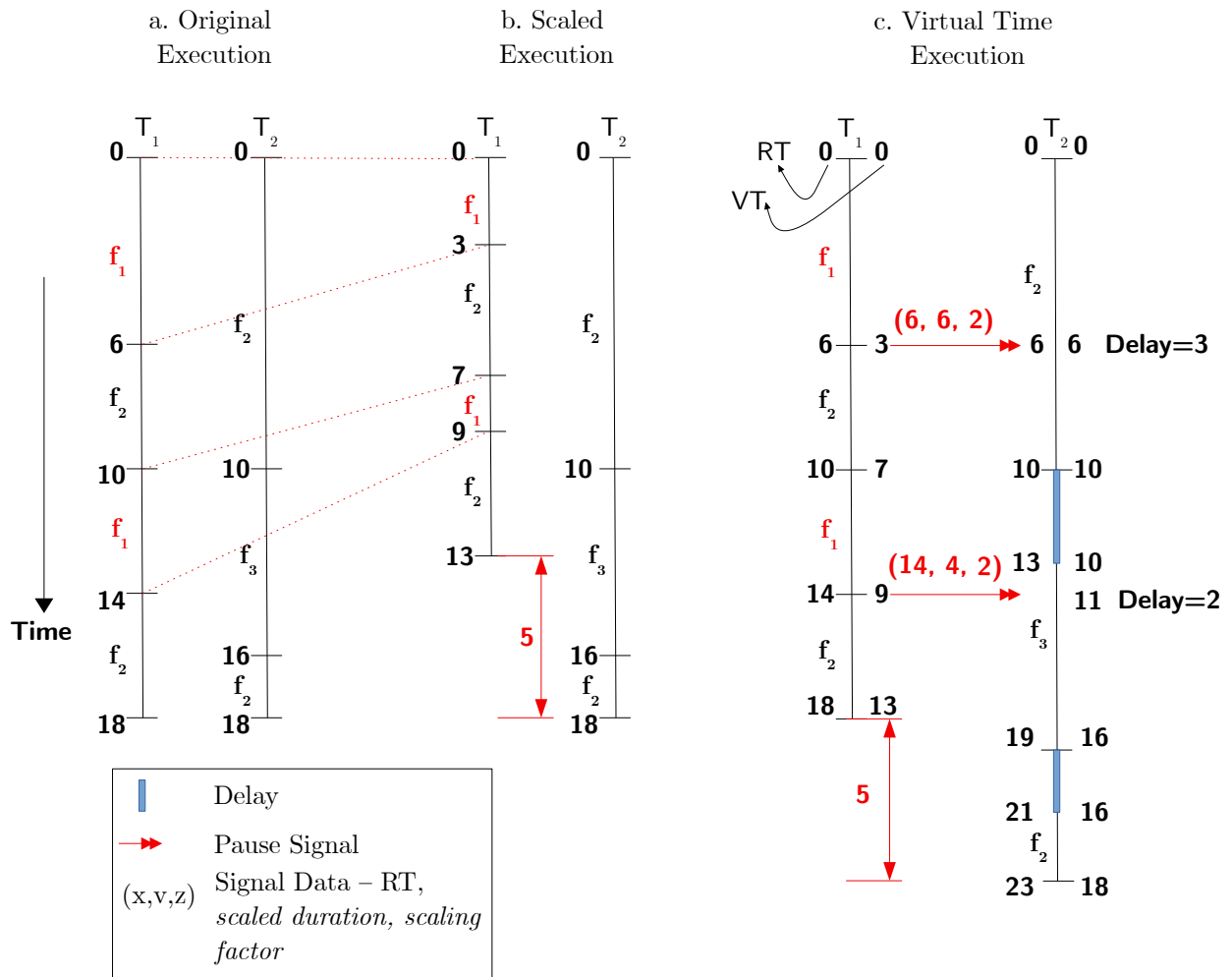


Figure 5.4: Annotated virtual-time execution

is queued in T_2 's signal descriptor will be processed by T_2 at the next *vte-safepoint*, viz., the exit point of function f_3 .

To process a signal, the receiver first compares the rate at which the virtual-time advanced in the sender, with the rate at which it advanced in the receiver, during the *scaled duration*. For the sender, the elapsed virtual-time during the scaled duration is equal to $\frac{\text{scaled duration}}{\text{scaling factor}}$. For the receiver, it is either $\frac{\text{scaled duration}}{\text{scaling factor}}$ or equal to *scaled duration*, depending upon whether the receiver was being scaled or not simultaneously. If the receiver was not scaled, then for a *scaling duration* t , and *scaling factor*, s , the delay to be inserted in the receiver will be equal to $(t - \frac{t}{s})$, which is basically the difference in rate of change of virtual-time between the sender and the receiver during the *scaled duration*. The delay value of 3 for the signal (6,6,2) from T_1 has been calculated as described before.

Now consider the second signal, (14,4,2), in Figure 5.4, sent from T_1 , at real-time 14. The elapsed virtual-time of T_1 for the real-time interval [10,14] is equal to $9 - 7 = 2$, and for T_2 , it is $11 - 10 = 1$. This is because T_2 was paused during the interval [10,13], in response to the previous *pause signal*, (6,6,2) from T_1 , and *VT* is not advanced when threads are paused. In principle, T_2 should not be paused for the second signal from T_1 , as its virtual-time advanced at a slower rate when compared to the sender. However, since T_2 was paused in response to a previous signal, the delay is calculated considering this as a non-scaled execution, spanning the real-time interval [10,14]. Consequently, the delay is estimated as $(t - \frac{t}{s})$, where $t = 4$ and $s = 2$.

Concurrent scaling

In the example discussed in the previous section, only one thread executed the scaled function. Now consider the case where two threads are scaled simultaneously, as in Figure 5.5. In the virtual-time execution, thread T_2 finishes scaling first and sends the *pause signal*. In response to this signal, (14,10,2), T_1 , the signal receiver, compares its change in virtual-time with that of the sender for the scaled duration [4,14]. During this real-time interval, T_2 's virtual-time advanced by $(9 - 4 = 5)$, where as T_1 's virtual-time advanced by $(11 - 4 = 7)$. The net delay by which T_1 should be paused is calculated as the difference in the rate at which the virtual-time

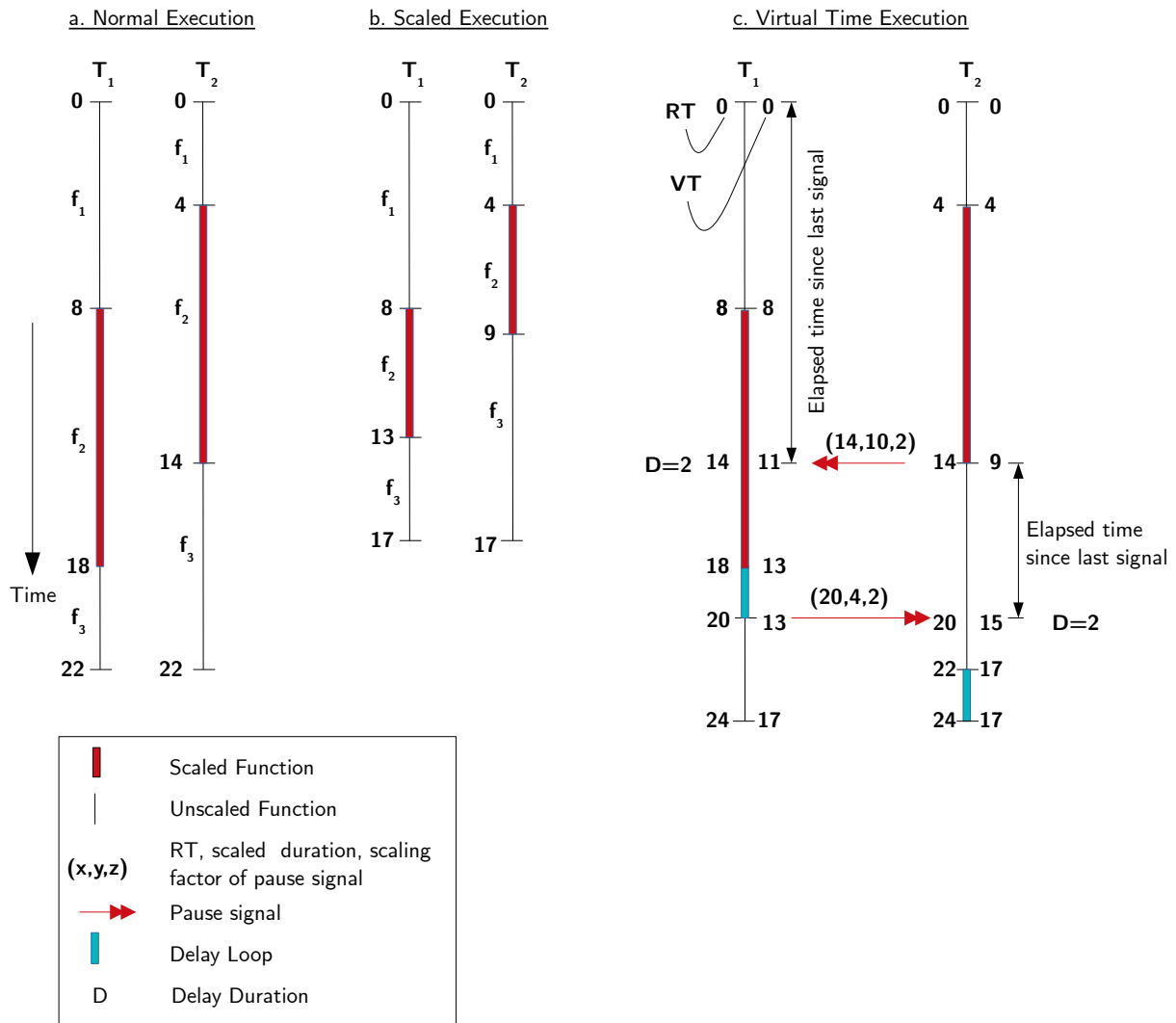


Figure 5.5: Virtual-time execution with concurrent scaling

advanced during the *scaled duration*, which is $7 - 5 = 2$.

A thread only considers the duration since the last signal when sending/processing *pause signals*. This is because all threads would have synchronized to the same virtual-time in response to the last *pause signal* in the system. In principle, the *pause signal* acts like a *global virtual time* in optimistic synchronization strategies of PDES [Fuj15]. For a thread, the last *pause signal* can be a signal sent from or received by it. Following this logic, the *pause signal* sent by T_1 , (20,4,2), limits the *scaled duration* to 4, equivalent to the real-time interval [14,18] (since the last signal was received at *RT* 14). Since T_2 is in a non-scaled function at *RT* 20, it calculates the *pause duration* as $(t - \frac{t}{s})$, as explained before. If a *pause signal's scaled duration* is greater than the time elapsed since the last signal, the receiver will only consider the time elapsed since the last signal.

The steps followed by the receiver in handling each *pause signal* is summarized below:

1. Calculate the effective scaled duration, *scaleTime*, as the minimum of the *scaled duration* in the signal and the receiver's elapsed time since last signal.

$$scaleTime = \min(elapsedTime_lastSignal_{recvr}, scaledDuration_{signal})$$

2. Determine the pause duration based on whether or not the receiver is inside a scaled function, when the signal is received.

- If the receiver is inside a non-scaled function,

$$pauseDuration = (scaleTime - \frac{scaleTime}{s}), \text{ where } s \text{ is the signal scaling factor.}$$

- If the receiver is inside a scaled function when the *pause signal* is delivered, and the virtual-clock of the receiver advanced at a faster rate when compared to that of the receiver,

$$pauseDuration = diffVT_{recvr} - diffVT_{sender}$$

where $diffVT_{recvr}$ is the rate of change of the receiver's *VT* for *scaleTime*, and $diffVT_{sender}$ is the same for the signal sender.

Calculating pause duration - Coz

Coz calculates the pause duration using a global delay counter and a per-thread local delay counter. Each experiment in Coz chooses a specific line of code to speedup. At any instant, the local delay counter represents the sample count of the chosen line of code so far in the thread, and the global counter, the total sample count of the line of code in the entire application, so far. Every 10 msec, each thread compares its local delay counter with the global delay counter and pauses if the local counter lags behind the global counter.

As an example, suppose a line of code in function f_2 (Figure 5.5a) was chosen to speedup, T_2 updates the global counter at real-time 14, and T_1 compares the global counter with its local counter at real-time 18. Assuming equal number of samples were retrieved from both executions of function f_2 , T_1 will not be paused because both the global and local counters match. However, in an ideal time-dilated execution, threads should pause for a time equal to the non-overlapped execution. This mismatch between the actual and calculated *pause duration* need not happen in every execution. However, it can lead to Heisenbugs causing predictions to vary across executions. VIRBS tries to minimize such disparities by measuring the elapsed time and comparing the virtual clocks, rather than following the sample-based approach used in Coz. Note that counter comparisons happen at the end of each 10 msec interval, so, similar to VIRBS, Coz also inserts delays sometime after the execution of the scaled code. We revisit this in Section 5.4.1 to discuss how this approach affects the correctness of prediction.

5.3.5 Implementation details - VIRBS

As discussed before, VIRBS implements virtual-time execution by statically injecting user space routines at *vte-safepoints*. An outline of the operations performed by these routines is given below.

1. *calledFirst()*: Invoked before the main routine of the application. It performs the following operations:

- Initializes *scalingFactor*.
 - Initializes *start-time* of the application.
 - Configures signal handling.
2. *initThread()*: Injected to the entry point of thread start functions.
- Initializes *RT*, *VT*, *signalfds*, *start-time* and shared data structure entries of a thread.
 - Adds thread ID to the list of threads in the application, *threadList*.
3. *checkPauseSignal()*:
- (a) Read signals from the *signalfd*.
 - (b) Sort pause signals as per sender's *RT*.
 - (c) For each signal,
 - i. Update receiver's *RT* to sender's *RT*.
 - ii. Calculate *pause duration*.
4. *enterScaling()*: Injected to entry point of scaled functions.
- (a) Handle pending *pause* signals (*checkPauseSignal*).
 - (b) Update *RT* and *VT*.
 - (c) Execute any pending delays.
 - (d) Update thread state as being scaled so that execution time is calculated correctly.
5. *exitScaling()*: Injected to scaled function exit points.
- (a) Handle pending *pause* signals (*checkPauseSignal*).
 - (b) Update *RT* and *VT*.
 - (c) Reset thread state as non-scaled.
 - (d) Execute any pending delays.
 - (e) Send *pause signal* to other threads in *threadList*.

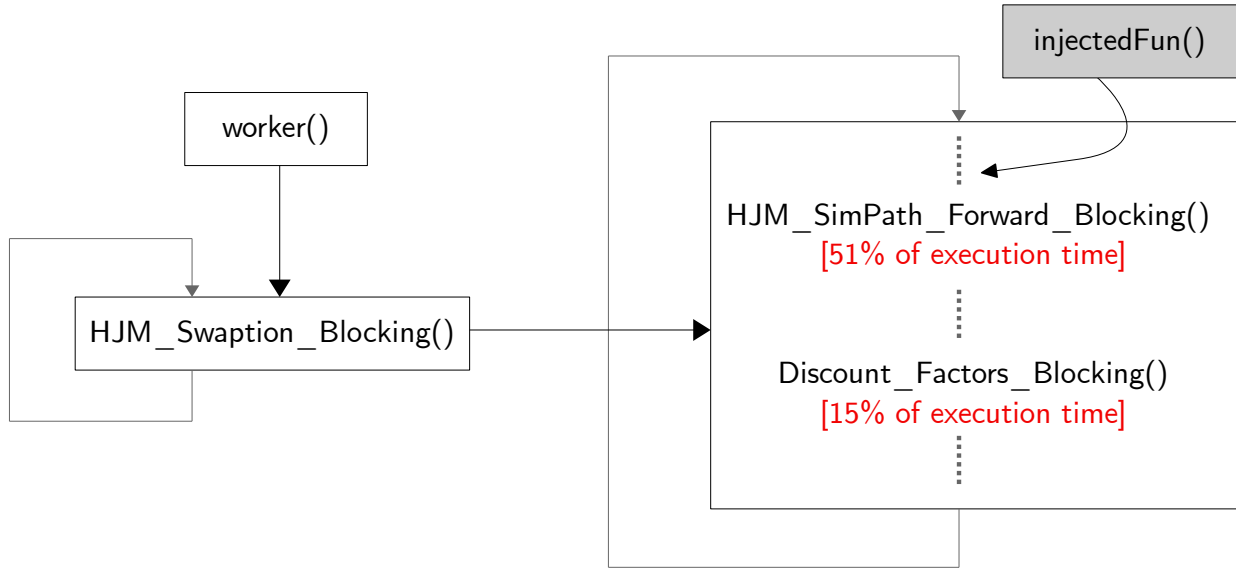
A thread can receive signals from multiple threads and there may be multiple pending signals to be processed at a *vte-safepoint*. These signals should be handled in the order of their *RT* to ensure that *pause duration* is calculated correctly. At the end of the virtual-time execution, the *VT* of the last thread to finish will be the predicted execution time of the application. This is compared with the normal execution time to estimate the program speedup achievable by scaling the function for the specified *scaling factor*.

The essence of both Coz and VIRBS is the same; simulate the effect of scaling a line of code/function by simultaneously pausing peer threads proportional to the duration of the line of code/scaled function and the *scaling factor*. Coz subtracts the delay inserted to estimate the effect of scaling, whereas VIRBS rolls back the *VT*. Nevertheless, if pausing threads speeds up/slows down the normal execution then that will be reflected in the *VT*/execution time of the application. The major difference between Coz and VIRBS is the method followed in calculating the *pause duration*.

The next section evaluates the correctness of VIRBS for different execution scenarios and compares the results with those generated by Coz.

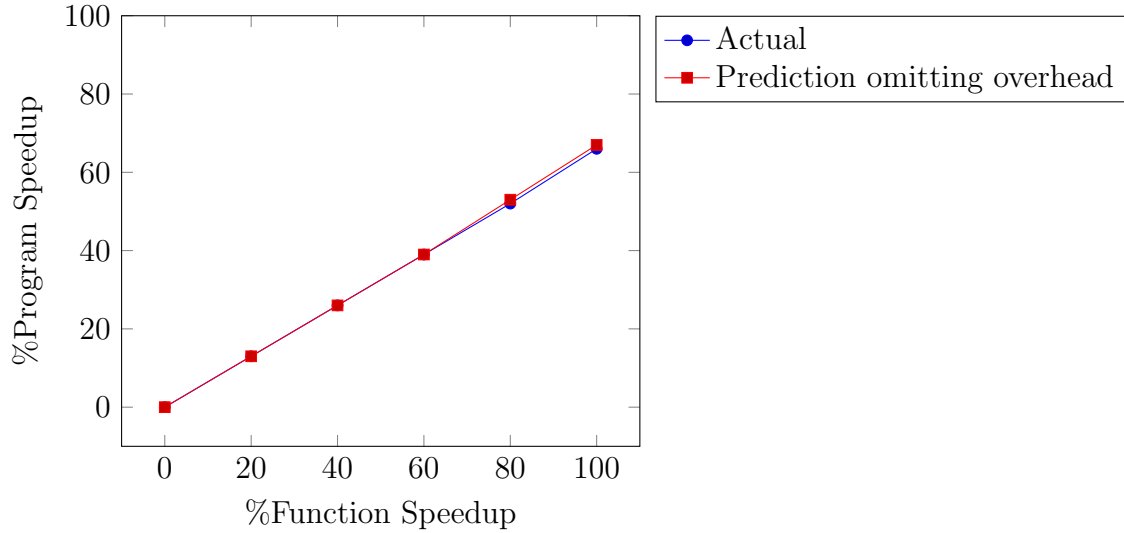
5.4 Evaluating Virtual-time Execution

The correctness of VIRBS was evaluated by injecting a function into the *Swaptions* application from the *Parsec3.0* benchmark suite, and scaling the function by different factors. The injected function does not do any useful work, except for looping for a specific duration determined by the argument passed to it. Figure 5.6 illustrates the general logic of *Swaptions* with the injected code. The injected function increased the program execution time from 51 secs to 152 secs. The application was executed on the *native* input with 8 threads. The function duration was varied to mimic the effect of speeding it up. For different function duration, the program execution time was measured and plotted as the actual performance improvement. Later, the prediction framework was used to estimate the performance improvement of the application by virtually scaling the injected function by different *scaling factors*. Figure 5.7 plots the

Figure 5.6: *Swaptions* - logic

actual and predicted program speedup for different speedup of the injected function. Note that the function speedup is the inverse of the *scaling factor*. For example, 10% function speedup implies that function duration is scaled to 90% of its execution time, equivalent to a *scaling factor* of 0.9. We have used percentage speedup in the plots to make them consistent with the notations followed in Coz, so that the results can be compared. The predicted program speedup is the difference between the actual program execution time and the virtual-time of the parent thread (the last thread to finish). It can be observed that the prediction closely matches the actual performance improvement.

Swaptions is a data parallel application without any inter-thread synchronization. A natural question that may arise is whether pausing peer threads will have any effect on such data parallel applications where threads do not interact with each other. Can we obtain the same result by simply rolling back *VT* at the end of a scaled function? The answer depends on whether threads interact indirectly during execution. In the case of *Swaptions*, there is no visible interaction between threads. However, threads may interact with each other at the operating system level when accessing shared resources such as the same address space (recall similar issues discussed with *Blackscholes* and *Dedup* in Section 4.2). In such cases, virtual-time execution should reflect the effect of avoiding contention.

Figure 5.7: Scaling injected function - *Swaptions*

VIRBS suggests checking for pending signals at every function exit point. However, instrumenting every function exit point will induce large overhead, especially if the functions are invoked frequently. For example, with the *native* input, *Swaptions* made more than 4 billion function invocations when executed with 16 threads. Instrumenting all function exit points with a dummy function (function that returns without doing any useful work) induced approximately 52% overhead to the *Swaptions* benchmark. In order to reduce the extra overhead caused due to instrumentation, we have chosen a function that is executed frequently as a checkpoint for pending signals. For this experiment, the *checkPauseSignal()* function was injected to the exit point of *HJM_Swaption_Blocking()* function. A design where threads periodically check for pending *pause signals* can be considered as an alternative approach. As mentioned previously, this prototype framework requires rigorous experimentation to compare the pros and cons of each approach. This section investigates the feasibility of VIRBS in terms of its correctness and consistency in prediction.

The overhead of VIRBS (excluding the delays inserted) on *Swaptions* with the injected function was $20\% \pm 0.14$. This was measured by running the framework with a *scaling factor*=1, where signals are sent at the end of the scaled duration, but peer threads do not execute any delay as scaling is not applied. The overhead was measured by repeating each experiment ten times. As the overhead of the prediction framework also gets added to the virtual-time, it was subtracted from the final virtual-time to correctly estimate the execution time.

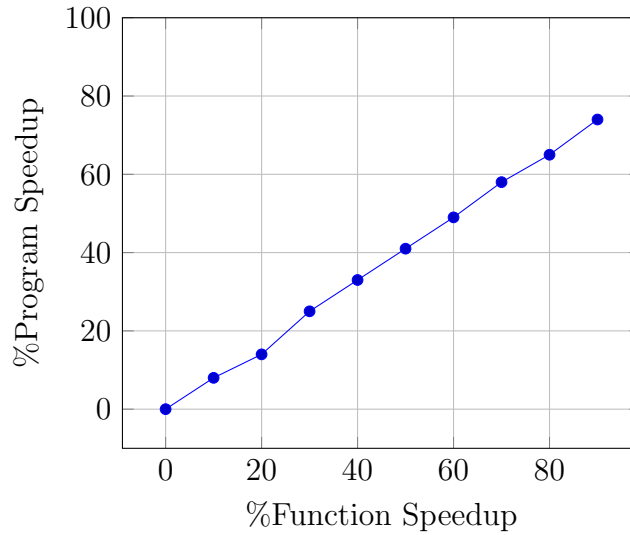


Figure 5.8: Scaling HJM_SimPath_Forward_Blocking() - *Swaptions*

For *Swaptions*, the contribution of the *pause duration* to the overhead was negligible. This is because, in the absence of any thread interaction, all threads would have executed the scaled function (in this case, the injected function) simultaneously. VIRBS does not pause threads during overlapped scaled execution and hence the delay inserted was minimal. However, approximately 70% of the overhead was attributed to the signal sending and handling mechanism. The remaining overhead was due to the code injected at *vte-safepoints*. A better design replacing signals with shared structures may reduce the overhead. Again, this needs to be experimented to quantify the overhead, and is left as a future work.

Optimizing *Swaptions* in Parsec2.1

The experiments published in Coz were conducted on the 2.1 version of the Parsec benchmark suite. To improve performance of *Swaptions*, changes were suggested by Coz for different lines of code in the dominant function, HJM_SimPath_Forward_Blocking(). Applying these patches reduced the function duration by approximately 18% and improved the application performance by circa 13%.

We simulated scaling the HJM_SimPath_Forward_Blocking() function by 18% using VIRBS, and the predicted program speedup was approximately 14%, which aligns closely with the actual improvement of 13%. Figure 5.8 shows the prediction for HJM_SimPath_Forward_Blocking()

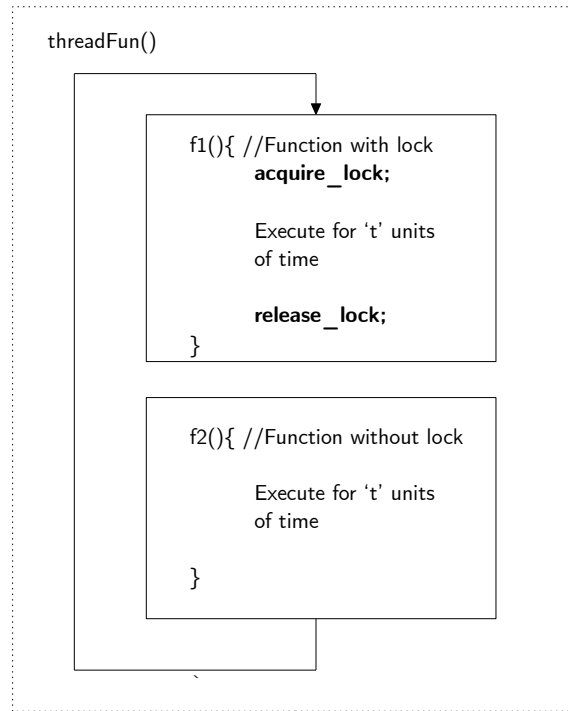


Figure 5.9: Threads with synchronization primitives - Sample program

with VIRBS. As with the injected function, the overhead due to the delay inserted was negligible; the maximum delay inserted for a *scaling factor* of 0.1 was only 0.2 secs (circa 0.4% of total execution time). Most of the overhead was due to the instrumented code and signal handling mechanism and this attributed to approximately 70% of the virtual execution time.

Coz tries to reduce the overhead by using the following strategies:

- Rather than delaying threads for each sample, samples are processed in batches of ten.
- The delay for a sample is approximated to a factor of the sampling period for a specific speedup.
- Instead of using *POSIX* signals, the delay to be inserted is tracked using a global counter which each thread checks and matches it to its local counter to determine pending delays.

5.4.1 Threads with synchronization primitives

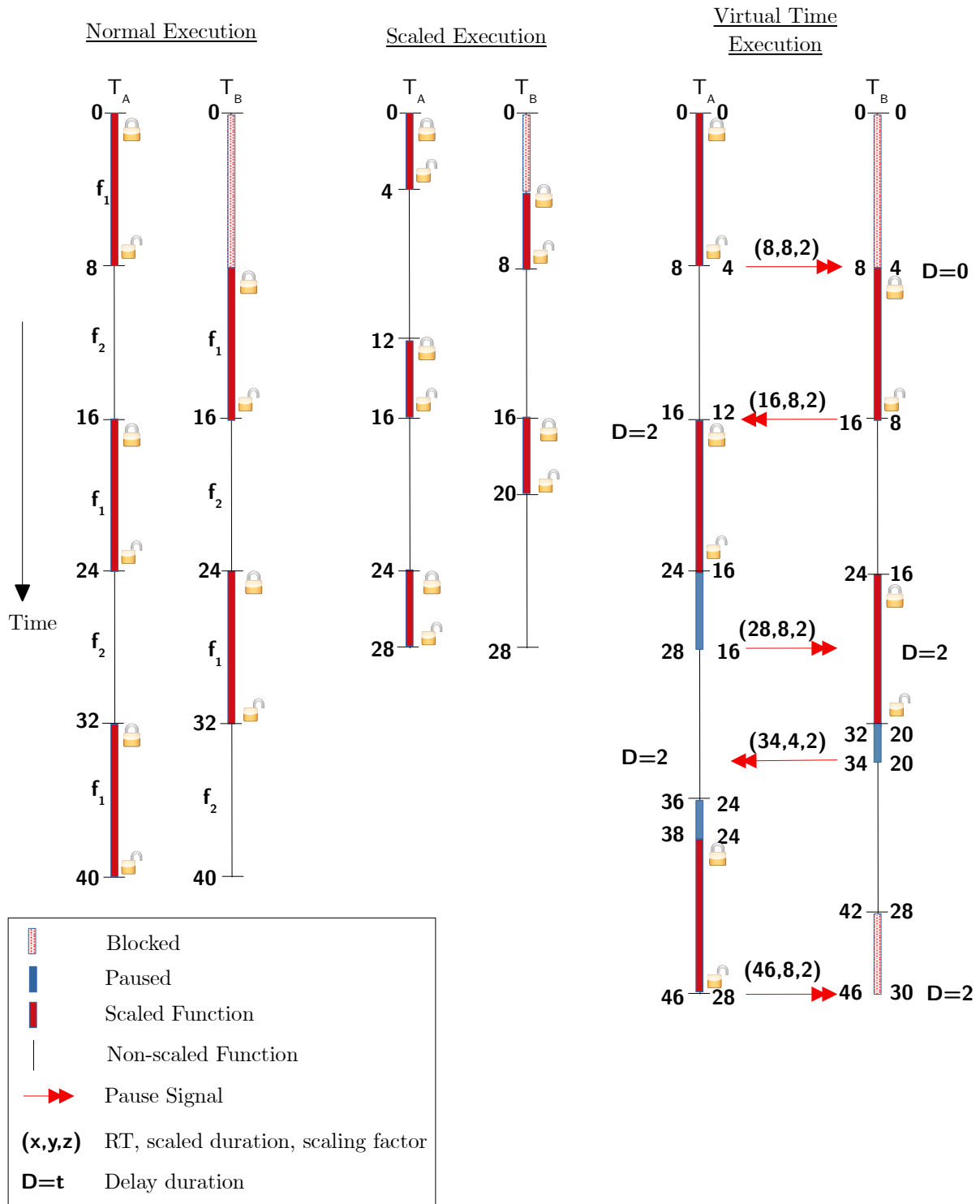
The previous section validates VIRBS on data parallel applications without any thread synchronization. We now evaluate the correctness of VIRBS on examples where threads interact with each other through shared resources.

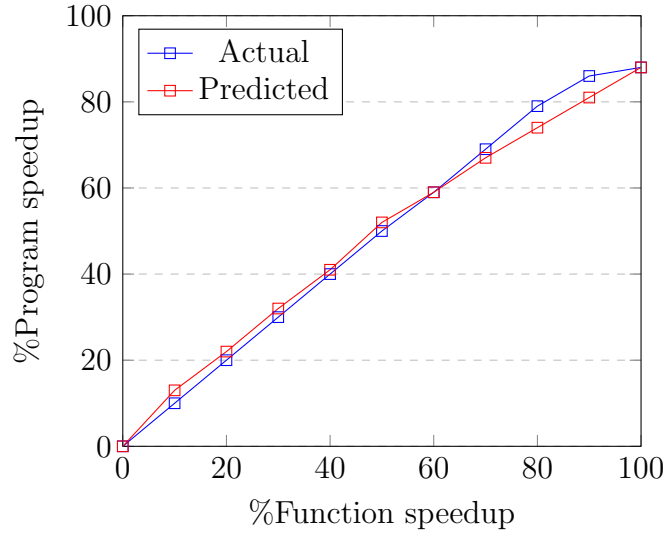
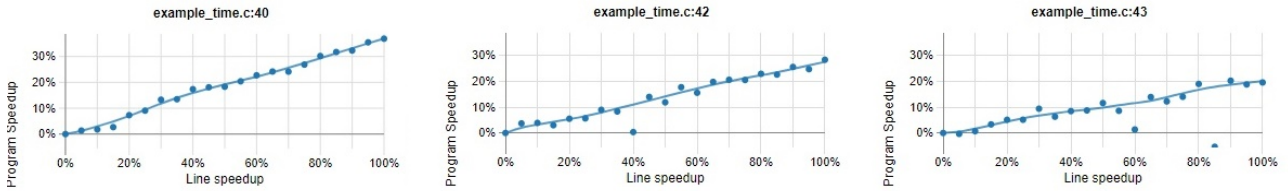
Consider an example program with 8 threads, where threads follow the logic illustrated in Figure 5.9. Each thread repeatedly execute two functions, f_1 and f_2 in a loop. Threads enter f_1 , acquire a lock, executes for t units of time, releases the lock and enters f_2 , where they iterate for another t units of time.

Figure 5.10 illustrates the virtual-time execution when $f_1()$ is scaled by a factor of 2. Since all threads are contending for the shared lock simultaneously, accelerating $f_1()$ scales all threads, including their blocked duration, uniformly. Hence the lock acquisition pattern in a scaled execution remains the same as in the non-scaled execution, as evident from Figure 5.10. The virtual-time execution follows the lock acquisition pattern in the scaled execution and hence the predicted performance improvement closely follows the actual performance improvement as shown in Figure 5.11. However, we can observe that sometimes the predicted program speedup varies from the actual speedup, especially for larger function speedup. This is because, delays inserted can alter the expected behaviour of the virtual-time execution. For example, the delays cause thread T_B to wait for the lock in the last iteration illustrated in the virtual-time execution. On the contrary, threads never have to wait in the scaled execution. The variance between the actual and predicted speedup depends on how often and for how long this behaviour happens in the system. For a higher function speedup (lower *scaling factor*), delays will be more and hence their effect will be more prominent. The plots generated by Coz, for the lines invoked from $f_1()$ shows a similar linear improvement in performance for different line speedup (Figure 5.12).

Scaling $f_2()$

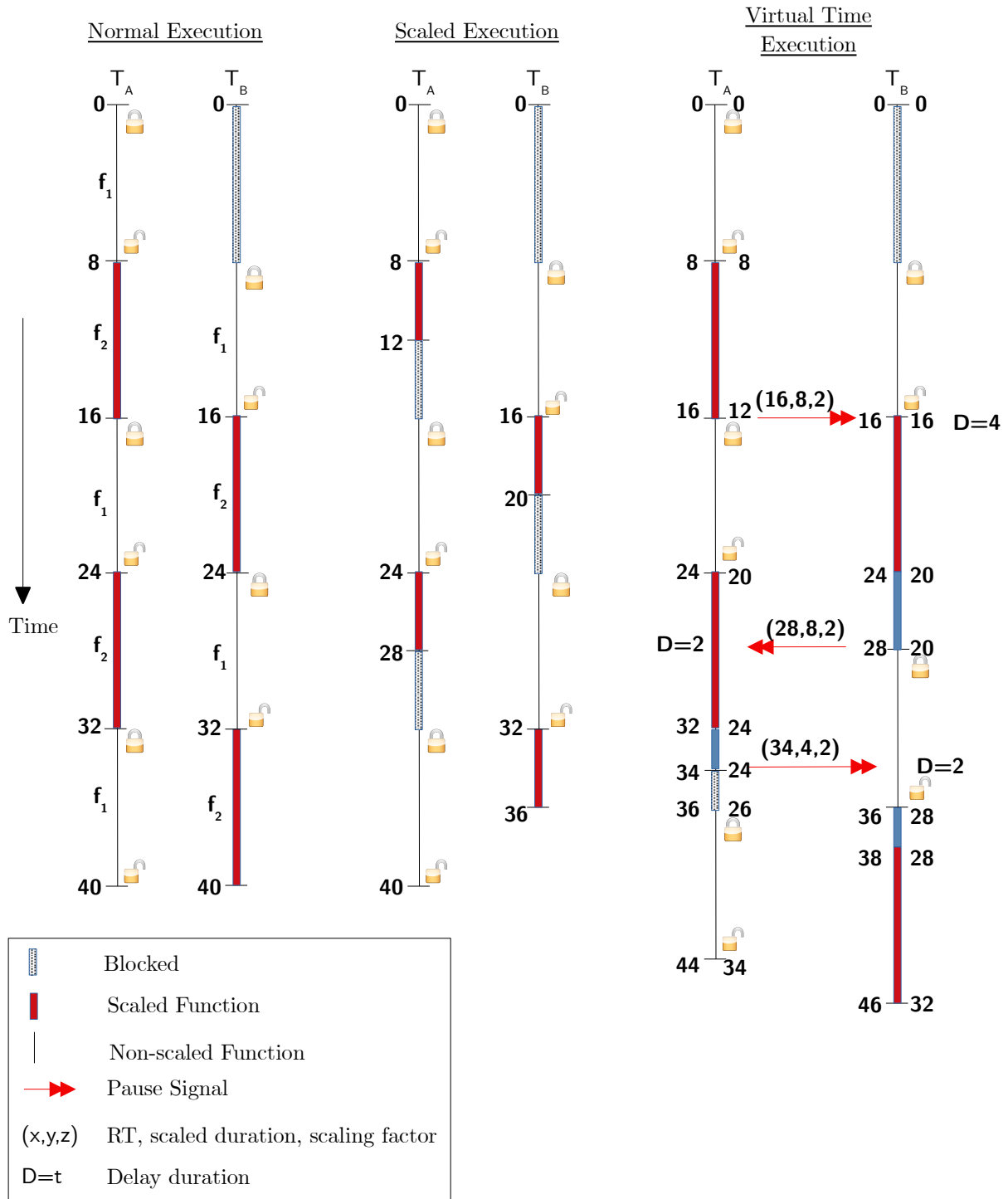
In the previous section, we have seen that both Coz and VIRBS correctly predicts the trend in performance improvement when function $f_1()$ was scaled. Consider the case where function

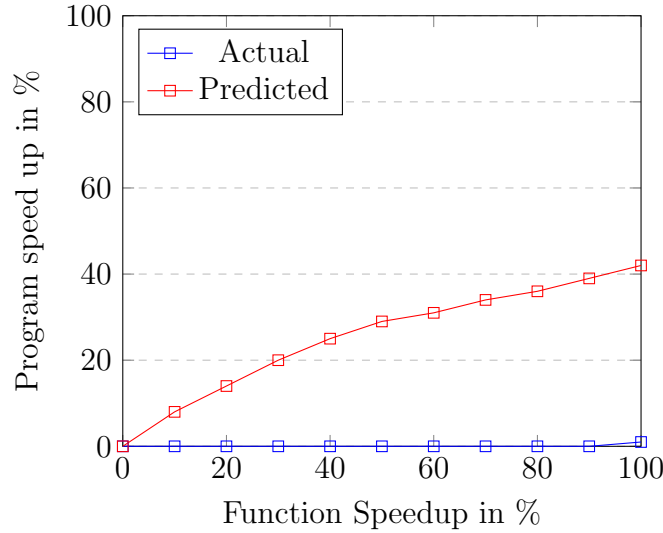
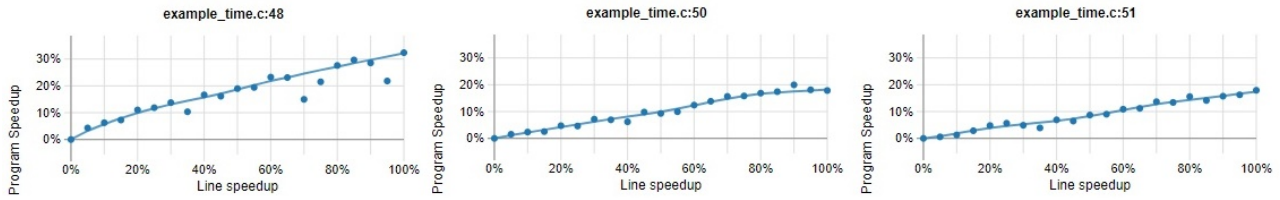
Figure 5.10: Scaling $f_1()$ in Figure 5.9

Figure 5.11: Scaling $f_1()$ - actual vs predictionFigure 5.12: Coz results - scaling $f_1()$

$f_2()$ in the example program illustrated in Figure 5.9 is scaled by a factor of 2. Figure 5.13 shows the actual and scaled execution for this case with 2 threads. As observed, scaling $f_2()$ has marginal improvement in execution time; only for the last iteration in the illustrated execution sequence. This is because the effect of scaling $f_2()$ is nullified by the extra waiting time induced for the shared lock. For example, in the scaled execution of Figure 5.13, even though thread T_A finishes $f_2()$ at time 12, it has to wait for another 4 units of time (the same time by which $f_2()$ was accelerated) to acquire the lock held by thread T_B . In other words, scaling $f_2()$ has no effect on performance as it is not the bottleneck.

However, it can be observed that the virtual-time execution in Figure 5.13 does not follow the execution pattern in the scaled execution. While thread T_A finishes at time 40 in the scaled execution, it finishes at virtual-time 34 in the virtual-time execution. As the number of iterations increases, the mismatch between the scaled and virtual execution increases. This mismatch can be observed in Figure 5.14 which plots the actual and predicted performance improvement when function $f_2()$ is scaled. Even though scaling $f_2()$ does not improve performance, VIRBS

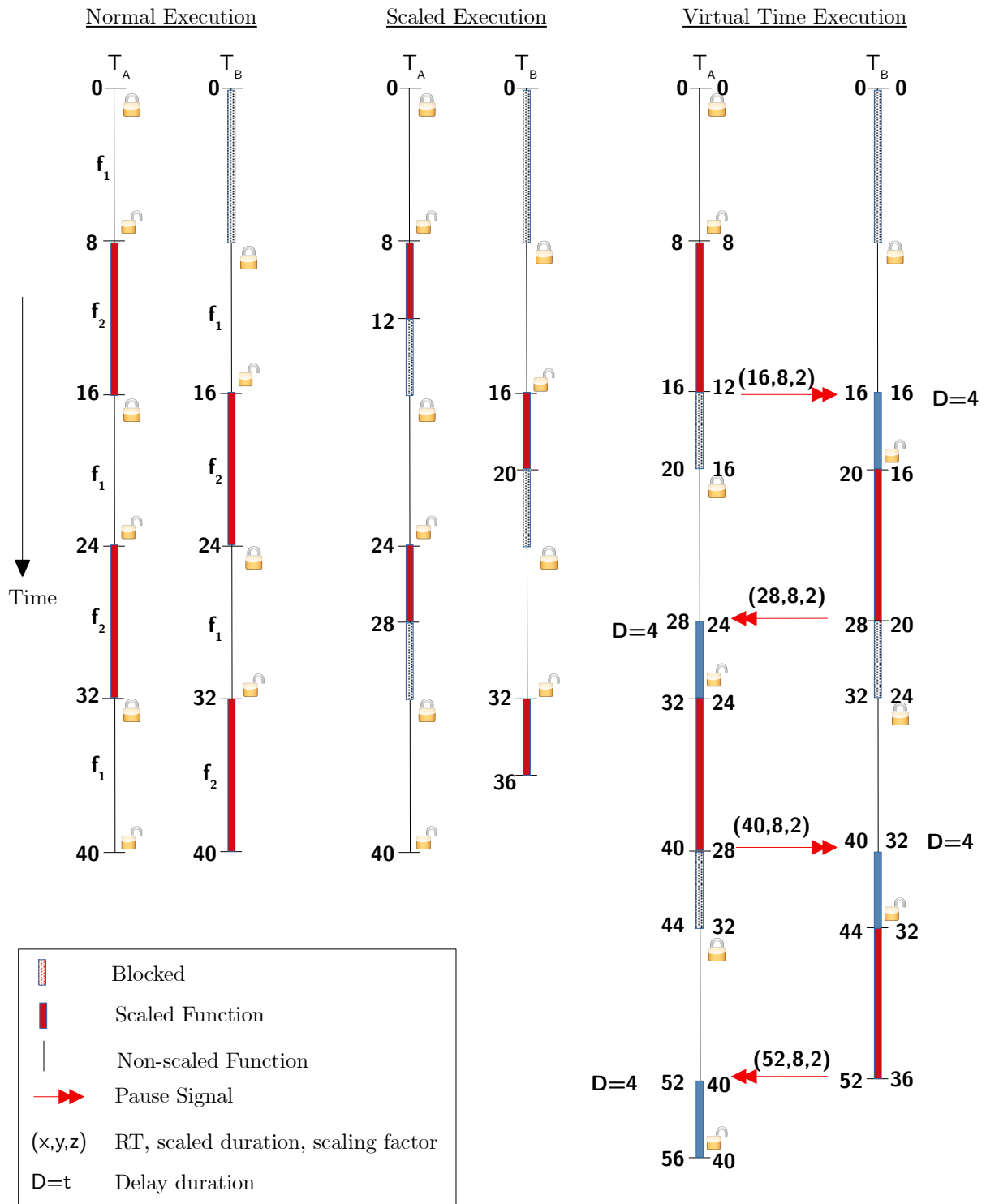
Figure 5.13: Scaling $f_2()$ of Figure-5.9

Figure 5.14: Scaling $f_2()$ - actual vs predictionFigure 5.15: Coz results - scaling $f_2()$

predicts a performance improvement of up to circa 40%.

Note that the above issue is an artefact of threads pausing in general, and not a feature of the implementation. In particular, Coz also predicts a similar positive improvement in performance when $f_2()$ is scaled (Figure 5.15). The reason for the error in prediction becomes clear if we examine the virtual-time execution when $f_2()$ is scaled (Figure 5.13). In the virtual-time execution, when T_A is scaled during the real-time interval $[8,16]$, T_B is executing the critical section. Ideally, T_B should be paused simultaneously when T_A is scaled. If that were done, then T_B 's lock holding time would have extended causing T_A to wait to acquire the lock. This idea is illustrated in Figure 5.16. It can be observed that if T_B is paused at the right instance, then the virtual-time execution simulates the required behaviour as in the scaled execution. From this we can conclude that both VIRBS and Coz fail for the same reason when $f_2()$ is scaled; neither approach results in pausing peer threads at the correct instance.

The reason for the prediction to fail when $f_2()$ is scaled can be analyzed from a different

Figure 5.16: Scaling $f_2()$ - modified logic

perspective, comparing the virtual-time of the *waker* and *target* threads when the wake-up signals are sent. For example, in the last iteration of Figure 5.13, T_B releases the lock at virtual-time 28, and T_A acquires it at virtual-time 26. For the virtual-time execution to follow the timestamps of a scaled execution, the threads should be synchronized along the virtual timeline when they communicate with each other. The scenario explained here is an example of an out of order synchronization event that happened during virtual-time execution. For a correct virtual-time execution, the *target* thread's (thread being woken) VT should be equal or greater than that of the *waker*. Section 6.2.1 discusses possible ways to handle such out of order events.

What we have achieved so far is a scaling engine similar to Coz. Section 6.2.1 gives directions on how VIRBS can be further improved to overcome its limitations.

Chapter 6

Conclusion and Future Work

This thesis presents the development, and evaluation, of two frameworks, GAPP and VIRBS, aimed at pinpointing bottlenecks and then subsequently evaluating the likely effect of fixing the bottleneck, through virtual-time scaling.

6.1 GAPP

GAPP presents a lightweight generic interface for identifying serialization bottlenecks in parallel Linux applications. Based on the experiments performed with GAPP, we can conclude that:

1. Tracing kernel events to capture execution intervals with reduced parallelism allows a range of bottlenecks to be detected, for example critical sections, bottlenecks identified from hardware events, busy-wait loops and resource contention.
2. The overhead of a profiler can be limited by selectively capturing events of interest, without losing necessary information.
3. Kernel tracing mechanisms available on modern operating systems can be leveraged to build profiling tools that work ‘out of the box’, without the need to instrument the source code, patch the operating system or undertake elaborate post-processing of data captured during a program’s execution.

4. Ranking bottlenecks based on the CMetric makes it possible to focus on performance issues that are more likely to provide better performance improvement.

The initial focus of this research was on identifying performance anomalies instigated by the application logic. However, experiments revealed that the execution environment can also trigger performance issues. GAPP was extended to retrieve kernel stack traces to reveal such bottlenecks that arise due to hardware-software interactions. In several experiments, such bottlenecks were ranked higher when compared to those triggered by the application logic. This also emphasizes the need to develop profilers that follow a holistic approach to identify performance issues rather than focusing on one specific type of anomaly, as other more important issues might be missed in the process. Still, following a blanket approach to detect bottlenecks has certain trade-offs. These are discussed in the following section.

6.1.1 Discussion and Future Work

Categorising critical stack traces

For synchronisation related bottlenecks, GAPP will report stack traces from both the blocking threads as well as from the *waker* threads. As far as performance benefits are considered, blocking events may not be worth optimizing, especially if blocking and waking happen at different call sites. With GAPP, the user must manually investigate the critical paths to identify those that belong to the *wakers*. This issue was further explored in a supervised Masters project, MGAPP [JDL19], where extra information is attached to critical paths triggered by synchronization and I/O bottlenecks to categorize the type of bottlenecks and also to differentiate between blocking and waking stack traces. MGAPP achieves this by tracing *futex* calls and *read/write* system calls. This prototype framework has been tested on a couple of applications, but requires further experimentation to evaluate its capability and overhead.

Even though the Linux tracing subsystem provides a wide variety of kernel events which can be monitored, the major challenge is in selectively capturing these events specific to the application. eBPF solves this issue to a great extent with the help of tailored APIs and eBPF maps.

Still, we have encountered cases where we could not distinguish between events triggered by the operating system and the same events triggered by the application. For example, a thread can be blocked to `TASK_UNINTERRUPTIBLE` state when performing I/O, as well as by the operating system. In such cases, tracing user-level routines can provide information tailored to the application. This shows a major trade-off that exists between generality and specificity when tracing applications through low-level kernel events.

Dynamic library functions

Dynamic shared libraries are built without frame pointers, and hence, GAPP cannot retrieve stack-traces if bottlenecks arise when threads are inside a shared library routine. Calls to dynamic shared libraries can be traced by inserting *uprobes* to the Procedure Linkage Table (PLT) of the binary, using eBPF. However, this can raise two issues: one is that eBPF puts a limit on the number of open probes allowed in an application, and sometimes this limit gets exceeded. The other is that *uprobes* are executed in the kernel context and hence switching between user and kernel context for each library call can incur extra overhead. Whilst eBPF becomes handy in tracing kernel events, for tracing user-space routines the overhead can be significant, depending on how frequently they are invoked. As a compromise, GAPP provides the `--trace-lib` option to list the critical stack traces retrieved from shared library calls. Even though such call paths might be incomplete, samples from the time-slices can guide the user to the corresponding code location.

GAPP on different threading platforms

It will be interesting to see how GAPP behaves with other parallel platforms such as Java, Intel TBB, Cilk etc. While the core concepts in GAPP is not attached with any language/library, experiments need to be conducted to see how stack traces and address mapping work in the presence of virtual environments or intermediate schedulers present in such platforms.

6.2 VIRBS

VIRBS hypothetically scales bottleneck functions following a Coz [CB15] like approach and quantifies the performance improvement that can be achieved by fixing the bottleneck. It tracks function duration using static instrumentation and implements function scaling using virtual clocks with time-dilation. Experiments with VIRBS have revealed that:

1. By using GAPP to determine the bottleneck functions, VIRBS avoids scaling functions that may not deliver any performance improvement, which reduces the number of scaling experiments to be performed.
2. Determining *pause duration* based on scaled-function execution time and inserting delays at *vte-safepoints* produce consistent results across repeated scaling experiments.
3. A strong trade-off exists between the correctness of predictions and the overhead of the framework when implementing performance prediction using time-dilation. Improving the correctness requires inserting the *pause duration* at the correct instance. This requires frequent pauses of peer threads during a scaled execution, which can increase the overhead of the framework.

6.2.1 Discussion and Future Work

Virtual-time execution and out of order events

VIRBS has been tested with data parallel applications without inter-thread synchronization. Applying scaling on applications with synchronization can cause inter-thread communication messages to arrive out of order. For example, if a wake-up signal is sent from thread T_1 to T_2 , such that $VT_1 > VT_2$, where VT_1 and VT_2 are the thread's virtual-time, T_2 will be processing a signal that would have arrived in the future. It may be possible to devise rollback mechanisms, as in optimistic synchronization strategies in *PDES*, to cancel the state changes due to out

of order events. However, as this can increase the overhead, a better strategy may be to re-synchronize the thread's execution across the virtual timeline when out of order events occur. This may introduce an offset between the timestamp of events in the virtual-time execution and an equivalent scaled execution. However, akin to many approaches in *PDES*, VIRBS's goal is to approximate the scaled execution rather than replicate it, without sacrificing the correctness.

To synchronize wake-up events across the virtual timeline, synchronization events need to be traced along with time dilation. This can be achieved by instrumenting specific threading library primitives. However, this will limit the scope of the approach to applications that use specific threading library primitives. Moreover, threads can also be blocked/woken-up in response to I/O operations, and such synchronization events will be missed if threading-library primitives alone are traced. So a better approach would be to trace blocking and wake-up events in the kernel.

Scaling I/O in virtual-time

In the proposed approach, if a function performs I/O, since function duration is determined by instrumenting its entry and exit points, scaling the function will also scale the time spent performing I/O or the time spent waiting for I/O, which is the expected behaviour. However, if the scaled function performs other computation along with I/O, scaling the function will uniformly scale all the operations inside the function. If the instrumentation framework supports injecting code to instruction boundaries, the logic can be easily extended to scale specific blocks of code rather than the whole function. This way, the effect of scaling I/O can be correctly quantified. Instrumenting basic blocks, rather than function boundaries has a further advantage that the effect of scaling a function call-site can be investigated.

Nevertheless, determining wake-up events due to I/O is challenging as wake-up signals can also be delivered from the interrupt context. For each wake-up due to I/O, we need to determine the thread that triggered this event to correctly compare the virtual clocks of the *waker* and the *target*. Tracing all types of wake-up events will require it to be done in the kernel. However, since virtual-time execution is simulated in the user-space, the challenge is in mapping the

timestamps of wake-up events in the kernel to the virtual execution time stamps in the user-space.

6.3 Summary

This thesis lays the foundation for the development of a performance analysis suite combining GAPP and VIRBS to provide a powerful framework for automatically detecting bottlenecks and answering ‘what if’ questions to predict whether an effort to fix a bottleneck is worthwhile before applying any code refactoring or investing in additional hardware resources. Software architects can then focus on redesigning code fragments that provide maximum performance improvement. This is especially important as fixing one bottleneck can sometimes shift the bottleneck to a different one. It can also be deployed at any stage in the software development life cycle making it a potentially valuable addition to existing test-driven development tools and frameworks, which play a major role in determining the ‘time to market’.

Bibliography

- [AAFM10] Erik Altman, Matthew Arnold, Stephen Fink, and Nick Mitchell. Performance analysis of idle programs. In *ACM Sigplan Notices*, volume 45, pages 739–753. ACM, 2010.
- [ACGS04] Glenn Ammons, Jong-Deok Choi, Manish Gupta, and Nikhil Swamy. Finding and removing performance bottlenecks in large systems. In *European Conference on Object-Oriented Programming*, pages 172–196. Springer, 2004.
- [AS90] John K Antonishek and Robert D Snelick. Emulation through time dilation. In *Distributed Memory Computing Conference, 1990., Proceedings of the Fifth*, volume 2, pages 1271–1275. IEEE, 1990.
- [ASE17] Shoaib Akram, Jennifer B Sartor, and Lieven Eeckhout. DEP+ BURST: Online DVFS Performance Prediction for Energy-Efficient Managed Language Execution. *IEEE Transactions on Computers*, 66(4):601–615, 2017.
- [BC06] Daniel Pierre. Bovet and Marco Cesati. *Understanding the Linux Kernel: from I/O ports to process management*. O’Reilly Media, Inc., 3rd edition, 2006.
- [BF11] Nikos Baltas and Tony Field. Software performance prediction with a time scaling scheduling profiler. In *2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 107–116. IEEE, 2011.
- [BF12] Nikos Baltas and Tony Field. Continuous performance testing in virtual time.

- In *2012 Ninth International Conference on Quantitative Evaluation of Systems*, pages 13–22. IEEE, 2012.
- [BGZ16] Zachary Benavides, Rajiv Gupta, and Xiangyu Zhang. Parallel execution profiles. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, pages 215–218, 2016.
- [BH00] Bryan Buck and Jeffrey K Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, 2000.
- [BHD⁺00] Russell P Blake, David O Hovel, Robert I Davidson, David E Heckerman, and John S Breese. Automatic bottleneck detection by means of workload reconstruction from performance measurements, May 23 2000. US Patent 6,067,412.
- [BJ18] Mario Badr and Natalie Enright Jerger. A high-level model for exploring multi-core architectures. *Parallel Computing*, 80:23–35, 2018.
- [BKSL08] 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, 2008.
- [BM11] Andrew R Bernat and Barton P Miller. Anywhere, any-time binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*, pages 9–16, 2011.
- [CB15] 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, 2015.
- [CF19] David Calavera and Lorenzo Fontana. *Linux Observability with BPF: Advanced Programming for Performance Analysis and Networking*. O’Reilly Media, 2019.

- [CMC⁺15] Chris D Cantwell, David Moxey, Andrew Comerford, Alessandro Bolis, Gabriele Rocco, Gianmarco Mengaldo, Daniele De Grazia, Sergey Yakovlev, J-E Lombard, Dirk Ekelschot, et al. Nektar++: An open-source spectral/hp element framework. *Computer physics communications*, 192:205–219, 2015.
- [Cor14] Jonathan Corbet. Rcu, cond_resched, and performance regressions. Available at <https://lwn.net/Articles/603252/>, 2014. Accessed 30th July, 2020.
- [DBSEE13] Kristof Du Bois, Jennifer B Sartor, Stijn Eyerman, and Lieven Eeckhout. Bottle graphs: Visualizing scalability bottlenecks in multi-threaded applications. *ACM SIGPLAN Notices*, 48(10):355–372, 2013.
- [Des] Mathieu Desnoyers. Using the linux kernel tracepoints. Available at <https://www.kernel.org/doc/Documentation/trace/tracepoints.txt>. Accessed 4th August, 2020.
- [DESE13] Kristof Du Bois, Stijn Eyerman, Jennifer B Sartor, and Lieven Eeckhout. Criticality stacks: Identifying critical threads in parallel programs using synchronization behavior. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 511–522. ACM, 2013.
- [DSDMT⁺17] Daniele De Sensi, Tiziano De Matteis, Massimo Torquati, Gabriele Mencagli, and Marco Danelutto. Bringing parallel patterns out of the corner: the p3 arsec benchmark suite. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(4):1–26, 2017.
- [DTLM14] Florian David, Gael Thomas, Julia Lawall, and Gilles Muller. Continuously measuring critical section pressure with the free-lunch profiler. *ACM SIGPLAN Notices*, 49(10):291–307, 2014.
- [FCW18] Tony Field, Robert Chatley, and David Wei. Software performance testing in virtual time. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 173–174, 2018.

- [Fle17] Matt Fleming. An introduction to the bpf compiler collection. Available at <https://lwn.net/Articles/742082/>, December 2017. Accessed 5th July, 2020.
- [Fle18] Matt Fleming. Using user-space tracepoints with bpf. Available at <https://lwn.net/Articles/753601/>, 2018. Accessed July 5th, 2020.
- [Fuj15] R. Fujimoto. Parallel and distributed simulation. In *2015 Winter Simulation Conference (WSC)*, pages 45–59, 2015.
- [GKM82] Susan L Graham, Peter B Kessler, and Marshall K Mckusick. Gprof: A call graph execution profiler. *ACM Sigplan Notices*, 17(6):120–126, 1982.
- [GS15] Anjana Gosain and Ganga Sharma. A survey of dynamic program analysis techniques and tools. In *Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA) 2014*, pages 113–122. Springer, 2015.
- [GSdS⁺11] Todd Gamblin, Martin Schulz, Bronis R de Supinski, Felix Wolf, Brian J N Wylie, and Others. Reconciling sampling and direct instrumentation for unintrusive call-path profiling of MPI programs. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 640–651. IEEE, 2011.
- [GYM⁺05] Diwaker Gupta, Kenneth Yocum, Marvin McNett, Alex C Snoeren, Amin Vahdat, and Geoffrey M Voelker. To infinity and beyond: time warped network emulation. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 1–2. ACM, 2005.
- [HDG⁺12] Shi Han, Yingnong Dang, Song Ge, Dongmei Zhang, and Tao Xie. Performance debugging in the large via mining millions of stack traces. In *Proceedings of the 34th International Conference on Software Engineering*, pages 145–155. IEEE Press, 2012.
- [HM94] 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.

- [IUNO19] Tatsushi Inagaki, Yohei Ueda, Takuya Nakaike, and Moriyoshi Ohara. Profile-based detection of layered bottlenecks. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, pages 197–208, 2019.
- [JDL19] Ashley J Davies-Lyons. Leveraging linux kernel tracing to classify and detail application bottlenecks. Technical report, Department of Computing, Imperial College London, 2019.
- [Jef85] David R Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(3):404–425, 1985.
- [JHHF09] Zhen Ming Jiang, Ahmed E Hassan, Gilbert Hamann, and Parminder Flora. Automated performance analysis of load tests. In *2009 IEEE International Conference on Software Maintenance*, pages 125–134. IEEE, 2009.
- [JSMP12] José A Joao, M Aater Suleman, Onur Mutlu, and Yale N Patt. Bottleneck identification and scheduling in multithreaded applications. *ACM SIGPLAN Notices*, 47(4):223–234, 2012.
- [KBH⁺14] Heiko Koziolk, Steffen Becker, Jens Happe, Petr Tuma, and Thijmen de Gooijer. Towards software performance engineering for multicore and manycore systems. *ACM SIGMETRICS Performance Evaluation Review*, 41(3):2–11, 2014.
- [Ker16] Michael Kerrisk. signal-safety(7) - linux manual page. Available at <https://man7.org/linux/man-pages/man7/signal-safety.7.html>, 2016. Accessed 15th July, 2020.
- [KTK12] Melanie Kambadur, Kui Tang, and Martha A Kim. Harmony: Collection and analysis of parallel block vectors. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 452–463. IEEE, 2012.
- [KTK14] 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.

- [LCM⁺05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices*, 40(6):190–200, 2005.
- [LNC14] Jereme Lamps, David M Nicol, and Matthew Caesar. Timekeeper: A lightweight virtual time system for linux. In *Proceedings of the 2nd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, pages 179–186. ACM, 2014.
- [LNGP17] Qi Luo, Aswathy Nair, Mark Grechanik, and Denys Poshyvanyk. Forepost: Finding performance problems automatically with feedback-directed learning software testing. *Empirical Software Engineering*, 22(1):6–56, 2017.
- [LST15] Hee Won Lee, Mihail L Sichitiu, and David Thuente. High-performance emulation of heterogeneous systems using adaptive time dilation. *The International Journal of High Performance Computing Applications*, 29(2):166–183, 2015.
- [LTCS10] Michael A Laurenzano, Mustafa M Tikir, Laura Carrington, and Allan Snaveley. Pebil: Efficient static binary instrumentation for linux. In *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, pages 175–183. IEEE, 2010.
- [LZM⁺17] Tongping Liu, Guangming Zeng, Abdullah Muzahid, et al. Syncperf: Categorizing, detecting, and diagnosing synchronization performance bugs. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 298–313. ACM, 2017.
- [MBDH99] Philip J Mucci, Shirley Browne, Christine Deane, and George Ho. PAPI: A portable interface to hardware performance counters. In *Proceedings of the department of defense HPCMP users group conference*, volume 710, 1999.
- [Moo14] Ryan W Moore. *Predicting application performance for chip multiprocessors*. PhD thesis, University of Pittsburgh, 2014.

- [mysa] Mysql 5.7 reference manual:: 14.8.9 configuring spin lock polling. Available at https://docs.oracle.com/cd/E17952_01/mysql-5.7-en/innodb-performance-spin_lock_polling.html. Accessed 15th September, 2019.
- [mysb] Mysql 5.7 reference manual::8.5.8 optimizing innodb disk i/o. Available at <https://dev.mysql.com/doc/refman/5.7/en/optimizing-innodb-diskio.html>. Accessed 15th September, 2019.
- [NS07] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007.
- [SD16] Suchakrapani Datt Sharma and Michel Dagenais. Enhanced Userspace and In-Kernel Trace Filtering for Production Systems. *Journal of Computer Science and Technology*, 31(6):1161–1178, 2016.
- [SDD⁺17] Tao B Schardl, Tyler Denniston, Damon Doucet, Bradley C Kuszmaul, I-Ting Angelina Lee, and Charles E Leiserson. The CSI framework for compiler-inserted program instrumentation. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 1(2):1–25, 2017.
- [SK] Alexei Starovoitov and Michael Kerrisk. bpf(2)-linux manual page. Available at <http://man7.org/linux/man-pages/man2/bpf.2.html>. Accessed 8th July, 2020.
- [SLPG15] Du Shen, Qi Luo, Denys Poshyvanyk, and Mark Grechanik. Automating performance bottleneck detection using search-based application profiling. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 270–281, 2015.
- [Thi06] Justin Thiel. An overview of software performance analysis tools and techniques: From gprof to dtrace. *Washington University in St. Louis, Tech. Rep*, 2006.
- [TMC09] Nathan R Tallent and John M Mellor-Crummey. Effective performance measurement and analysis of multithreaded applications. In *Proceedings of the 14th*

- ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 229–240, 2009.
- [TMCP10] Nathan R Tallent, John M Mellor-Crummey, and Allan Porterfield. Analyzing lock contention in multithreaded applications. In *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 269–280, 2010.
- [YLB⁺12] Wucherl Yoo, Kevin Larson, Lee Baugh, Sangkyum Kim, and Roy H Campbell. ADP: automated diagnosis of performance pathologies using hardware events. *ACM SIGMETRICS Performance Evaluation Review*, 40(1):283–294, 2012.
- [YM88] C Yang and Barton Miller. Critical path analysis for the execution of parallel and distributed programs. In *The 8th International Conference on Distributed*, pages 366–367, 1988.
- [YN17] Adarsh Yoga and Santosh Nagarakatte. A fast causal profiler for task parallel programs. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 15–26, 2017.
- [YP16] Tingting Yu and Michael Pradel. SyncProf: detecting, localizing, and optimizing synchronization bottlenecks. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 389–400. ACM, 2016.
- [ZC18] Yu Zhang and Jiankang Chen. A scalable pthreads-compatible thread model for vm-intensive programs. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 121–137. Springer, 2018.
- [ZGMW18] Fang Zhou, Yifan Gan, Sixiang Ma, and Yang Wang. wPerf: Generic Off-CPU analysis to identify bottleneck waiting events. In *13th USENIX Symposium on Operating Systems Design and Implementation*, pages 527–543, 2018.