μProfiler: A Concurrent Profiler for Concurrent C++ (μC++)

by

Justyna Gidzinski

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2007

AUTHOR'S DECLARATION FOR ELECTRONIC SUBMISSION OF A THESIS

Abstract

A concurrent program, unlike a sequential program, has multiple threads of execution, resulting in numerous advantages (e.g., faster execution), but also in complex and unpredictable interaction. As a consequence, a concurrent program can easily underutilize available parallelism, and performance can be extremely difficult for users to predict and analyze on their own.

A profiler is a tool that can help a user identify as well as locate potential performance problems in a program. Profiling is accomplished through monitoring of the program execution, and analyzing and visualizing the collected performance data. A profiler must display useful information in a way that allows a user to effectively and efficiently understand and analyze a program's behaviour.

This thesis describes the advancement in design and implementation of μProfiler, a profiler for sequential and concurrent programs written in μC++. μC++ is a concurrent dialect of the C++ programming language, which executes in uni-processor and multi-processor shared-memory environments. Major advancements to three μProfiler metrics are presented: the Execution State, the Exact Routine Call-Graph and the Statistical Routine Call-Graph. The Execution State metric charts each state for every thread over the entire execution of the program. With high overhead and perfect accuracy, the Exact Routine Call-Graph metric provides an exact call-graph profile of the program's dynamic execution, describing the control flow among routines. With low overhead and less accuracy, the Statistical Routine Call-Graph metric provides a statistical call-graph profile of the program's dynamic execution. For each metric, advancements were made throughout the profiling process (i.e., monitoring, analysis and visualization), addressing goals such as scalability, functionality, usability and performance. The metrics provide reasonable memory overhead and, based on the comparison to related work, are state-of-the-art in functionality and provide similar run-time performance.

Acknowledgements

First of all, I would like to sincerely thank my supervisor Dr. Peter Buhr for all his time, understanding and guidance. His efforts and active involvement made this process a great learning experience and my work all that much more enjoyable. I would like to thank my readers, Dr. David Taylor and Dr. Steve MacDonald, for their valuable suggestions and comments. I would also like to graciously acknowledge funding from the Natural Sciences and Engineering Research Council of Canada.

Many thanks go to Richard Bilson for his technical assistance with μC++ and μProfiler, for his willingness to answer my many questions and give constructive suggestions, and for his many hours spent editing my thesis. I would also like to thank my other lab mates Ashif Harji and Roy Krischer. Richard, Ashif and Roy made my time working in the lab truly enjoyable.

Finally, special thanks go to my family and friends, especially my parents. Undoubtedly, I would not have made it this far without their unconditional love and support. They always believed I could do anything I set my mind to and they made me believe the same.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Programming is divided into two forms: sequential and concurrent. A sequential program has a single thread of execution, while a concurrent program has multiple threads of execution (see Section 1.2). The transition from one to many threads significantly increases programming complexity across all dimensions of development and maintenance. Applications employ concurrency to get work done faster, to handle larger quantities of work, or to simplify the structure of the application [HH04].

Initially concurrency was a system-level programming technique for use within operating systems. Now, concurrency is a necessary user-level programming technique used in many common user applications: database and web servers, Internet search engines, web applications, and graphical user interfaces [XMN99]. The need for concurrent programming techniques and tools is especially urgent given the changing direction of hardware. CPU performance increases are now the result of increasing hardware parallelism, via multi-threading and/or multi-core CPUs, and not increasing CPU clock-speed, as a plateau has been reached in clock-speed. Concurrent programming techniques and tools are required to successfully utilize the available parallelism at both the system and user levels.

Many programming languages support user-level concurrency. Languages such as Ada [Uni95], Java [AGH00], and C# [HWG03] support user-level concurrency through built-in concurrency constructs. Other originally sequential languages such as C [KR88] and C++ [Str97] have been extended to support user-level concurrency. New language dialects such as Concurrent C [GR89], pC++ [MMB$^+$94] and μC++ [BDS$^+$92] are the result of such extensions.

1.1 Performance of Concurrent Programs

Although concurrency is an invaluable user-level programming tool, the performance of concurrent programs can be extremely difficult for a user to predict and analyze [HLM95]. Concurrent programs are inherently more complex than their sequential counterparts and their performance can be affected by a greater number of factors [JFL98]. However, it should be noted that predicting and analyzing the performance of a sequential program can also be a difficult task.

The factors affecting performance include nondeterminism, synchronization, mutual exclusion and context switching.

- **Nondeterminism:** Concurrent programs are inherently nondeterministic; threads interact with each other in unpredictable ways [CL00]. Synchronization and mutual exclusion [BH05, §4] (described below) reduce the unpredictable behaviour, but can potentially cause performance bottlenecks.

- **Synchronization:** Synchronization is used to make thread interaction predictable, i.e., ensure operations happen in a certain temporal order. Synchronization is achieved through the blocking and scheduling of threads. One or more threads are blocked until the constraints on their execution order are satisfied. If threads block too frequently or for too long, performance in reduced.

- **Mutual exclusion:** A resource shared by multiple threads must be protected within a critical section (i.e., a piece of code manipulating the resource). Mutual exclusion restricts the number and type of threads given access to a critical section at any given time. If a critical section is fully occupied (maximum number and type of threads), then any subsequent thread wanting to enter is blocked until other threads leave the critical section. If a critical section is large and frequently executed, the number of threads awaiting entry and the length of their wait increases, and again performance is reduced.

- **Context switching:** Each time a thread is blocked or preempted, a typically small amount of overhead is incurred to save its state and schedule another thread. However, excess context switching due to unnecessary synchronization, excessive mutual exclusion, a poor scheduling algorithm or an inappropriate time-slice value can cause a reduction in performance.

1.1.1 Locating Performance Problems

To achieve a high-performance program (sequential or concurrent), a user must identify potential performance problems as well as locate the part of the program where the problems occur. Given the complexity of concurrent programs and the number of factors affecting performance, high-level tools are essential for effective and efficient performance analysis [HLM95]. These high-level tools are called profilers, which monitor, analyze and visualize the execution performance of a program to help a user verify its expected behaviour, and locate bottlenecks and hotspots.

- **Expected Behaviour:** A programmer has an expectation of how a program should behave during execution; however, expected and actual program behaviour often differ. A profiler provides a programmer with information about the program's actual execution to compare against the expectation. This comparison helps the programmer identify the location of any divergence.

- **Bottlenecks:** Bottlenecks are specific areas of a program that throttle performance (i.e., rate determining steps). For example, bottlenecks include areas of resource contention. A profiler can help identify bottlenecks, allowing a programmer to focus on areas of the program that can significantly affect performance.

- **Hotspots:** Hotspots are areas of a program that are frequently executed. For example, hotspots include the specific routines in which the greatest amount of execution time is spent. Although such areas may not be the direct cause of a performance reduction, any optimization could significantly improve performance due to the disproportionate amount of execution time spent in the areas. A profiler can help locate these hotspots, again allowing a programmer to focus on areas of the program that can significantly affect performance.

While the primary focus of a profiler is performance, analyzing and understanding the performance of a concurrent program may help a programmer establish its correctness. This thesis is mainly concerned with the performance side of profiling. Numerous concurrent debugging tools, whose primary purpose is correctness, also exist [PN93].

This thesis presents the design and implementation of a concurrent profiler called μProfiler. μProfiler profiles concurrent programs written in μC++ (a concurrent dialect of C++) and is itself written in μC++.

1.2 Definitions

This section provides definitions for terms used extensively throughout this thesis.

- A **thread** is an independent sequential execution path through a program [BS07].
- A **task** is a programming language object that contains a thread and an execution state (including a stack). Tasks share a common memory and their threads are called user threads. User threads are implicitly scheduled across one or more kernel threads provided by the operating system.
- A **coroutine** is a programming language object that contains an execution state (including a stack). A coroutine uses the thread of its caller to advance its own execution state. What differentiates a coroutine from a routine is that a coroutine can suspend its execution and return to its caller *without* terminating. The caller can then resume the coroutine at a later time and it restarts from the point where it suspended, continuing with the local state that existed at the point of suspension.
- An **execution stack** stores information about the currently active routines as well as the parameters, return addresses and local variables of those routines. A stack is the major

component of a task and coroutine's execution state. The term "execution entity" refers to any language entity with its own execution stack.

- **Concurrency** is the logical notion of threads executing simultaneously [BH05, §2]. Hence, concurrency can occur on a uni-processor system by rapidly interleaved execution of multiple threads on the single processor, or on a multi-processor system by interleaved execution of multiple threads across the processors.

- **Parallelism** is the physical notion of threads executing simultaneously [BH05, §2]. Hence, parallelism can only occur on a multi-processor system where threads execute simultaneously.

Hence, any multithreaded (multi-tasking) program is a concurrent program with the potential for parallelism if run on a multi-processor system.

1.3 Thesis Organization

This thesis is organized as follows. Chapter 2 presents a detailed description of profiling. Chapter 3 presents the design and implementation of μProfiler, which is the profiling tool advanced for this thesis, including a brief overview of the μC++ programming language, which is μProfiler's target environment. The next three chapters present the major contributions of this thesis through three μProfiler metrics. Chapter 4 explains the Execution State Chart as part of the Execution State metric, which charts each task's states during execution of the program. Chapter 5 explains the Exact Routine Call-Graph metric, which provides an exact call-graph of a profiled program. Chapter 6 explains the Statistical Routine Call-Graph metric, which provides a statistical call-graph of a profiled program. Finally, Chapter 7 summarizes the contributions of this thesis and presents possible directions for future work.

Chapter 2

Profiling

A profiler is a high-level tool to help a user understand a program's run-time behaviour as well as locate potential performance problems (e.g., bottlenecks and hotspots). Profiling is accomplished through monitoring of the program execution, and analyzing and visualizing the collected performance data.

Profiling a program consists of three phases:

- **Instrumentation insertion:** instrumentation is inserted into a program to monitor its run-time behaviour.
- **Execution and monitoring of instrumented program:** the instrumented program is run and performance data (also called profiling data) is collected.
- **Analysis and visualization:** the performance data is analyzed to extract useful information to be visually presented to a user.

Profiling is often an iterative process. Once a user analyzes the visualized performance data for a profiled program, further data may be required to understand the performance, or changes can be made to the problematic areas of the program. In both cases, a user profiles the program again, possibly refining the instrumentation. The process continues until a program's performance is acceptable to a user.

2.1 Instrumentation

In the instrumentation insertion phase, additional code is added at specific locations in a program to generate performance data during execution. Instrumentation can be broken down into points, primitives and predicates [GKM82, MCC$^+$95]:

- An **instrumentation point** is a location in a program's code where instrumentation is inserted.
- An **instrumentation primitive**, a counter or timer with operations to change its value, is used to collect performance data.
- An **instrumentation predicate** is a boolean expression that guards the execution of an instrumentation primitive (e.g., an **if** statement).

For example, in the Exact Routine Call-Graph metric described in Chapter 5, a counter counts the number of times a routine is called and a timer tracks the time spent executing a routine. Hardware counters are also available to count hardware events, such as the number of instructions executed, over a given period. However, although a program may make the same number of routine calls, for example, each time it is run, the number of routine calls counted up to a specific time in the program's execution may differ between multiple runs of the program because of the unpredictable interaction of tasks (nondeterminism) in a concurrent program.

An instrumentation primitive and instrumentation predicate form a hook, which is inserted at various instrumentation points throughout the program to be profiled.

Probe Effect

The insertion of instrumentation into a program results in an overhead, with respect to both time and space, called a probe effect. The extent of the probe effect is dependent on the amount and type of instrumentation inserted, the frequency at which instrumentation is executed, as well as the type of program being profiled (i.e., sequential or concurrent). The probe effect can change the run-time behaviour and performance characteristics of a program; therefore, minimizing the probe effect is an important goal for a profiler [LP85, MH89].

In sequential programs, the probe effect results in an increase in running time for the program, but no change in program behaviour, unless the program's behaviour depends on time. However, in concurrent programs, the probe effect can lead to the disappearance of existing performance problems or their movement to different (or unexpected) locations. New performance problems may also appear [HM93].

2.1.1 Direct and Indirect Instrumentation

Instrumentation is either direct or indirect. In direct instrumentation, code is placed at instrumentation points (see Figure 2.1). In indirect instrumentation, execution jumps from an instrumentation point to a profiling routine, called a trampoline, and returns once the instrumentation code in the trampoline executes (see Figure 2.1). Although indirect instrumentation has a higher probe-effect as a result of the routine-call-like jump, modularizing the instrumentation code reduces code duplication and facilitates the dynamic insertion, modification and removal of instrumentation.

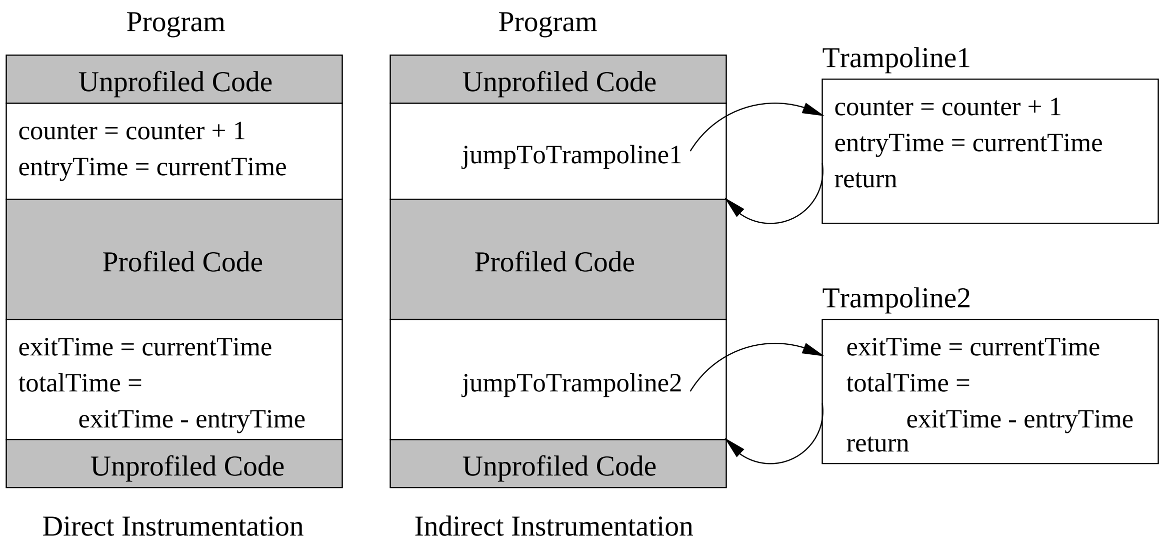


Figure 2.1: Direct and Indirect Instrumentation

2.1.2 Instrumentation via Insertion

Instrumentation insertion can be done at almost any point along the compilation/execution chain, e.g., during program composition, preprocessing, compilation, linking, executable re-writing or execution. If instrumentation is inserted higher in the chain (e.g., at program composition), it is programming language dependent, but system/architecture independent (as long as the language is supported) [She99]. If instrumentation is inserted lower in the chain (e.g., by executable re-writing), it is programming language independent, but system/architecture dependent.

The two broad categories generally considered for instrumentation insertion are static and dynamic.

Static Insertion

Static insertion is instrumentation inserted at any point before program execution [Zak00]. Static insertion is used by the majority of profilers, as it is both easier and less time-consuming than dynamic insertion, and it can collect performance data that is very difficult to obtain by other methods [Den97]. An example of such data is information regarding the callers and callees of a routine, which is required when generating a call-graph. Sometimes instrumentation is inserted at a point in a program that does not help in the location of performance problems, resulting in unnecessary performance data and probe effect. Through the use of instrumentation predicates (see Section 2.1), statically inserted instrumentation can be disabled; however, the instrumentation (and hence the cost and effect) cannot be completely removed without stopping program execution and recompiling. Static insertion is best suited for short to medium running programs since long-running programs magnify the negative effects of unnecessary instrumentation.

Dynamic Insertion

Dynamic insertion is instrumentation inserted during program execution [Hol94]. In an iterative process, algorithms are run by the profiler to determine when, where and what type of instrumentation needs to be added to or removed from an executing program. These decisions are often related to how effective an instrumentation point is at locating a performance problem and the movement of performance problems during execution (especially if the profiled program is a concurrent program). Dynamic insertion is best suited for long-running programs as decisions about insertion can take time, but the instrumentation can be selective.

No Instrumentation

Profiling can be done without inserting any instrumentation into a program; instead the state of the program is polled or sampled at regular intervals (see Section 2.2.2).

2.1.3 Instrumentation via Hardware Counters

Hardware counters can with low cost, and hence low probe-effect, collect information (e.g., the number of CPU cycles elapsed) inaccessible by any other method of instrumentation. Once configured, hardware counters run in parallel with the executing program at the hardware level. Therefore, the cost associated with hardware counters comes almost entirely from the reading of the counters, storing of the count values and writing of the counters.

2.2 Monitoring

Monitoring is the process of collecting, filtering and storing performance data generated by the instrumentation during a program's execution (filtering is optional, but can substantially reduce profiling storage requirements by eliminating irrelevant data). For a concurrent program, data collection and storage is often done on a per-task basis; i.e., data is stored according to the task executing at the time the data is generated, rather than aggregated across all tasks. This task separation is carried forward into the analysis and visualization phases. Sometimes it is also beneficial to collect and store performance data based on other constructs such as coroutines or objects. For a user, the separation of the performance data allows for a more precise understanding of a program's performance problems.

Monitoring is divided into two forms: exact and statistical.

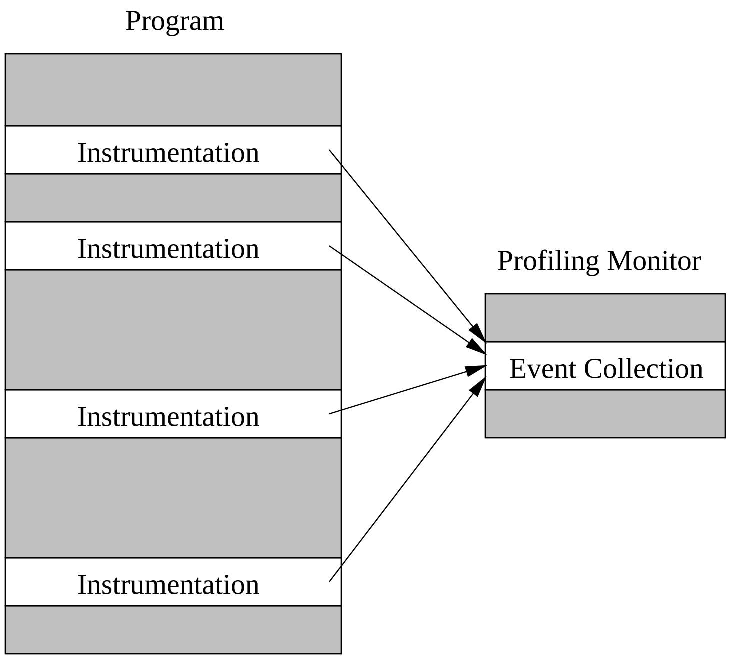


Figure 2.2: Exact Monitoring

2.2.1 Exact Monitoring

Exact monitoring (also called event-driven monitoring) collects data at each occurrence of all relevant events. In this case, the profiling monitor is notified when instrumentation associated with a relevant event is triggered during program execution (see Figure 2.2). A routine call, for example, is a relevant event when generating a call-graph. By collecting data at the occurrence of each event, exact monitoring provides accurate performance data, but at the cost of high overhead, both in time and space, and consequently a higher probe-effect. The amount of data collected (i.e., space requirements) can be reduced by restricting the scope of the monitoring (to specific program segments), dynamically filtering unnecessary data or by aggregating data on-the-fly. Overall, exact monitoring is used to provide an accurate event trace for a short-running application or short segment of a long-running application, or a summary of execution rather than a full trace.

2.2.2 Statistical Monitoring

Statistical monitoring (also called polling or sampling) collects data only at specific intervals called sampling intervals or periods. In this case, the profiling monitor polls the executing program at these specific intervals to obtain information about the program's execution state (see Figure 2.3). The structure of the call-stack, for example, is state information relevant when generating a call-graph. The sampling interval can be based on time (e.g., every 10 milliseconds) or the occurrence of hardware events (see Section 2.2.3). By collecting data only at specific intervals, statistical monitoring has lower overhead, both in time and space, and consequently a lower probe-effect, but at the cost of less accurate performance data. The smaller the sampling interval, the greater the accuracy and overhead of the information; however, this approach can never replace exact monitoring when complete accuracy or event coverage is essential. For example, if statistical monitoring is used to trace a program's state transitions, the resulting trace does not cover all transitions, resulting in anomalies like a transition from a blocked state to another blocked state with no intervening execution.

2.2.3 Hardware Counters and Monitoring

Hardware counters are useful for both exact and statistical monitoring. In exact monitoring, hardware counters are used to determine the number of hardware events that occurred during the execution of a specific section of code by subtracting the counter value read at the start of the code section from the counter value read at the end of the code section. In statistical monitoring, the sampling interval can be based on the occurrence of a specific number of hardware events.

In general, hardware counters count from 0 to $2^w - 1$, where w is the architecture-dependent width of the counters in bits. However, to generate a sampling interval of n events, the hardware counter is set to a value of $2^w - n$. When the count exceeds $2^w - 1$, an overflow signal is generated and delivered to the profiling monitor, which samples the program's execution state before resetting the counter to $2^w - n$. For a concurrent program, hardware-event counts are virtualized across threads by storing/restoring the counters during context switching.

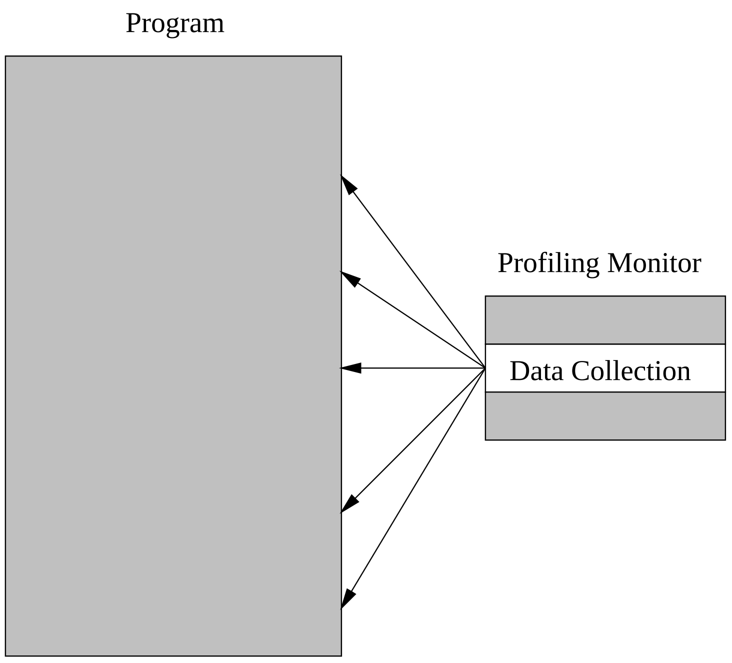


Figure 2.3: Statistical Monitoring

2.3 Analysis

Performance data must be analyzed to extract useful information about a program's behaviour for visualization. Before the data is analyzed, it can be optionally filtered in order to reduce the size of the data set, and hence, the time required for analysis. Various calculations and/or algorithms are performed on the data, and, if possible, the data is mapped back to the program's source code. Once visualized, such processing results in information that is much more understandable to a user and conducive to locating performance problems. Additional preparation of the information may be required depending on the format of visualization (e.g., summary view, detailed view).

Analysis of performance data for a concurrent program, versus a sequential program, is more complex because of the separation of the data by task (and possibly other constructs). Not only does each separate group of data need to be individually analyzed, but the data from the separate groups should be compared so that performance problems due to their interactions can be discovered (such analysis can be left to the user during visualization).

Analysis can be done in real-time (also called on-the-fly), post-mortem or a combination of the two.

2.3.1 Real-Time Analysis

Real-time analysis is done during program execution. Two advantages of real-time analysis are the ability to dynamically filter unnecessary performance data and the ability to process performance data on-the-fly, in both cases reducing storage requirements for profiling. A further advantage is that the information extracted during the analysis can form the basis for decisions made by the profiler or user regarding dynamic (i.e., decisions regarding the insertion and removal of instrumentation) or static (i.e., decisions regarding the enabling or disabling of instrumentation predicates) instrumentation before the program has finished execution.

The major disadvantage of real-time analysis is the higher probe-effect that results from performing the analysis and the display of any visualization. Furthermore, the analyzed information may only be available after a delay, if the events generating performance data are occurring quickly and/or the analysis and visualization is time-consuming; therefore, effective decisions regarding instrumentation adjustment are difficult for the profiler and almost impossible for a user to make, especially for short-running programs. For this reason and the ability to reduce storage requirements, real-time analysis is best suited for long-running programs.

2.3.2 Post-Mortem Analysis

Post-mortem analysis is done after the program has finished execution, meaning the monitoring process is the only contributor to the probe effect. However, dynamic filtering of unnecessary performance data is impossible and no information is available to make instrumentation adjustments. For these reasons, post-mortem analysis is best suited for short to medium running programs. Finally, some profilers save performance data to a file to allow post-mortem analysis at any time, i.e., even after the visualizations of the performance data have been terminated.

2.3.3 Combination

A combination of real-time and post-mortem analysis can also be used. For example, real-time analysis can be done to allow for dynamic instrumentation by the profiler and post-mortem analysis can be done to process the collected data for user visualization.

2.4 Visualization

Visualization is the last step in the profiling process. The goal of visualization is to display the performance data so that a user can understand it and ultimately make decisions regarding the performance of a program. To achieve this goal, the visualization needs to effectively and clearly convey all pertinent information without overwhelming a user.

As previously mentioned, for a concurrent program, data is collected, stored and analyzed according to how it is separated (e.g., per appropriate constructs). In general, the data needs to be visualized in a similar manner to reflect the user's high-level execution model. Because visualization can be complex, displaying summary information for different groupings on a single screen can help direct a user in choosing which group to examine in greater detail. Performance data can be visualized as tables, charts and graphs (for further visualization techniques refer to [Tuf83]).

Tables

Tables, the simplest form of visualization, display discrete values (e.g., numerical data) arranged in rows and columns, and are often used when significant detail needs to be conveyed.

Charts

Charts are pictures or diagrams that display discrete values. The pictorial format makes trends in a set of data (as well as among multiple data sets) easier to see. Examples are bar charts or histograms, pie charts and Gantt charts [MR82].

Graphs

Graphs use points, lines and surfaces to represent multi-dimensional relations [Den97]. Like charts, graphs use a pictorial format to make trends easier to see, but unlike tables and charts, graphs can display continuous values.

Chapter 3

μProfiler

μProfiler is a concurrent, object-oriented profiler for concurrent, object-oriented programs written in μC++. μProfiler provides multiple metrics for displaying information about the dynamic behaviour of a program, where each metric is composed of monitoring, analyzing and visualizing one or more aspects of program performance. Initial work on μProfiler was done in 1997 by Robert Denda [Den97]. In 2000, Dorota Zak [Zak00] added a number of new metrics, and in 2005, Josh Lessard [Les05] added hardware counters and metrics utilizing these counters.

This chapter describes the design and implementation of μProfiler, covering Robert, Dorota and Josh's previous work, in addition to changes I have made during the development of this thesis.

3.1 Target Environment

A profiler can be loosely or tightly coupled with its execution environment. Loose coupling indicates a weak integration of the profiler with the target language(s) and run-time system; hence, the profiler makes few (high level) or no assumptions about the environment's execution model. On the other hand, tight coupling indicates a strong integration of the profiler with the target language(s) and run-time system; hence, the profiler is aware of and can access constructs intrinsic to the environment's execution model and provide fine-grained performance data based on those constructs. Consequently, loosely-coupled profilers can often profile programs written in a variety of programming languages, whereas tightly-coupled profilers concentrate on a single or very small subset of similar programming languages.

μProfiler is tightly-coupled with its target execution environment; the profiler collects performance data from the execution environment and expresses results in terms of the environment's concurrent execution model, i.e., performance data is separated by and related back to the concurrency constructs in the environment. Tight coupling facilitates performance analysis and deeper understanding by allowing a programmer to continue to think in terms of the specific execution model used during program implementation.

This section describes the execution environment needed to understand the design and implementation of μProfiler presented in the remainder of this thesis.

3.1.1 μC++

The target environment for μProfiler is a concurrent extension of the C++ programming language [Str97] called μC++ [BDS+92, BS07]. μC++ extends C++ with new language constructs providing advanced control-flow, including lightweight concurrency, on uni-processor shared-memory computers (by interleaving task execution) and parallel execution on multi-processor shared-memory computers (by interleaving and true parallel execution).

μC++ is implemented using a translator and a run-time library (called the μC++ kernel), and provides an M:N user-to-kernel-thread model. The translator reads a μC++ program containing language extensions and transforms each extension into C++ statements. A C++ compiler generates the program's object code and links it to the μC++ run-time library. The μC++ kernel is responsible for creating, managing and destroying the new language constructs as well as for task scheduling.

3.1.2 μC++ Language Constructs

μC++ provides its own execution model through the introduction of six new language constructs that support concurrent execution. These constructs are coroutines, monitors, coroutine-monitors, tasks, virtual processors, and clusters. Only the coroutine and task constructs are relevant to this thesis (see [BS07] for details on the other constructs).

3.1.2.1 Coroutine

A coroutine is a programming language object that contains its own execution state (including a stack). Like a routine, a coroutine does not have its own thread of control; it uses the thread of its caller to advance its own execution state. Unlike a routine, a coroutine's execution can be inactivated as control returns to its caller (task or coroutine) without terminating. Therefore, the coroutine can be again activated at a later time and it restarts from the point where it was last inactivated (rather than from the beginning), continuing with the local state (i.e., local variables) that existed at the point of inactivation.

A coroutine has one distinguished member routine called `main`. Direct interaction with the `main` routine is not permitted, so a coroutine can only be activated indirectly through a call to one of its public member routines. A public member routine executes a resume statement which explicitly activates `main`, at the point of the last inactivation, and execution of the caller's thread moves from the caller's stack to the activated coroutine's stack. A coroutine can be inactivated by executing a suspend statement and reactivating its caller, or by activating another coroutine by calling one of its public member routines containing a resume statement. In either case, execution of the currently active thread moves from the inactivated coroutine's stack to the activated coroutine's stack.

3.1.2.2 Task

A task is a programming language object that contains its own execution state (as for a coroutine), mutually exclusive execution of its member routines, and its own thread of control. A task has a distinguished member routine called `main` in which the new thread starts execution, and as for a coroutine, interaction with the task is through public member routines. A task's thread can execute on the task's stack as well as on the stack of another coroutine. A task's thread runs concurrently with all other task threads in the same program.

Tasks and coroutines are known as execution entities because both contain their own execution stack.

3.2 Design Objectives

The current implementation of μProfiler fulfills six main objectives. These objectives stem from μProfiler's original design requirements [Den97].

3.2.1 Profiling on a Per-Thread Basis

For a profiler to profile an individual thread or effectively aggregate data across threads, it must be aware of how its execution environment (μC++ in the case of μProfiler) handles thread management and scheduling. Per-thread profiling is essential for concurrent profilers because threads form the basis of a concurrent language's execution model.

3.2.2 Profiling at Different Levels of Detail

Collecting, analyzing and visualizing performance data at different levels of detail is required to provide a user with the most helpful information as well as provide multiple perspectives of the collected data. μProfiler can profile at the cluster, virtual processor, task, coroutine, object, and routine levels, across a number of appropriate metrics.

3.2.3 Selective Profiling

Users may be interested in profiling only certain aspects (e.g., specific tasks, routines etc.) of a program, rather than the entire program. μProfiler provides selective profiling of μC++ programs (i.e., instrumentation control) by allowing a user to specify which program modules and which tasks within the modules are profiled. Per-module profiling is enabled by compiling a module with the `-profile` flag, and profiled and unprofiled modules are compatible. Per-task profiling can be dynamically enabled and disabled for a task during execution by calling the profileActivate and profileInactivate routines.

3.2.4 Support Different Forms of Visualization

Different metrics collect different performance data, each requiring various forms of visualization. Sometimes the same data needs to be presented in multiple ways. μProfiler supports several different visualization forms, from textual to graphical, and provides a custom Motif widget [HF94] for each one.

3.2.5 Extendibility

Programs, and especially concurrent ones due to their complexity, often require a wide range of metrics to measure the various performance problems that arise. Profilers provide a set of built-in metrics, but when situations arise that cannot be adequately handled by those metrics, the profiler should allow users to add their own metrics. μProfiler can be extended in this way through inheritance, allowing a user to derive a new monitor, analyzer and visualizer for a new metric from a corresponding set of base classes. The new metric can be attached to μProfiler without recompilation.

3.2.6 Portability, Interoperability, and Maintainability

μC++ supports several operating system/architecture pairs, with μProfiler currently running on three of those operating system/architecture pairs. However, nothing in μProfiler's design or implementation prevents a port to any other systems.

Maintainability is an essential design consideration during the software development process as it makes future corrections, improvements and adaptations easier. Maintainability has been an important objective of the work done in μProfiler for this thesis. Numerous new reusable components have been developed, and consistency, in data structures and visualization, across metrics has been greatly increased.

3.3 Instrumentation Insertion

μProfiler uses both direct and indirect instrumentation insertion. For direct instrumentation, hooks are inserted into the μC++ kernel, and for indirect instrumentation, shared trampoline calls are inserted into the user code of the μC++ program during compilation.

3.3.1 μC++ Kernel Instrumentation

Hooks have been inserted at various locations in the μC++ kernel and are present whether or not a target program is being profiled. However, a hook is only triggered, and hence performance data is only collected, if the instrumentation predicate guarding its execution evaluates to true.

Figure 3.1 is an example of a μC++ kernel hook that can be triggered when a task changes its execution state. This hook is triggered for the Execution State metric described in Chapter 4. The **if** statement surrounding the routine call is the predicate, and its boolean expression must evaluate to true in order for the hook to be triggered. The boolean expression is true if the profileActive flag is true and if uProfiler::uProfiler_registerTaskExecState is non-null. The profileActive flag is true when profiling is enabled for the currently active task (i.e., the task changing state). uProfiler::uProfiler_registerTaskExecState is a routine pointer that, if non-null, points to the uProfiler::registerTaskExecState member routine. uProfiler::uProfiler_registerTaskExecState is non-null when at least one module of the program is compiled with the -profile flag and at least one metric requiring this hook is selected, i.e., that metric's execution monitor (see Section 3.5.1) has registered to receive notifications upon triggering. All μC++ kernel hooks are structured and activated in the same way.

3.3.2 User Code Instrumentation

Shared trampoline calls are inserted into a target program during compilation. The -profile flag tells the μC++ translator to activate the -finstrument-functions flag, which in turn tells the C++ compiler gcc [GCC] to insert the trampoline calls. gcc inserts an entry trampoline call at each routine entry and an exit trampoline call at each routine exit, for each routine in a module. The

```
void uBaseTask::setState( uBaseTask::State s ) {
    ...
     if ( profileActive && uProfiler::uProfiler_registerTaskExecState ) {
          (*uProfiler::uProfiler_registerTaskExecState)( uProfiler::profilerInstance, *this, state );
     }

    ...
}
```

Figure 3.1: A μProfiler Instrumentation Hook

trampoline calls are passed the address of the routine being entered or exited, and the address of the call site in its caller routine. These shared trampolines are inserted for the Routine Call-Graph metrics described in Chapters 5 and 6. Figure 3.2 shows the execution of the trampoline during a routine call in a μC++ program. However, if the -profile statistical flag is specified all profiling is activated except the -finstrument-functions flag as trampoline calls are not required for statistical metrics. By specifying the -profile statistical flag, the large number of trampoline calls is avoided, which has a positive impact on the running time of a profiled program.

For modules with inserted trampolines, if routine-level profiling is not enabled for at least one active metric requiring it then the trampoline code is not executed and execution returns to the instruction immediately following the trampoline call. Otherwise, metric-specific data structures are updated to reflect the new execution-state (i.e., reflect the current state of the stack - a new routine being entered or a routine being exited), data collection is performed and, if active, the corresponding hook is triggered.

3.4 μProfiler Kernel

The μProfiler kernel provides the main functionality of μProfiler. Figure 3.3 shows, using the object-oriented notation described in Appendix A, the relationship between the μProfiler kernel

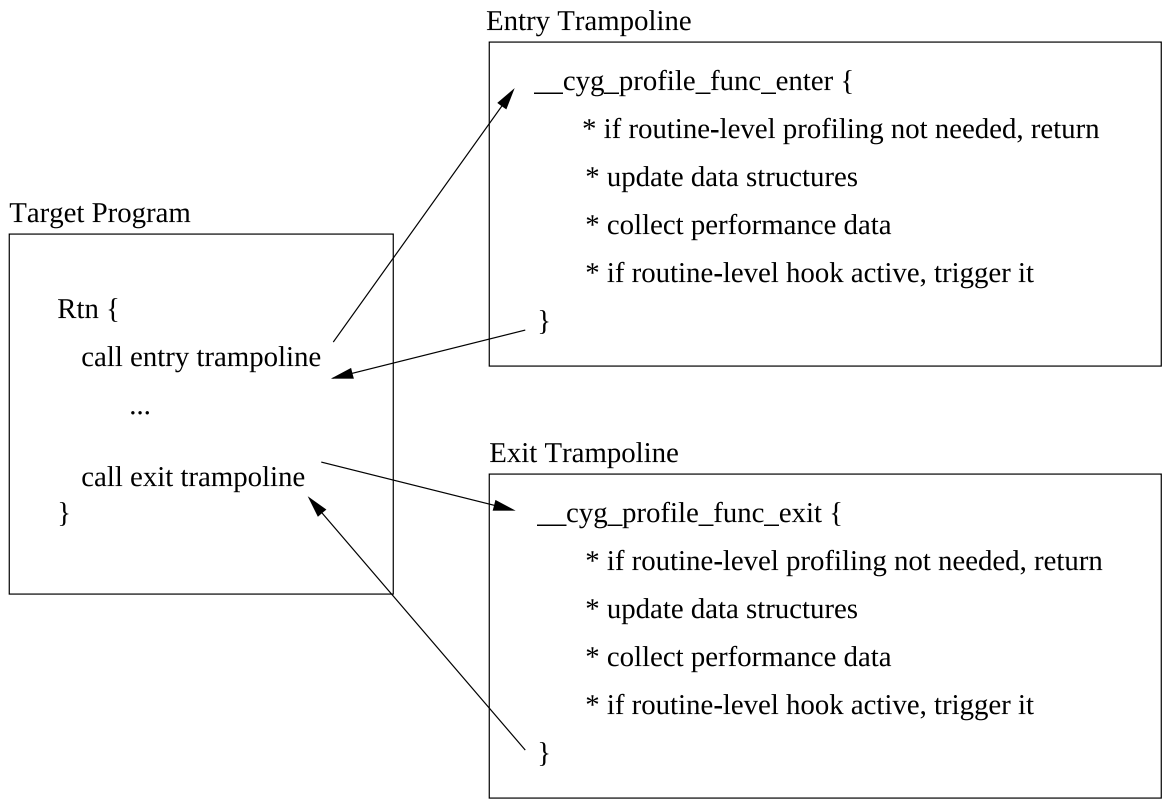

Entry Trampoline
__cyg_profile_func_enter {
* if routine-level profiling not needed, return
* update data structures
* collect performance data
* if routine-level hook active, trigger it
}
Target Program
Rtn {
call entry trampoline
...
call exit trampoline
}
Exit Trampoline
__cyg_profile_func_exit {
* if routine-level profiling not needed, return
* update data structures
* collect performance data
* if routine-level hook active, trigger it
}

Figure 3.2: Flow of Control for Routine-Level Profiling in μProfiler

and the metrics' execution monitors, analyzers and visualizers. The μProfiler kernel consists of the following objects: uProfiler, StartMenuWindow, uProfileTaskSampler, uExecutionMonitor, Analyze, ProfilerAnalyze, and SymbolTable.

uProfiler is a task that acts as a proprietor [Gen81, p. 446] for all active metrics, handling registration and management. Once created, the execution monitor for each active metric registers itself with uProfiler, and registers for any required instrumentation hooks (see Section 3.5.1). Monitors for metrics doing exact profiling are notified by uProfiler when their registered instrumentation hooks are triggered during program execution, indicating to the monitor that an event has occurred. Monitors for metrics doing statistical profiling are notified by uProfiler at specific intervals, indicating it is time for the monitor to sample the program's execution state. Once

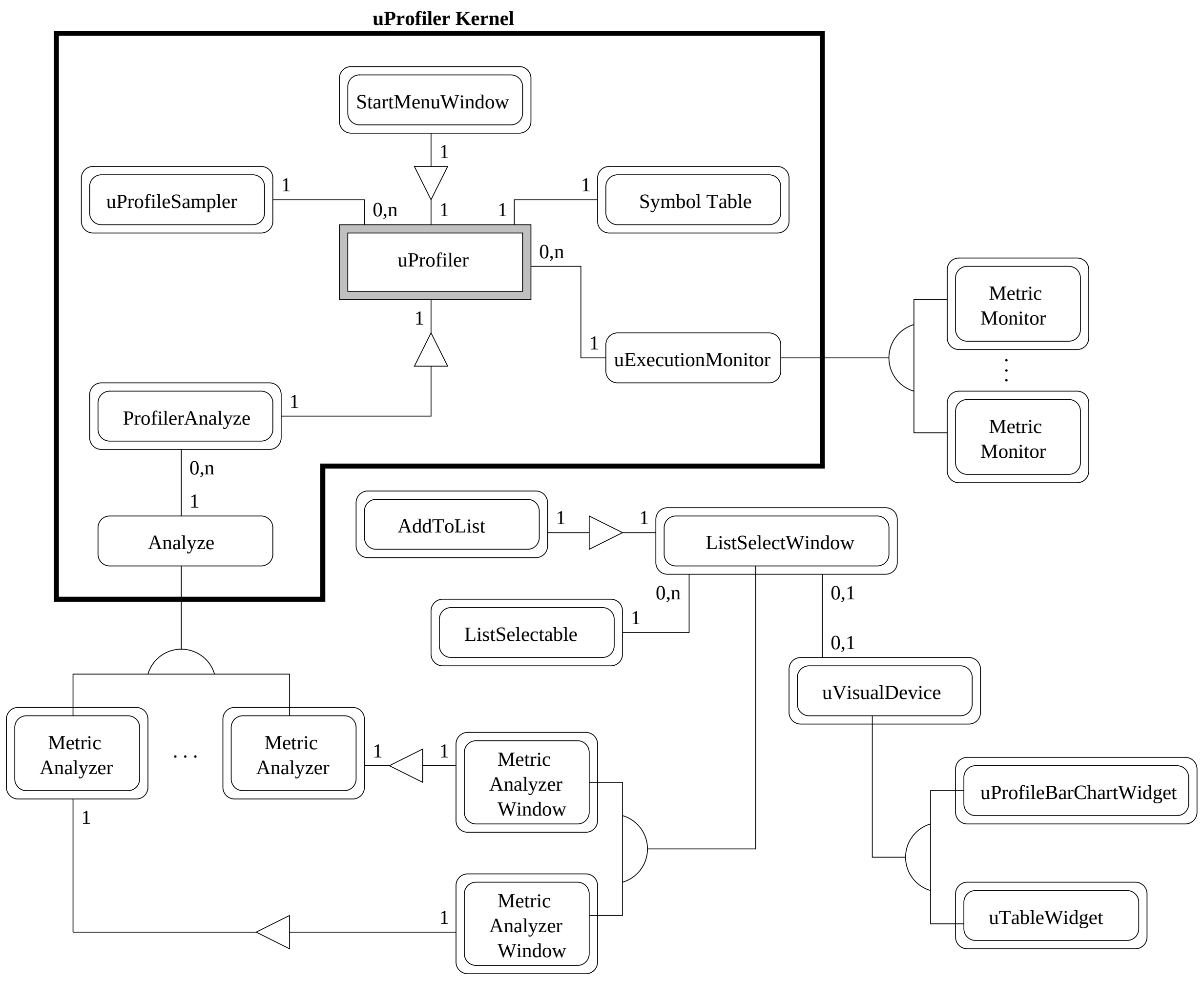


Figure 3.3: Object-Oriented Design of the μProfiler Kernel

monitoring is complete, uProfiler invokes ProfilerAnalyze which creates and invokes an analyzer (see Section 3.5.2) for each registered monitor.

Before the target program is executed, a user must select all desired metrics from the list of available metrics presented on the startup window (see Figure 3.4). StartMenuWindow creates the startup window and also creates and invokes an execution monitor for each selected metric.

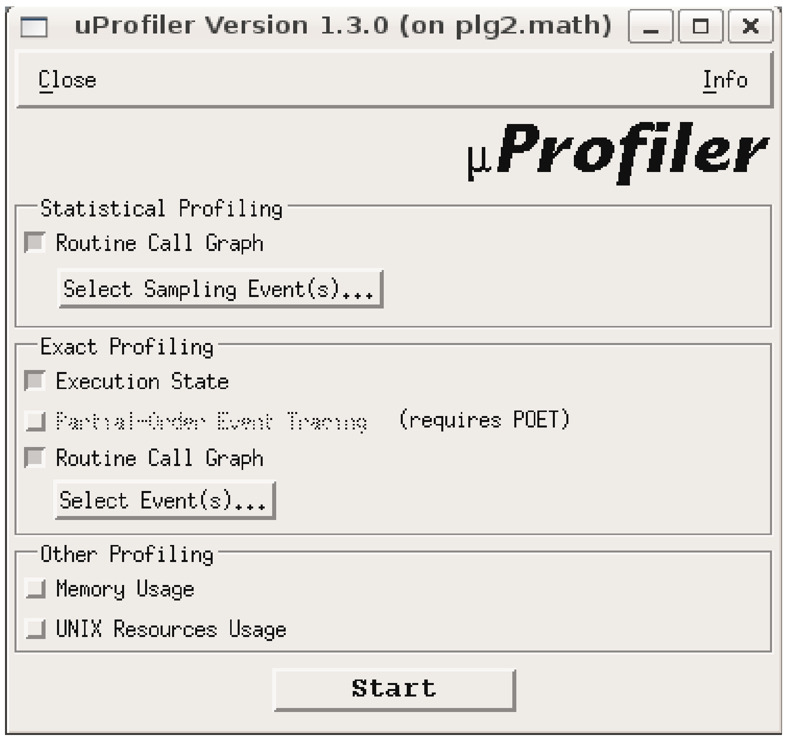


Figure 3.4: μProfiler Startup Window

A uProfileTaskSampler is created for each profiled task and coroutine to store, in various per-metric data structures, the related performance data collected during monitoring. uExecutionMonitor and Analyze are abstract base-classes explained in Section 3.5.

When the target program is compiled, the compiler generates an architecture-dependent symbol table. The program symbol table is accessible through the Binary File Descriptor (BFD) Library [Cha91]. SymbolTable provides a high-level interface to the BFD library, abstracting the symbol table details and providing access to its information (e.g., routine names and locations in files).

3.5 μProfiler Metrics

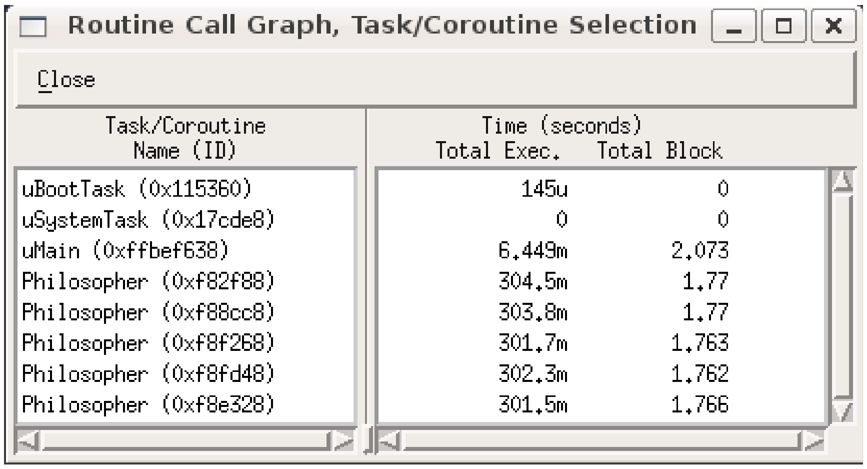
Reflecting the profiling process, a μProfiler metric consists of an execution monitor, analyzer and visualizer. Firstly, an execution monitor, derived from the uExecutionMonitor abstract base-class, collects performance data during the monitoring phase of profiling. Secondly, an analyzer, derived from the Analyze abstract base-class, processes the performance data during the analysis phase of profiling. Finally, a visualizer, using a device provided by μProfiler or derived from the uVisualDevice base class, displays the processed performance data on screen during the visualization phase. In this way, work related to the various phases remains separated and each metric becomes a separate entity allowing for easy extendibility and maintenance.

3.5.1 Execution Monitors

μProfiler's execution monitors are passive objects that monitor a target program's run-time behaviour. A monitor registers with and is managed by uProfiler, as described in Section 3.4. Furthermore, for those monitors which register for hooks with uProfiler, the uExecutionMonitor abstract base-class includes one hook-notification routine for each hook, which is defined by the derived monitor and called when that particular hook is triggered.

3.5.2 Analyzers and Visualizers

μProfiler does post-mortem analysis. Therefore, only after monitoring (and program execution) is complete does uProfiler invoke ProfilerAnalyze to create and invoke the analyzers for all registered monitors. The uExecutionMonitor base class has a virtual routine called createAnalyze, which is defined by the derived execution monitor and called by ProfilerAnalyze to create the analyzer object.



Task/Coroutine Name (ID)	Time (seconds) Total Exec.	Total Block
uBootTask (0x115360)	145u	0
uSystemTask (0x17cde8)	0	0
uMain (0xffbef638)	6.449m	2.073
Philosopher (0xf82f88)	304.5m	1.77
Philosopher (0xf88cc8)	303.8m	1.77
Philosopher (0xf8f268)	301.7m	1.763
Philosopher (0xf8fd48)	302.3m	1.762
Philosopher (0xf8e328)	301.5m	1.766

Figure 3.5: μProfiler Task/Coroutine Selection Window

Once analysis is complete, visualization of data begins; however, further analysis may occur as a user makes selections or chooses certain options on the various visualization windows. Each μProfiler analyzer creates an analyzer window (a selection window), derived from a common base-class called ListSelectWindow. ListSelectWindow is useful for summary information, providing routines for left/right panes with selection for drilling down in the data. For example, in Figure 3.5, call-graph summary information is displayed for each profiled task and coroutine listed on the left-hand pane. By clicking on a task or coroutine, another window is displayed providing specific information for that selection (e.g., the per-task or per-coroutine call-graph). The windows displaying specific information derive from a common base-class, called ListSelectable, and often also derive from the class TextInfoWindow. TextInfoWindow is useful for detailed information, providing routines for creating and managing various types of window panes (e.g., hideable panes, clickable panes etc.). These powerful base classes are available in μProfiler to simplify construction of complex graphical user interfaces.

3.5.3 Alternative Profiler Design

Other profiler designs exists to the one monitor, analyzer and visualizer metric-design described for μProfiler. The alternatives decouple the monitors, analyzers and visualizers from one another as opposed to the tightly-couple approach in μProfiler. Monitors would deposit performance data into a common repository that is accessible to all the analyzers and visualizers, allowing data to be more easily analyzed and visualized in multiple ways as well as enabling the addition of new metrics into a profiler. Clearly, this design is very different and much more complex than the current μProfiler design, but is a potential long-term goal for μProfiler.

3.6 Accessing Hardware Counters

μProfiler has support for hardware counters on three different architectures: the UltraSPARC I/II/III running Solaris, the x86 (including Intel Pentium/MMX/Pro/II/III/4 and AMD Athlon) running Linux, and the IA-64 (Itanium 2) running Linux.

Each processor has a fixed number of hardware counters, each with a set of countable hardware events. Because the hardware counter properties of each processor vary, this information is encapsulated in per-processor event tables. Usually, the number of hardware counters is fairly small (e.g., only 2 counters for the UltraSPARC III [Sun04]) and a particular hardware event is often bound to only a subset of those counters; therefore, only a limited number of events can be counted at any given time.

A substantial amount of fairly complex code must be executed in order to cause the hardware counters to count specific hardware events, but these underlying details are encapsulated and abstracted away by the HWCounters class [Les05]. The HWCounters API provides the programmer with routines to choose events to be counted as well as to read from and write to the various counters counting those events. User-level events, system-level events, or both can be counted. For metrics such as the Exact and Statistical Routine Call-Graph metrics described in Chapters 5 and 6, a user can decide which level of events to count via an options box.

Chapter 4

Execution State Chart

This chapter describes the advances made in μProfiler's Execution State Chart (**ESC**) within the Execution State (**ES**) metric.

The **ESC** displays the states of individual tasks during execution. In μC++, a task can transition through five states during execution:

- **start:** the task has been created but has not started execution.
- **ready:** the task is ready to execute but is not currently scheduled for execution.
- **running:** the task is executing on a processor.
- **blocked:** the task is waiting for an event to occur.
- **end:** the task has finished but has not been deleted.

The **ES** metric collects the required data through tracing: each state entered and the duration of the state is recorded on a per-task basis. The **ESC** uses a Gantt Chart [MR82] to display the states for every task over the entire execution of the program, i.e., one continuous line per-task. An example display of the initial implementation is presented in Figure 4.1. Each line is subdivided into segments representing the states. The colour of a segment indicates the type of state, and the segment length indicates the duration of the state. The states **start** and **end** are in yellow, **ready** in blue, **running** in green, and **blocked** in red. The name of the μC++ task associated with each line appears to the left of the chart, and the X-axis shows the elapsed time of execution.

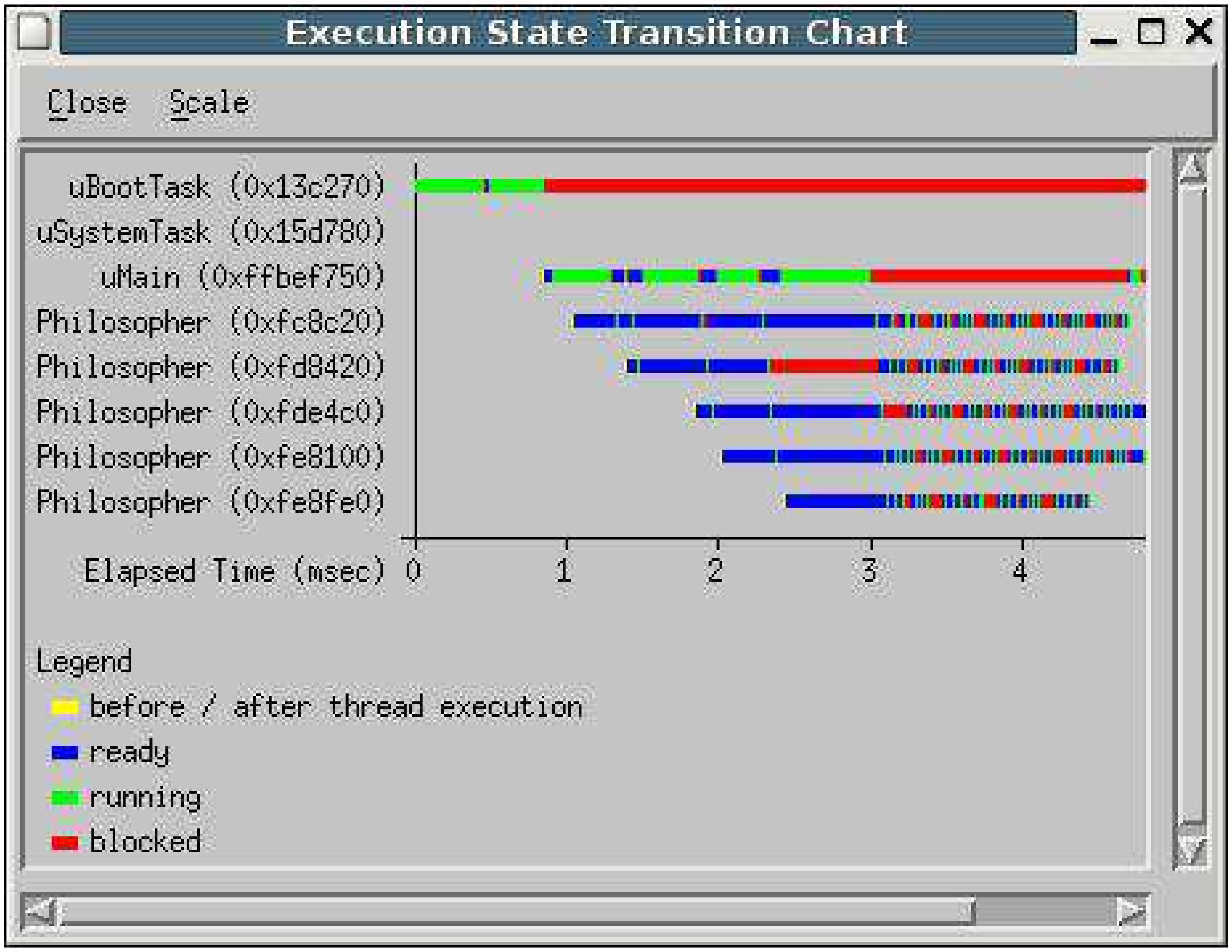


Figure 4.1: Initial Implementation: Execution State Chart Display

A user is able to magnify the chart (i.e., zoom-in and zoom-out). Zooming-in increases the magnification, showing the chart in greater detail so each pixel represents a smaller duration of execution time, and hence, each state (and line) expands in length. The duration of time represented by one pixel is the scale ratio. In the initial implementation of the **ESC** [Zak00], the scale ratio is updated as a user changes the scale factor in the “Scale” pull-down menu (see top menu bar in Figure 4.1). The scale factor is an integer value between 0 and 9. To compute the scale ratio the entire execution duration is divided by a number associated with the current scale factor. At scale factor 0 the number is 300, and the number increases to 30,000 by scale factor 9. Therefore, the larger the scale factor, the smaller the scale ratio and the higher the magnification.

The overall goal of the advances discussed in this chapter is to develop μProfiler’s **ES** metric, primarily the **ESC**, into a state-of-the-art metric with good performance that scales to programs of long duration and with large numbers of tasks and states.

4.1 Initial Implementation Issues

This section describes several issues arising in the initial implementation of the **ESC**. I addressed each issue in the advanced implementation of the **ESC** and the solutions are discussed in Section 4.2.2.

The first issue involves scaling the **ESC** to programs with large numbers of tasks and long execution. In the initial implementation, the entire chart (i.e., the entire execution of the program) is drawn, all at once, into an X-window drawing area. A user then uses the horizontal and vertical scrollbars to move the window over the drawing area. However, an X-window drawing area is restricted to 32,000 x 32,000 pixels. Given this restriction, if the number of tasks within a program is sufficiently large, the drawing area cannot vertically accommodate a line for each task, or if a line is sufficiently long (i.e., long execution or high magnification), the drawing area cannot horizontally accommodate the entire line. In fact, some instances of the X-server, which handles the display and input devices, terminate the profiler application if the drawing area exceeds the X-server’s size restrictions, and hence, no information is displayed for a user. The second issue involves a loss of information at lower magnification. At lower magnification, many states are represented by line segments too small to draw, i.e., less than one pixel in width. Such a loss of information can result in a chart that is very confusing and misleading for a user. For example, two distinct states of the same type can appear as one continuous state if the states occurring between them are not drawn because they are too small at the current magnification. In an attempt to deal with this problem, the initial implementation always draws at least one pixel for each line segment. However, this artificially lengthens the line, and consequently the elapsed time of execution does not always correctly match with the X-axis tick-marks.

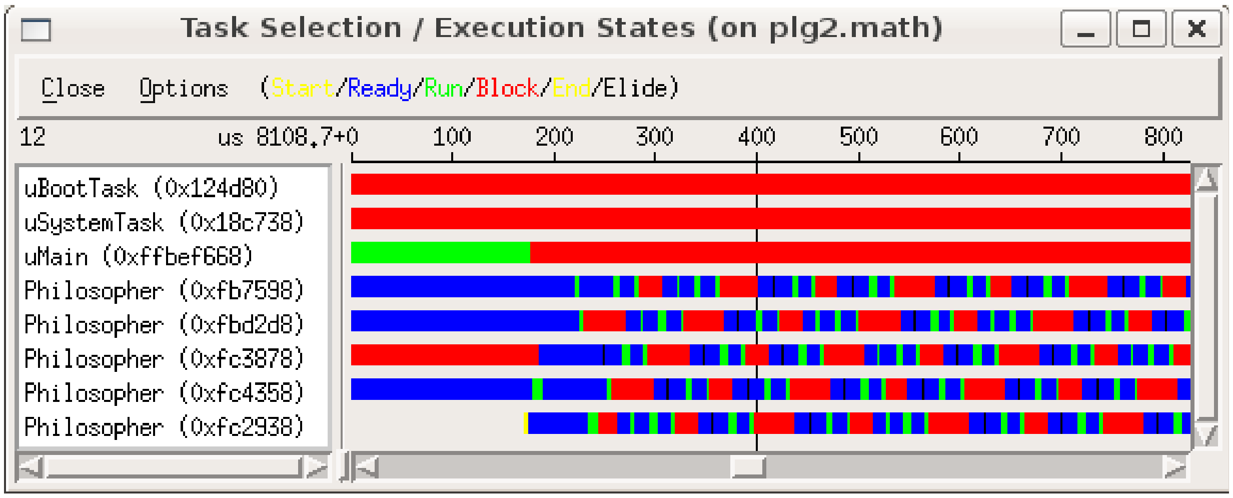
The third issue involves the X-axis. In the initial implementation, the axis always shows the elapsed time of execution in millisecond time units. Additionally, no fractional tick-mark intervals (duration of time between two consecutive tick-marks on the axis) are used, and therefore, the smallest tick-mark interval is one millisecond. Millisecond time units are not always the best choice. Using nanoseconds for programs of very short execution and at higher magnification, and seconds for programs of longer execution and at lower magnification is much more appropriate, intuitive and allows for greater precision. Similarly, the use of fractional tick-mark intervals allows a more precise division of the axis so it is easy to associate a position along a line with a specific execution time.

The fourth issue involves the visibility of the legend and the task names. Both the legend and the task names appear to the left of the chart, but are drawn into the same drawing area as the chart itself (see Figure 4.1). Therefore, as a user moves the horizontal scrollbar to the right these items disappear from the window. The consequence of the legend no longer being visible is often minimal. However, given a large number of tasks it can be difficult to remember which line corresponds to which task. This forces a user to constantly move the horizontal scrollbar between the region under examination and the far left side.

The final issue involves visualization performance. As the number of states grows larger (e.g., several million states), the amount of time needed to process the state data in order to display the corresponding lines also becomes larger. The decline in performance is noticeable and can be frustrating for a user (e.g., 5-30 seconds of delay when zooming as the entire set of state data must be processed and redisplayed). This issue is significant because programs that make several million state transitions are common.

4.2 Advanced Implementation

In addition to addressing the issues from the initial implementation of the **ESC**, the advanced implementation has also progressed in other areas. An example display is presented in Figure 4.2. The advanced display consists of two panes, the task pane (left) containing the task names and the chart pane (right) containing the states.

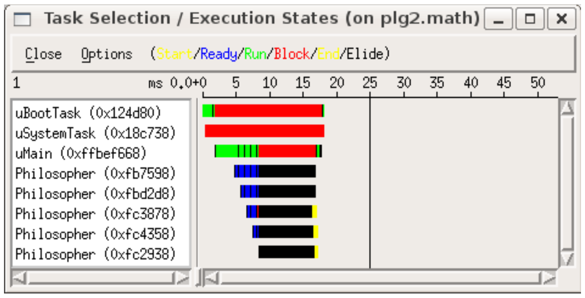

Task Selection / Execution States (on plg2.math)
Close
Options
(Start/Ready/Run/Block/End/Elide)
12
us 8108.7+0
100
200
300
400
500
600
700
800

```
uBootTask (0x124d80)
uSystemTask (0x18c738)
uMain (0xffbef668)
Philosopher (0xfb7598)
Philosopher (0xfbd2d8)
Philosopher (0xfc3878)
Philosopher (0xfc4358)
Philosopher (0xfc2938)
```

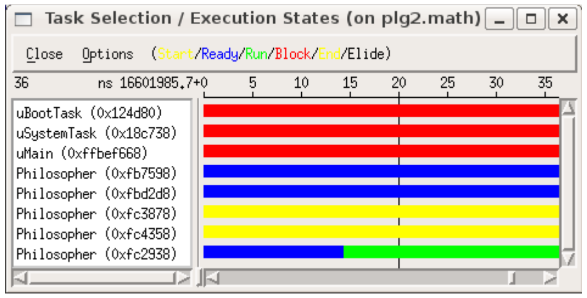
Figure 4.2: Advanced Implementation: Execution State Chart Display

4.2.1 Implementation Details

Each pixel of a line in the chart pane corresponds to a specific duration of execution time. As in the initial implementation, the number of nanoseconds corresponding to one pixel is represented by a scale ratio. The scale ratio is a continuous (floating-point) value. Requiring a user to set the scale ratio in order to adjust the magnification is problematic because a user is forced to relate particular magnifications to long floating-point values. Furthermore, the existence of a large number of magnifications can clearly complicate user interaction. Therefore, I chose to have a user indirectly control the magnification (i.e., the chart detail) by adjusting either of the two parameters, the scale factor or the magnification step, instead of directly adjusting the scale



(a) Minimum



(b) Maximum

Figure 4.3: Advanced Implementation: Magnification Range

ratio, restricting adjustment to discrete integer values. The purpose of the restriction is solely to simplify user interaction by providing small, repeatable values for controlling the magnification.

The scale factor is a unit-less integer value between 1 and a maximum. The maximum indicates the number of magnifications between the minimum magnification (i.e., at scale factor 1, see Figure 4.3(a)), where each state is maximally compressed in length, and the maximum magnification (i.e., at the maximum scale factor, see Figure 4.3(b)), where each state is maximally expanded in length. As the scale factor is increased, the scale ratio decreases, and hence, the magnification increases (see Equation 4.1). The number of integral scale factors (i.e., the maximum scale factor) is determined by the magnification step. The magnification step defines the percentage change in magnification for each step in the scale factor. Defining the change in this way allows a user to magnify more quickly, but still have fine-grained control with the selection of the magnification step. As a result, the scale ratio is an (inverse) exponential function of the $magStep$ and $scaleFactor$ (see Equation 4.1). The lower the magnification step, the greater the number of scale factors because each step in scale factor represents a smaller percentage change in magnification; therefore, more steps are required to reach the maximum magnification. For example, a magnification step of 100% means that at each step of the scale factor the magnification is doubled. The chart displayed at the minimum and maximum magnifications is the same regardless of the magnification step.

The scale ratio (and correspondingly the chart displayed) is updated given any change in the scale factor (*scaleFactor*) or the magnification step (*magStep*). The scale ratio is computed by the following formula

$$scaleRatio = \frac{totalDuration}{magStep^{\,scaleFactor-1} \times MinDisplayPixels} \tag{4.1}$$

The smaller the scale ratio, the higher the magnification because each pixel represents a smaller duration of execution time. The smallest scale ratio possible, *MinScaleRatio*, is 0.1, meaning that one nanosecond is represented by 10 pixels.

Solving for *scaleFactor* in Equation 4.1 gives the following formula for computing the scale factor

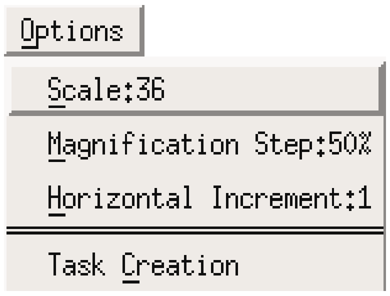
$$scaleFactor = \left\lfloor log_{magStep}\left(\frac{totalDuration}{scaleRatio \times MinDisplayPixels}\right)\right\rfloor + 1 \tag{4.2}$$

The current scale factor is updated given a change in the magnification step. The new magnification step and updated scale factor are then used to compute the scale ratio. The scale factor calculation attempts to preserve the current scale ratio (i.e., the current magnification) while keeping the scale factor an integer value. *MinDisplayPixels* is 100 and represents the number of pixels used to display the entire chart at scale factor 1. Similarly, the maximum scale factor is updated given a change in the magnification step, but computed using Equation 4.2 with *MinScaleRatio* instead of *scaleRatio*. However, the maximum scale factor does not always lead to the maximum magnification (i.e., the lowest scale ratio, *MinScaleRatio*) because the scale factor is maintained as an integer value. Therefore, in many situations, when the maximum scale factor is reached, the corresponding scale ratio is still noticeably above *MinScaleRatio*. I considered having the ability to reach the maximum magnification important, but it required an exception to the scale factor being an integer value. To avoid the fractional scale factor (just above the integral maximum) one extra scale factor is made available. The maximum integral scale factor followed by a '+' sign is used to represent this value.

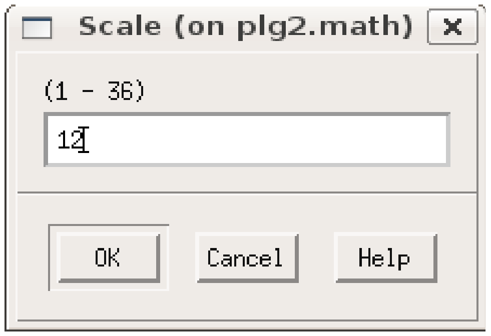
The magnification step has both a display and an internal value. The display value is the percentage form of the internal value and that value is presented to and set by a user. The maximum magnification step, a display value which remains constant, is computed by the following formula

$$maxMagStep = \left(\frac{totalDuration}{MinScaleRatio \times MinDisplayPixels} - 1 \right) \times 100 \tag{4.3}$$

Setting the magnification step to the maximum results in the availability of only two scale factors, i.e., it results in a maximum scale factor of 2. In this case, only one step is required to go from the minimum magnification to the maximum magnification.



(a) Options Menu



(b) Dialog Box

Figure 4.4: Advanced Implementation: User Adjustable Options

As previously mentioned, a user can set both the scale factor and the magnification step, which is accomplished via a pull-down menu associated with the "Options" button on the menu bar (see Figure 4.2). This pull-down menu is presented in Figure 4.4(a). The "Scale" option can be used to set the scale factor between 1 and the current maximum scale factor (or maximum+ if a final fractional part exists). Figure 4.4(b) shows the dialog box for setting a new scale factor. Alternatively, the scale factor can be increased or decreased sequentially by one using keyboard and mouse shortcuts (see Section 4.4). The default scale factor is 5. The "Magnification Step" option can be used to set the magnification step between 1 and the maximum magnification step. The default magnification step is 50%. The current scale factor and magnification step values are displayed beside their respective options in the pull-down menu. In addition, the current scale factor is displayed directly above the task pane on the left side (see Figure 4.2, the scale factor is 12).

4.2.2 Addressing Initial Issues

Whereas in the initial implementation every line and every line's entire execution duration is drawn into the drawing area, in the advanced implementation only the lines and the execution duration visible within the chart pane are drawn. Consequently, for each movement of the horizontal scrollbar in the chart pane, the section of the chart starting at the execution time of the beginning of the chart pane needs to be drawn for the length of the chart pane, i.e., $chartWidth \times scaleRatio$ nanoseconds of time, where $chartWidth$ is the width of the chart pane in pixels. By only rendering the section of the chart visible within the chart pane, the **ESC** now scales to programs of long execution (hours and days) and to higher magnification (0.1 nanoseconds per pixel). As the vertical scrollbar is moved, the chart is also redrawn to include lines for the tasks that are now visible and remove lines for the tasks that are no longer visible, accommodating large numbers of tasks. Programs with large numbers of tasks (e.g., 10,000 tasks or more) are not unrealistic and do exist in practice. While only displaying the visible lines and execution duration in the chart pane is an obvious solution, it required a complete transformation of the initial implementation with significant complexity. For example, the explicit management of some scrollbar functionality was required to ensure the proper section of the chart is displayed at all times.

To address the loss of information at lower magnification, states represented by line segments too small to draw must still be addressed. When the duration of a state is less than the scale ratio, the corresponding line segment is considered to be *invisible*, but is indicated in black on the line. A black state signals that one or more states exist within that duration of time, but at the current magnification, further detail cannot be displayed. The legend gives the black state the name "Elided", meaning information is being omitted. In the line drawing algorithm, the durations of successive *invisible* line segments are summed until a visible line segment is encountered. At that point, a black line segment is drawn to represent the *invisible* line segments, unless the black line segment is less than one pixel in width. In this case, a pixel is stolen from the front of the upcoming visible line segment. Stealing a pixel from the visible line segment may in turn result in that line segment becoming *invisible*, and hence, also being represented within the current black line segment. The advantage over the solution in the initial implementation is that a line is not artificially lengthened, so the elapsed time of execution always correctly matches with the

X-axis tick-marks.

To address the issue of the static X-axis, the elapsed time of execution is no longer limited to millisecond time units. Nanoseconds, microseconds, milliseconds, seconds, and kiloseconds are all available as axis time-units. The most appropriate time unit is determined based on the magnitude of the duration of time (i.e., $chartWidth \times scaleRatio$ nanoseconds) visible within the chart pane. Magnitudes of less than 10^3 use nanoseconds, 10^3 to $10^6 - 1$ use microseconds, 10^6 to $10^9 - 1$ use milliseconds, 10^9 to $10^{12} - 1$ use seconds, and 10^{12} or greater use kilo-seconds. The selected time unit is displayed to the left of the axis immediately followed by the starting time of the currently drawn chart section and a '+' sign (see below the menu bar in Figure 4.2, 'us' means microseconds). Adding the value of a tick-mark label to the starting time gives the elapsed time of execution for that chart position in the selected time unit. The maximum precision of the tick-mark interval (and the tick-mark labels) is 3 decimal places. The precision of the starting time is equal to the precision of the tick-mark interval, with one exception. If the tick-mark interval is an integer value the starting time still displays 1 decimal place.

To address the visibility of the legend and the task names, both are no longer drawn into the same drawing area as the chart. Therefore, the legend and the task names remain visible to a user at all times. The legend appears in the menu bar after the "Options" button. The task names are now drawn into a separate task pane (see left column in Figure 4.2). In addition to the task pane maintaining a separate horizontal scrollbar, the line separating the task pane from the chart pane can be pulled to the left or right, adjusting the width of the task and chart panes. Both features accommodate longer task names. The vertical scrollbar scrolls both the task pane and chart pane at the same time.

To address the visualization performance issue, the data structure storing the states was altered; states are now stored in blocks of equal size. Each block is 32 kilobytes in size and stores 2046 states. Therefore, a line consisting of 1,000,000 states is separated into 489 blocks. The maximum state duration is calculated and stored for each block. This information is then used to improve the performance of the line-drawing algorithm. Before each state within a block is processed and drawn, the maximum state duration for the block is compared against the scale ratio. If the maximum state duration is less than the scale ratio, then the entire block is *invisible*;

therefore, the individual states within the block do not need to be processed and a black line segment (representing the duration of the block) can be drawn. The improvement in performance is most apparent for programs with large numbers of state transitions at the minimum magnification, where the entire chart is drawn and consists entirely of *invisible* line segments. Using the example above, 489 blocks are processed rather than 1,000,000 states.

4.3 Implementation Issues

This section describes implementation issues I encountered and solved during the writing of the advanced implementation of the **ESC**.

4.3.1 Scrollbar Scaling

This issue involves scaling with respect to the horizontal scrollbar in the chart pane. A scrollbar manages its position, representing the distance from the beginning of execution, as well as a width for the sliding bar and a maximum position. The meaning of the X-window scrollbar values is up to the application, within the limits of a signed 32-bit integer value. To simplify the implementation, the scrollbar values are chosen to be the pixel equivalent of the time values. For example, the maximum position is given by the following formula

$$maximum = \left\lceil \frac{totalDuration}{scaleRatio} \right\rceil + 1 \tag{4.4}$$

This formula computes the number of pixels needed to represent the entire execution duration ($totalDuration$) at the current scale ratio. Figure 4.5 illustrates the simple conversion between time and pixel units:

Multiplying a position in pixel units by the scale ratio results in the corresponding time units; dividing a position in time units by the scale ratio results in the corresponding pixel units. However, given a scale ratio of 5 nanoseconds per pixel and an execution duration of 1 minute, with nanosecond precision, the maximum scrollbar position in pixels requires more than 32 bits (in

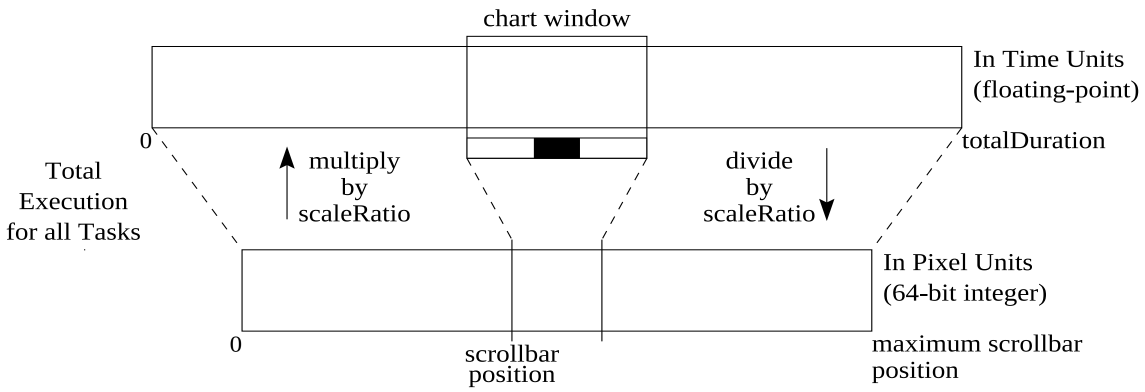


Figure 4.5: Advanced Implementation: Converting Between Time and Pixel Units

this example, 34 bits are required); hence, a 64-bit integer is needed to store this value in pixel units. Unfortunately, an X-window scrollbar manages values as signed 32-bit integers, limiting the scalability to programs of longer execution as well as the ability to increase magnification. Since the internal representations of the scrollbar cannot be changed, I adopted another solution: the scrollbar values are scaled. Given a change in the scale ratio, the maximum position is computed (see Equation 4.4) and used to determine $numShifts$. $numShifts$ is the number of bit shifts required to convert the maximum position (i.e., the largest value given to the scrollbar) into a signed 32-bit integer. Any value given to the scrollbar is shifted to the right by $numShifts$, and any value retrieved from the scrollbar is shifted to the left by $numShifts$. Consequently, a $numShifts$ of 3, for example, means that a one unit move of the scrollbar moves the chart by 2^3 pixels (or $2^3 \times scaleRatio$ nanoseconds) rather than 1 pixel. As $numShifts$ becomes large, a one unit move of the scrollbar can result in a large movement of the chart. If the movement is greater than the width of the chart pane then some areas of the chart become inaccessible. For example, if the visible area of the chart pane is 500 pixels in width and a one unit move of the scrollbar moves the chart by 512 pixels, then for every 512 pixels the last 12 are inaccessible. However, such a situation can only occur for programs of long execution at high magnification. In the previous example, at a magnification of one nanosecond per pixel, the program must run for 1099.5 seconds.

I also considered an alternative solution of keeping two separate numbers for each scrollbar value. One number is maintained by the scrollbar and consists of the 31 most-significant bits of the whole value. The other number is maintained by the program and consists of the remaining least-significant bits of the whole value (i.e., $numShifts$ bits). This solution does overcome the issue of inaccessible chart areas because the extra bits are not discarded; however, it generates significant complexity. Functionality currently handled independently by the scrollbar needs to be overridden because the scrollbar maintains only 31 bits of the whole value. Before a decision on an appropriate scrollbar action can be made, the 31 most-significant bits need to be combined with the remaining least-significant bits. The complexity introduced by the alternative solution seemed greater than the benefit gained for the extreme cases where it is needed, and therefore, the previous solution was selected.

4.3.2 X-Axis Labelling

This issue involves the selection of appropriate tick-mark intervals and labels for the X-axis. The length of the longest tick-mark label must be considered because each label needs a certain number of pixels for display; therefore, the tick-marks must be sufficiently spaced apart to allow for the display of the longest label. Ideally, the tick-marks should be frequent. Yet, as the elapsed time of execution increases, the labels become very lengthy, resulting in fewer tick-marks. To maintain frequent tick-marks the starting time of the currently drawn chart section is subtracted from the label values and, as previously mentioned, displayed to the left of the X-axis. The tick-mark labels and lengths are now limited to the incremental increase of the tick-mark interval. Therefore, the number of digits in the longest possible tick-mark label is equal to the number of digits in the integer part of the duration of time being represented within the chart pane (converted into the selected time unit), plus one for the decimal place and plus the maximum allowable number of decimal places (i.e., 3). This maximum length in digits is converted into pixels ($\mathsf{maxPixels}$) and used to compute the initial tick-mark interval, $\mathsf{tickInt}$, as follows:

```
// compute numTicks given length of longest label
unsigned int numTicks = (unsigned int)(chartWidth / maxPixels);

// compute tickInt given number of tick marks
double tickInt = dur / numTicks;

// round tickInt up to one significant digit
double exponent = floor( log10( tickInt ) );
double mantissa = tickInt / pow( 10, exp );
tickInt = ceil( man ) * pow( 10, exp );

// recompute numTicks
numTicks = (unsigned int)(dur / tickInt);
```

The number of tick-marks along the X-axis is numTicks. The tickInt is rounded up to one significant digit in order to provide a reasonable interval. However, as a result of the rounding the initial tick-mark interval may need to be iteratively increased. If numTicks is less than the number of tick-marks possible given the longest label, the tick-marks are increased (i.e., tickInt $\times 0.5$) as long as the longest label can still be displayed. In the majority of cases, at most one increase is made. Overall, the procedure produces a well divided X-axis.

4.4 Other Considerations

I also considered additional factors relating to the user experience in the advanced implementation of the **ESC**.

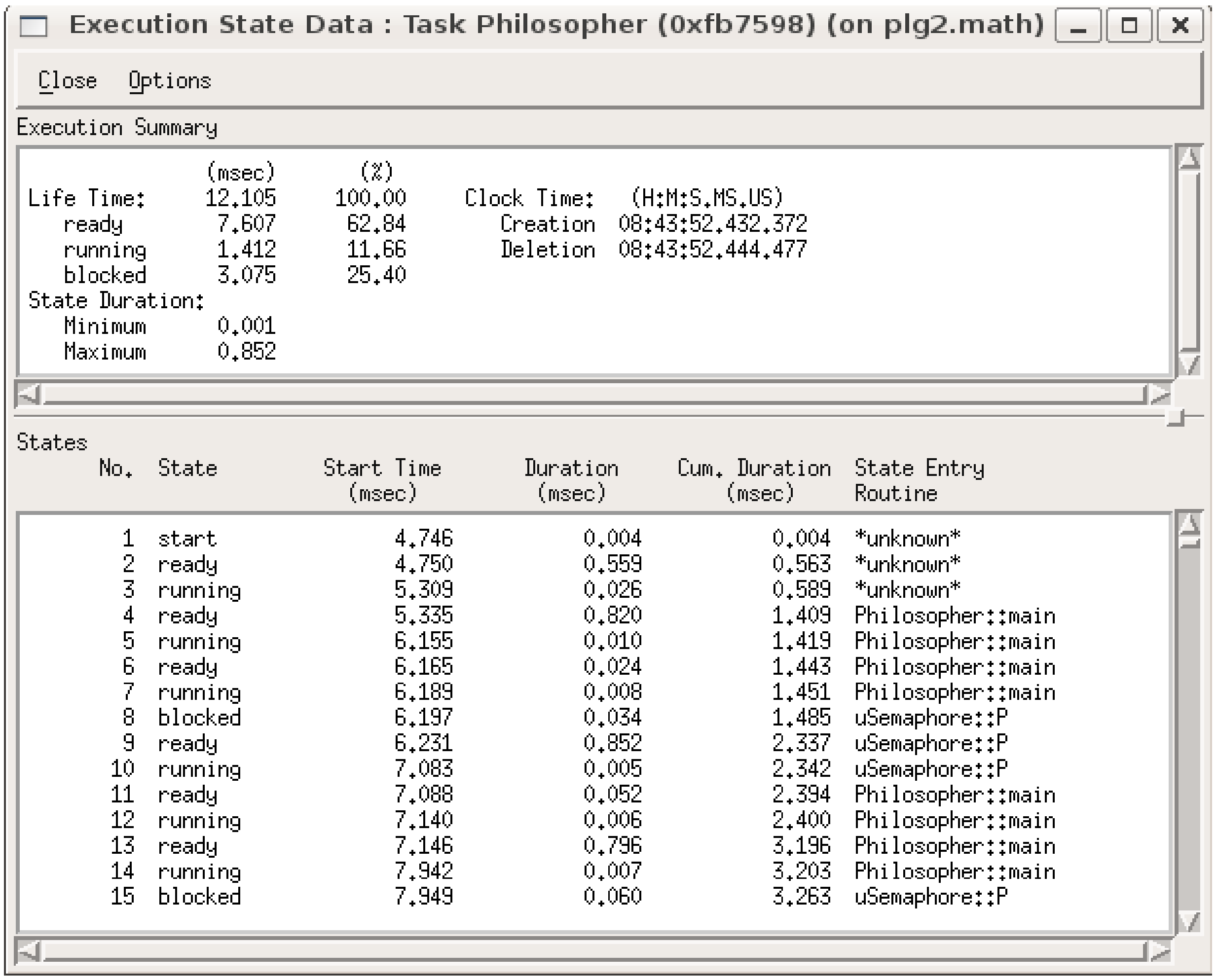
One way to move the horizontal scrollbar in the chart pane is to click the arrow button located at each end of the scrollbar. The default action is that one click of an arrow button moves the scrollbar by one unit. The corresponding movement of the chart, in pixels, depends on the current value of $numShifts$ (see Section 4.3.1). However, users can use the “Horizontal Increment” option in the pull-down menu associated with the “Options” button to define the number of units that the scrollbar moves given one click of an arrow button (see Figure 4.4), but again the corresponding movement of the chart depends on $numShifts$. This option allows users to precisely adjust the movement of the scrollbar according to the requirements of their current situation.

The mechanisms to control the interface to the data are repeated in multiple forms to satisfy different user preferences. The keyboard keys, the mouse buttons and the scroll wheel can be used as shortcuts to perform various actions. The ‘i’ key and the middle mouse button can be used to increase magnification; both increase the scale factor by one. The ‘o’ key and the right mouse button can be used to decrease magnification; both decrease the scale factor by one. The left/right arrow keys can be used to move the horizontal scrollbar in the chart pane by the user-defined number of units, as can the scroll wheel, or by the width of the chart pane if holding the ‘control’ key. The up/down arrow keys can be used to move the vertical scrollbar by one line, and the page up/down keys can be used to move the vertical scrollbar by the number of lines currently visible. Similar control functionality exists for the task pane. For example, the left/right arrow keys can be used to move the task names by one pixel. Scrolling through the task names in the task pane, using the up/down arrow keys, also appropriately scrolls the lines within the chart and moves the vertical scroll bar on the right.

A single gridline, positioned at the middle-most X-axis tick-mark, is displayed behind the lines in the chart pane (see gridline descending from 400 in Figure 4.2). A user can line-up any state precisely at the gridline to aid in the reading of the chart. The single gridline, as opposed to a full grid, does not clutter the chart pane.

4.5 Task Details

By clicking on a task name in the task pane of the **ESC**, detailed information about the selected task’s execution is displayed in a new window (see Figure 4.6). The upper pane of the window displays the execution summary information for the task. The summary information includes the total lifetime of the task, the total duration of time spent in various states, the minimum and maximum state durations, and the creation and deletion clock times. The lower pane of the window lists the task’s states, including details for each state. The details include the start time of the state, the duration of the state, the cumulative duration of the task’s execution at the time


Execution State Data : Task Philosopher (0xfb7598) (on plg2.math)
Close
Options

```
Execution Summary

                 (msec)       (%)
Life Time:       12.105     100.00     Clock Time:   (H:M:S.MS.US)
   ready          7.607      62.84        Creation  08:43:52.432.372
   running        1.412      11.66        Deletion  08:43:52.444.477
   blocked        3.075      25.40
State Duration:
   Minimum        0.001
   Maximum        0.852

States
       No.  State          Start Time      Duration      Cum. Duration  State Entry
                             (msec)         (msec)          (msec)      Routine

         1  start              4.746          0.004           0.004  *unknown*
         2  ready              4.750          0.559           0.563  *unknown*
         3  running            5.309          0.026           0.589  *unknown*
         4  ready              5.335          0.820           1.409  Philosopher::main
         5  running            6.155          0.010           1.419  Philosopher::main
         6  ready              6.165          0.024           1.443  Philosopher::main
         7  running            6.189          0.008           1.451  Philosopher::main
         8  blocked            6.197          0.034           1.485  uSemaphore::P
         9  ready              6.231          0.852           2.337  uSemaphore::P
        10  running            7.083          0.005           2.342  uSemaphore::P
        11  ready              7.088          0.052           2.394  Philosopher::main
        12  running            7.140          0.006           2.400  Philosopher::main
        13  ready              7.146          0.796           3.196  Philosopher::main
        14  running            7.942          0.007           3.203  Philosopher::main
        15  blocked            7.949          0.060           3.263  uSemaphore::P
```

Figure 4.6: Advanced Implementation: Execution State Task Details Display

the state is entered, and the name of the routine in which the task entered the state. This window was available in the initial implementation, but now interacts with the **ESC** (and vice versa). Clicking on a state or scrolling in the state list results in the **ESC** being scrolled horizontally to the corresponding state location. Similarly, clicking on a line segment in the **ESC** scrolls to and highlights the corresponding state in the state list. Such functionality allows a user to quickly obtain detailed information about any area of the **ESC**. Additionally, a user can use the “File

Info” option in the “Options” pull-down menu to display, for each state, the file name (and path) containing the routine in which the state was entered and the line number corresponding to the start of the routine within the file.

4.6 Performance

This section describes the performance of the **ESC** with respect to both time and space. I was unable to compare the **ESC**, in these respects, to the related profiling tools (described in Section 4.7) because I did not have access to the tools or the environment required to run them.

4.6.1 Time

To evaluate the time cost of a state transition, I constructed a worst-case test program (see Figure 4.7), profiled this program with the **ES** metric, and compared its running time to the same test program run without profiling. Only the running time of the test program itself was measured, i.e., the time includes monitoring and data collection but not time spent during analysis or visualization.

The test program simply calls the task yield routine, causing a state transition from running to ready; then the task is immediately scheduled (because no other tasks are in the system), causing a state transition from ready to running. In other words, one call to the yield routine results in two state transitions. The test program is a worst-case scenario because it does no work, other than change state by calling the yield routine. In most applications, the cost of a task’s computation would dominate the cost of its state transitions.

The test program was compiled with optimization (i.e., O2 flag) and run multiple times with an increasing number of state transitions (2000 to 30000). Table 4.1 shows the results of the performance testing on a per-transition basis (in microseconds).

The per-transition time is calculated by dividing the total running time by the number of state transitions. As the number of state transitions increases, the running time of the program increases for both the profiling and no profiling cases; as seen in the table, the time per-transition

```
#include <uC++.h>

_Task Worker {
    int loop;
    void main() {
        for ( int i = 0; i < loop; i += 1 ) {
            yield();
        } // for
    } // Worker::main
  public:
    Worker( int loop ) : loop( loop ) {} // Worker::Worker
}; // Worker

void uMain::main() {
    int loop = 1;
    if ( argc == 2 ) loop = atoi( argv[1] );
    Worker w( loop );
} // uMain::main
```

Figure 4.7: Execution State Chart: Time Performance Test Program

remains relatively constant. However, the average time per-transition for the profiling case is 70% higher than that of the no profiling case, signifying that the **ES** metric increases running time by 70% in this worst-case program. Such an increase is reasonable given the overhead of creating the storage data structures, and at each state transition, collecting and storing the necessary data.

4.6.2 Space

To determine the space cost of a state, the space cost of a state object and a block (see Section 4.2.2) needs to be considered. A state object stores data for an individual state. Each state object consists of a long integer to store the start time of the state, an integer to store the type of state and a pointer to the routine in which the state was entered. Therefore, in a standard 32 bit system with 4 byte pointers, 4 byte integers and 8 byte long integers, each state object requires 16 bytes of space.

No. State Transitions	No Profiler Time per-Transition (μs)	μProfiler Time per-Transition (μs)	% Increase
2000	0.81	1.42	76.09
4000	0.79	1.35	71.79
6000	0.78	1.33	71.01
8000	0.77	1.31	70.39
10000	0.75	1.30	72.05
12000	0.76	1.30	70.74
14000	0.77	1.29	68.40
16000	0.76	1.29	69.31
18000	0.75	1.29	72.58
20000	0.76	1.29	70.03
22000	0.75	1.29	71.80
24000	0.76	1.28	69.40
26000	0.75	1.28	70.46
28000	0.76	1.28	68.03
30000	0.76	1.29	70.02

Table 4.1: Execution State Chart: Time Performance Results

Each block object consists of a pointer to link the blocks, header information, and an array of N state objects. The header contains an integer to store the number of elements in the array, 4 bytes of padding to maintain proper data structure alignment, and a long integer to store the maximum state duration in the block. Therefore, each block object requires 24 bytes of space in addition to the space required for the state objects stored in the block's array.

A block index is also created to provide random access and for performing searches. The index (an array) has one entry for each block object consisting of a pointer to the block object and a long integer to store the maximum start time in the block. Therefore, an index entry requires 12 bytes of space and, in order to compute the per-block cost, needs to be added to the cost of a block object.

As previously mentioned each block object stores 2046 states, so the total space cost per-block is

$$Total\ space\ per\text{-}block = 12\ bytes + 24\ bytes + (2046 \times 16\ bytes) = 32,772\ bytes \tag{4.5}$$

All state objects equally share the cost of the block in which they reside. To compute the space cost per-state the above result is divided by 2046. Therefore, in a standard 32 bit system, the space cost per-state is 16.018 bytes. In a standard 64 bit system with 8 byte pointers, no padding is necessary, leaving storage for 2045 states, so the space cost per-state is 20.018 bytes.

4.7 Related Work

This section describes three current profiling tools that include execution-state charts. HP Visual Threads [HP04] provides profiling metrics for programs using a POSIX threads library, including C, C++, and Java; however, its use with Java applications is limited. The NetBeans Profiler [Net] and Borland Optimizeit Thread Debugger [Bor03] both provide profiling metrics for multithreaded Java programs. Other profiling tools that include execution-state charts exist in the literature [HP06, ejt07, App, gra96].

4.7.1 HP Visual Threads

The HP Visual Threads tool analyzes programs to detect problems associated with multithreading, including performance, data protection, and deadlock [HP04]. The profiling data undergoes real-time analysis and visualization; therefore, the information presented to a user is continuously updated. The data can also be saved to a tracefile for later visualization. Two visualizations provide the state data: the Main Window and the State Transitions Window.

Main Window

The Main Window provides a high-level overview of the global execution-state (see Figure 4.8). The displayed graph is generated by statistical profiling (i.e., sampling), so data is collected only at specific time intervals (i.e., sampling interval). The graph shows the number of active threads

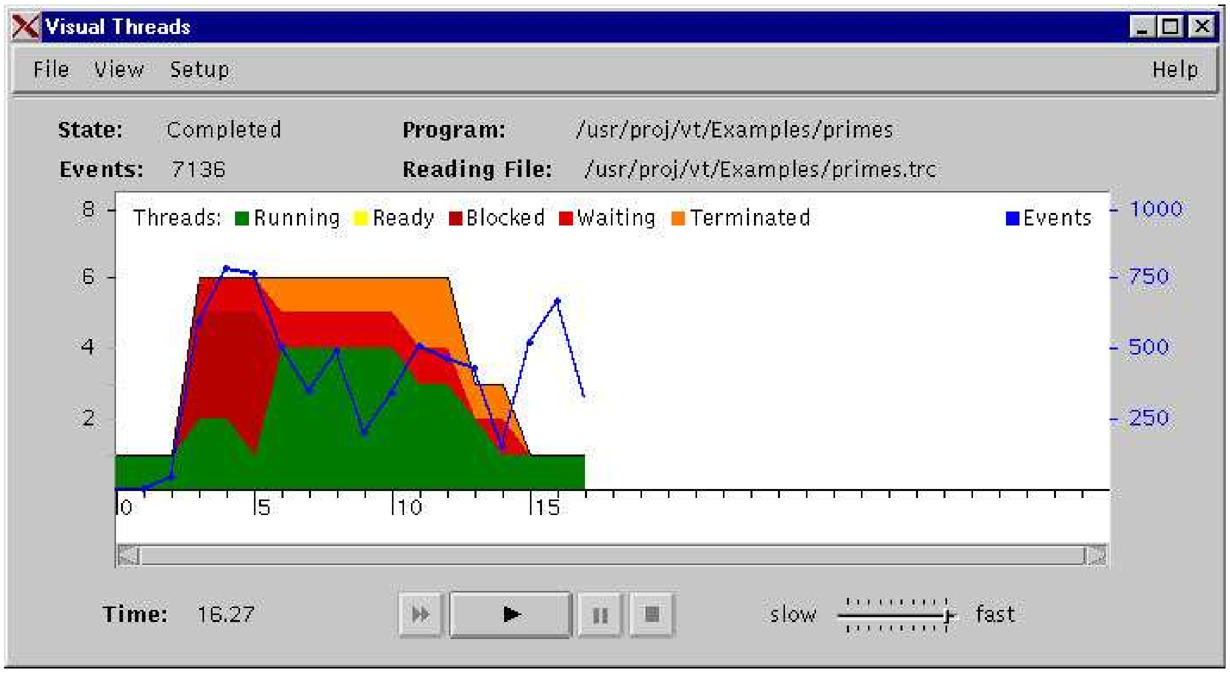

Visual Threads
File View Setup Help
State: Completed
Events: 7136
Program: /usr/proj/vt/Examples/primes
Reading File: /usr/proj/vt/Examples/primes.trc
Threads: Running Ready Blocked Waiting Terminated
Events
Time: 16.27
slow
fast

Figure 4.8: HP Visual Threads Main Window

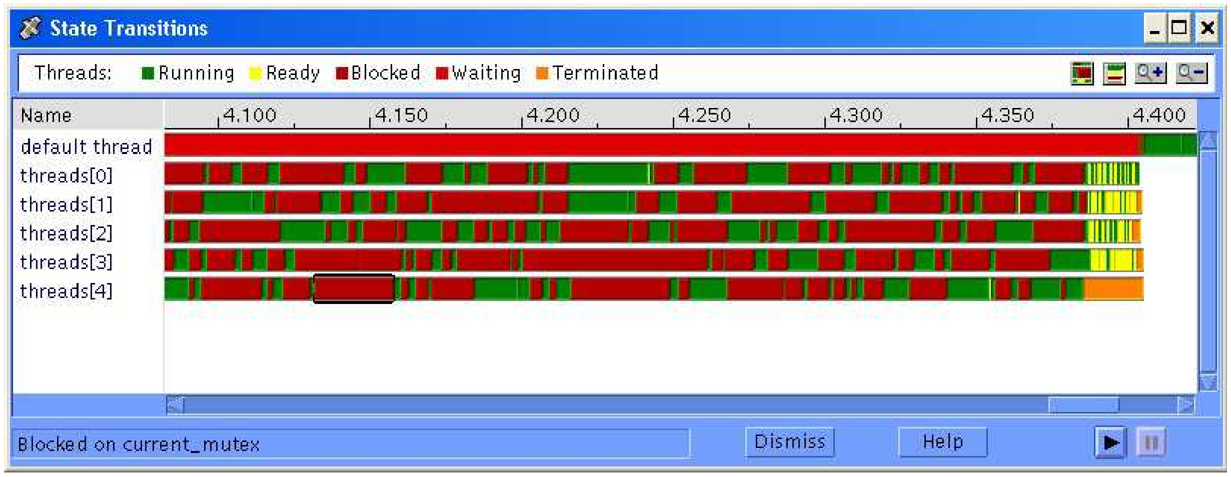


Figure 4.9: HP Visual Threads State Transitions Window

within the program over time. At any point in time, each band of colour shows the portion of the active threads in the associated state: running, ready, blocked, waiting, or terminated. The states are defined similarly to those in the **ESC**, with the addition of the waiting state indicating a thread is blocked on an event such as a system call, page fault, join, or on a condition variable, rather than on a mutex object or lock as is the case in the blocked state. The sampling interval is indicated by the X-axis tick-mark interval and can be changed by a user. All subsequently graphed data is displayed using the new interval, so the currently graphed data is removed. The connected line segments on the graph show the number of events processed per interval, where events include acquiring a mutex object, a deadlock, entering or exiting a routine, etc. Severity icons can be displayed at various times along the X-axis to indicate that a violation (e.g., of a deadlock condition or performance threshold) has occurred. Several control buttons (i.e., fast forward, play, pause, stop) are also available to a user for controlling the profiling of a program. For example, the play button allows a user to start, restart or resume profiling. The speed slider allows a user to control how often the graph is updated with new data.

State Transitions Window

The State Transitions Window displays an execution-state chart (see Figure 4.9). The chart is generated by exact profiling, meaning all states are recorded in the chart. The State Transitions Window, like the **ESC**, shows the elapsed time of execution across the top of the chart and the list of threads on the left. The colour coded states represented in the chart are the same as those described for the Main Window. Buttons are available to sort the threads by name or their current state, to overlay colours on blocked states indicating which mutex objects threads are blocked on, and to zoom-in and zoom-out. Waiting and blocked states are also sorted by the blocking reason (e.g., the mutex object the thread is blocked on). The display order of the threads changes according to the sorting criteria as the chart is updated in real-time (e.g., threads currently in a particular state are grouped together). The highest magnification is 2 milliseconds per pixel and the lowest magnification is 12 seconds per pixel. Moving the cursor over a line segment displays the text description of the corresponding state (and blocking reason if applicable) in the status box below the chart. Horizontal and vertical scrollbars exist to move through the chart. As new data is displayed in real-time, the chart is automatically scrolled to ensure the new data is visible. As in the Main Window, several control buttons are available for controlling the profiling. Clicking a line segment displays the Event Details Window which provides information about the event that caused the thread to enter the selected state. The information includes the event type, the time the event occurred, the threads involved in the event, and a call-stack. Clicking on a thread name displays the Object Details Window, which provides general information about the thread. The information includes statistics about the thread, related objects, and details such as the thread ID.

4.7.2 NetBeans Profiler

The NetBeans Profiler tool is a profiler for the NetBeans Integrated Development Environment and provides CPU, memory and threads profiling as well as basic Java Virtual Machine monitoring to analyze and solve memory and performance problems [Net]. The profiling data collected undergoes real-time analysis and visualization; therefore, the information presented to a user is continuously updated. The data can also be saved as snapshots for later visualization. Two visualizations provide the state data: the Threads Timeline Tab and the Threads Details Tab.

Threads Timeline Tab

The Threads Timeline Tab displays an execution-state chart (see Figure 4.10). The chart is generated by statistical profiling, so thread states are collected only at specific time intervals. The Threads Timeline Tab, like the **ESC**, shows the elapsed time of execution across the top of the chart and the list of threads on the left. The colour coded states represented in the chart are: running, sleeping, wait, and monitor. Here, the running state means the thread is running or ready to run, the sleeping state means the thread has called the sleep function, the wait state means the thread is blocking on a condition variable in a monitor (i.e., executing a wait function), and the monitor state means the thread is waiting to enter a monitor held by another thread. Buttons are available to zoom-in and zoom-out as well as to scale the chart to fit the window. The threads can be filtered to display all threads, active threads, or finished threads. Horizontal and verti-

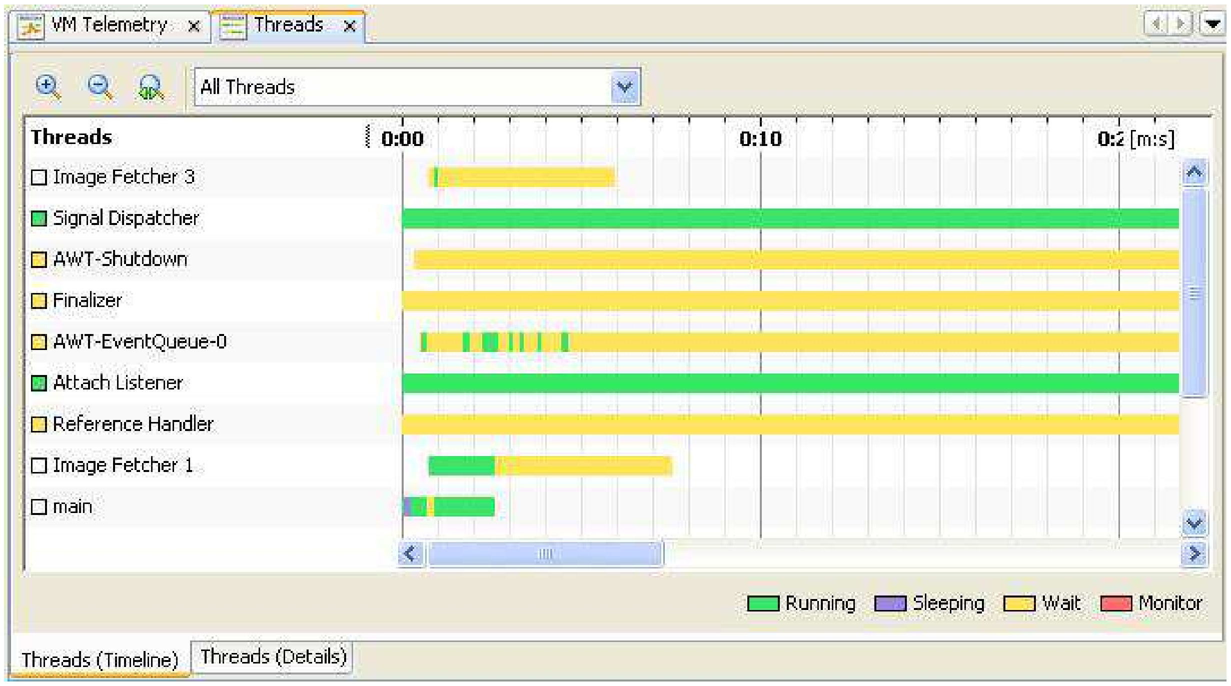
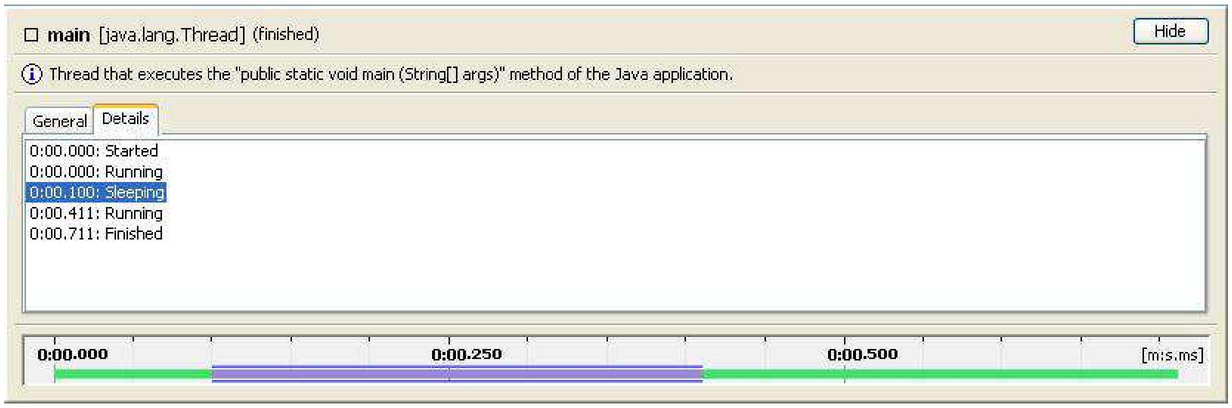


Figure 4.10: NetBeans Profiler Threads Timeline


main [java.lang.Thread] (finished)

Hide

Thread that executes the "public static void main (String[] args)" method of the Java application.

General Details

```
0:00.000: Started
0:00.000: Running
0:00.100: Sleeping
0:00.411: Running
0:00.711: Finished
```

Figure 4.11: NetBeans Profiler Threads Details

cal scrollbars exist to move through the chart, and the current state of a thread is indicated next to the thread name using colour. Gridlines are also displayed to ease the reading of the chart. Control buttons are available to stop the profiling and to rerun the previous profiling command (e.g., same program, same options). Double clicking a thread displays the thread's details in the Threads Details Tab. Additionally, a VM Telemetry Tab displays a graph showing the number of active threads within the program over time (separated into user and system threads).

Threads Details Tab

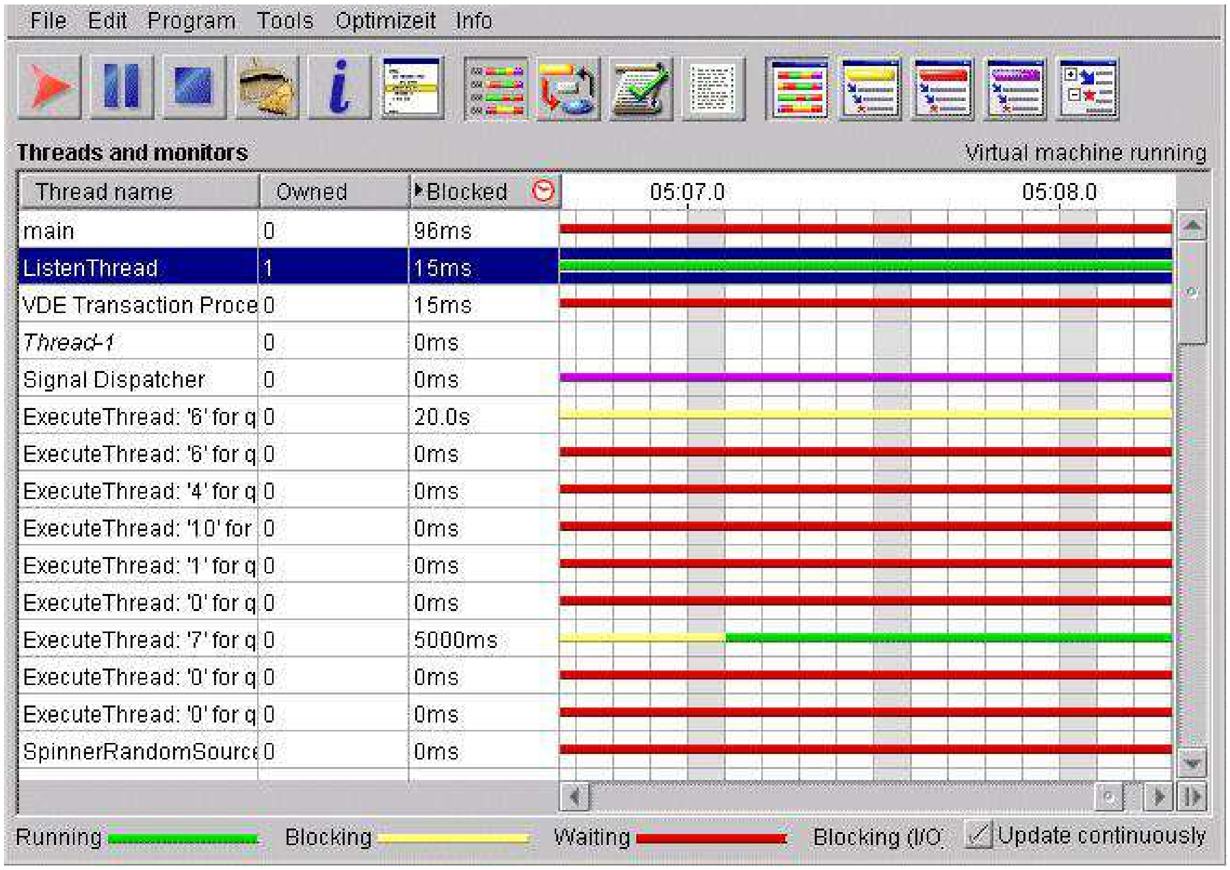
The Threads Details Tab provides detailed information about the threads (see Figure 4.11). A user can display the details for all threads, active threads, finished threads, or for particular threads selected in the Threads Timeline Tab. The state line for the thread and a list of the thread's states (corresponding to the line), including the start times of the states, is displayed for each selected thread. Clicking a line segment in the state line highlights the corresponding state in the list. A pie chart illustrating the percentage of time the thread spent in each state is displayed on the General Tab (chart not visible in Figure 4.11) to provide a quick overview of the thread's activity. Finally, a short text description of the thread is provided.

4.7.3 Borland Optimizeit Thread Debugger

The Borland Optimizeit Thread Debugger tool reveals how a Java program uses computer resources, identifying thread contentions, thread starvation, unnecessary locking, and deadlocks to understand and improve the performance and reliability of a Java program [Bor03]. The profiling data collected undergoes real-time analysis and visualization; therefore, the information presented to a user is continuously updated. The main visualization that provides the state data is the Thread View.

Thread View

The Thread View displays an execution-state chart (see Figure 4.12). The chart is generated by exact profiling, meaning all states are recorded in the chart. The colour coded states repre-



Thread name	Owned	Blocked
main	0	96ms
ListenThread	1	15ms
VDE Transaction Proce	0	15ms
Thread-1	0	0ms
Signal Dispatcher	0	0ms
ExecuteThread: '6' for q	0	20.0s
ExecuteThread: '6' for q	0	0ms
ExecuteThread: '4' for q	0	0ms
ExecuteThread: '10' for	0	0ms
ExecuteThread: '1' for q	0	0ms
ExecuteThread: '0' for q	0	0ms
ExecuteThread: '7' for q	0	5000ms
ExecuteThread: '0' for q	0	0ms
ExecuteThread: '0' for q	0	0ms
SpinnerRandomSource	0	0ms

Figure 4.12: Borland Optimizeit Thread Debugger Thread View

sented in the chart are: running, blocking, waiting, and blocking (I/O). Here, blocking means the thread is waiting to enter a monitor held by another thread, waiting means the thread is blocking on a condition variable in a monitor (i.e., executing a wait function), and blocking (I/O) means the thread is not making progress as a result of waiting on an I/O operation. On the left, the Thread View displays several columns of information for each thread, in addition to the thread name. The default information includes the number of monitors the thread currently holds and the length of time the thread has blocked for a monitor. A user can display further information such as the number of times the thread has blocked for a monitor, the number of times the thread has waited in a monitor, etc. The information is aggregated for all monitors. A user can sort the threads in the chart on any of the available columns. The Thread View shows the elapsed time of execution across the top of the chart. Gridlines are also displayed to ease the reading of the chart. Horizontal and vertical scrollbars exist to move through the chart. As new data is displayed in real-time, the chart is automatically scrolled to ensure the new data is visible. Several control buttons (i.e., play, pause, stop) are available to a user for controlling the profiling of a program. For example, the play button allows a user to start or resume profiling. A thread can be selected by clicking anywhere on the thread's state line and a time range can be selected by highlighting an area right on the chart. The selected thread or time range determines the range of information displayed in the other views. A Source Code Viewer is available to display code related to a detected event (e.g., routine where a thread is blocking).

Other Views and Displays

The Contention View provides information to understand why contention among threads occurs for a monitor. The view displays a backtrace of routine calls leading to the routine where a thread is blocking. Upon selecting a contended monitor, details are provided explaining all the threads involved in the contention. The Waiting View provides information to understand why a thread is not making progress (e.g., waiting in a monitor, blocked on an I/O operation). The view displays a backtrace of routine calls leading to the location of the thread's stalled progress and provides the wait time. The Monitor Enter View describes where a thread enters and holds monitors to understand and correct unnecessary locking. The view provides a backtrace of routine calls indicating locations where the thread enters various monitors. The number of times a routine enters any monitor and the corresponding percentage of the total entrances are provided for each routine.

The Monitor Display provides deadlock detection by providing a real-time graph showing the relationship of threads and monitors within a deadlock cycle. Selecting a relationship (e.g., thread blocking on a monitor) in the graph displays the backtrace of routine calls resulting in the relationship. The Monitor Usage Analyzer Display provides warnings about possible unsafe situations that can lead to deadlock and identifies the threads involved in the warnings. For example, the lock and wait warning occurs when a thread enters one monitor and then waits for another monitor before releasing the first, possibly causing deadlock.

4.7.4 Comparison

Table 4.2 summarizes and compares the relevant features of μProfiler's **ES** metric and the three profiling tools discussed in the previous sections. Some of the important features are discussed in detail.

	μProfiler ES Metric	HP Visual Threads	NetBeans Profiler	Borland Debugger
Real-time Analysis and Visualization		√	√	√
Aggregate Views		√	√	
Exact Profiling	√	√		√
Saving Tracefile or Snapshot			√	√
Blocking Reasons or Backtrace	√ (minimal)	√		√
State List	√		√	
Thread Summary Information	√	√	√	√
Zooming-in and out	√	√	√	
Fine-grained Zooming Control	√			
Sorting or Filtering		√	√	√
Gridlines	√		√	√
Eliding	√			

Table 4.2: Execution State Chart: Comparison of Related Profilers

Using statistical versus exact profiling to generate an execution-state chart is an important difference. The choice of profiling approach comes down to a trade-off between accuracy and overhead. In statistical profiling, thread states are only collected at specific time intervals, so state transitions occurring within the time interval are lost, thereby reducing the accuracy of the chart. Exact profiling provides complete accuracy because each state transition is recorded as it occurs. In statistical profiling, only collecting data at specific time intervals gives a lower data collection overhead, both in time and space, than in exact profiling and consequently a lower probe effect. The NetBeans Profiler uses statistical profiling for its execution-state chart. HP Visual Threads also uses statistical profiling, not for its execution-state chart, but for its global execution-state graph. In such a situation, the loss of accuracy may be less of an issue as the graph is only trying to illustrate an overall view of the execution state. μProfiler, HP and Borland use exact profiling for their execution-state charts, choosing complete accuracy over a reduction in overhead.

Unlike the other profiling tools, μProfiler's **ES** metric does not provide real-time analysis and visualization. Real-time analysis and visualization is useful when a user pauses profiling (e.g., by using a control button or due to the interactive nature of the program) and proceeds to examine the data displayed up to that point. Otherwise, because the display is updated in real-time, a user would be overwhelmed by the constantly changing data since many programs generate hundreds to thousands of state transitions per second.

Although two out of the three profiling tools provide magnification (i.e., the ability to zoom-in and out), the **ESC** provides higher magnification and, unlike all the other profiling tools, fine-grained control. HP Visual Threads, for example, has a maximum magnification of only 2 milliseconds per pixel, whereas the **ESC** has the maximum magnification of 0.1 nanoseconds per pixel. High magnification is essential for analyzing the execution of threads with states of short duration (e.g., micro or nanosecond duration). States of short duration are common in many concurrent programs. The **ESC** additionally provides fine-grained control through the "Magnification Step" option, allowing a user to control the percentage change in magnification for each step in the scale factor.

The other profiling tools do not provide an elided state or similar functionality; therefore, at lower magnification, states represented by line segments too small to draw are not drawn. The loss of states at lower magnification can result in a chart that is very misleading and confusing for a user because the chart is not accurate and may be logically inconsistent. A chart is logically inconsistent when it displays adjacent states, resulting from a loss of states in between them, such that the second state is not logically reachable from the first state.

Overall, μProfiler's **ES** metric provides many important features, and furthermore, includes new features unavailable in the other profiling tools. Some features not currently provided in μProfiler, such as aggregate views, saving tracefiles and sorting, are possible enhancements for future work.

4.8 Summary

The advancements made to μProfiler's **ES** metric have achieved the goals stated at the beginning of the chapter. Firstly, based on functionality and the comparison to related work, μProfiler's **ES** metric is similar to state-of-the-art vendor execution-state metrics. Secondly, the evaluation of time and space costs reveals good performance in both areas. Finally, the **ESC** scales to programs of long duration and with large numbers of tasks and states.

Chapter 5

Exact Call-Graph

This chapter describes the advancements made in μProfiler's Exact Routine Call-Graph (**ECG**) metric.

The **ECG** generates an exact profile of a μC++ program's dynamic execution (called a call-graph). The profile provides the dynamic calling relationship among routines in the program and gives a user some indication about the program's control flow. A dynamic call-graph includes only those routines called during a particular execution of the program, in contrast to a static call-graph that includes all routines in a program (called or not). To express the calling relationships among routines, the parents and children of each routine are indicated. The set of routines that call a specific routine one or more times are that routine's parents or callers. The set of routines that a specific routine calls one or more times are that routine's children or callees. Figure 5.1 is a graphical representation of a call-graph where A, B, C, D, E and F represent routines. A directed edge represents a call being made from one routine (the caller) to another routine (the callee). In Figure 5.1, routine A calls routine D; therefore, routine D has routine A as a caller and routine A has routine D as a callee (as well as routine B). The routines, starting at the root (i.e., routine A), along the call-path leading to a specific routine are that routine's ancestors (e.g., routines A and D are ancestors of routine E). The routines having a specific routine as an ancestor are that routine's descendants (e.g., routines B, C, D, E and F are descendants of routine A). However, if a routine is an ancestor of itself, then there exists a call cycle in the call-graph. In Figure 5.1, routines A, B and C are ancestors of routine B; therefore, routine B is an ancestor of itself and the call-graph includes a cycle, namely B $\rightarrow$ C $\rightarrow$ B.

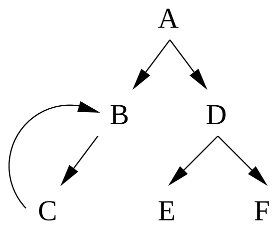


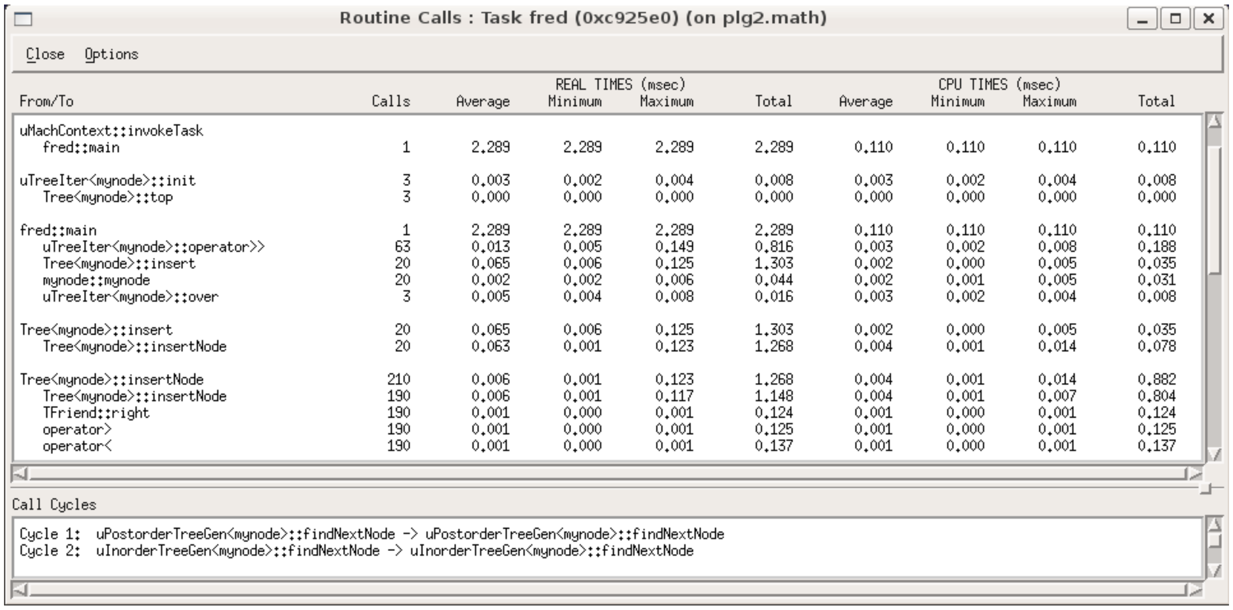
Figure 5.1: Call-Graph

For each routine, various data can be collected. The data can include the number of calls to a routine, the inclusive time of a routine, the self or exclusive time of a routine, the block time of a routine, and the descendant time of a routine. Inclusive time is the time spent for the entire execution of the routine (i.e., total time between routine enter and exit). The inclusive time is the sum of the routines exclusive, block and descendant times. Exclusive time is the time spent executing the routine itself and does not include the time spent blocking. Block time is the time spent while the task executing the routine is blocked in the routine itself. Descendant time is the time spent executing the descendants of the routine. The descendant time is the sum of the inclusive times of the routine's callees. The total number of calls, inclusive time, exclusive time, descendant time and block time for a routine can be broken down by caller. Similarly, the total descendant time for a routine can be broken down by callee. It is also possible to measure hardware events (e.g., number of completed instructions) corresponding to these times.

The exact nature of the **ECG** implies profiling data is collected at each occurrence of a relevant event; therefore, accurate information is provided at the cost of higher overhead (in both time and space). For the **ECG** the relevant events include:

- **routine enter:** the routine is called by a parent routine and starts executing.
- **routine exit:** the routine completes and execution returns to the parent routine.

- **task block:** the task stops executing while waiting for an event to occur (including a voluntary yield or involuntary preemption).
- **task unblock:** the task starts executing when an event occurs.
- **coroutine discontinue:** the coroutine stops executing.
- **coroutine continue:** the coroutine starts executing.



```
                                               REAL TIMES (msec)                                CPU TIMES (msec)
From/To                          Calls   Average   Minimum   Maximum     Total   Average   Minimum   Maximum     Total

uMachContext::invokeTask
   fred::main                        1     2.289     2.289     2.289     2.289     0.110     0.110     0.110     0.110

uTreeIter<mynode>::init              3     0.003     0.002     0.004     0.008     0.003     0.002     0.004     0.008
   Tree<mynode>::top                 3     0.000     0.000     0.000     0.000     0.000     0.000     0.000     0.000

fred::main                           1     2.289     2.289     2.289     2.289     0.110     0.110     0.110     0.110
   uTreeIter<mynode>::operator>>    63     0.013     0.005     0.149     0.816     0.003     0.002     0.008     0.188
   Tree<mynode>::insert             20     0.065     0.006     0.125     1.303     0.002     0.000     0.005     0.035
   mynode::mynode                   20     0.002     0.002     0.006     0.044     0.002     0.001     0.005     0.031
   uTreeIter<mynode>::over           3     0.005     0.004     0.008     0.016     0.003     0.002     0.004     0.008

Tree<mynode>::insert                20     0.065     0.006     0.125     1.303     0.002     0.000     0.005     0.035
   Tree<mynode>::insertNode         20     0.063     0.001     0.123     1.268     0.004     0.001     0.014     0.078

Tree<mynode>::insertNode           210     0.006     0.001     0.123     1.268     0.004     0.001     0.014     0.882
   Tree<mynode>::insertNode        190     0.006     0.001     0.117     1.148     0.004     0.001     0.007     0.804
   TFriend::right                  190     0.001     0.000     0.001     0.124     0.001     0.000     0.001     0.124
   operator>                       190     0.001     0.000     0.001     0.125     0.001     0.000     0.001     0.125
   operator<                       190     0.001     0.000     0.001     0.137     0.001     0.000     0.001     0.137

Call Cycles

Cycle 1:  uPostorderTreeGen<mynode>::findNextNode -> uPostorderTreeGen<mynode>::findNextNode
Cycle 2:  uInorderTreeGen<mynode>::findNextNode -> uInorderTreeGen<mynode>::findNextNode
```

Figure 5.2: Initial Implementation: Exact Call-Graph Display

The initial μProfiler **ECG** implementation allows a user to display a call-graph for each task's execution. An example display of the initial implementation is presented in Figure 5.2. The information provided in the display includes the number of calls to the routine, the exclusive time of the routine (called CPU time), the inclusive time of the routine (called real time), and the average, maximum and minimum of those values [Zak00]. This information is displayed for each routine executed (totalled over all calls to it) as well as for each callee routine of each caller routine (totalled over all calls to the callee by the caller). In addition to time, the information can be displayed in terms of hardware events [Les05]. However, only the hardware events counted during the execution of the routine itself (exclusive events) are displayed, and no maximum or minimum values are provided. The initial implementation also displays a list of call cycles.

The overall goal of the advances discussed in this chapter is to develop μProfiler's **ECG** metric into a state-of-the-art metric with good performance that scales to programs of long duration and complex behaviour, providing an environment conducive to more thorough yet simpler user analysis of a call-graph.

5.1 Initial Implementation Issues

This section describes several issues arising in the initial implementation of the **ECG**. I addressed each issue in the advanced implementation of the **ECG** and the solutions are discussed in Section 5.2.3.

The first issue involves the **ECG** existing as two separate metrics. In the initial implementation, one version of the metric displays the call-graph in terms of time, and another version of the metric displays the call-graph in terms of hardware events. Having two separate starting points and displays for the same metric is an unnecessary complication and confusing for a user. Furthermore, as previously mentioned, the hardware-event-based version only displays a subset of the information displayed in the time-based version. Both versions of the metric can collect and analyze the same type of profiling data, and thus, should consistently display the same information to a user.

The second issue involves the lack of separation between tasks and coroutines. During a task's execution, the task may execute one or more coroutines (i.e., execute on one or more coroutine stacks). In the initial implementation, each per-task call-graph combines information regarding task routines (executed on the task's stack) and coroutine routines (executed on the coroutine stacks) without any indication of which routine comes from which context. The lack of separation between task and coroutine routines can result in an increased amount of analysis time for a user. Also, by only providing a per-task break down, the call-graph for each coroutine is split across the various per-task call-graphs, again complicating the analysis.

The third issue involves the simplicity of the visualization. The display provided by the initial implementation is very simple in nature, providing all information at once, in one format, in a static window. There is no opportunity for a user to interact in any way with the call-graph data. One purpose of the call-graph is to give a user some indication about the program's control flow; however, such information can be made clearer if a user can in some way progress through and view the call-graph step-by-step. Also, since only the callees of a routine are listed, a user must analyze the entire call-graph in order to determine the set of callers of any given routine.

The final issue involves the lack of scaling of the displayed values. All values displayed are either in millisecond time units or un-scaled hardware-event counts. With respect to time, millisecond time units are not always the best choice. Using nanoseconds for routines of very short duration and seconds for routines of longer duration is much more appropriate. With respect to hardware events, the hardware-event counts need to be scaled to an appropriate unit, as event counts can quickly become very large even for routines of short duration. Appropriately scaled values can make the call-graph much more readable as well as easier to understand and analyze. Furthermore, there is only a limited amount of space on the window, and hence, displaying un-scaled values of small or large magnitude uses up valuable space.

5.2 Advanced Implementation

While addressing the issues from the initial implementation of the **ECG**, the advanced implementation has progressed in two major areas: the data structures used to store the profiling data collected during monitoring and the visualization of the collected data.

5.2.1 Data Collection

At each occurrence of the previously described relevant events, profiling data is collected and stored in specific data structures.

At any time during execution of the program, there exists a caller-callee routine pair (i.e., a call edge) corresponding to the current state of execution. The callee represents the routine executing at that time and the caller represents the routine that called the callee.

In the initial implementation of the **ECG**, the data structure consisted of a hash table. One hash table was maintained for each task. Each hash-table entry represented a caller routine (i.e., a routine with one or more callees) and it maintained a list of callee objects (one for each routine called by the caller routine). The hash-table key was the routine address of the caller and each callee in the caller's callee list was also identified by its routine address. The profiling data collected during monitoring, at the occurrence of a relevant event, was stored within the callee object corresponding to the current state of execution. Before any profiling data could be stored, the corresponding caller-callee routine pair needed to be found in the hash table. This involved a hash-table look-up using the routine address of the caller, followed by a linear search of the caller's callee list comparing the routine address of the callee. The profiling data stored included the number of calls to the callee from the caller as well as the inclusive and exclusive time for the callee when called by the caller. Furthermore, at each occurrence of the routine-enter event, the corresponding caller-callee routine pair may need to be inserted into the hash table (i.e., add a hash-table entry for the caller and/or a callee object to the caller's list for the callee).

The advanced implementation of the **ECG** replaces the hash table with a calling context tree (CCT) [FFMC03]. A CCT is a space-efficient refinement of a dynamic call tree (DCT), which represents the calling behaviour of a program [ABL97]. In a DCT, a tree node represents a routine activation and a tree edge represents a call from one routine to another routine (i.e., directed edge going from the caller routine to the callee routine). In other words, a DCT adds a new tree node and edge for every routine call (or activation), so the size of a DCT is proportional to the number of calls in the execution of the program. Figure 5.3 shows an example DCT and the corresponding CCT. A CCT maintains unique calling contexts (i.e., the set of call-paths in the CCT and DCT are identical) while removing the redundant data in the DCT by reducing its vertex set according to the following equivalence: two tree nodes in a DCT are equivalent if they represent the same routine and they have the same caller in the tree. In the example in Figure 5.3, the CCT maintains the two unique calling contexts for routine C (i.e., M $\rightarrow$ A $\rightarrow$ C and M $\rightarrow$ D $\rightarrow$ C), while discarding the redundant data present in the DCT. In other words, in a CCT, a tree node represents a routine, and tree nodes and edges aggregate data for multiple routine calls. The CCT was chosen over the DCT because it is a significantly more space-efficient data structure than the DCT. However, aggregation in the CCT means data on a per-routine-call basis is lost, unlike the DCT. Fortunately, this data is not required for the **ECG** because it provides information on a per-routine basis only.

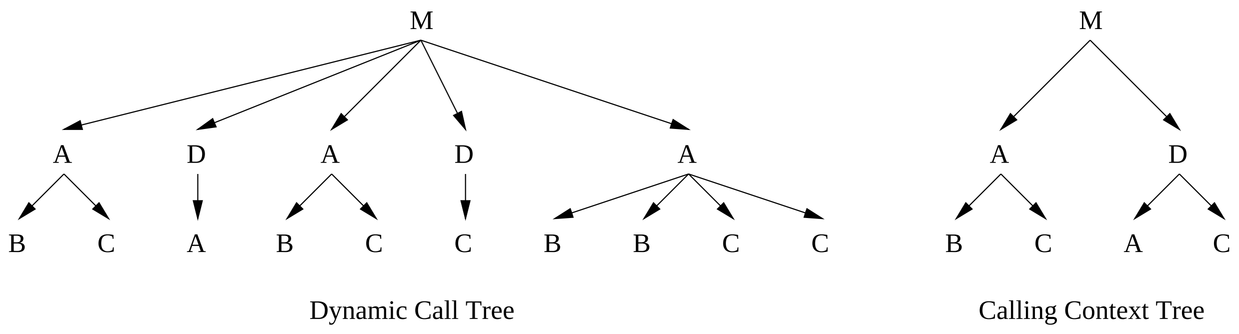


Figure 5.3: Comparison of DCT and CCT

A further refinement has been made to the CCT with the addition of back-edges [ABL97]. A back-edge is an edge from a specific routine to an ancestor of that routine; therefore, a back-edge represents a call cycle. Figure 5.4 shows a CCT with and without back-edges. There are two advantages of back-edges. Firstly, back-edges can reduce the space requirements of a CCT (i.e., provide a bound on the size of a CCT); given no back-edges and a recursive program, the size of a CCT is unbounded. Secondly, back-edges indicate the locations of call cycles in the CCT, reducing the amount of time required for analysis. There are also two disadvantages of back-edges. Firstly, there is a small amount of additional work required to determine when to add a back-edge to the CCT (described in Section 5.2.1.1). Secondly, back-edges can destroy the context-uniqueness property of a CCT. Using a conservative approach to path detection, some call-paths fail to be detected. For example, in Figure 5.4, the call-path A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ B $\rightarrow$ D is not detected in the CCT with back-edges; only the call-paths A $\rightarrow$ B $\rightarrow$ D and A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ B are detected. On the other hand, a liberal approach to path detection can detect some call-paths not executed. For example, in Figure 5.4, the call-path A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ B $\rightarrow$ D is detected in the CCT with back-edges even if only the call-paths A $\rightarrow$ B $\rightarrow$ D and A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ B are executed. I chose the liberal approach to path detection and the consequences of this and the corresponding solutions are discussed in Section 5.3.2. I chose to include back-edges because I felt the advantages outweigh the disadvantages.

$$\begin{tikzcd}
 & A \arrow[d] & \\
 & B \arrow[dl] \arrow[dr] & \\
C \arrow[ur, bend left] & & D
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
A \arrow[d] \\
B \arrow[d] \\
C \arrow[d] \\
B \arrow[d] \\
D
\end{tikzcd}$$

CCT with back-edges CCT without back-edges

Figure 5.4: CCT Refinement

In the advanced implementation, a CCT is a collection of node objects rooted at a single node, where each node represents a routine. Each node maintains a pointer to its caller node in the tree (i.e., the routine's caller), except the root, and a list of callee edge objects (i.e., the routine's callees). Each edge maintains a pointer to its corresponding callee node in the tree. The profiling data collected during monitoring is stored within the edge object corresponding to the current state of execution. Figure 5.5 illustrates the specific data structures composing the CCT. The data objects are discussed in Section 5.2.1.2.

As just mentioned, the size of the CCT is bounded. Let n be the number of routines executed in the program. Since back-edges are used in the CCT, the depth of the CCT is bounded by n. For each node in the CCT, its number of callee edges (and hence nodes) is at most $n-1$ because each callee represents a unique routine called from the node and if the node calls itself that call is instead represented by a back-edge. Thus at each depth in the tree, there can be at most $n-1$ nodes for each node at the previous depth in the tree. Since the tree starts with one node at

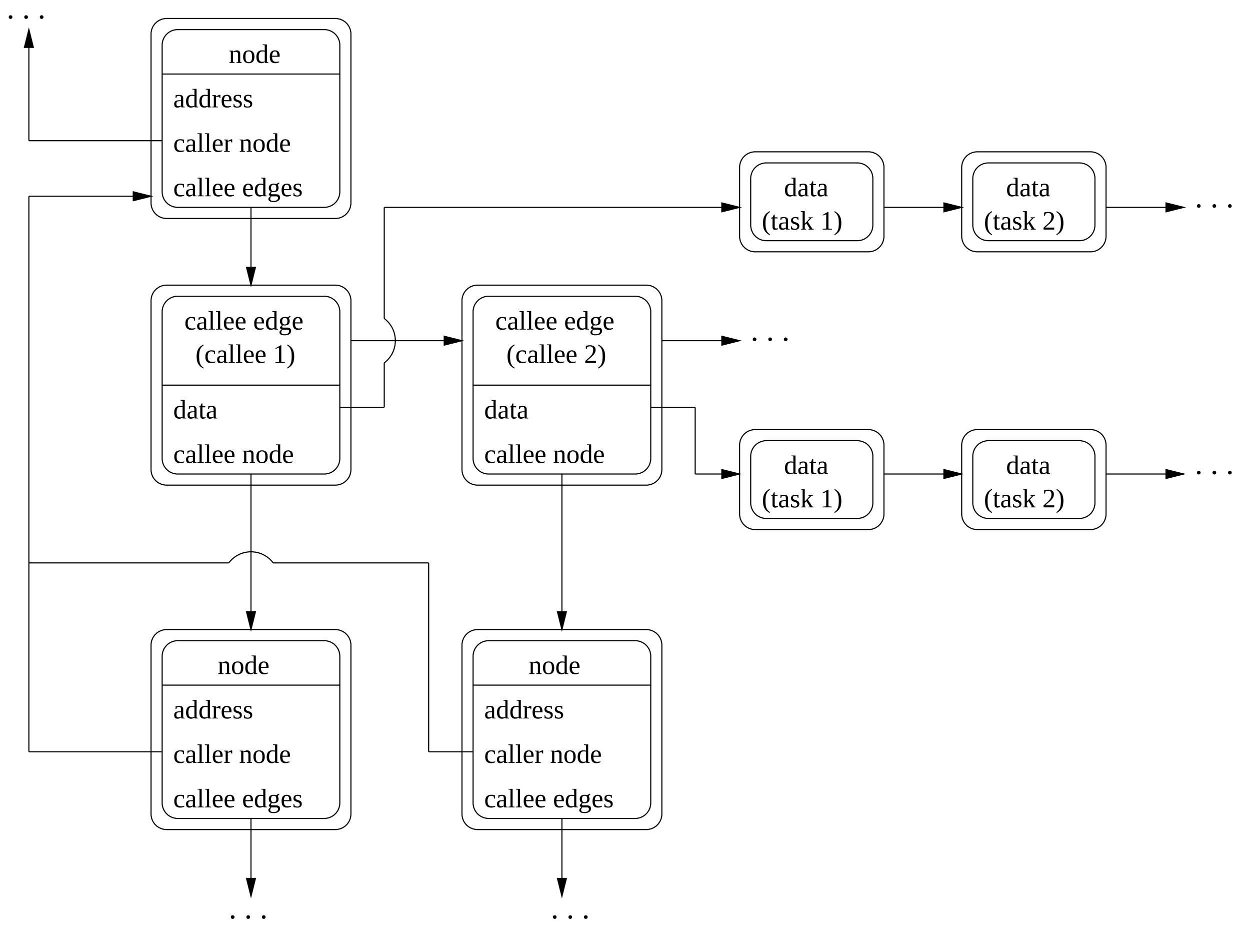


Figure 5.5: Advanced Implementation: Exact CCT Data Structures

the root and has a depth of n, the maximum number of nodes at the lowest depth of the tree is $(n-1)^{n-1}$. In other words, the breadth of the CCT is bounded by $(n-1)^{n-1}$.

The use of a CCT is more appropriate than a hash table because its structure mirrors the calling structure or call-graph of a task's execution in the program. The call-graph reconstructed from a hash table is not as accurate; the hash table does not maintain all the unique calling contexts that a CCT does because it only maintains the caller-callee routine pairs rather than entire call-paths. Furthermore, a CCT lends itself to the use of efficient tree traversal algorithms needed during analysis (e.g., depth-first-search algorithm).

5.2.1.1 Creating and Updating a CCT

During execution of the program, the edge associated with the current state of execution is stored and changes at each routine-enter and routine-exit event. This edge represents the currently executing caller-callee routine pair and provides access to the current node (i.e., node representing the callee routine currently executing).

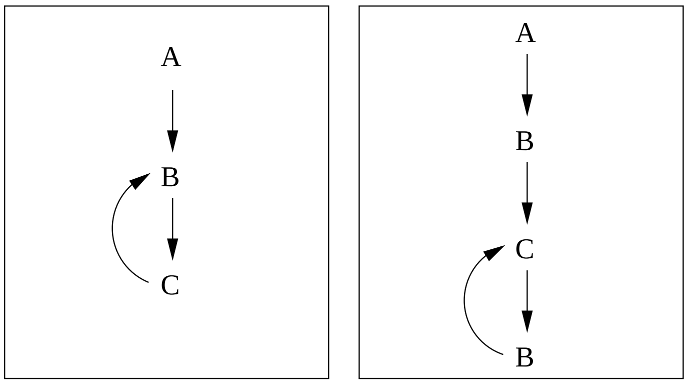
The structure of the CCT is formed during routine-enter events. When a routine-enter event occurs, this means the routine represented by the current node has made a call to another routine, say routine N. A call to N can result in one of three mutually exclusive actions:

1. One of the current node's edges is reused. If the current node previously called routine N, then there exists in its list of edges, one edge representing a call to routine N. This edge is reused.

2. A new back-edge is created. If a node representing routine N appears as an ancestor of the current node then a new edge representing a call to routine N is added to the current node's list of edges. This edge is marked as a back-edge and represents a cycle in the tree. Finding an ancestor node requires following the caller pointer maintained by each node from the current node up to the root node. At each node along this path, the routine address of N is compared to the routine address of the routine represented by the node. More efficient ways of finding an ancestor node are possible (e.g., using a Bloom filter [Blo70]); however, in practice a call-path of length 16 is considered long and a linear search along such a path is reasonable.

3. A new node and edge are created. The new edge representing a call to routine N (represented by the new node) is added to the current node's list of edges. The new edge maintains a pointer to the new node.

In all three cases, the reused or new edge becomes associated with the current state of execution and is updated with the appropriate profiling data. The profiling data is stored at the edge objects, rather than the node objects, because of back-edges in the CCT. Since a back-edge represents a call from a node to one of its ancestor nodes, the routine represented by the ancestor node now



(a) Current Approach (b) Alternative Approach

Figure 5.6: Adding Back-Edges

has multiple callers. For example, in Figure 5.4, routine B has two callers: routine A and routine C via the back-edge. However, in a call-graph, data for a specific routine needs to be broken down by caller, so storing data at the edge objects provides a means of keeping that data separated.

An alternative approach exists for adding back-edges to the CCT (proposed by thesis reader, David Taylor). A back-edge from the current node to an ancestor node is added only if the ancestor-node's caller-routine, represented by its caller pointer, is the same as the routine represented by the current node. For example, given the call-path A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ B $\rightarrow$ C, Figure 5.6(a) shows the resulting CCT for the current approach and Figure 5.6(b) shows the resulting CCT for the alternative approach. The alternative approach does result in one extra node being added to the CCT for each cycle path, but now each node has a uniquely defined caller (e.g., caller B for routine C in Figure 5.6(b)). Therefore, data can be stored in the node objects and the edge objects are no longer required. This alternative approach can be considered for future work.

When a routine-exit event occurs, this means the routine represented by the current node has completed and execution is being transferred back to its caller routine. Therefore, the edge associated with the current state of execution changes. Unfortunately, an issue arises given the presence of back-edges, as this edge cannot always be determined by following the caller pointer of the current node. As previously mentioned a routine can have multiple callers. The current node's caller may not be the routine represented by its caller pointer, but may instead be a routine represented by a back-edge leading to the current node. In the example in Figure 5.7, routine A calls routine B, which then calls routine C, which then calls routine B. The routine represented by B's caller pointer is A (corresponding to E_{AB}); however, for this call-path, routine B's caller is actually routine C represented by the back-edge E_{CB}. To solve this issue, I store an edge path (an array of edges representing the current runtime stack) associated with the current state of execution instead of one edge. This edge path (see Figure 5.7) simply represents the current call-path in the tree. Therefore, when a routine-enter event occurs, the reused or new edge is added to the end of the array and the array pointer is incremented. When a routine-exit event occurs, the array pointer is decremented. At all times the array pointer points to the edge associated with the current state of execution.

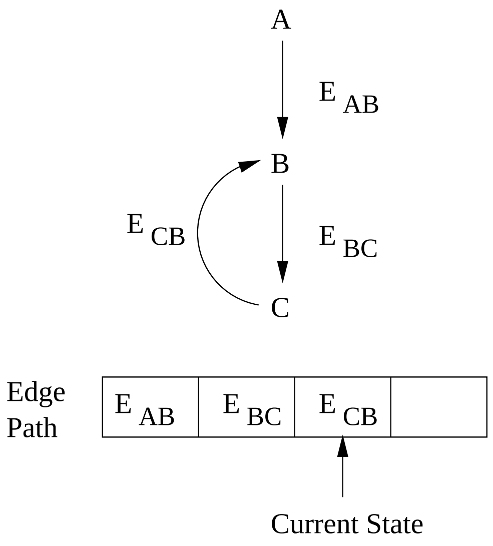


Figure 5.7: Advanced Implementation: Exact CCT Edge Path

The profiling data stored in an edge object, which represents a caller-callee routine pair, includes the number of calls to the callee from the caller (*calls*) and the total self or exclusive time (*totalExclTime*), total block time (*totalBlockTime*), and total inclusive time (*totalInclTime*) for the callee when called by the caller. The total descendant-time is computed according to the following formula:

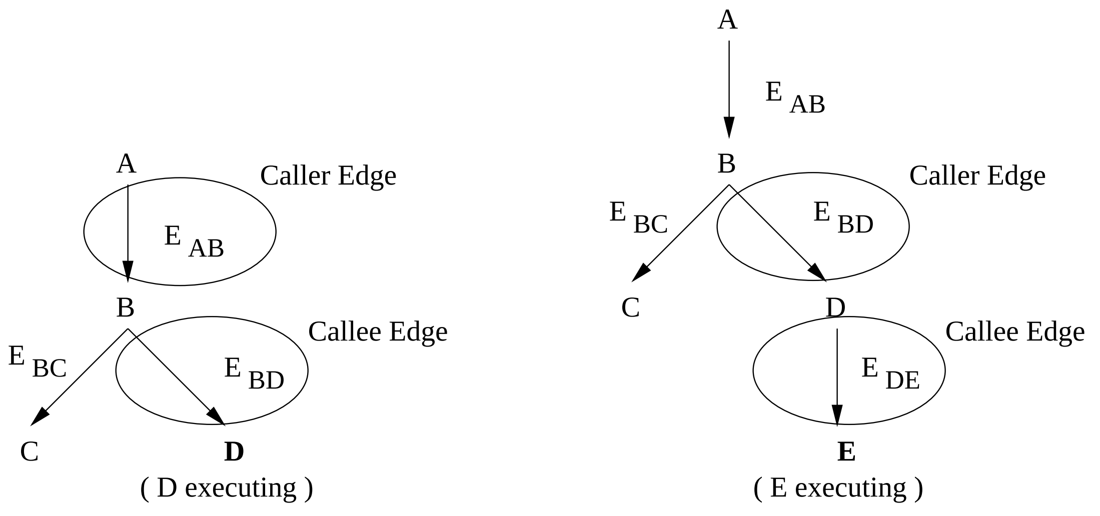
$$totalDescTime = totalInclTime - totalExclTime - totalBlockTime \tag{5.1}$$

To compute the totals during monitoring, additional variables corresponding to each total are required ($startInclTime$, $startExclTime$, $startBlockTime$). These variables keep track of the starting times of the various relevant events. Since the information provided by the **ECG** can be displayed in terms of hardware events as well as time, each of the previously mentioned time variables also exists in hardware-event count form and all calculations are similarly executed. One exception, discussed shortly, is the hardware-event counts associated with the blocking period.

Subsets of the variables are updated at the various relevant events. At the routine-enter event, profiling data is updated at two edges: the caller edge and the callee edge. Figure 5.8 illustrates a CCT before (Figure 5.8(a)) and after (Figure 5.8(b)) a routine-enter event where routine D calls routine E. The caller edge represents the edge associated with the current state of execution before the routine call (e.g., edge E_{BD} in Figure 5.8(b)) and the callee edge represents the edge associated with the current state of execution after the routine call, or in other words, after the routine-enter event (e.g., edge E_{DE} in Figure 5.8(b)). Therefore, the callee edge before a routine-enter event occurs (e.g., edge E_{BD} in Figure 5.8(a)) becomes the caller edge after the routine-enter event occurs. For the callee edge, execution is starting (e.g., for routine E), so number of $calls$ is incremented and both $startInclTime$ and $startExclTime$ are initialized to the current time. For the caller edge, exclusive execution is temporarily stopped (e.g., for routine D), so $totalExclTime$ is increased according to the following formula:

$$totalExclTime = totalExclTime + (current\ time - startExclTime) \tag{5.2}$$

At the routine-exit event, profiling data is also updated at the caller edge and the callee edge. The callee edge represents the edge associated with the current state of execution before the routine exit and the caller edge represents the edge associated with the current state of execution after the routine exit. Therefore, these edges correspond to the same edges as in the



(a) CCT Before Routine Enter (b) CCT After Routine Enter (D calls E)

Figure 5.8: Advanced Implementation: Routine-Enter Event

previous routine-enter event. For the callee edge, execution is ending (e.g., for routine E), so $totalExclTime$ is increased (see Equation 5.2) and $totalInclTime$ is increased (replace $Excl$ with $Incl$ in Equation 5.2). For the caller edge, exclusive execution is restarting (e.g., for routine D), so $startExclTime$ is reinitialized to the current time.

At the task block and unblock events, profiling data is updated only at the callee edge, the edge associated with the current state of execution. At a task block, exclusive execution is temporarily stopping, so $totalExclTime$ is increased and $startBlockTime$ is initialized to the current time. At a task unblock, exclusive execution is restarting, so $startExclTime$ is reinitialized to the current time and $totalBlockTime$ is increased (replace $Excl$ with $Block$ in Equation 5.2). However, for the hardware-events, the event counts for the period spent blocking are not computed because of the multi-processor environment. In a multi-processor environment, each processor has its own set of hardware counters with different hardware-event counts. When a task becomes blocked, it does so on a particular processor, and so, it would record the current hardware-event counts of that processor's hardware counters (in $startBlockCounts$ for example). When a task becomes unblocked, it may do so on a different processor with a different set of hardware counters.

The current hardware-event counts for this processor cannot be used to compute the event counts of the blocking processor (e.g., *startBlockCounts*) because these event counts are unrelated; therefore the increase to *totalBlockCounts* cannot be computed. There is no analogous problem for the time event because the system clock is global and synchronized across the processors. Although *totalBlockCounts* is not computed, additional inclusive event-count computations are required at the block and unblock events. Because of a blocking event, a task may be running on one processor when a specific routine starts, but on another processor when that routine ends. Therefore, at a task block, *totalInclCounts* is increased and at a task unblock *startInclCounts* is reinitialized to the current event counts for all active routines (i.e., these computations are done not only at the callee edge) because any of the active routines may have started on a different processor and execution eventually returns to each. To accomplish this, the edge path representing the current call-path in the tree is walked and these computations are done for each edge along that path.

An alternative approach to walking up the edge path on each block and unblock event is to carry an adjustment up on each routine return (proposed by thesis reader, David Taylor). At the block event, totalInclCounts is increased only at the callee edge, and at the unblock event, startInclCounts is reinitialized only at the callee edge. After the unblock event, the increment to totalInclCounts is propagated up one level in the edge path at the next routine-exit event. The time between the unblock event and the routine-exit event (at the first propagation) and the time between routine-exit events (at each following propagation) must be added to the increment because startInclCounts is only reinitialized for the callee edge at the unblock event. The time of any subsequent blocking periods must also be added to the increment. During coroutine execution, the current task may not exit all routines along the edge path before a coroutine-discontinue event occurs (see Section 5.3.1); however, the edge path is walked up at each coroutine-discontinue event, so the increment can be propagated all the way up to the root at that time. This approach does require an additional variable (for the increment) to be maintained and additional computations to be made at each routine-exit event. The efficiency of each implementation approach depends heavily on program behaviour. Programs that block infrequently and have short call-paths (e.g., 16 or less), but make frequent routine calls (as often occurs in object-oriented programs), would benefit from the first implementation approach. Programs that block frequently and have long call-paths, but make few routine calls, would benefit from the second implementation approach. I believe the first category of programs are more common than the latter, and hence, feel the first implementation approach would be more efficient in general.

5.2.1.2 Coroutines

A CCT, and its associated edge path, is maintained for each execution entity having its own stack; i.e., one for each task and each coroutine. However, because multiple tasks may execute a single coroutine (i.e., multiple tasks' threads may execute on a coroutine's stack), a CCT must keep profiling data separated by task. Therefore, each edge object maintains a list of profiling data objects, one for each task executing along that edge, rather than a single set of data (see Figure 5.5). For each per-task CCT, the lists consists of only one element because only the task's thread may execute on the task's stack.

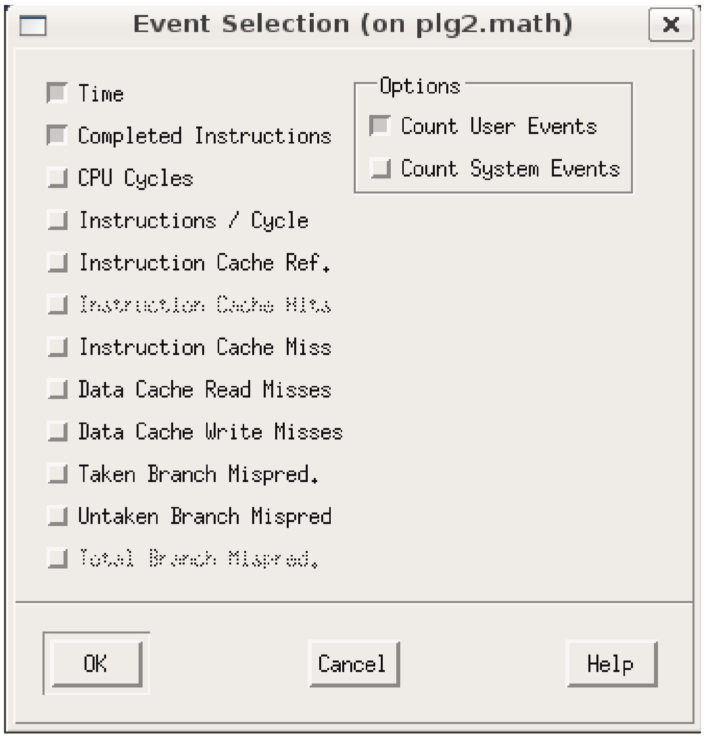
Any execution entity can activate a coroutine (i.e., a task can activate a coroutine and a coroutine can activate another coroutine). When an execution entity activates a coroutine (e.g., coroutine resume), the entity (i.e., the task's thread executing the entity) begins execution of the coroutine at the point of the last inactivation (e.g., coroutine suspend). During the coroutine's execution, routines may be called and executed until the coroutine becomes inactive. An inactivation may occur anywhere within a routine. Therefore, execution of the coroutine can start and end anywhere within a routine. Furthermore, the subsequent activation may be made by an execution entity executed by a different task's thread, meaning execution of the coroutine starts from the point of the last inactivation reached by an execution entity executed by another task's thread. For this reason, a coroutine activation is treated similarly to a routine call and a coroutine inactivation is treated similarly to a routine return. When a coroutine becomes active, a coroutine-discontinue event occurs for the execution entity activating the coroutine and a coroutine-continue event occurs for the coroutine. Execution moves from the execution entity and its stack to the coroutine and its stack. When a coroutine becomes inactive, a coroutine-discontinue event occurs for the coroutine and a coroutine-continue event occurs for another execution entity. Execution moves from the coroutine and its stack to the execution entity and its stack.

Different actions are taken at a coroutine-discontinue and a coroutine-continue event. A coroutine-discontinue event is treated similarly to a routine-exit event. Therefore, $totalExclTime$ and $totalInclTime$ are increased. A coroutine-continue event is treated similarly to a routine-enter event. Therefore, $startExclTime$ and $startInclTime$ are reinitialized to the current time. However, instead of incrementing the number of $calls$, the number of *continues* is incremented. The name continues represents the continuation of execution from the point of the last inactivation. The profiling data is updated, as just described, at the callee edge (edge associated with the current state of execution), but other edges, discussed in Section 5.3.1, are also updated.

5.2.2 Visualization

The first step in running the **ECG** metric involves selecting the events (i.e., time and/or hardware events) for which the profiling data is collected and subsequently displayed. The event-selection window is shown in Figure 5.9 with the "Time" and "Completed Instructions" events selected. As hardware events are selected the remaining hardware events become greyed out depending on the number of hardware counters available and the capabilities of those counters, i.e., which events they are able to count [Les05]. For hardware events, a user can also choose to have profiling data collected while executing user and/or system code (see upper right options box of Figure 5.9).

Once program execution and monitoring are completed, the task/coroutine-selection window is displayed (see Figure 5.10). A user can select the call-graph for any task or coroutine by clicking on a task or coroutine name in the left column. The tasks are listed above the dashed separator line and the coroutines below it. Summary information, broken down for each event selection from Figure 5.9, is displayed for each task and coroutine. The summary information includes total execution-time or event-counts as well as total block-time for the time event. The purpose of the summary information is to give a user some direction as to which call-graph to analyze first. For example, a task with a large time or count value is a potential "hot-spot" of execution.


Event Selection (on plg2.math)

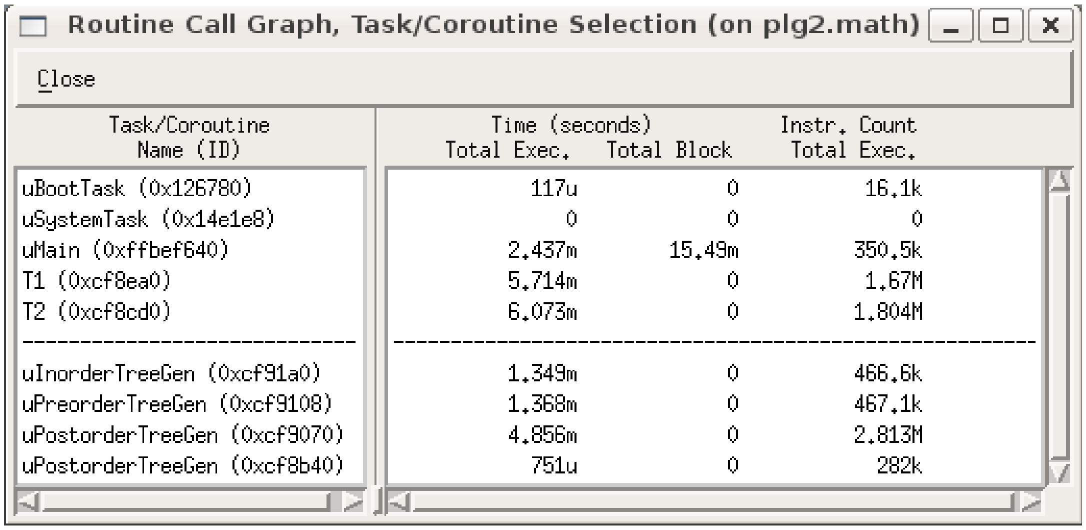
- [✓] Time
- [✓] Completed Instructions
- [ ] CPU Cycles
- [ ] Instructions / Cycle
- [ ] Instruction Cache Ref.
- [ ] Instruction Cache Hits
- [ ] Instruction Cache Miss
- [ ] Data Cache Read Misses
- [ ] Data Cache Write Misses
- [ ] Taken Branch Mispred.
- [ ] Untaken Branch Mispred
- [ ] Total Branch Mispred.

Options
- [✓] Count User Events
- [ ] Count System Events

Figure 5.9: Advanced Implementation: Exact Event-Selection Window

The call-graph window in Figure 5.11 is displayed after selecting a task in the task/coroutine-selection window. Figure 5.11 shows the call-graph window for task T1 from Figure 5.10. The call-graph window contains several panes (from top to bottom): routine pane, callers pane, callees pane, callees-visited pane, cycles pane and coroutine-selection pane. All data displayed in the call-graph window is for a single selected event (i.e., time or a hardware event). In Figure 5.11, the call-graph is currently displaying the time-event data for task T1.

The routine pane lists each routine executed by a task. For each routine, the information includes (left to right) the number of calls/continues to the routine as well as a histogram showing that number as a percentage of the total, the self time of the routine as well as a histogram showing that number as a percentage of the total, the descendant time of the routine, the block time of the routine, and the routine name (optionally followed by a file and line number of the

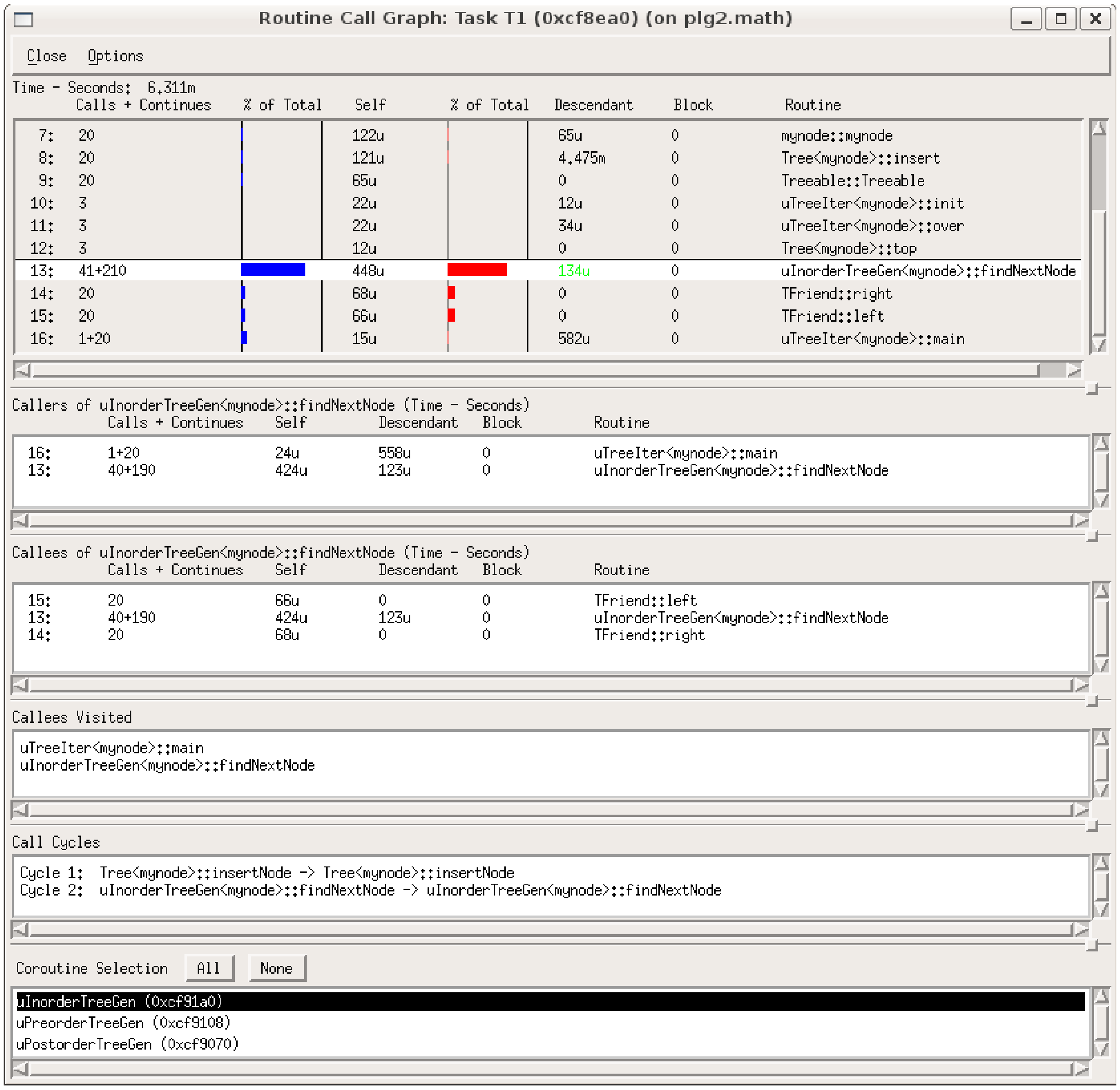


```
        Task/Coroutine                 Time (seconds)           Instr. Count
          Name (ID)                Total Exec.   Total Block     Total Exec.
uBootTask (0x126780)                      117u            0           16.1k
uSystemTask (0x14e1e8)                       0            0               0
uMain (0xffbef640)                      2.437m       15.49m          350.5k
T1 (0xcf8ea0)                           5.714m            0           1.67M
T2 (0xcf8cd0)                           6.073m            0          1.804M
-----------------------------  ---------------------------------------------------------
uInorderTreeGen (0xcf91a0)              1.349m            0          466.6k
uPreorderTreeGen (0xcf9108)             1.368m            0          467.1k
uPostorderTreeGen (0xcf9070)            4.856m            0          2.813M
uPostorderTreeGen (0xcf8b40)              751u            0            282k
```

Figure 5.10: Advanced Implementation: Exact Task/Coroutine-Selection Window

routine source). All values displayed for a specific routine are totalled over all calls to the routine. The percentage values are represented by histograms, allowing a user to quickly analyze and understand the distribution of calls and time among the various routines. The routines are sorted by self time and the currently selected routine is highlighted in white (e.g., routine `findNextNode` in Figure 5.11). Any routine in this pane can be selected by clicking on its line. Once a routine is selected, the callers and callees panes are updated for the selected routine.

For each caller, the callers pane lists the number of calls/continues and times attributed to the caller by the selected routine. Therefore, the sum of the values of the callers for each individual field (calls, self, descendant and block) is equal to the total value displayed for the selected routine in the routine pane. For each callee, the callees pane lists the times attributed to the selected routine by the callee as well as the number of calls/continues from the selected routine to the callee. Therefore, the sum of the values of the callees for all time fields (self, descendant and block) is equal to the total descendant time displayed for the selected routine in the routine pane. A caller or callee routine can be selected from their respective panes, by clicking on its line, and this routine then becomes the selected routine in the routine pane.



```
Time - Seconds:  6.311m
          Calls + Continues    % of Total   Self      % of Total   Descendant   Block       Routine
  7:   20                                  122u                    65u          0          mynode::mynode
  8:   20                                  121u                    4.475m       0          Tree<mynode>::insert
  9:   20                                  65u                     0            0          Treeable::Treeable
 10:   3                                   22u                     12u          0          uTreeIter<mynode>::init
 11:   3                                   22u                     34u          0          uTreeIter<mynode>::over
 12:   3                                   12u                     0            0          Tree<mynode>::top
 13:   41+210                              448u                    134u         0          uInorderTreeGen<mynode>::findNextNode
 14:   20                                  68u                     0            0          TFriend::right
 15:   20                                  66u                     0            0          TFriend::left
 16:   1+20                                15u                     582u         0          uTreeIter<mynode>::main
```

```
Callers of uInorderTreeGen<mynode>::findNextNode (Time - Seconds)
             Calls + Continues    Self          Descendant   Block         Routine
 16:         1+20                 24u           558u         0             uTreeIter<mynode>::main
 13:         40+190               424u          123u         0             uInorderTreeGen<mynode>::findNextNode
```

```
Callees of uInorderTreeGen<mynode>::findNextNode (Time - Seconds)
             Calls + Continues    Self          Descendant   Block         Routine
 15:         20                   66u           0            0             TFriend::left
 13:         40+190               424u          123u         0             uInorderTreeGen<mynode>::findNextNode
 14:         20                   68u           0            0             TFriend::right
```

```
Callees Visited
 uTreeIter<mynode>::main
 uInorderTreeGen<mynode>::findNextNode
```

```
Call Cycles
 Cycle 1:  Tree<mynode>::insertNode -> Tree<mynode>::insertNode
 Cycle 2:  uInorderTreeGen<mynode>::findNextNode -> uInorderTreeGen<mynode>::findNextNode
```

```
uInorderTreeGen (0xcf91a0)
uPreorderTreeGen (0xcf9108)
uPostorderTreeGen (0xcf9070)
```

Figure 5.11: Advanced Implementation: Exact Call-Graph Window

The callees-visited pane keeps track of the current path visited by a user in the call-graph. As callee routines are selected the path increases (routines are added to the visit pane list) because a user is moving down the call-graph. As caller routines are selected the path decreases (routines are removed from the visit pane list) because a user is moving up the call-graph. This pane allows a user to keep track of the position in the call-graph at all times.

The cycles pane displays all cycles detected in the call-graph. If no cycles exist the cycles pane is not shown.

The final pane is the coroutine-selection pane. This pane lists each coroutine executed by the task. If the task executes no coroutines this pane is not shown. A user can select any number of coroutines from the list by clicking on the coroutine name. Also, the “All” and “None” buttons allow a user to quickly select or deselect all coroutines. By default no coroutines are selected because I cannot without user input determine which coroutines to select. Once one or more coroutines are selected, the coroutine’s routines executed by the task are displayed below a separator line in the routine pane. The separator line provides a clear division between the task and coroutine routines. The same information displayed for a task routine is displayed for a coroutine routine. The percentages represented by the histograms are, for task routines above the separator line, percentages of the total for the task, and for coroutine routines below the separator line, percentages of the total for the selected coroutines.

The coroutine selections also affect the total execution time displayed in the top title bar of the routine pane. If no coroutines are selected, the total represents the total execution time for the task while executing task routines. If one or more coroutines are selected, the total execution time for the task while executing the selected coroutines’ routines is included in the total.

Various options are available from the pull-down menu associated with the “Options” button (see Figure 5.12). The “Histogram” option allows a user to show or hide the histograms in the routine pane. The “File Info” option allows a user to show or hide file information for each routine displayed in the routine, callers, callees and cycles panes. The file information includes the file name (and path) containing the routine and the line number corresponding to the start of the routine within the file. The pull-down menu associated with the “Events” option allows a user to choose the event for which the call-graph data is displayed. The events available are those events previously selected on the events-selection window (e.g., “Time” and “Completed Instructions” in Figure 5.9). If the event is changed from “Time” to “Completed Instructions”, then the data displayed in the call-graph window (Figure 5.11) changes from time to hardware-event counts (i.e., the number of completed instructions). When a hardware event is selected, the “Block” field is hidden in the routine, callees and callers panes because this data is not collected for hardware events (see Section 5.2.1).

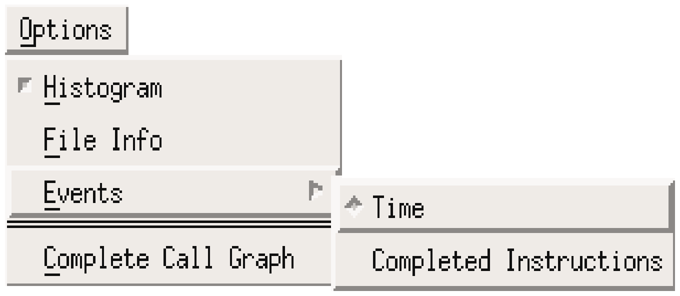
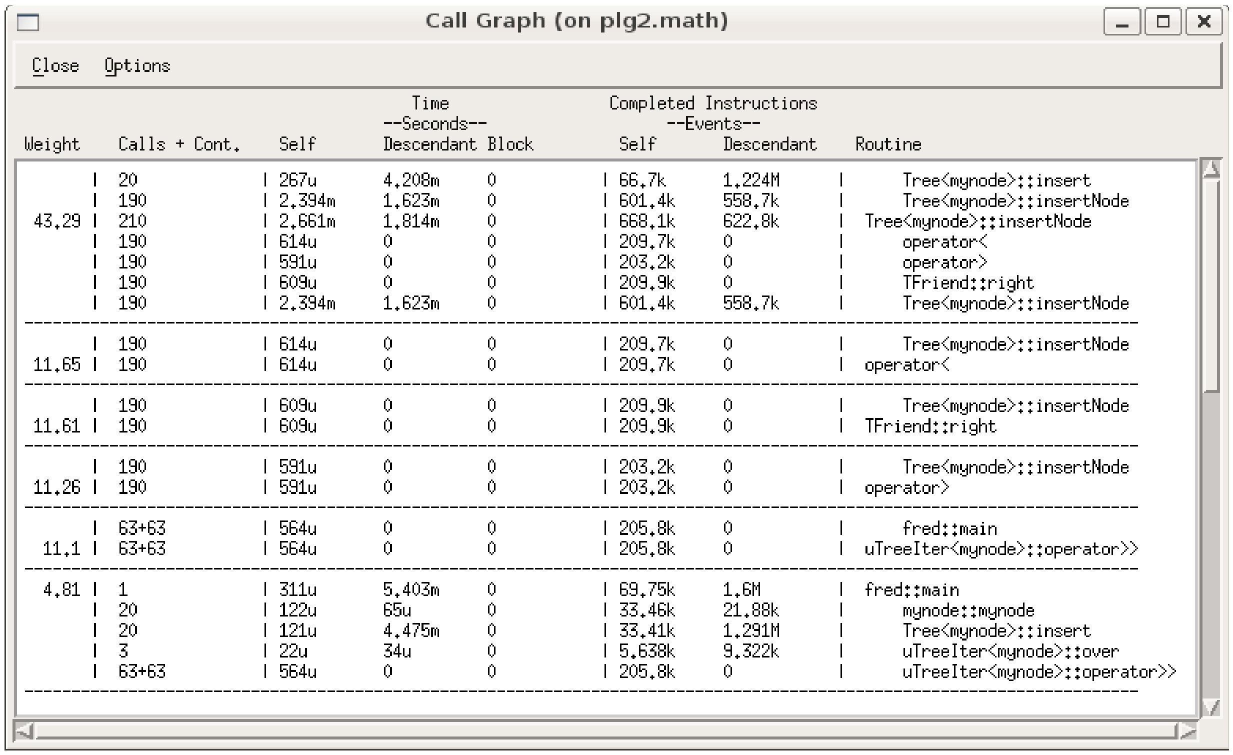


Figure 5.12: Advanced Implementation: Exact Options Menu

The final option, “Complete Call-Graph”, opens a complete call-graph window, displaying caller and callee information for all routines rather than just a selected routine (see Figure 5.13). For each routine, the callers are listed above the routine and the callees are listed below the routine. Information is displayed for all task routines and the routines of the currently selected coroutines. The complete call-graph window provides another means by which a user can view the call-graph data; a user can see and analyze the entire call-graph at once. However, a user may not wish to view the entire call-graph for all events at one time, so from the “Options” menu on the complete call-graph window, a user can show or hide data for individual events. As well, from the “Options” menu the user can show or hide file information for each routine. The routines displayed in the complete call-graph window are sorted by the value of the “Weight” column. For each event, a routine’s self time or hardware-event counts, as a percentage of the total for the event, is computed. A routine’s weight is the average percentage over the events currently displayed in the window. For example, given two events displayed, if a routine accounts for 20% of the total for one event and 11% of the total for the other event, then the routine’s weight is



```
                                     Time                  Completed Instructions
                                  --Seconds--                    --Events--
Weight    Calls + Cont.    Self     Descendant Block         Self        Descendant     Routine

       |  20             | 267u     4.208m     0          | 66.7k      1.224M       |      Tree<mynode>::insert
       |  190            | 2.394m   1.623m     0          | 601.4k     558.7k       |      Tree<mynode>::insertNode
 43.29 |  210            | 2.661m   1.814m     0          | 668.1k     622.8k       |  Tree<mynode>::insertNode
       |  190            | 614u     0          0          | 209.7k     0            |      operator<
       |  190            | 591u     0          0          | 203.2k     0            |      operator>
       |  190            | 609u     0          0          | 209.9k     0            |      TFriend::right
       |  190            | 2.394m   1.623m     0          | 601.4k     558.7k       |      Tree<mynode>::insertNode
--------------------------------------------------------------------------------------------------------------
       |  190            | 614u     0          0          | 209.7k     0            |      Tree<mynode>::insertNode
 11.65 |  190            | 614u     0          0          | 209.7k     0            |  operator<
--------------------------------------------------------------------------------------------------------------
       |  190            | 609u     0          0          | 209.9k     0            |      Tree<mynode>::insertNode
 11.61 |  190            | 609u     0          0          | 209.9k     0            |  TFriend::right
--------------------------------------------------------------------------------------------------------------
       |  190            | 591u     0          0          | 203.2k     0            |      Tree<mynode>::insertNode
 11.26 |  190            | 591u     0          0          | 203.2k     0            |  operator>
--------------------------------------------------------------------------------------------------------------
       |  63+63          | 564u     0          0          | 205.8k     0            |      fred::main
  11.1 |  63+63          | 564u     0          0          | 205.8k     0            |  uTreeIter<mynode>::operator>>
--------------------------------------------------------------------------------------------------------------
  4.81 |  1              | 311u     5.403m     0          | 69.75k     1.6M         |  fred::main
       |  20             | 122u     65u        0          | 33.46k     21.88k       |      mynode::mynode
       |  20             | 121u     4.475m     0          | 33.41k     1.291M       |      Tree<mynode>::insert
       |  3              | 22u      34u        0          | 5.638k     9.322k       |      uTreeIter<mynode>::over
       |  63+63          | 564u     0          0          | 205.8k     0            |      uTreeIter<mynode>::operator>>
--------------------------------------------------------------------------------------------------------------
```

Figure 5.13: Advanced Implementation: Exact Complete Call-Graph Window

15.5 (i.e., $(20+11)/2$).

If a coroutine is selected on the task/coroutine-selection window (see Figure 5.10), the call-graph window displayed and the functionality of that window is very similar to that for a task. One difference is the presence of a task-selection pane instead of a coroutine-selection pane. This pane lists each task that executed the coroutine. Like the coroutine-selection pane, a user can select any number of tasks from the list. By default all tasks are selected because I chose not to initially display an empty window, since without user input I cannot determine which tasks to select. The call-graph data displayed is aggregated for all the selected tasks. In other words, if a coroutine's routine is executed by more than one currently selected task, the data displayed for this routine is the summation of values for those tasks. If a routine is not executed by any of the currently selected tasks the routine is not displayed in the routine pane. Also, the cycles pane only lists those cycles fully executed by the selected tasks. If no tasks are selected all panes are empty.

5.2.3 Addressing Initial Issues

To address the existence of the initial **ECG** as two separate metrics, both versions, time and hardware-event based, have been combined into one metric with a single starting point and display in the advanced implementation. From the event-selection window (see Figure 5.9), a user can now select the time event as well as various hardware events. After profiling is completed and a task or coroutine is selected, a single call-graph window (see Figure 5.11) is displayed and a user can, via the pull-down menu associated with the “Events” option, choose the specific event to display. Such functionality provides a consistent view across the various events, as nothing changes in the call-graph window except the actual values. Furthermore, the complete call-graph window now allows a user to easily analyze the call-graph across any subset of the selected events.

Whereas in the initial implementation coroutine information is combined with information associated with the task, in the advanced implementation coroutines and tasks are cleanly separated in both the per-task and per-coroutine call-graphs. On the per-task call-graph window, a user can select which coroutines to include/exclude from the display and that information is visually separated from that of the task. Such functionality allows a user to more easily and accurately analyze the execution of the task on each individual execution stack (task or coroutine) as well as the execution as a whole. On the per-coroutine call-graph window, a user can again select which tasks to include/exclude from the display. Such functionality allows a user to aggregate coroutine information across tasks in a single call-graph window. Having both a per-task and per-coroutine call-graph provides two means by which to view the call-graph data associated with a specific coroutine. Multiple views can be invaluable depending on the program and the purpose of the analysis. The improvements all stem from the careful separation of the task and coroutine profiling data during monitoring (see Sections 5.2.1.2 and 5.3.1), allowing for analysis on a per-task and per-coroutine basis.

To address the simplicity of the visualization, the call-graph window was completely re-designed with functionality, interactivity and understanding in mind (see Section 5.2.2). The window now provides a more interactive experience for a user. The ability to select coroutines or tasks allows a user to control the amount of information displayed in the call-graph window. Such control provides an environment more conducive to thorough analysis and understanding. Also, a user can progressively move down and up the call-graph by selecting routines in the routine, callers and callees panes, with that movement being tracked in the callees-visited pane. Such functionality allows a user to better understand and visualize the program's control flow. Since multiple views are often beneficial, the complete call-graph window also provides a user with the means to view the entire call-graph at once.

To address the lack of scaling in the initial implementation of the **ECG**, the values displayed in the call-graph window are no longer limited to millisecond time units or un-scaled hardware-event counts. In the advanced implementation, the scaling ranges from nano units all the way up to peta units, taking into account very small and very large values. All values are scaled on an individual basis. The most appropriate unit is determined based on the size of the value. For example, values of less than 10^{-6} use nano units and values of 10^{12} or greater use peta units. Such scaling allows the call-graph to be scalable to routines (and programs) of short and long duration. Also, scaling individual values makes the call-graph window much easier to read and analyze given that the values displayed are often a mixture of both smaller and larger size (e.g., microseconds and milliseconds in Figure 5.11).

5.3 Implementation Issues

This section describes implementation issues I encountered and solved during the writing of the advanced implementation of the **ECG**.

5.3.1 Handling Coroutines

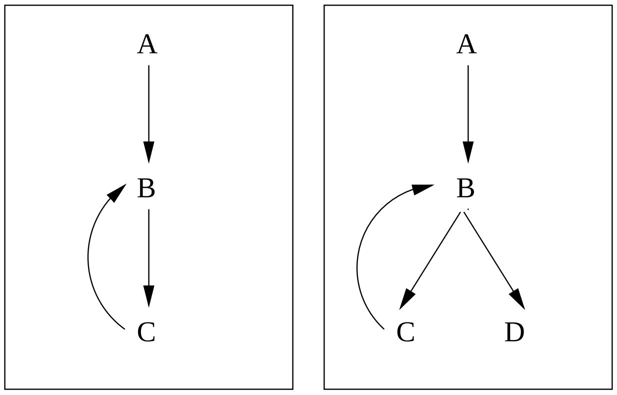
This issue involves the creation and maintenance of the per-coroutine CCT. I chose to store profiling data related to coroutine execution in separate CCTs (see Section 5.2.1.2). This choice was made after considering an alternative solution. The alternative solution maintains one CCT per-task only. The per-task CCT has multiple tree branches (or subtrees), one representing execution on the task's stack and one or more representing execution on the coroutine stacks of the coroutines executed by the task. During monitoring, before profiling data can be stored, a linear search of the CCT's subtrees is required to find the subtree associated with the current execution. However, as previously explained, multiple tasks may execute one coroutine, and on a coroutine activation, a task may start execution of that coroutine at the point of the last inactivation reached by a different task. In other words, a task does not necessarily execute the entire coroutine; therefore, on any particular coroutine activation, the subtree corresponding to that coroutine may not be structurally up-to-date in the task's CCT. This means that at each coroutine-continue event, the corresponding subtree needs to be updated before any profiling data can be stored. The subtree either needs to be compared against the subtree of the task which previously executed the coroutine or a separate tree structure needs to be maintained for the coroutine to store the structure of the overall coroutine execution. Such a comparison and subsequent update is a time consuming and complicated process, which would substantially increase the profiling overhead, and therefore, this solution was rejected.

Per-coroutine CCTs do require some additional work at each coroutine continue and discontinue event, also because a task does not necessarily execute the entire coroutine. Updating is required for all active routines. On a coroutine-continue event, the edge path representing the current call-path in the tree needs to be walked down from the root to the current node. While walking down the path, the data object at each edge corresponding to the task (i.e., the currently executing task thread) needs to be initialized. Initialization is required because the current task may not have executed along all the edges of the current path, so if a coroutine's routine exits, it may exit into a routine never initialized for that task. Initialization includes setting $startInclTime$ to the current time. On a coroutine-discontinue event, the edge path representing the current call-path in the tree needs to be walked up from the current node to the root. While walking up the path, the data object at each edge corresponding to the task (i.e., the currently executing task thread) needs to be finalized. Finalization is required because the current task may never actually exit all the routines along the edge path, leaving values in an incomplete state. Finalization includes increasing $totalInclTime$. When a coroutine continue or discontinue event occurs for a task (update per-task CCT versus per-coroutine CCT), such actions are taken not because the path needs to be initialized or finalized (since only the task's thread executes on the task's stack), but in order to maintain consistency across execution entities in terms of time calculations. In other words, for any execution entity, the inclusive time for a routine does not include the time spent executing another execution entity.

In the per-coroutine CCT, a linear search is required when updating an edge in order to find the data object associated with the executing task. A hash table can be used instead of a list in order to prevent linear searches. However, in general it is good programming practice to keep the number of tasks executing a particular coroutine (or accessing any shared resource) relatively small to minimize contention, which can cause a program to run slowly and/or require complex synchronization. If the number of tasks is large the linear search is at most compounding an existing inefficiency rather than creating a new one.

5.3.2 Handling Cycles

This issue involves the handling of cycles during monitoring, analysis and visualization. During monitoring, the current call-path may include an edge of the CCT multiple times (recursive invocations). An edge can be recursively invoked given cycles such as A $\rightarrow$ A $\rightarrow$ A (recursive invocation of edge E_{AA}) and A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ A $\rightarrow$ B (recursive invocation of edge E_{AB}). A variable *inRoutineCount* is used to detect recursive invocations to prevent double counting of the inclusive time at an edge. If an edge is recursively invoked, the inclusive times of all invocations after the first are contained within the inclusive time of that first invocation. *inRoutineCount* is incremented at the end of the routine-enter event and is decremented at the beginning of the routine-exit event. At the routine-enter event, $startInclTime$ is initialized to the current time only if *inRoutineCount* is zero, meaning this is not a recursive invocation so inclusive time should be



(a) Call-Graph 1 (b) Call-Graph 2

Figure 5.14: Example Cycles

counted. At the routine-exit event, *totalInclTime* is increased only if *inRoutineCount* is zero, meaning this invocation is the first in a recursive sequence (or not part of any recursion) so the inclusive time should be added to the total.

During analysis, the CCTs are analyzed using a depth-first-search. During this search, if a back-edge is detected, a cycle has been found. A cycle can be collapsed or the back-edge can be ignored. The profiler gprof first introduced the notion of collapsing cycles [GKM82]. Collapsing a cycle involves reducing the cycle to a single node or pseudo-routine. Therefore, the number of calls and time totals for all members (i.e., routines) of the cycle are summed together to arrive at totals for the single-cycle node. Unfortunately, when a cycle is collapsed, the execution behaviour of the cycle members is lost. Therefore, I chose to ignore back-edges rather than collapse cycles. To compute the total inclusive time for each routine, the inclusive time from each of the routine's callers is summed together. However, given a cycle, such as the one in Figure 5.14(a), this summation can lead to double counting. In the example, routine B has two callers: routine A and routine C via a back-edge. However, the inclusive time for routine B, when called by routine A, already includes the inclusive time for routine B when called by routine C; therefore, the inclusive times from all back-edges are ignored.

One consequence of ignoring back-edges becomes apparent during the analysis of the callers and callees of a given routine. The total descendant time for each routine is accurately computed by subtracting the total exclusive and block times for a routine from the total inclusive time for a routine (see Section 5.2.1.1). The total descendant time for a routine should also equal the sum of the descendant times of its callers as well as the sum of the exclusive, descendant and block times for its callees (see Section 5.2.2). This is the desired result for each routine, including those routines involved in a cycle. However, for a routine that starts and ends a cycle (e.g., routine B in Figure 5.14(a)), this is not the case. In the example, the descendant time from caller A and from callee C includes routine B's exclusive time when called by caller C, leading to inflated values. In this example (and other similar cycles), the exclusive time of the back-edge simply needs to be subtracted from the descendant time of the cycle's caller edge (E_{AB}) and the cycle's first callee edge (E_{BC}).

Unfortunately, such adjustments are insufficient for all cycles because back-edges are included in the CCT. The inclusion of back-edges in a CCT prevents some call-paths from being detected. In Figure 5.14(b), the call-path A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ B $\rightarrow$ D cannot be detected. Only the call-paths A $\rightarrow$ B $\rightarrow$ D and A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ B can be detected. In this example, the descendant time from both caller A and caller C includes the exclusive time for routine D when routine D is called through the call-path A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ B $\rightarrow$ D; however, this double counting cannot be adjusted given that the call-path A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ B $\rightarrow$ D is not detected.

Since adjustments for routines that start and end cycles can only be made for a limited group of cycles, I chose not to make adjustments in any case; therefore, for those routines, the total descendant time does not add up. In order to bring the issue to a user's attention on the call-graph window, the descendant time of any routine which starts and ends a cycle is highlighted in green on the routine pane (see descendant value 134u on line 13 in the routine pane of Figure 5.11).

5.4 Related Work

This section describes two current profiling tools that include exact call-graphs. gprof profiles C and C++ programs, but provides limited support for multithreading. Intel VTune is programming language and compiler independent, so it provides profiling metrics for programs, including multithreaded programs, written in C, C++, Java, Fortran and other languages.

5.4.1 gprof

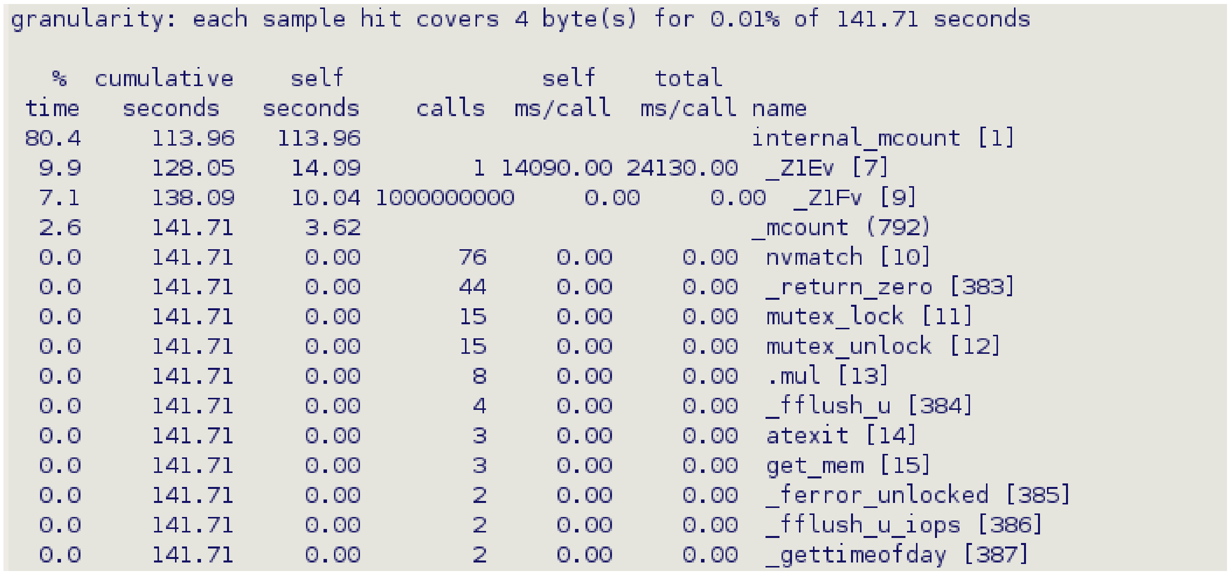
gprof uses a combination of exact and statistical call-graph profiling [GKM82]. Each call made to every routine is recorded, providing the exact call counts and the structure of the call-graph. However, the time spent in each routine is derived by sampling. Samples are taken at a sampling interval of approximately 10 milliseconds. By default, gprof combines the static and dynamic call-graphs of the executing program. The profiling data collected is later processed, upon user instruction, to produce a file containing a flat profile (lists each routine) and a call-graph profile (lists each routine with its callers and callees) of the program. No graphical user interface is available, but gprof does allow data from multiple profiling runs to be combined into one file.

Flat Profile

The flat profile lists each routine in the program in decreasing order of self time (see Figure 5.15). The information provided for each routine includes the self time of the routine as well as that number as a percentage of the total, the cumulative self time of the routine (i.e., self time of the routine plus those routines above it), the number of calls to the routine, and the per-call self and total (or inclusive) time of the routine.

Call-Graph Profile

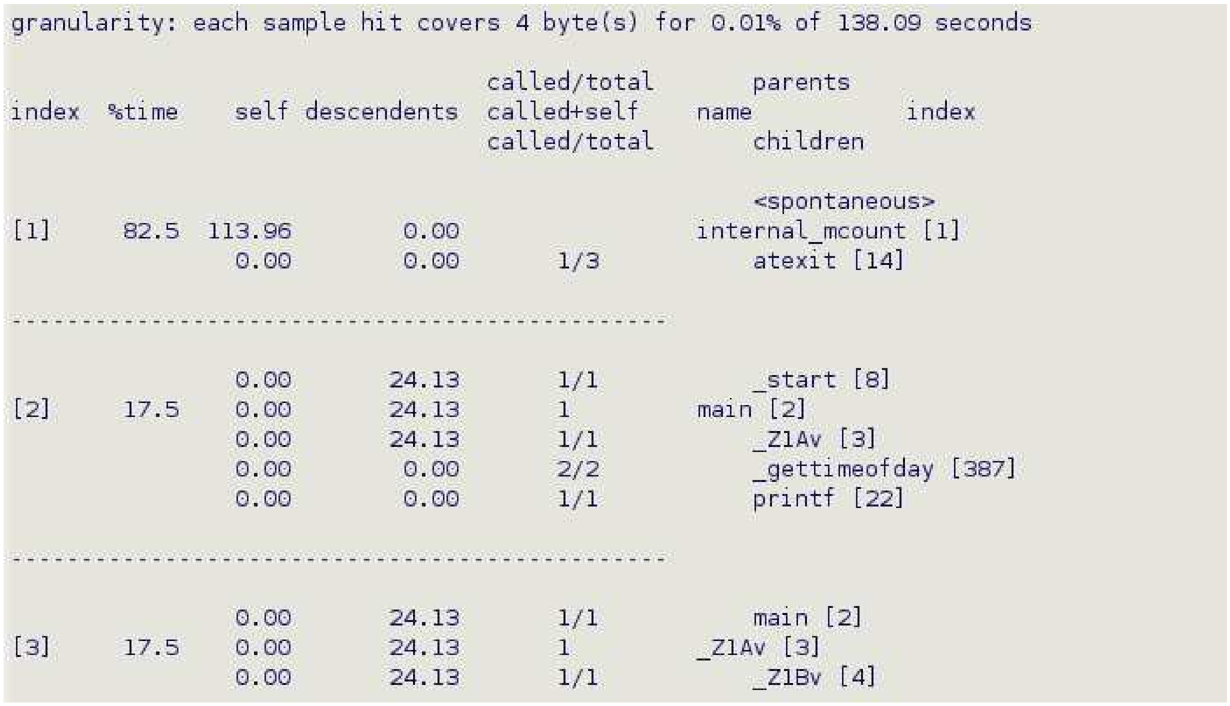
The call-graph profile lists each routine with the callers of the routine listed above and the callees of the routine listed below (see Figure 5.16). The total self and descendant times are provided for the routine as well as the total number of calls and recursive calls to the routine. For each caller of the routine, the self and descendant time attributed to the caller (from the routine) is displayed as well as the number of calls from the caller to the routine over the total number of calls to the routine. For each callee of the routine, the self and descendant time attributed from the callee (to the routine) is displayed as well as the number of calls from the routine to the callee over the total number of calls to the callee. The routines are sorted by the percentage of the total



```
granularity: each sample hit covers 4 byte(s) for 0.01% of 141.71 seconds

  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call name
 80.4     113.96   113.96                            internal_mcount [1]
  9.9     128.05    14.09        1 14090.00 24130.00  _Z1Ev [7]
  7.1     138.09    10.04 1000000000     0.00     0.00  _Z1Fv [9]
  2.6     141.71     3.62                            _mcount (792)
  0.0     141.71     0.00       76     0.00     0.00  nvmatch [10]
  0.0     141.71     0.00       44     0.00     0.00  _return_zero [383]
  0.0     141.71     0.00       15     0.00     0.00  mutex_lock [11]
  0.0     141.71     0.00       15     0.00     0.00  mutex_unlock [12]
  0.0     141.71     0.00        8     0.00     0.00  .mul [13]
  0.0     141.71     0.00        4     0.00     0.00  _fflush_u [384]
  0.0     141.71     0.00        3     0.00     0.00  atexit [14]
  0.0     141.71     0.00        3     0.00     0.00  get_mem [15]
  0.0     141.71     0.00        2     0.00     0.00  _ferror_unlocked [385]
  0.0     141.71     0.00        2     0.00     0.00  _fflush_u_iops [386]
  0.0     141.71     0.00        2     0.00     0.00  _gettimeofday [387]
```

Figure 5.15: gprof Flat Profile



```
granularity: each sample hit covers 4 byte(s) for 0.01% of 138.09 seconds

                                    called/total       parents
index  %time    self  descendents  called+self    name           index
                                    called/total       children

                                                       <spontaneous>
[1]     82.5  113.96        0.00                  internal_mcount [1]
                0.00        0.00       1/3             atexit [14]

-----------------------------------------------

                0.00       24.13       1/1             _start [8]
[2]     17.5    0.00       24.13       1          main [2]
                0.00       24.13       1/1             _Z1Av [3]
                0.00        0.00       2/2             _gettimeofday [387]
                0.00        0.00       1/1             printf [22]

-----------------------------------------------

                0.00       24.13       1/1             main [2]
[3]     17.5    0.00       24.13       1          _Z1Av [3]
                0.00       24.13       1/1             _Z1Bv [4]
```

Figure 5.16: gprof Call-Graph Profile

execution time of the program accounted for by the inclusive time of the routine. In gprof, cycles are collapsed, and therefore, reduced to single pseudo-routines.

gprof provides options for a user to limit data collection to specific routines and their callees during monitoring (i.e., instrumentation control), remove specific routines from calculations during analysis, and prevent the output of data for specific routines (including routines that exist only in the static call-graph) as well as specific edges (caller-callee routine pairs) in the produced profiles.

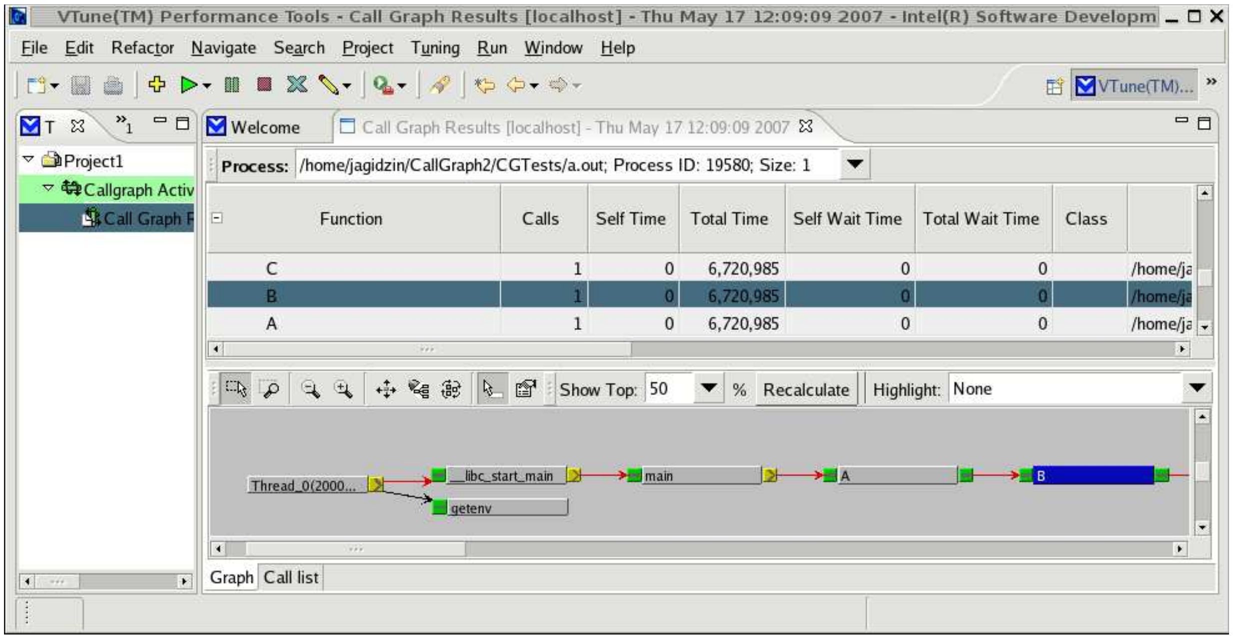
5.4.2 Intel VTune

Intel VTune helps a user maximize application performance by providing various profiling metrics including an exact call-graph metric and a system-wide performance metric (which uses sampling) [Int07]. A graphical user interface and command line version of VTune are available. Profiling data can also be saved for later visualization. Two visualizations provide the exact call-graph (dynamic) data: the Graph Tab and the Call List Tab.

Graph Tab

The Graph Tab displays a flat profile and a call-graph in tree form (see Figure 5.17). The flat profile lists the routines, which can be grouped by module, thread or class. The information provided for each routine includes the number of calls to the routine, the self, total (or inclusive) and block times of the routine, the number of callers to and callees of the routine, file information about the routine as well as various average and percentage values. A user can sort on any column and show, hide or reorder columns of data. Clicking on a routine highlights the routine in the tree and updates the Call List Tab for that routine.

The tree provides a graphical and interactive view of the program's call-graph. Each tree node represents a routine and is colour coded. The nodes are divided into colour groups according to their self time (e.g., nodes with the highest self-time are bright orange) and the colour scheme is customizable. If a user chooses to view data by thread, each thread is represented as a separate tree, otherwise each thread is represented as a separate root of a single tree where routines exe-


VTune(TM) Performance Tools - Call Graph Results [localhost] - Thu May 17 12:09:09 2007 - Intel(R) Software Developm
File Edit Refactor Navigate Search Project Tuning Run Window Help
VTune(TM)...
T
Project1
Callgraph Activ
Call Graph F
Welcome
Call Graph Results [localhost] - Thu May 17 12:09:09 2007
Show Top:
%
Recalculate
Highlight:
Thread_0(2000...
__libc_start_main
main
getenv
A
B
Graph
Call list

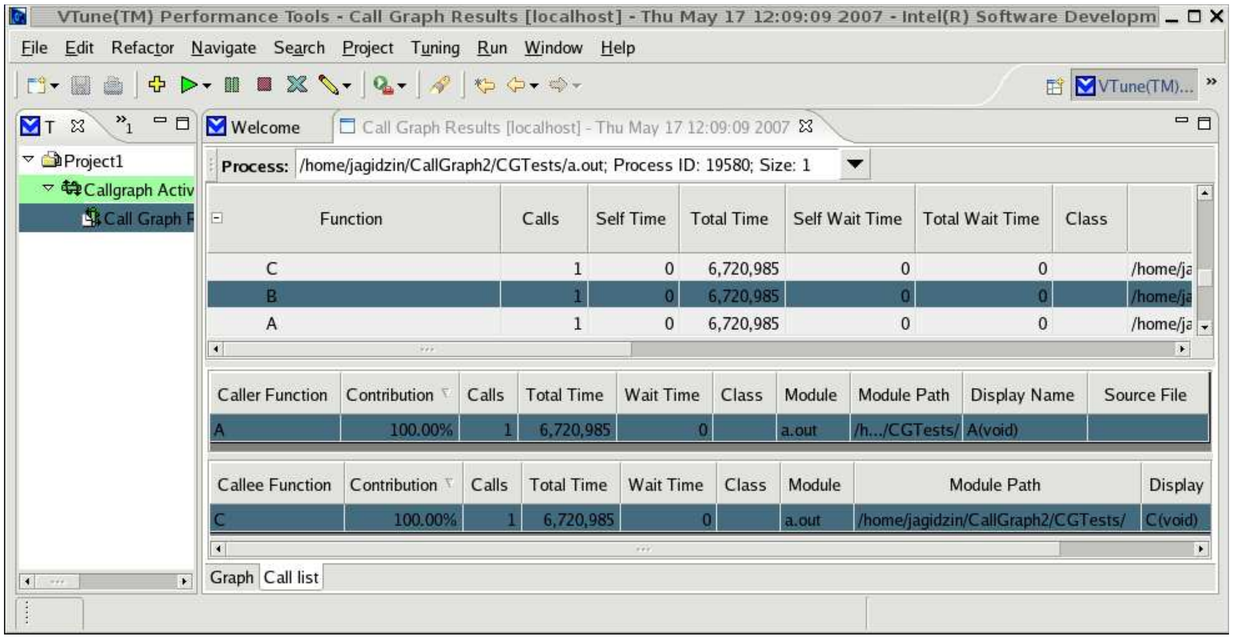
Process: /home/jagidzin/CallGraph2/CGTests/a.out; Process ID: 19580; Size: 1

Function	Calls	Self Time	Total Time	Self Wait Time	Total Wait Time	Class	
C	1	0	6,720,985	0	0		/home/ja
B	1	0	6,720,985	0	0		/home/ja
A	1	0	6,720,985	0	0		/home/ja

Show Top: 50 %

Highlight: None

Figure 5.17: Intel VTune Graph Tab


VTune(TM) Performance Tools - Call Graph Results [localhost] - Thu May 17 12:09:09 2007 - Intel(R) Software Developm
File Edit Refactor Navigate Search Project Tuning Run Window Help
VTune(TM)...
T
Project1
Callgraph Activ
Call Graph F
Welcome
Call Graph Results [localhost] - Thu May 17 12:09:09 2007
Graph
Call list

Process: /home/jagidzin/CallGraph2/CGTests/a.out; Process ID: 19580; Size: 1

Function	Calls	Self Time	Total Time	Self Wait Time	Total Wait Time	Class	
C	1	0	6,720,985	0	0		/home/ja
B	1	0	6,720,985	0	0		/home/ja
A	1	0	6,720,985	0	0		/home/ja

Caller Function	Contribution	Calls	Total Time	Wait Time	Class	Module	Module Path	Display Name	Source File
A	100.00%	1	6,720,985	0		a.out	/h.../CGTests/	A(void)	

Callee Function	Contribution	Calls	Total Time	Wait Time	Class	Module	Module Path	Display
C	100.00%	1	6,720,985	0		a.out	/home/jagidzin/CallGraph2/CGTests/	C(void)

Figure 5.18: Intel VTune Call List Tab

cuted by multiple threads are displayed as a single node with aggregated data. A user can show or hide a node as well as show or hide various callers and callees of a node. Indicator buttons on each node tell a user whether all, none or some of its callers and callees are currently displayed. Information about a particular node or edge can be obtained by hovering over the node or edge with the mouse. Red edges indicate the critical path (i.e., most time-consuming call path on the basis of self time), which is recomputed after any routine is hidden or shown. The critical path to the root or to the bottom of the tree can be displayed from any node. A user can zoom-in and out as well as highlight specific nodes and edges (e.g., top 10 nodes by self time, nodes involved in a cycle etc.). A user can also define what percentage of routines to view at any time and this percentage is used to determine which callees to display for each node. Clicking a node in the tree highlights the corresponding routine in the flat profile.

Call List Tab

The Call List Tab displays information for the caller and callee routines of the routine currently selected in the flat profile on the Graph Tab (see Figure 5.18). For each caller of the routine, the percentage of the total time of the routine attributed to the caller is displayed as well as the total and wait time attributed to the caller and the number of calls from the caller to the routine. For each callee of the routine, the percentage of the total time of the routine attributed from the callee is displayed as well as the total and wait time attributed from the callee and the number of calls from the routine to the callee. The data displayed on the Call List Tab can be further broken down by call site. Therefore, instead of listing only each caller and callee routine, the list displays each call site in the individual caller and callee routines. Also, by right-clicking on a routine and selecting “View Source”, a user can see the source code line of the call site. VTune does not collapse cycles, and furthermore, does not make adjustments in order to prevent double counting.

VTune provides numerous profiling options. A user can pause and resume data collection for the running program (but not perform real-time analysis) using control buttons or an application program interface (API). Also, VTune allows a user to select the level of instrumentation (i.e., for which routines profiling data is collected). A user can choose to instrument only specific routines as well as all routines in a particular module.

5.4.3 Comparison

Table 5.1 summarizes and compares the relevant features of μProfiler's **ECG** metric and the two profiling tools discussed in the previous sections. Some of the important features are discussed in detail.

	μProfiler ECG Metric	gprof	Intel VTune
Hardware Events	$\surd$		
Combine Profiling Runs		$\surd$	
Instrumentation Control	$\surd$	$\surd$	$\surd$
No gprof Fallacy	$\surd$		$\surd$
Data Saved to File		$\surd$	$\surd$
Graphical User Interface	$\surd$		$\surd$
Interactive Caller-Callee Display	$\surd$		$\surd$
Interactive Tree with Critical Path			$\surd$
Cycles Information	$\surd$	$\surd$	(via tree)
Complete Call-Graph	$\surd$	$\surd$	(via tree)
Call-Graph Break Down	$\surd$		$\surd$
Sorting			$\surd$
Source Information	$\surd$	$\surd$	$\surd$
Callees Visited List	$\surd$		
Histograms of %s	$\surd$		

Table 5.1: Exact Routine Call-Graph: Comparison of Related Profilers

The gprof fallacy is an assumption made by a profiler that the time spent in a routine is independent of the routine's caller (i.e., execution time is always the same). However, this assumption is often incorrect as the amount of time spent in a routine can depend on which routine calls it, and making such an assumption can lead to misleading results. gprof, unlike the other profiling tools, suffers from this fallacy because it estimates the amount of time spent in a routine when called by a particular caller from the number of calls to the routine by that caller, regardless of the time actually spent in the routine [GKM82]. Neither of the exact call-graph metrics of μProfiler or Intel VTune suffer from the gprof fallacy, and hence, provide a user with more accurate call-graph information.

Although the **ECG** does not allow a user to combine data from multiple profiling runs as gprof does, the **ECG** can collect profiling data for multiple events (e.g., time and hardware events) during one run. gprof and VTune only provide the time event. Also, in the **ECG**, a user can view and compare the data from the multiple events on a single display (complete call-graph window). Multiple events allow a user to view the call-graph information from several different perspectives. Furthermore, combining data from multiple profiling runs is not always appropriate in a concurrent (or sequential) system as each run of the program can produce a very different pattern of execution.

Unlike gprof, the **ECG** and VTune have a graphical user interface that provides a user with an interactive environment. VTune also includes an interactive tree which is used to present the complete call-graph, call cycles, etc. to a user. The **ECG** does not include such a tree, but does instead provide a list of call cycles and a complete call-graph window. Also, by providing a callees-visited list, the **ECG** allows a user to keep track of the path in the call-graph. Although the two forms of visualization are different, they can be equally valuable. A tree does provide a user with a quick visual representation of the call-graph; however, it is often easier to analyze a call-graph presented in table form (e.g., complete call-graph window in Figure 5.13). In table form, the data is visible in its entirety and a user does not need to highlight or select individual tree nodes or edges in order to view the associated data.

Although all three profilers provide instrumentation control, μProfiler only provides control at the module and task level. gprof and VTune allow a user to enable or disable instrumentation at the routine level, allowing for more precise control of the data collection.

Overall, μProfiler's **ECG** metric provides many important features, and furthermore, includes features unavailable in the other profiling tools. Some features not currently provided in μProfiler, such as routine-level instrumentation control, saving data to file, sorting and critical path display, are possible enhancements for future work.

5.5 Performance

This section describes the performance of the **ECG** with respect to both time and space.

5.5.1 Time

To evaluate the running time of the **ECG**, I constructed a worst-case test program (see Appendix B.1), profiled the program with the **ECG** metric, and compared its running time to the same test program run without profiling. The program was also run with gprof and Intel VTune for a cross-profiler comparison. Only the running time of the test program itself was measured, i.e., the time includes monitoring and data collection but not time spent during analysis or visualization.

The test program produces a call-graph of depth 8 (routines A through H). Starting at routine A, each routine calls 6 subsequent routines. Therefore, routine A calls routines B1, B2, B3, B4, B5 and B6, each routine B1 through B6 calls routines C1, C2, C3, C4, C5 and C6, etc. This calling process continues until routines G1 through G6 are called. Routines G1 through G6 simply call routine H1 30,000 times. The test program is a worst-case scenario because it does no work, other than make routine calls. Also, the depth and breadth of the call-graph is large.

The test program was compiled with optimization (i.e., O2 flag). Table 5.2 shows the results of the performance testing in milliseconds. The percentage increase is with respect to the no profiling case.

No Profiler	μProfiler		gprof		Intel VTune	
Time	Time	% Increase	Time	% Increase	Time	% Increase
9349.69	2023384.23	21541.19	83387.16	791.87	1811403.12	19273.94

Table 5.2: Exact Routine Call-Graph: Time Performance Results

As seen in the table, the percentage increase over the no profiling case for the **ECG** and the other profilers is large. Such an increase is to be expected for this worst-case program because every routine call in the program is instrumented (i.e., profiling data is collected at each routine call). Furthermore, in the **ECG**, each routine exit, task block, task unblock, etc. is also instrumented. The percentage increase for gprof is considerably smaller because gprof is not a completely exact call-graph profiler (see Section 5.4.1); therefore, much of the work is done at a less frequent sampling interval. As well, gprof instruments only routine enter and not routine exit. The percentage increase for the **ECG** compared to Intel VTune is slightly larger (difference of 2267.25%).

5.5.2 Space

To examine the total space cost of a CCT, the space costs of the individual data structures (see Section 5.2.1) are examined. The data structures include the general tree object, the node object, the edge object and the data object. The general tree object maintains a pointer to the root node of the tree, an array (of size 240) of edge object pointers representing the edge path and an integer storing the position of the edge (in the path) associated with the current state of execution. Therefore, in a standard 32 bit system with 4 byte pointers and 4 byte integers, each tree object requires 968 bytes of space (i.e., $(4\ bytes \times 240) + 8\ bytes$).

Each node object consists of a routine address (a pointer), a pointer to its caller node in the tree and a queue of pointers to its callee edge objects. A queue requires 8 bytes of space; and therefore, each node object requires 16 bytes of space. Each edge object consists of a pointer to link the edge object in a node's queue, a pointer to its callee node in the tree, a boolean specifying whether or not the edge is a back-edge and a queue of pointers to its data objects. Therefore, given a 1 byte boolean (with 3 bytes of padding to maintain proper alignment), each edge object requires 20 bytes of space. Each data object consists of a pointer to link the data object in an edge's queue, a pointer to the task associated with the data and the variables storing the number of calls, number of continues, times and hardware-event counts. The variables include 4 integers, 6 long integers and 4 arrays of long integers (with one array entry for each hardware counter). Therefore, given 8 byte long integers, the space cost of each data object is computed according to the following formula

$$Total\ space\ per\text{-}data\text{-}object = 72\ bytes + 4 \times (number\ of\ hardware\ counters \times 8\ bytes) \quad (5.3)$$

The test program used to evaluate the running time of the **ECG** creates one tree (single task program) with 102,644 node objects, 102,643 edge objects and 102,643 data objects. Assuming the test program is run on a processor with 4 hardware counters, the total space cost for the tree is

$$\begin{aligned} Total\ space\ cost &= 968\ bytes + (102644 \times 16\ bytes) + (102643 \times 20\ bytes) \\ &+ (102643 \times 200\ bytes) = 24,224,732\ bytes \end{aligned} \tag{5.4}$$

The space cost of 23.10 MB is reasonable given the depth and breadth of the call-graph is significantly large, i.e., large number of node, edge and data objects in the tree.

5.6 Summary

The advances made to μProfiler's **ECG** metric have achieved the goals stated at the beginning of the chapter. Firstly, based on functionality and the comparison to related work, μProfiler's **ECG** metric is similar to state-of-the-art vendor exact call-graph metrics. Secondly, the run-time of the **ECG** metric compared to the related work shows similar performance and reasonable space costs. Thirdly, the **ECG** metric, through the careful collection, separation, analysis and visualization of profiling data, scales to programs of long duration and complex behaviour. Finally, I believe the collective achievement of the goals results in a call-graph metric providing an environment conducive to more thorough and simpler user analysis.

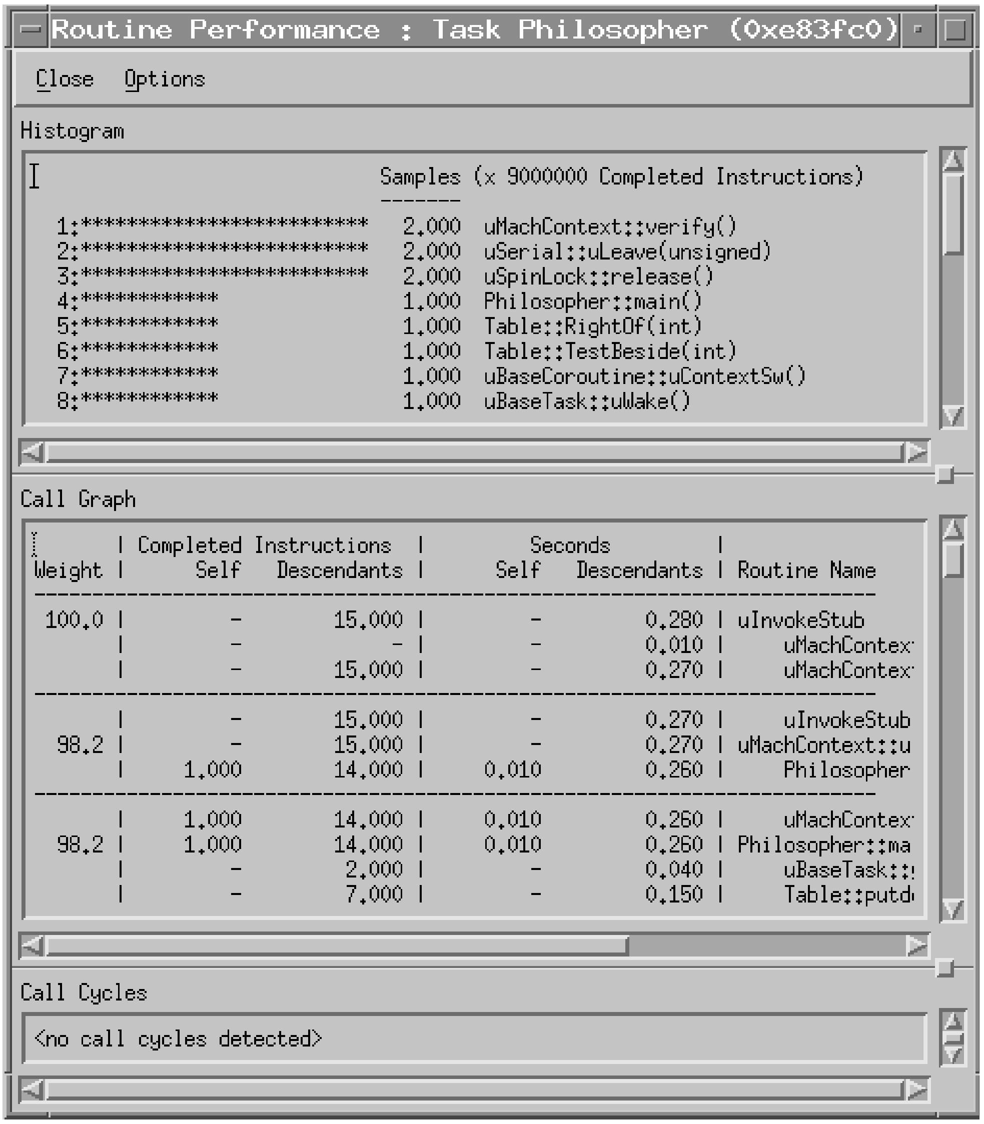
Chapter 6

Statistical Call-Graph

This chapter describes the advances made in μProfiler's Statistical Routine Call-Graph (**SCG**) metric.

The **SCG** generates a statistical profile of a μC++ program's dynamic execution. As in the **ECG**, the profile provides the dynamic calling relationship among routines in the program and gives a user some indication about the program's control flow. The statistical nature of the call-graph metric implies that profiling data is collected (or a sample is taken) only at specific intervals, called a sampling interval; therefore, the **SCG** has a lower overhead (in both time and space) at the cost of less accurate information. A sampling event can be time or a hardware event (e.g., completed instructions), and a user can choose a custom sampling interval for each sampling event (e.g., sample every 10ms and every 10,000 completed instructions). The overall structure of the call-graph is the same as described for the **ECG**, but the data is collected in terms of samples taken versus exact time or exact hardware-event counts.

The initial μProfiler **SCG** implementation allowed a user to display a call-graph for each task's execution. An example display of the initial implementation is presented in Figure 6.1. The information provided in the display includes a histogram showing the distribution of samples across the routines executed by the task, the complete call-graph for the task (similar to the complete call-graph window in the **ECG**) and a list of call cycles [Les05]. The complete call-graph shows the number of samples taken while executing the routine itself (self or exclu-



```
                             Samples (x 9000000 Completed Instructions)
                             -------
  1:*************************  2,000  uMachContext::verify()
  2:*************************  2,000  uSerial::uLeave(unsigned)
  3:*************************  2,000  uSpinLock::release()
  4:************               1,000  Philosopher::main()
  5:************               1,000  Table::RightOf(int)
  6:************               1,000  Table::TestBeside(int)
  7:************               1,000  uBaseCoroutine::uContextSw()
  8:************               1,000  uBaseTask::uWake()
```

```
       | Completed Instructions  |          Seconds         |
Weight |       Self  Descendants |       Self  Descendants | Routine Name
-------------------------------------------------------------------------
 100.0 |          -       15,000 |          -        0.280 | uInvokeStub
       |          -            - |          -        0.010 |     uMachContex
       |          -       15,000 |          -        0.270 |     uMachContex
-------------------------------------------------------------------------
       |          -       15,000 |          -        0.270 |     uInvokeStub
  98.2 |          -       15,000 |          -        0.270 | uMachContext::u
       |      1,000       14,000 |      0.010        0.260 |     Philosopher
-------------------------------------------------------------------------
       |      1,000       14,000 |      0.010        0.260 |     uMachContex
  98.2 |      1,000       14,000 |      0.010        0.260 | Philosopher::ma
       |          -        2,000 |          -        0.040 |     uBaseTask::
       |          -        7,000 |          -        0.150 |     Table::putd
```

```
<no call cycles detected>
```

Figure 6.1: Initial Implementation: Statistical Call-Graph Display

sive samples) and the number of samples taken while executing the descendants of the routine (descendant samples). This information is displayed for each routine (totalled over all calls to the routine) as well as for each of its caller routines (totalled over all calls to the routine by the caller) and callee routines (totalled over all calls to the callee by the routine). Whereas the complete call-graph displays information for all hardware events selected (as sampling events), the histogram only displays samples for one hardware event currently selected using the "Options" menu. When a user chooses time as a sampling event, a display virtually identical to Figure 6.1 is displayed; however, time and hardware sampling events cannot be simultaneously selected for one profiling run.

The overall goal of the advances discussed in this chapter, as in the **ECG**, is to develop μProfiler's **SCG** metric into a state-of-the-art metric with good performance that scales to programs of long duration and complex behaviour, providing an environment conducive to more thorough yet simpler user analysis of a call-graph.

6.1 Initial Implementation Issues

This section describes several issues arising in the initial implementation of the **SCG**. I addressed each issue in the advanced implementation of the **SCG** and the solutions are discussed in Section 6.2.3. The issues are very similar to those of the **ECG**.

The first issue involves the **SCG** existing as two identical yet mutually exclusive metrics. In the initial implementation, one run of the profiler can collect and display data for the time event, but another run of the profiler is required to collect and display data for the hardware events. Having to run the profiler multiple times is an unnecessary complication for a user. By not combining the information into one display, a user must switch between two windows and cannot easily compare the data. Furthermore, given a concurrent program and the statistical nature of the metric, data from different runs can often not be compared in detail.

The second issue involves the lack of separation between tasks and coroutines. This issue is identical to the one described for the **ECG** in Section 5.1.

The third issue involves the simplicity of the visualization. The display provided by the initial implementation is again very simple in nature, providing almost all the information at once with little opportunity for a user to interact with the call-graph data. One purpose of the call-graph is to give a user some indication about the program's control flow; however, as in the **ECG**, such information can be made clearer if a user can in some way progress through and view the call-graph step-by-step. Also, the display is inconsistent as some information (e.g., complete call-graph) is displayed for all events selected while other information (e.g., histogram) is displayed for only one event.

The final issue involves the unnecessary differences between the implementations of the **ECG** and **SCG** in terms of the visualization of profiling data and the data structures used to store profiling data. Unnecessary differences lead to maintainability issues for μProfiler. Also, differences in visualization lead to increased learning time for a user.

6.2 Advanced Implementation

While addressing the issues from the initial implementation of the **SCG**, the advanced implementation has progressed in two major areas: the data structures used to store the profiling data collected during monitoring and the visualization of the collected data.

6.2.1 Data Collection

When a sample is taken at each sampling interval, the profiling data collected is stored in specific data structures.

In the initial implementation of the **SCG**, the data structure consists of a list of sample objects. One list is maintained for each processor, where a processor corresponds to a kernel thread. Each sample object maintains all the information for one sample. The information collected and stored in each sample object includes an array of routine addresses (program counters) representing the call-stack at the time of the sample, a bitmask indicating the event triggering the sample and a pointer to the task executing at the time of the sample. Since sample objects for all tasks executing on a particular processor are stored in one list, before any analysis can take place, the list entries must be separated by task.

The advanced implementation of the **SCG** replaces the list with a calling context tree (CCT) [FFMC03]. The CCT stores all call-stacks (routine addresses representing the entries in the call-stacks) taken during sampling. Here, the CCT is referred to as an approximate CCT because it is only being approximated through sampling [AS00]. In other words, since a full call-stack is stored at each sample (versus just the executing caller-callee routine pair), the call-graph represented by the CCT is connected, but not necessarily complete as sampling may not cover all routines executed by a task.

In the advanced implementation, a CCT is a collection of node objects rooted at a single node, where each node represents one call-stack entry (i.e., one routine address). Each node maintains a pointer to its caller node in the tree, except the root, and a list of callee nodes. The profiling data collected during monitoring is stored within the node objects. Figure 6.2 illustrates the specific data structures composing the approximate CCT. The data objects are discussed in Section 6.2.1.1.

Unlike the CCT for the **ECG**, the approximate CCT does not include back-edges. Adding a back-edge requires the same process as described for the **ECG** (i.e., searching ancestor nodes), but the process must be executed for each entry in the call-stack as it is being added to the CCT, considerably increasing the time required at each sample. Because the **SCG** is a statistical metric there is an expectation that the profiling overhead is relatively low; therefore, the increase in overhead related to back-edges is a major disadvantage. As a consequence of not including back-edges, the profiling data can be stored within the node objects, and hence, no edge objects are required (see Section 5.2.1 p. 71 for a comparison to the **ECG** where the CCT stores data at the edges). Although not including back-edges is opposite to the goal of making the SCG and ECG data structures consistent, in the case of a statistical metric the reduction in overhead is of primary importance to reduce the probe effect. If deemed necessary, back-edges can be added to the CCT in the future.

Another consequence of not including back-edges relates to the size of the CCT. No back-edges means the depth of the CCT is bounded by the length of the longest call-stack observed during program execution which, given recursion, can be much greater than the number of rou-

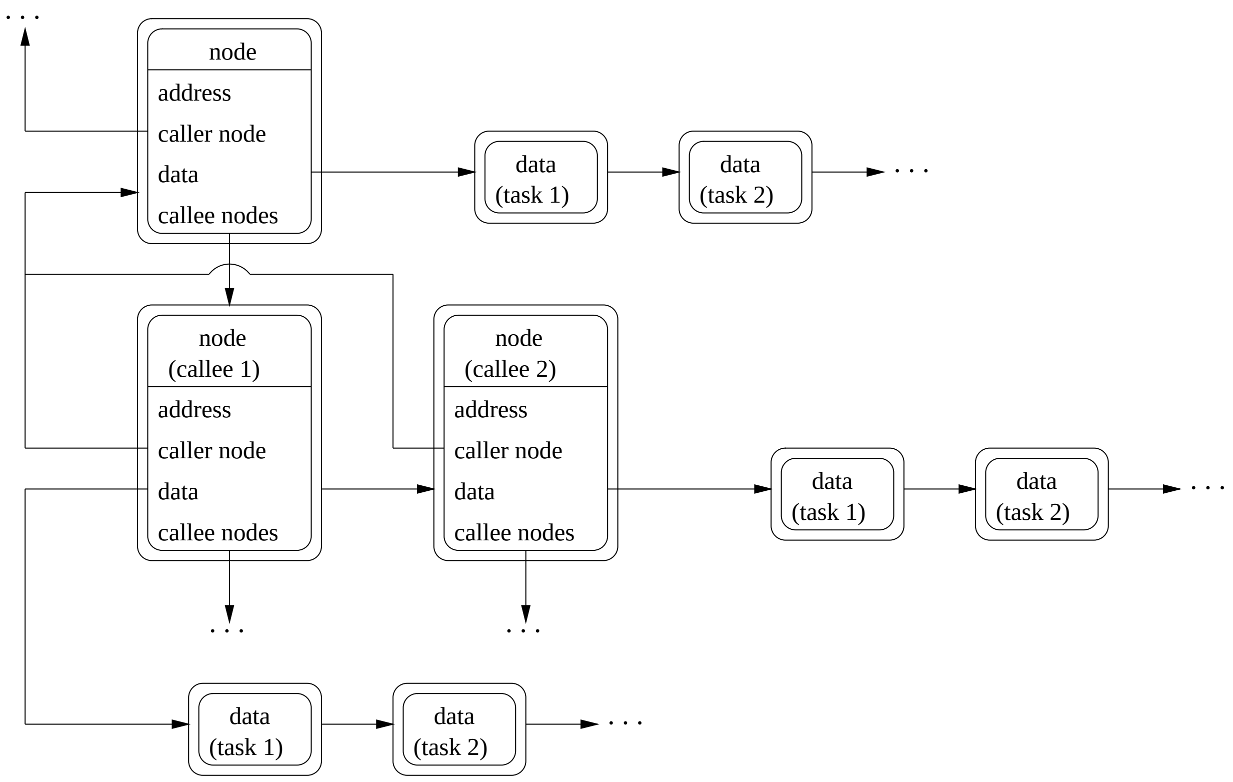


Figure 6.2: Advanced Implementation: Statistical CCT Data Structures

tines executed in the program. Let d be the depth of the CCT and n be the number of routines executed in the program. For each node in the CCT, its number of callee nodes is at most n because each callee represents a unique routine called from the node. Thus, at each depth in the tree, there can be at most n nodes for each node at the previous depth in the tree. Since the tree starts with one node at the root and has a depth of d, the maximum number of nodes at the lowest depth of the tree is n^{d-1}. In other words, the breadth of the CCT is bounded by n^{d-1}.

6.2.1.1 Creating and Updating a CCT

At each sample, the call-stack at the time of the sample is inserted into the CCT. Insertion starts at the root of the CCT and moves downward with the last node object added representing the address of the routine executing at the time of the sample. However, as call-stacks are inserted into the CCT, subsequent insertions may require the addition of fewer or no node objects due to existing entries.

Inserting a call-stack into the CCT requires three steps:

1. The longest prefix of the call-stack already in the CCT is found. The first address on the call-stack (i.e., root entry of the call-stack) is compared to the addresses represented by the callee nodes of the root of the CCT. The root itself represents a null address to allow for the possibility of call-stacks rooted at different addresses. Once a node with an identical address is found, the second address on the call-stack is compared to the addresses represented by the callee nodes of that node. This process continues until, at a particular node along the path, no address identical to the current call-stack address is found among its callees or the end of the call-stack is reached (indicating the call-stack already exists in the CCT). The node at which the process terminates represents the longest prefix.

2. If the longest prefix found in the first step is not the entire call-stack, the remaining portion of the call-stack (suffix) is inserted into the tree starting at that terminating node. One new node is created for each address in the suffix of the call-stack and becomes a callee of the preceding node (i.e., added to the preceding node's list of callee nodes).

3. The last node found or added is updated to indicate that a call-stack terminating at the address represented by this node was sampled (i.e., address of the routine executing at the time of the sample). The update involves incrementing a counter associated with the sampling event triggering the sample. An array of counters is maintained with one entry for the time event and one entry for each of the hardware counters used to count the hardware events.

A CCT is maintained for each execution entity having its own stack; i.e., one for each task and each coroutine. As for the **ECG**, multiple tasks may execute a single coroutine (i.e., multiple tasks' threads may execute on the coroutine's stack), so the CCT must keep profiling data separated by task. To handle this issue, each node object maintains a list of profiling data objects,

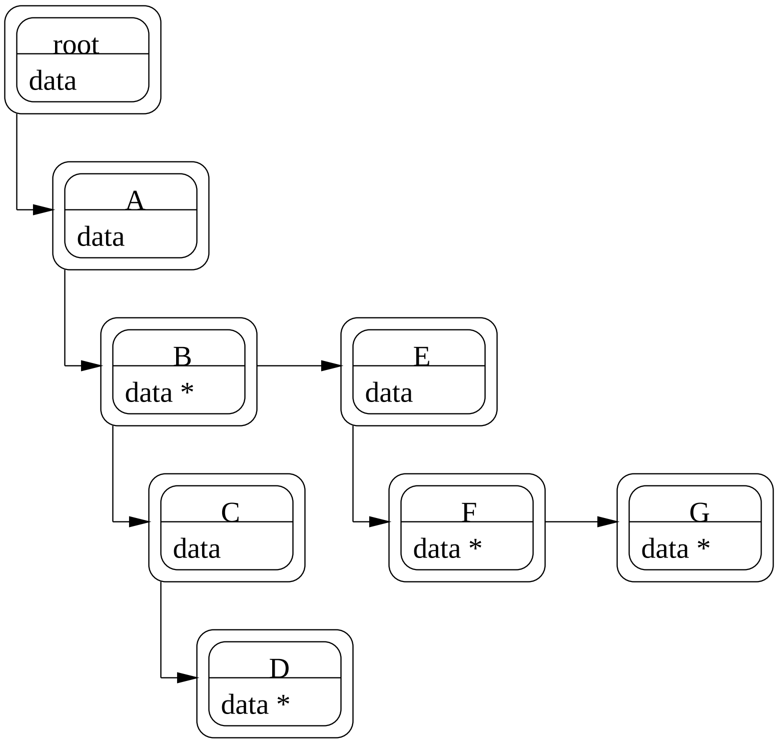


Figure 6.3: Advanced Implementation: CCT Call-Stack Paths

rather than a single set of data (see Figure 6.2). Each data object consists of the array of counters described above. For each per-task CCT, the lists consist of at most one element because only the task's thread may execute on the task's stack.

Every call-path (i.e., path from the root to any node) in the CCT represents a call-stack; however, every call-path does not represent a call-stack sampled during profiling. Each path of nodes in the CCT, starting at the root and ending at a node with one or more data objects, represents a call-stack sampled at least once during profiling. Therefore, an empty list at a node means that a call-stack terminating at the address represented by this node was never sampled. In the example CCT in Figure 6.3 (a simplified representation of a CCT), each node marked with an "*" has a list with one or more data objects. Therefore, the paths in the CCT that represent sampled call-stacks are: A $\rightarrow$ B, A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ D, A $\rightarrow$ E $\rightarrow$ F and A $\rightarrow$ E $\rightarrow$ G.

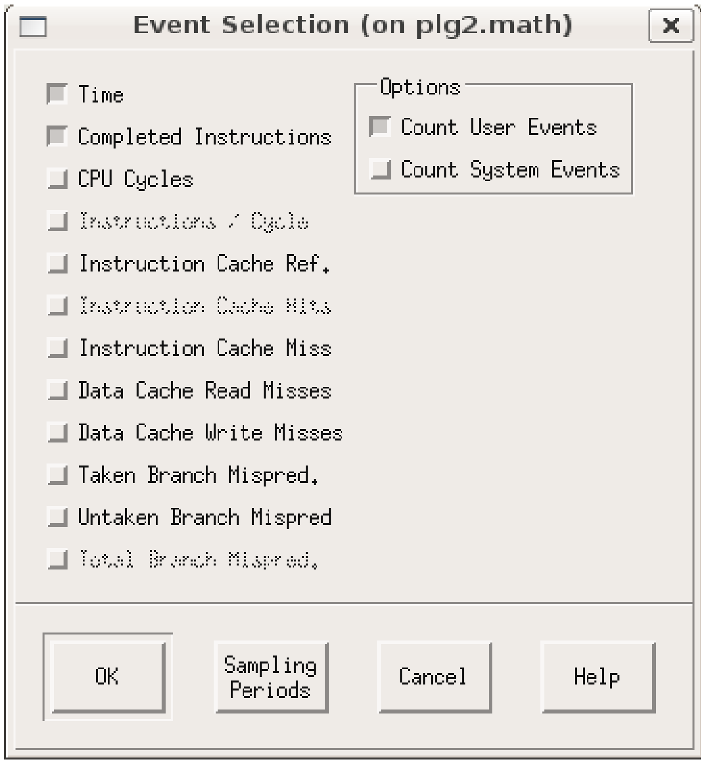
The CCT is a more space efficient data structure than the list in the initial implementation. Whereas the list stores every call-stack in its entirety (one for each sample), the CCT stores the common prefixes of the various call-stacks only once with counters indicating the number of times and for which sampling events the call-stack is sampled. The simultaneous occurrence of multiple sampling events is detected. As a consequence, in the CCT, profiling data for duplicate call-stacks is aggregated during monitoring, significantly reducing the number of call-stacks to be analyzed, and hence, the time needed for analysis. With the list, analysis is required for every call-stack including duplicate call-stacks. However, space efficiency and reduced analysis time are achieved at the cost of increased monitoring time. The time required to insert a call-stack into the CCT (as previously described) is greater than the time required to add a call-stack to the end of a list. Also, storing call-stacks in a list preserves the temporal ordering of the samples. The temporal information can be useful during analysis; however, this information was not used in the initial implementation.

6.2.2 Visualization

The first step in running the **SCG** metric involves selecting the sampling events (i.e., time and/or hardware events) for which samples are taken. The event-selection window, similar to that of the **ECG**, is shown in Figure 6.4 with the "Time" and "Completed Instructions" events selected.

By clicking the "Sampling Periods" button at the bottom, a user can choose a custom sampling interval for each selected event. For example, in Figure 6.5 a user has specified samples are taken every 10 ms. and every 10,000 completed instructions. For the time event, the interval can range from a minimum value dependent on the operating system's clock resolution to 1000 ms., and for the hardware events, the interval must be greater than zero, although very small intervals may cause problems because of the high number of interrupts (i.e., signals specifying a sample is to be taken). If a user enters an invalid interval the textbox is highlighted in red, signalling an error.

Once program execution and monitoring are completed, the task/coroutine-selection window is displayed (see Figure 6.6). A user can select the call-graph for any task or coroutine by clicking



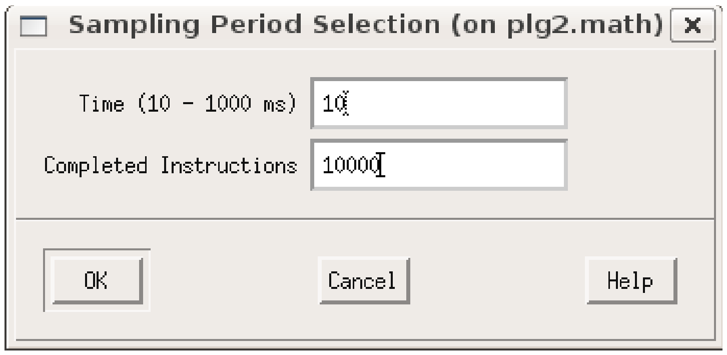
[✓ Time]
[✓ Completed Instructions]
CPU Cycles
Instructions / Cycle
Instruction Cache Ref.
Instruction Cache Hits
Instruction Cache Miss
Data Cache Read Misses
Data Cache Write Misses
Taken Branch Mispred.
Untaken Branch Mispred
Total Branch Mispred.

Options
[✓ Count User Events]
Count System Events

Figure 6.4: Advanced Implementation: Statistical Event-Selection Window

on a task or coroutine name in the left column. The tasks are listed above the dashed separator line and the coroutines below it. The total number of samples taken, totalled over all sampling events, is displayed for each task and coroutine. The purpose of this information is to give a user some direction as to which call-graph to analyze first. For example, a task with a large number of samples is a potential "hot-spot" of execution.

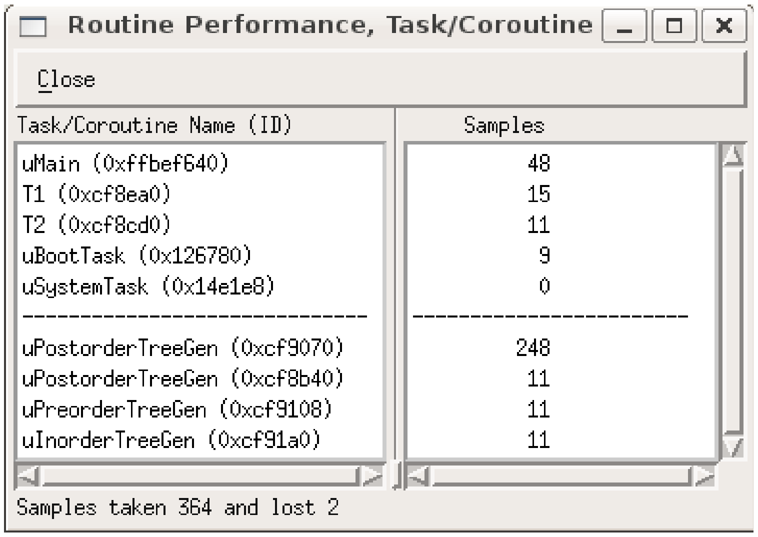
The call-graph window in Figure 6.7 is displayed after selecting the task T1 on the task/coroutine-selection window. All data displayed in the call-graph window is for a single selected event (i.e., time or a hardware event). The call-graph, in Figure 6.7, is currently displaying the completed-instructions data for task T1. The call-graph window contains several panes (from top to bottom): routine pane, callers pane, callees pane, callees-visited pane, cycles pane and coroutine-selection pane. The panes and the overall functionality of the window is



```
Sampling Period Selection (on plg2.math)

Time (10 - 1000 ms)       10
Completed Instructions    10000
```

Figure 6.5: Advanced Implementation: Sampling-Interval-Selection Window



```
Routine Performance, Task/Coroutine
Close
Task/Coroutine Name (ID)            Samples
uMain (0xffbef640)                    48
T1 (0xcf8ea0)                         15
T2 (0xcf8cd0)                         11
uBootTask (0x126780)                   9
uSystemTask (0x14e1e8)                 0
------------------------------  ------------------------
uPostorderTreeGen (0xcf9070)         248
uPostorderTreeGen (0xcf8b40)          11
uPreorderTreeGen (0xcf9108)           11
uInorderTreeGen (0xcf91a0)            11
Samples taken 364 and lost 2
```

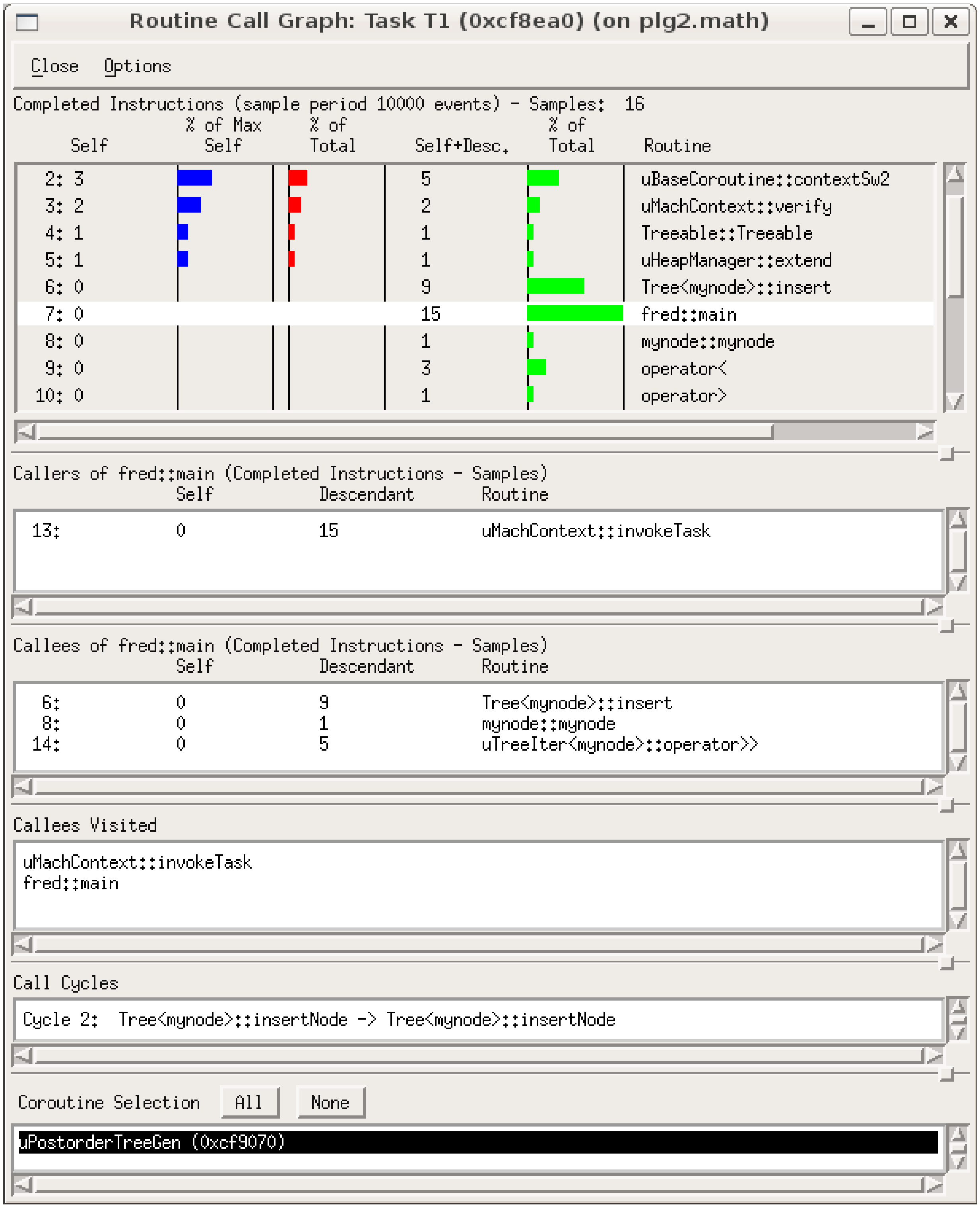
Figure 6.6: Advanced Implementation: Statistical Task/Coroutine-Selection Window

nearly identical to that of the **ECG** described in Section 5.2.2; therefore, only the differences are examined in this section.

The main difference is that the data displayed in the routine pane (as well as the callers and callees panes) is presented in terms of samples taken versus exact time or exact hardware-event counts. For each routine, the information includes (left to right) the number of self samples for the routine as well as a histogram showing that number as a percentage of the total and a percentage of the maximum (for any routine), and the number of self plus descendant samples for the routine as well as a histogram showing that number as a percentage of the total. All values displayed for a specific routine are totalled over all calls to the routine. The routines are sorted by the number of self samples, and as in the **ECG**, all individual values are scaled.

The callers pane lists, for each caller, the samples attributed to the caller by the selected routine (highlighted in the routine pane). Therefore, the sum of the samples of the callers for each individual field (self and descendant) is equal to the total value displayed for the selected routine on the routine pane. The callees pane lists, for each callee, the samples attributed to the selected routine by the callee. Therefore, the sum of the samples of the callees for all fields (self and descendant) is equal to the total number of descendant samples displayed for the selected routine on the routine pane.

There is also one additional option called "Format" available from the pull-down menu associated with the "Options" button (see Figure 6.8). This option allows a user to choose the format to display the data in the call-graph window. The formats available are uninterpreted samples and samples interpreted by the sampling period/interval. The uninterpreted samples format, currently chosen for the call-graph window in Figure 6.7, simply displays all the data as the number of samples. If the samples interpreted by the sampling period/interval format is chosen, the number of samples is multiplied by the sampling interval, changing the units of the data to the number of events executed (for hardware events) or the time spent (for the time event). Therefore, in Figure 6.7, the fields displaying 1 sample, for example, would instead display 10,000 completed instructions (i.e., $1 \times 10,000$) in the alternate format (and scaled to 10k). If the call-graph window is displaying data for the CPU cycles event, then there is one additional format, called CPU cycles time, available. When this format is chosen the following calculation



```
Completed Instructions (sample period 10000 events) - Samples:  16
                    % of Max     % of              % of
      Self            Self       Total   Self+Desc.  Total    Routine
   2: 3                                       5               uBaseCoroutine::contextSw2
   3: 2                                       2               uMachContext::verify
   4: 1                                       1               Treeable::Treeable
   5: 1                                       1               uHeapManager::extend
   6: 0                                       9               Tree<mynode>::insert
   7: 0                                      15               fred::main
   8: 0                                       1               mynode::mynode
   9: 0                                       3               operator<
  10: 0                                       1               operator>

Callers of fred::main (Completed Instructions - Samples)
                 Self           Descendant         Routine
  13:            0              15                 uMachContext::invokeTask

Callees of fred::main (Completed Instructions - Samples)
                 Self           Descendant         Routine
   6:            0              9                  Tree<mynode>::insert
   8:            0              1                  mynode::mynode
  14:            0              5                  uTreeIter<mynode>::operator>>

Callees Visited
 uMachContext::invokeTask
 fred::main

Call Cycles
 Cycle 2:  Tree<mynode>::insertNode -> Tree<mynode>::insertNode

Coroutine Selection   All   None
 uPostorderTreeGen (0xcf9070)
```

Figure 6.7: Advanced Implementation: Statistical Call-Graph Window

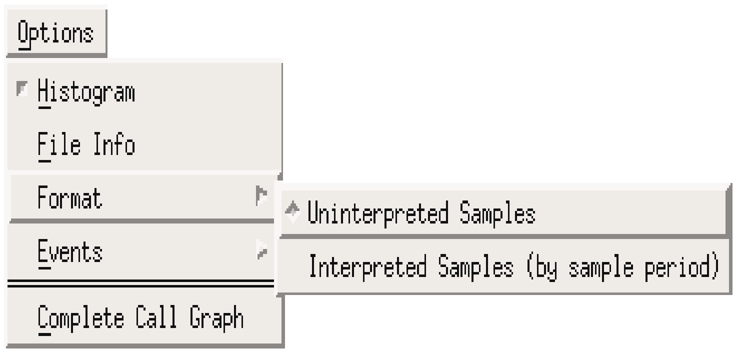


Figure 6.8: Advanced Implementation: Statistical Options Menu

is executed to convert the number of samples into a corresponding time value

$$time\ value = (number\ of\ samples \times sampling\ interval)\ /\ processor\ speed \tag{6.1}$$

where the *processor speed* is the number of cycles executed per-second. This calculation relies on the assumption that the CPU cycle rate is constant. While this is true for many microprocessors, some do vary the cycle rate (e.g., to conserve power). If the program is sampled during an interval of variable clock speed, the time values calculated are inaccurate.

Finally, there is also a complete call-graph window available similar to that of the **ECG**. A user can show or hide data for individual events, change the format of the data for individual events, etc. The routines displayed in the complete call-graph window are sorted by the value of the “Weight” column. For each event, a routine’s number of self samples as a percentage of the total for the event can be computed. A routine’s weight is the average percentage over the events currently displayed in the window.

6.2.3 Addressing Initial Issues

In the advanced implementation, to address the existence of the initial **SCG** as two mutually exclusive metrics, one run of the profiler can now collect data for both the time and hardware events, providing more comparable data. After profiling is completed and a task or coroutine is selected, a single call-graph window (see Figure 6.7) is displayed and a user can, via the pull-down menu associated with the "Events" option, choose the specific event to display. Such functionality decreases the number of windows a user needs to manage and allows a user to easily compare all the available data via the complete call-graph window.

The issue of the lack of separation between tasks and coroutines in the initial implementation is addressed identically to that of the **ECG** in Section 5.2.3. Again, the improvements all stem from the careful separation of the task and coroutine profiling data during monitoring (see Section 6.2.1.1), allowing for analysis on a per-task and per-coroutine basis.

To address the simplicity of the visualization, the call-graph window was completely redesigned in the same fashion as the call-graph window of the **ECG** (see Section 5.2.3). The call-graph window is now interactive, conducive to thorough analysis and understanding, and consistent.

The changes made to data collection and visualization in the advanced implementations of the **ECG** and **SCG** have resulted in greater consistency between the metrics. In the initial implementations, the data structures used were very different (a hash table versus a list), but now both metrics use a CCT, although slightly different forms. The consistency in visualization decreases the amount of time a user needs to become familiar with the metrics and also allows a user to employ similar approaches in the analysis of the call-graphs.

6.3 Implementation Issues

This section describes implementation issues I encountered and solved during the writing of the advanced implementation of the **SCG**.

6.3.1 Dynamic Memory Allocation

At the time a sample is taken, memory often needs to be allocated to store the profiling data. For example, memory is required to store the nodes of the CCT. However, conditions related to the behaviour and state of the μC++ kernel and task can preclude the successful execution of this action [Les05]. Since dynamic memory allocation is a potentially blocking operation in a concurrent system, a task requesting memory may need to be blocked and another task may need to be scheduled. When a sample is taken, and the collected data needs to be stored, a task may be executing in the kernel; however, the μC++ kernel cannot block. Also, when a sample is taken, a task may be holding a spinlock (used by the μC++ kernel to protect critical data-structures) and in this case a task cannot enter the kernel and block. Therefore, in both of these situations blocking is impossible, and hence, dynamic memory allocation is not permitted.

If dynamic memory allocation is not permitted at the time of a sample, then the sample is lost (i.e., the profiling data collected for the sample is not stored). In order to substantially reduce the number of lost samples, I created a node pool and a data pool. Per-task pools were chosen because they exclude the need for mutual exclusion, and hence, there is no corresponding contention. Each task maintains an array of node objects (CCT nodes) and an array of data objects (per-task objects at each node) representing the pools. The arrays (of size 20) are initialized with dynamically allocated node and data objects during creation of the task's metric specific data structures. At the time of a sample, the nodes in the pool are used if the addition of nodes to the CCT is required, but dynamic memory allocation is not permitted. If dynamic memory allocation is permitted, then the nodes are simply dynamically allocated at the time of the sample and the pool is replenished. The possibility of a lost sample still exists because the number of nodes to be added to the CCT may be greater than the number of nodes currently available in the pool. The data pool is used in the same way.

The number of lost samples depends on the machine as well as the amount of time the program spends in the run-time system. The total number of lost samples is reported, in addition to the total number of samples taken, at the bottom of the task/coroutine-selection window (see Figure 6.6).

6.3.2 Handling Cycles

This issue involves the handling of cycles during analysis. In the initial implementation, all cycles are collapsed (i.e., reduced to a single node or pseudo-routine) [Les05]. Unfortunately, when a cycle is collapsed, the execution behaviour of the cycle members is lost; therefore, in the advanced implementation, I chose not to collapse cycles.

During analysis, each CCT is analyzed using a depth-first-search. Each sampled call-stack (i.e., each node path representing a sampled call-stack, see Section 6.2.1.1) in the CCT is traversed from its lowest node in the tree (i.e., the node representing the routine executing at the time of the sample) up to the root node. Walking up the path simply requires following the caller pointer maintained by each node object up to the root node. During this analysis, self and descendant samples are assigned to the routines, represented by the nodes along the path. In the usual case (without a cycle), the lowest node in the tree is assigned a self sample and every other node is assigned a descendant sample. While assigning a sample to a node, a counter is incremented corresponding to the node's current caller (the node above it in the call-stack, except for the root) and corresponding to the node's current callee (the node below it in the call-stack, except for the lowest node). The value of each counter equals the number of samples assigned to a routine that must be attributed to the counter's associated caller or callee. These counters allow for a routine's assigned samples to be properly attributed to its callers and callees in the visualization (i.e., to populate the callers and callees panes of Figure 6.7 for a routine). For example, while examining the lowest node of the path in Figure 6.9, a self sample is assigned to routine B and one self sample is counted for routine D as a caller of routine B. No sample is counted for a callee of routine B because routine B is executing at the time of the sample.

When the call-graph includes a cycle, the traversal may encounter more than one node representing the same routine (e.g., routine B in Figure 6.9). In this case, some nodes are not assigned descendant samples in order to prevent double counting. Each distinct routine, represented along the path, is assigned only a single sample, and therefore, at most only one caller and one callee counter is incremented. In the example, routine B is assigned a self sample (as described above), and therefore, does not get a descendant sample when encountered for the second and third times in the path. These conditions ensure that the total descendant samples for a routine equals the sum of the descendant samples for its callers as well as the sum of the self and descendant samples for its callees. In the **ECG**, similar adjustments were often impossible because of the presence of back-edges in the CCT (see Section 5.3.2).

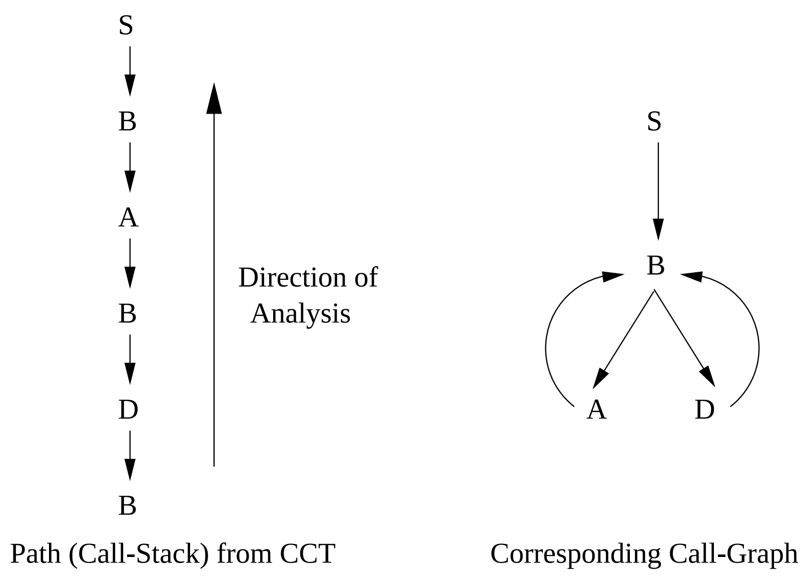


Figure 6.9: Advanced Implementation: Example Call-Stack

6.4 Related Work

This section describes two current profiling tools that include statistical call-graphs. HP Caliper provides profiling metrics for C, C++, Java, Fortran and Assembly programs, including multi-threaded programs. Sun Studio Performance Analyzer profiles C, C++, Fortran and Java programs, also supporting multithreaded programs.

6.4.1 HP Caliper

HP Caliper is a general-purpose performance analysis tool that helps a user understand the execution performance of a program [HP07]. A graphical user interface and command line version of Caliper are available. Profiling data is saved in databases for later visualization. Two visualizations provide the statistical call-graph (dynamic) data of the Sampled Call-Graph metric: the Histogram Tab and the Call Graph Tab.

Histogram Tab

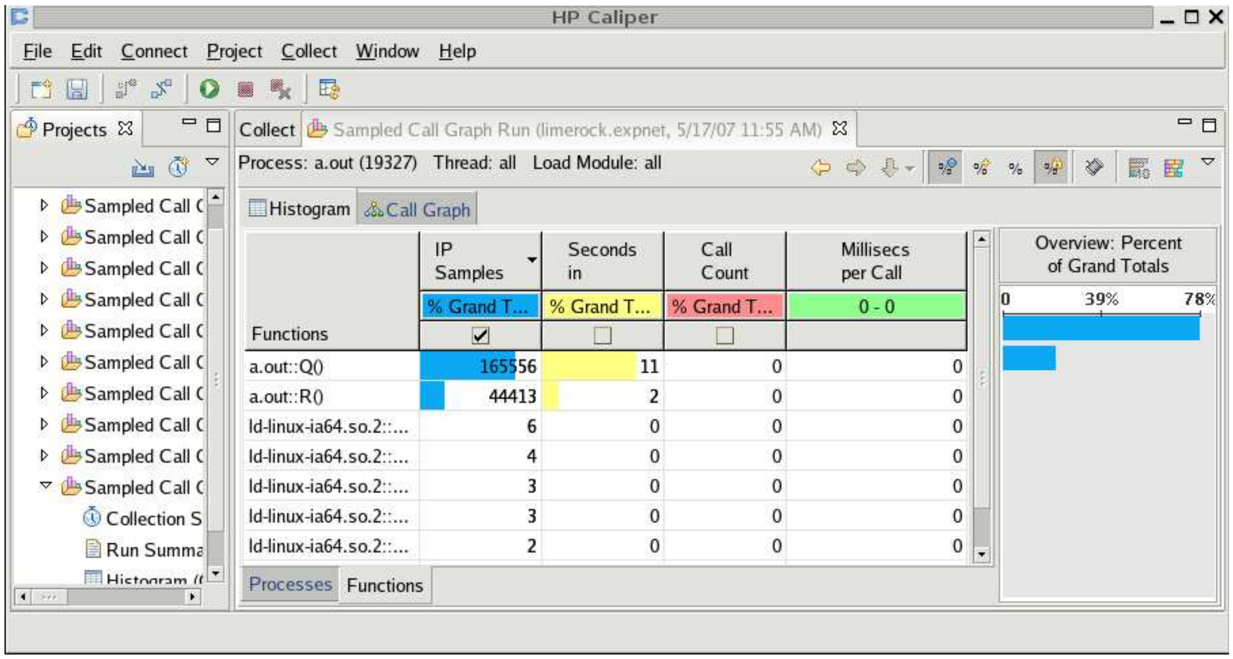
Caliper uses a CPU cycles hardware event to trigger sampling at a user defined interval. The Histogram Tab provides information on a process, module, thread or routine basis (see Figure 6.10). The per-routine data displayed includes the total number of self samples and self time for the routine, the total number of calls to the routine and the self time per-call. The data can be displayed as raw numbers or as percentages of a grand total, local total (e.g., with respect to the current process, module or thread) or cumulative total. The percentages are shown as numbers as well as histograms within the individual table fields and a separate bar graph. The table can be sorted on any column.

Call Graph Tab

The Call Graph Tab also lists each routine, and for the routine currently selected in the list, it updates a callers and callees pane (see Figure 6.11). The information displayed for each routine includes the number of self plus descendant samples for the routine as a percentage of the total, the number of self samples for the routine as a percentage of the total, the percentage of self samples relative to self plus descendant samples and the number of calls (and recursive calls) to the routine. The percentages are again displayed as raw numbers and histograms. For each caller of the selected routine, the total number of samples of the routine attributed to the caller is displayed as well as the number of calls from the caller to the routine over the total calls to the routine (as a percentage and fraction). For each callee of the selected routine, the total number of samples of the routine attributed from the callee is displayed as well as the number of calls from the routine to the callee over the total calls to the callee (as a percentage and fraction). A Callees Visited pane keeps track of the current path in the call-graph.

Sampling

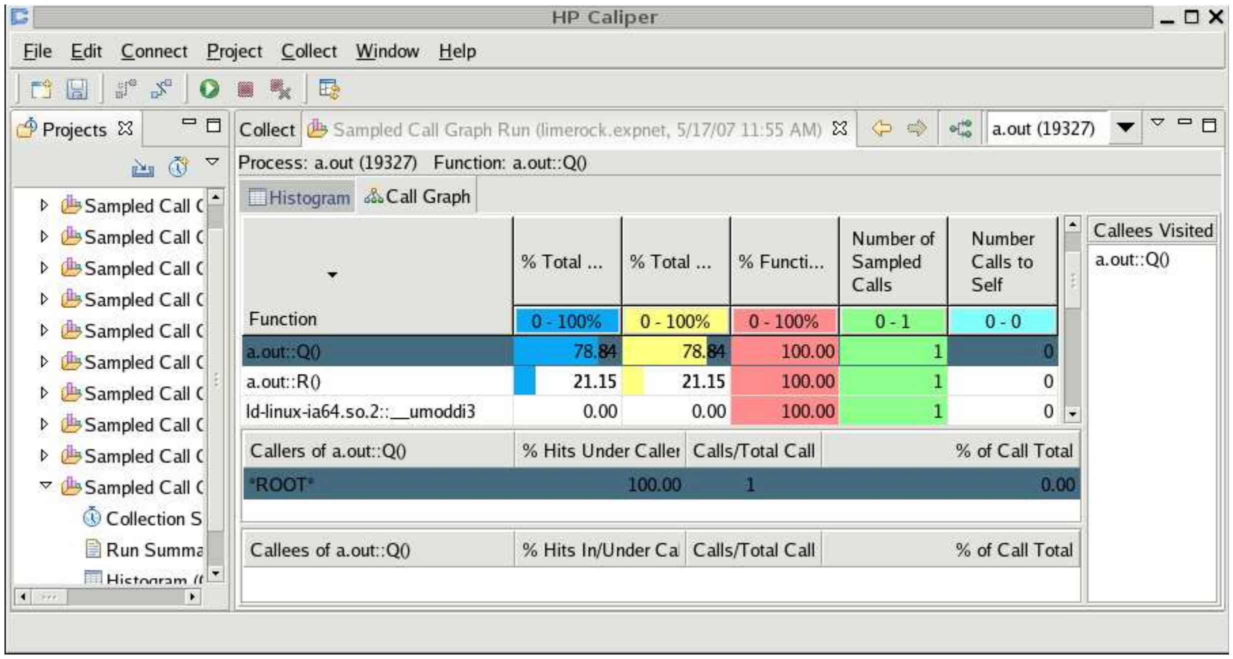
At each sample, the Sampled Call-Graph metric stores a copy of the branch-trace-buffer (BTB). The BTB is a circular buffer of size 8 implemented in hardware on the Itanium 2 processor. Two addresses are stored in the buffer for each routine call, so at any time at most 4 routine calls exist in the buffer. Therefore, routine addresses can be overwritten in the buffer before a sample is


HP Caliper
File Edit Connect Project Collect Window Help
Projects
Collect
Sampled Call Graph Run (limerock.expnet, 5/17/07 11:55 AM)
Histogram
Call Graph
Processes
Functions
Overview: Percent of Grand Totals
0
39%
78%

Process: a.out (19327) Thread: all Load Module: all

Functions	IP Samples	Seconds in	Call Count	Millisecs per Call
	% Grand T...	% Grand T...	% Grand T...	0 - 0
	✓	☐	☐	
a.out::Q()	165556	11	0	0
a.out::R()	44413	2	0	0
ld-linux-ia64.so.2::...	6	0	0	0
ld-linux-ia64.so.2::...	4	0	0	0
ld-linux-ia64.so.2::...	3	0	0	0
ld-linux-ia64.so.2::...	3	0	0	0
ld-linux-ia64.so.2::...	2	0	0	0

Figure 6.10: HP Caliper Histogram Tab


HP Caliper
File Edit Connect Project Collect Window Help
Projects
Collect
Sampled Call Graph Run (limerock.expnet, 5/17/07 11:55 AM)
a.out (19327)
Histogram
Call Graph
Callees Visited
a.out::Q()

Process: a.out (19327) Function: a.out::Q()

Function	% Total ...	% Total ...	% Functi...	Number of Sampled Calls	Number Calls to Self
	0 - 100%	0 - 100%	0 - 100%	0 - 1	0 - 0
a.out::Q()	78.84	78.84	100.00	1	0
a.out::R()	21.15	21.15	100.00	1	0
ld-linux-ia64.so.2::__umoddi3	0.00	0.00	100.00	1	0

Callers of a.out::Q()	% Hits Under Caller	Calls/Total Call	% of Call Total
ROOT	100.00	1	0.00

Callees of a.out::Q()	% Hits In/Under Ca	Calls/Total Call	% of Call Total

Figure 6.11: HP Caliper Call Graph Tab

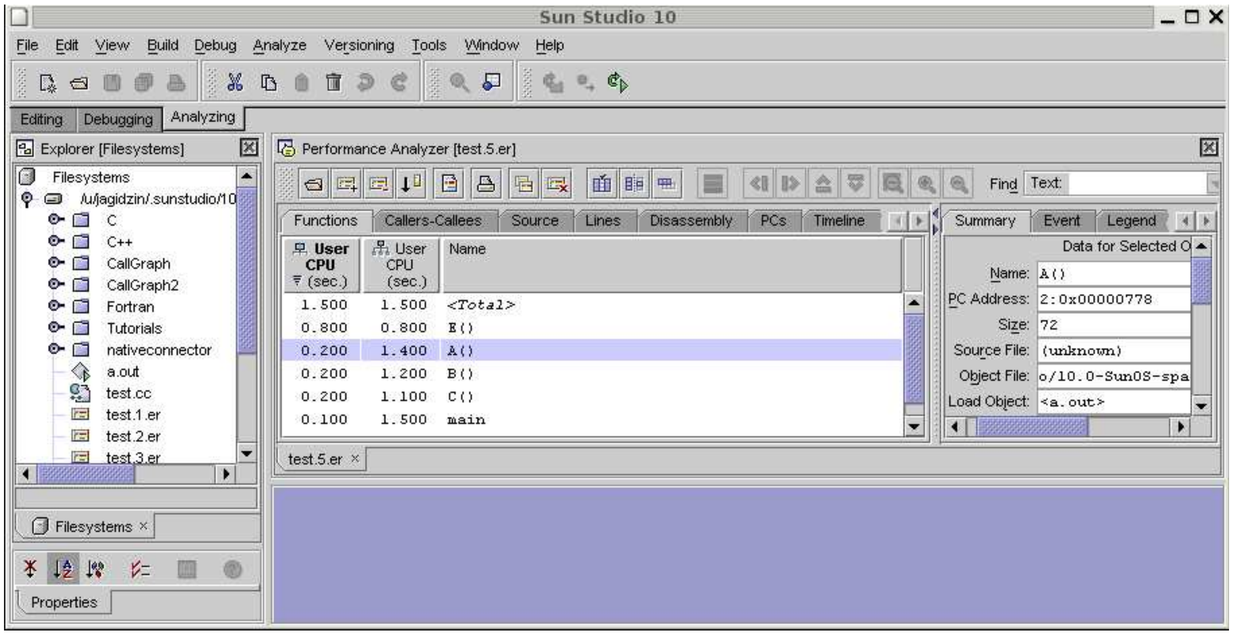
taken; such routines may go unreported in the call-graph. HP Caliper now has a new statistical call-graph metric called the Sampled Call-Stack Profile. At each sample, instead of storing a copy of the BTB, a copy of the full call-stack is stored. The time event triggers sampling and the visualization is similar to that of the Sampled Call-Graph metric. The Sampled Call-Stack Profile is currently only available for the HP-UX operating system, which I do not have access to; therefore, the Sampled Call-Stack Profile is not discussed further.

6.4.2 Sun Studio Performance Analyzer

Sun Studio Performance Analyzer helps a user identify potential performance problems, and locate the part of the program where the problems occur [Sun05]. A graphical user interface and command line version of the Performance Analyzer are available. Profiling data is also saved (as experiments) for later visualization. Two visualizations provide the statistical call-graph (dynamic) data: the Functions Tab and the Callers-Callees Tab. Data from multiple experiments can be combined and viewed on one tab.

Functions Tab

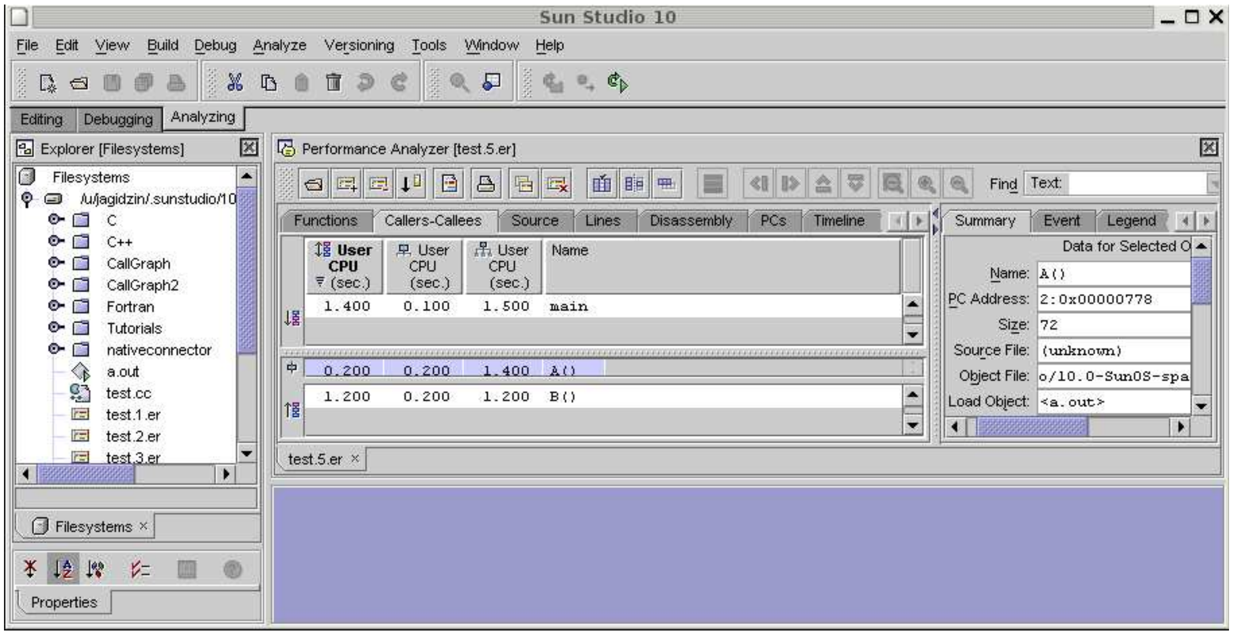
The sampling events available in the Performance Analyzer include time and various hardware events (e.g., CPU cycles, completed instructions, cache misses, etc.). A user can define a custom sampling interval for each event. The Functions Tab lists the routines and displays the self and total (self plus descendant) time or hardware-event counts for each routine (see Figure 6.12). The information on the Functions Tab is displayed for each sampling event selected by a user. The top routine represents the entire program, and hence, reveals the total time or hardware events executed during the running of the program. The data can be filtered by process, module or thread, and individual routines can be hidden. Furthermore, a user can modify the presentation of the tab. A user can sort on any column, display the data in any column as raw numbers or percentages, and show or hide columns of data. The Callers-Callees Tab is updated for the routine selected on the Functions Tab.


Sun Studio 10
File Edit View Build Debug Analyze Versioning Tools Window Help
Editing Debugging Analyzing
Explorer [Filesystems]
Filesystems
/u/jagidzin/.sunstudio/10
C
C++
CallGraph
CallGraph2
Fortran
Tutorials
nativeconnector
a.out
test.cc
test.1.er
test.2.er
test.3.er
Filesystems
Properties
Performance Analyzer [test.5.er]
Find Text:
Functions Callers-Callees Source Lines Disassembly PCs Timeline
Summary Event Legend
test.5.er

User CPU (sec.)	User CPU (sec.)	Name
1.500	1.500	*<Total>*
0.800	0.800	E()
0.200	1.400	A()
0.200	1.200	B()
0.200	1.100	C()
0.100	1.500	main

Data for Selected O
Name: A()
PC Address: 2:0x00000778
Size: 72
Source File: (unknown)
Object File: o/10.0-SunOS-spa
Load Object: <a.out>

Figure 6.12: Sun Studio Performance Analyzer Functions Tab


Sun Studio 10
File Edit View Build Debug Analyze Versioning Tools Window Help
Editing Debugging Analyzing
Explorer [Filesystems]
Filesystems
/u/jagidzin/.sunstudio/10
C
C++
CallGraph
CallGraph2
Fortran
Tutorials
nativeconnector
a.out
test.cc
test.1.er
test.2.er
test.3.er
Filesystems
Properties
Performance Analyzer [test.5.er]
Find Text:
Functions Callers-Callees Source Lines Disassembly PCs Timeline
Summary Event Legend
test.5.er

User CPU (sec.)	User CPU (sec.)	User CPU (sec.)	Name
1.400	0.100	1.500	main
0.200	0.200	1.400	A()
1.200	0.200	1.200	B()

Data for Selected O
Name: A()
PC Address: 2:0x00000778
Size: 72
Source File: (unknown)
Object File: o/10.0-SunOS-spa
Load Object: <a.out>

Figure 6.13: Sun Studio Performance Analyzer Callers-Callees Tab

Callers-Callees Tab

The Callers-Callees Tab displays the callers and callees for the selected routine (see Figure 6.13). The selected routine, with its corresponding information, is also displayed in between the Callers and Callees panes. For each caller of the selected routine, the total time or hardware-event counts for the routine attributed to the caller are displayed as well as the self and total values for the caller routine as a whole. For each callee of the selected routine, the total time or hardware-event counts for the routine attributed from the callee are displayed as well as the self and total values for the callee routine as a whole. Similar to the Functions Tab, a user can modify the presentation of the Callers-Callees Tab.

Other Tabs

Sun Studio Performance Analyzer provides some further useful information in other tabs. The Source Tab shows the file containing the source code of the selected routine, annotated with profiling data for each line. The Timeline tab shows a chart of the sampling points recorded during monitoring as a function of time. Clicking a sampling point displays the data for that sample in the Event Tab. The Event Tab displays information such as the time the sample was taken, the duration of time since the previous sample, and a colour-coded representation of the call-stack at the time of the sample.

A user can also pause and resume data collection for the running program (but not perform real-time analysis) using control buttons or an application program interface (API).

6.4.3 Comparison

Table 6.1 summarizes and compares the relevant features of μProfiler's **SCG** metric and the two profiling tools discussed in the previous sections. Some of the important features are discussed in detail.

HP Caliper's Sampled Call-Graph metric, like gprof, does suffer from the gprof fallacy (see Section 5.4.3). Also, as a consequence of storing a copy of the BTB at each sample, the resulting call-graph may be disconnected. Compared to a full call-stack, the BTB only provides the last four call-stack entries (or caller-callee routine pairs); hence, routines and routine calls existing elsewhere in the call-stack are not recorded. Therefore, in general HP Caliper's Sampled Call-Graph metric provides less accurate call-graph information than the statistical call-graph metrics of μProfiler and Sun Studio Performance Analyzer.

	μProfiler SCG Metric	HP Caliper	Sun Studio Perf. Analyzer
Hardware Events	$\surd$	$\surd$	$\surd$
Combine Profiling Runs		$\surd$	(side-by-side display)
Instrumentation Control	$\surd$	$\surd$	$\surd$
No gprof Fallacy	$\surd$		$\surd$
Data Saved to File		$\surd$	$\surd$
Graphical User Interface	$\surd$	$\surd$	$\surd$
Interactive Caller-Callee Display	$\surd$	$\surd$	$\surd$
Multiple Formats	$\surd$		
Cycles Information	$\surd$	$\surd$	
Complete Call-Graph	$\surd$		
Call-Graph Break Down	$\surd$	$\surd$	$\surd$
Sorting		$\surd$	$\surd$
Source Information	$\surd$	$\surd$	$\surd$
Callees Visited List	$\surd$	$\surd$	
Histograms of %s	$\surd$	$\surd$	

Table 6.1: Statistical Routine Call-Graph: Comparison of Related Profilers

Both the **SCG** and Sun Studio Performance Analyzer have multiple sampling events available for a user (e.g., time and various hardware events). HP Caliper only provides a single hardware event of CPU cycles. The Performance Analyzer displays all data as hardware-event counts or as time. Caliper displays all data as time by converting the CPU cycles. However, unlike the other profiling tools, the **SCG** allows a user to view the call-graph data in multiple formats such as the number of samples taken, time or hardware-event counts. Multiple formats allow users to view the call-graph information from several different perspectives as well as choose which format best suits their analysis.

Another feature only available in the **SCG** is a complete call-graph. By not including a complete call-graph, Caliper and Performance Analyzer fail to provide a user with a means of viewing and analyzing the entire call-graph at one time.

Overall, μProfiler's **SCG** metric provides many important features, and furthermore, includes features unavailable in the other profiling tools. Some features not currently provided in μProfiler, such as routine-level instrumentation control, saving data to a file and sorting, are possible enhancements for future work.

6.5 Performance

This section describes the performance of the **SCG** with respect to both time and space.

6.5.1 Time

To evaluate the running time of the **SCG**, I constructed a worst-case test program (see Appendix B.1), profiled the program with the **SCG** metric, and compared its running time to the same test program run without profiling. The program was also run with HP Caliper and Sun Studio Performance Analyzer for a cross-profiler comparison. Only the running time of the test program itself was measured, i.e., the time includes monitoring and data collection but not time spent during analysis or visualization.

The test program is the same program used to evaluate the running time of the **ECG** (see Section 5.5) except Routines G1 through G6 call routine H1 2,100,000 times (versus 30,000 times). The test program was compiled with optimization (i.e., O2 flag) and run multiple times with a decreasing sampling interval (i.e., increasing number of samples). Testing for the **SCG** and HP Caliper was done on one machine (Itanium II 1499 Mhz) and testing for the Sun Studio Performance Analyzer was done on a different machine (UltraSPARC III 1062 Mhz). Unfortunately, a direct comparison between μProfiler and Sun Studio on the same UltraSPARC architecture was impossible because μProfiler is only ported to the hardware-event interface for the Solaris-8 operating system and Sun Studio only runs on Solaris-10. Porting μProfiler to Solaris-10 is future work. Table 5.2 shows the results of the performance testing in microseconds. The CPU-

Sampling Interval (ms)	μProfiler		HP Caliper		Sun Studio Analyzer	
	Time per Sample (μs)	% Increase	Time per Sample (μs)	% Increase	Time per Sample (μs)	% Increase
10.00	28.80	0.29	9.80	0.10	83.777	0.92
7.80	24.32	0.31	9.76	0.13	50.84	0.96
5.60	19.93	0.36	8.24	0.15	45.56	1.13
1.10	19.72	1.77	2.73	0.25	28.08	2.52
0.78	19.45	2.50	2.31	0.30	27.97	3.57
0.56	19.97	3.59	2.13	0.38	27.81	4.95
0.33	19.47	5.84	2.40	0.72	27.70	8.17
0.22	18.96	8.53	2.52	1.13	27.86	12.29
0.11	19.05	17.14	2.26	2.03	28.01	25.60
0.06	18.87	33.99	2.02	3.64	27.92	53.13

Table 6.2: Statistical Routine Call-Graph: Time Performance Results

cycles hardware event was used as the sampling event, and the sampling intervals in the table are computed by converting CPU cycles to millisecond time units. The percentage increase is with respect to the no profiling case.

The per-sample time is calculated by dividing the difference between the profiled running time and corresponding no-profiler running time by the number of samples taken. As the sampling rate increases (i.e., the sampling interval decreases), the running time of the program increases for all profilers as seen in the table with the increasing percentages. The time per-sample quickly stabilizes as the sampling rate increases for all profilers. Although the test program was run for a sufficient duration for stabilization to occur across all rates, when fewer samples are taken the time per-sample values are higher. Further experiments have revealed that this is the result of a lower data cache hit-rate, which slows the entire program. The numbers for HP Caliper are significantly lower than those for the **SCG** and Sun Studio Performance Analyzer because the metric does less work at each sample and provides a much less accurate call-graph. Compared to the Sun Studio Performance Analyzer, the time per-sample and percentage increase numbers for the **SCG** are somewhat lower. However, it must be noted again that these two profilers run on different machines, and hence the numbers may not be directly comparable. The percentage increases for the **SCG**, for this worst-case program, are reasonable given the overhead of creating the storage data structures and, at each sample, collecting and storing the necessary data.

6.5.2 Space

To examine the space cost of a CCT, the space costs of the individual data structures (see Section 6.2.1) are examined. The data structures include the general tree object, the node object and the data object. The general tree object maintains a pointer to the root node of the tree, an array (of size 240) of routine addresses (pointers) representing the call-stack at the last sample, an integer to store the size of the last call-stack, an integer to store the sampling event triggering the last sample and a boolean specifying whether or not the last sample was lost. Therefore, in a standard 32 bit system with 4 byte pointers, 4 byte integers and a 1 byte boolean (with 3 bytes of padding to maintain proper alignment) each tree object requires 976 bytes of space (i.e., $(4\ bytes \times 240) + 16\ bytes$).

Each node object consists of a pointer to link the node object in a node's queue, a routine address (a pointer), a pointer to its caller node in the tree, a queue of pointers to its callee node objects and a queue of pointers to its data objects. A queue requires 8 bytes of space; and therefore, each node object requires 28 bytes of space. Each data object consists of a pointer to link the data object in a node's queue, a pointer to the task associated with the data and an array of integers (with one array entry for the time event and each hardware counter). Therefore, the space cost of each data object is computed according to the following formula

$$Total\ space\ per\text{-}data\text{-}object = 12\ bytes + (number\ of\ hardware\ counters \times 4\ bytes) \tag{6.2}$$

Each task maintains a node pool which consists of an array of 20 pointers to node objects and a data pool which consists of an array of 20 pointers to data objects. For each array, two integers are required to store the size of the array and the number of available objects. Therefore, the space cost for the pools is computed according to the following formula

$$Total\ space\ for\ pools = 24\ bytes + (20 \times 28\ bytes) + (20 \times size\ of\ a\ data\ object) \tag{6.3}$$

The test program used to evaluate the running time of the **SCG** creates one tree (single task program) with 102,644 node objects and 102,644 data objects. The number of node and data objects is for a worst case scenario where the tree includes all possible sampled call-stacks. Assuming the test program is run on a processor with 4 hardware counters, the total space cost for the tree is

$$\begin{aligned} Total\ space\ cost &= 976\ bytes + (102644 \times 28\ bytes) + (102644 \times 28\ bytes) \\ &+ 584 + (20 \times 28) = 5,750,184\ bytes \end{aligned} \tag{6.4}$$

The space cost of 5.48 MB is reasonable given that the depth and breadth of the call-graph is large, i.e., large number of node and data objects in the tree. For the **ECG**, the space cost is approximately 4.2 times larger (see Section 5.5.2 p. 101) as a result of the size of the data objects; 200 bytes for the **ECG** compared to 28 bytes for the **SCG**.

6.6 Summary

The advances made to μProfiler's **SCG** metric have achieved the goals stated at the beginning of the chapter. Firstly, based on functionality and the comparison to related work, μProfiler's **SCG** metric is similar to state-of-the-art vendor statistical call-graph metrics. Secondly, the run-time of the **SCG** metric compared to the related work shows similar performance and space costs are reasonable. Thirdly, the **SCG** metric, through the careful collection, separation, analysis and visualization of profiling data, scales to programs of long duration and complex behaviour. Finally, I believe the collective achievement of the goals results in a call-graph metric providing an environment conducive to more thorough and simpler user analysis.

Chapter 7

Conclusions and Future Work

This thesis focused on profiling user threads in concurrent, object-oriented programs running in a shared-memory, uni/multi-processor environment. Profiling is accomplished using μProfiler, a concurrent object-oriented profiler written in μC++, a concurrent dialect of the C++ programming language.

The contributions of this thesis include major advances to the following μProfiler metrics: the Execution State Chart as part of the Execution State metric, the Exact Routine Call-Graph metric and the Statistical Routine Call-Graph metric.

The Execution State metric charts each task's states during execution of the program. By only drawing the visible area of the chart, the Execution State Chart is now scalable to programs of long duration and with large numbers of tasks and states. The chart provides high magnification as well as fine-grained control through options such as the "Magnification Step". The introduction of the "Elided" state ensures the chart is always accurate and logically consistent. The dynamic nature of the X-axis provides a precise division of the axis, allowing for more exact reading of the chart. Furthermore, the evaluation of time and space costs reveals good performance, and a cross-profiler comparison indicates the metric is functionally state-of-the-art.

The Exact Routine Call-Graph metric and the Statistical Routine Call-Graph metric both provide a call-graph profile of the program's dynamic execution. The exact metric provides higher accuracy at the cost of higher overhead, whereas the statistical metric provides lower overhead at the cost of lower accuracy. For both metrics, major advances were made to the data structures storing the profiling data and the visualization of the data. A space efficient data structure called the context calling tree now stores the profiling data. The visualization provides an interactive experience, allowing a user to display data for a particular event, and for specific coroutines (coroutine-selection pane on per-task call-graph) or tasks (task-selection pane on per-coroutine call-graph). Also, a user can progress through the call-graph step-by-step by selecting routines in the various panes, with that movement being tracked in the callees-visited pane. The per-task and per-coroutine call-graphs provide separation of data and multiple perspectives for analysis. Furthermore, cross-profiler comparisons reveal similar run-time performance and state-of-the-art functionality.

To ensure μProfiler is well-designed and includes common features expected by profiler users, μProfiler was compared to several vendor profilers to establish a baseline for both features and performance. Through this comparison, it was found that μProfiler has some unique functionally unavailable in other commonly used profilers, suggesting some advancement in the state-of-the-art.

Overall, for a user, the advances have resulted in more powerful, scalable, functional and intuitive metrics with good performance, yet at the same time resulted in simpler analysis. A user can spend less time and effort learning the metrics, while spending their analysis time more effectively and efficiently.

7.1 Future Work

There are a number of possible directions for future work for μProfiler. Currently, performance data cannot be saved to or loaded from a file. Saving and loading data to/from a file gives a user easy access to the performance data for later analysis (i.e., after the visualizations of the performance data have been terminated). Such functionality is particularly important because the nondeterministic nature of concurrent programs makes it difficult to reproduce specific results by simply re-running and re-profiling a program. Also, μProfiler currently does not do any real-time analysis; all data is analyzed and visualized post-mortem. Real-time analysis and visualization could be beneficial for some metrics, especially given certain types of long running programs, as a user would not have to wait for the program to finish to start viewing performance data.

For the **EST** metric, the addition of an aggregate view, such as the one provided by some of the related execution state profiling tools, would provide a user with a high-level overview of the global execution-state of their program. Also, given a large number of tasks displayed on the **ESTC**, it would be advantageous to be able to sort the tasks by various criteria.

A number of improvements can also be made to the **ECG** metric. Routine-level instrumentation control would allow a user to selectively choose which routines are profiled, and hence, allow a user to precisely focus the analysis. Also, by only instrumenting certain routines in a program, the probe effect can be reduced. Given a large number of routines displayed, it would also be advantageous to sort the routines by various criteria. A simple graphical tree representation of the call-graph could also be beneficial. By highlighting the paths in the tree where the program is consuming the most time, a user could quickly focus the analysis.

In the **SCG** metric, in addition to instrumentation control, sorting and a graphical tree, improvements related to the sampling interval and hardware-event selection can be made. Currently, a user is presented with general defaults for the sampling intervals; however, it would be valuable to provide customized defaults based on certain criteria such as characteristics of the program being profiled. Also, because a processor has only a limited number of hardware counters and each of those counters can count only certain hardware events, a user can simultaneously select only a limited number of hardware events. Multiplexing (or time sharing) can mitigate this restriction by having different counters count different events during different periods of time. Future work should explore the multiplexing of hardware counters for the **ECG** and **SCG** metrics.

Finally, for both the **ECG** and **SCG** metrics, another possible enhancement is object-based profiling. For example, μC++ monitor objects could form a call-graph, in addition to the per-task and per-coroutine call-graphs. A per-monitor call-graph would include data related to the execution of the particular monitor and its routines. This further breakdown provides yet another perspective for a user when analyzing a call-graph.

Appendix A

Object-Oriented Notation

The notation described in this appendix is based on the object-oriented design notation of Peter Coad and Jill Nicola [CN93]. The original notation has been simplified and extended to include objects specific to μC++. The design of the μProfiler kernel in Chapter 3 and the design of the data structures in Chapters 5 and 6 are illustrated using this notation.

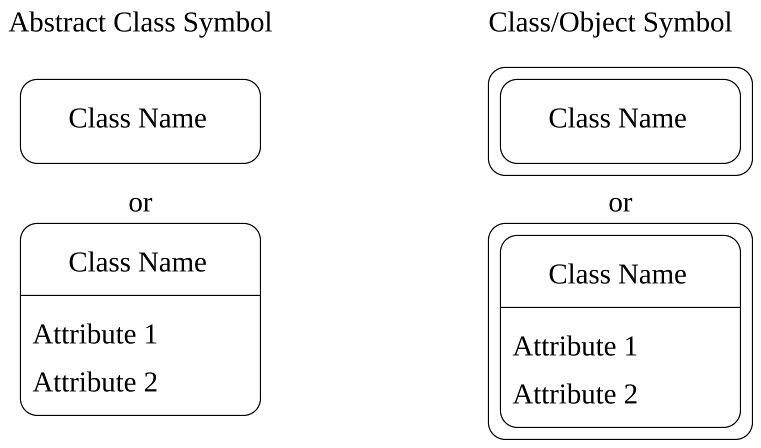


Figure A.1: Class and Object Notation

Figure A.1 shows the symbols used to illustrate classes and objects. A class defines the behaviour and properties of all objects instantiated from it, and therefore, an object is an instance of a class. The Abstract Class symbol represents an abstract class, one that cannot be instantiated. The Class/Object symbol represents a class that can be instantiated with one or more objects instances. The inner rounded rectangle represents the class definition and the outer rounded rectangle represents the instances of that class. Each of these symbols contains the name of the class and below the name are the class attributes. Only attributes relevant to the design being described are included, if any at all.

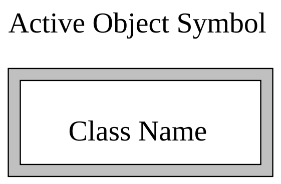
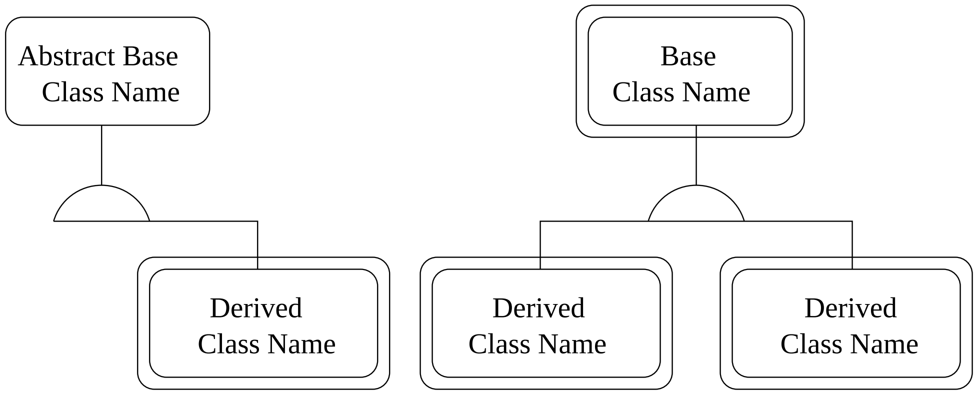


Figure A.2: Active Object Notation

μC++ has active objects, such as tasks, containing a thread of control and an execution state. The Active Object symbol shown in Figure A.2 is not part of Coad and Nicola's original notation, but was introduced by Dorota Zak [Zak00].

There are three types of relations that can be indicated between classes and objects: inheritance, aggregation and association. The inheritance relation is represented by a line with a semicircle drawn between classes (see Figure A.3). The derived classes, connected to the bottom of the semicircle, inherit the attributes and routines of the base class, connected to the top of the semicircle. A derived class "is-a" base class with additional specialization. Because inheritance occurs between classes, and not objects, the lines are connected to the inner rectangle of the Class/Object symbols. Derivation can occur from an abstract base-class (abstract derivation) or a concrete base-class (concrete derivation).

The aggregation relation is represented by a line with a triangle drawn between objects (see Figure A.4). The triangle points away from a member object and towards a containing object. The containing object "has-a" member object as an attribute. Because aggregation occurs between objects, and not classes, the lines are connected to the outer rectangle of the Class/Object symbols.



(a) Abstract Derivation (b) Concrete Derivation

Figure A.3: Inheritance Notation

The association relation is represented by a line drawn between objects (see Figure A.5). Each object “uses” (or is “aware of”) the other object. Because association occurs between objects, and not classes, the lines are connected to the outer rectangle of the Class/Object symbols.

The aggregation and association relations display cardinality symbols next to each object, representing how many objects of one class are connected to how many objects of the other class. In Figure A.4, the containing object contains zero or more instances of the member object, and each member object is contained by only one containing object. In Figure A.5, an object of class A is associated with one object of class B, and an object of class B is associated with one or more objects of class A.

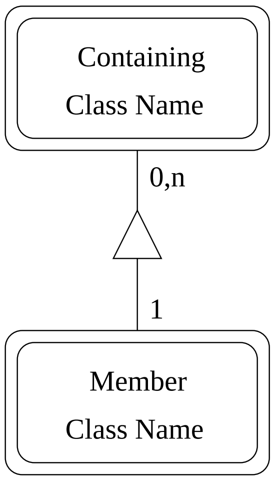


Figure A.4: Aggregation Notation

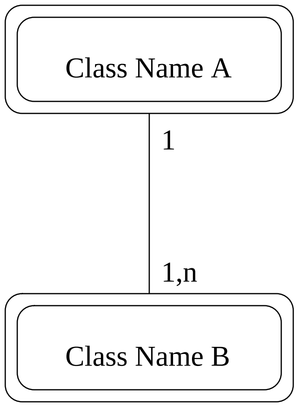


Figure A.5: Association Notation

Appendix B

Program Source Code

B.1 Call-Graph Test Program

```
include <uC++.h>

void A();
void B1(); void B2(); void B3(); void B4(); void B5(); void B6();
void C1(); void C2(); void C3(); void C4(); void C5(); void C6();
void D1(); void D2(); void D3(); void D4(); void D5(); void D6();
void E1(); void E2(); void E3(); void E4(); void E5(); void E6();
void F1(); void F2(); void F3(); void F4(); void F5(); void F6();
void G1(); void G2(); void G3(); void G4(); void G5(); void G6();
void H1();

void A() {
    B1();
    B2();
    B3();
    B4();
    B5();
    B6();
} // A


void B1() {
    C1();
    C2();
    C3();
    C4();
    C5();
    C6();
} // B1

...

void B6() {
    C1();
    C2();
    C3();
    C4();
    C5();
    C6();
} // B6

void C1() {
    D1();
    D2();
    D3();
    D4();
    D5();
    D6();
} // C1

...

void C6() {
    D1();
    D2();
    D3();
    D4();
    D5();
    D6();
} // C6


void D1() {
    E1();
    E2();
    E3();
    E4();
    E5();
    E6();
} // D1

...
```

```
void D6() {
    E1();
    E2();
    E3();
    E4();
    E5();
    E6();
} // D6

void E1() {
    F1();
    F2();
    F3();
    F4();
    F5();
    F6();
} // E1

...

void E6() {
    F1();
    F2();
    F3();
    F4();
    F5();
    F6();
} // E6
```

```
void F1() {
    G1();
    G2();
    G3();
    G4();
    G5();
    G6();
} // F1

...

void F6() {
    G1();
    G2();
    G3();
    G4();
    G5();
    G6();
} // F6

void G1() {
    for ( int i = 0; i < 30000; i++ ) { H1(); }
} // G1

...

void G6() {
    for ( int i = 0; i < 30000; i++ ) { H1(); }
} // G6

void H1() {
} // H1

void uMain::main() {
    A();
} // uMain::main
```

Bibliography

[ABL97] G. Ammons, T. Ball, and J.R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–96, 1997. 68, 69

[AGH00] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language*. Addison-Wesley, 2000. 1

[App] AppPerfect. *AppPerfect Java Profiler Data Sheet*. http://www.appperfect.com/-products/devsuite/jp.html. Last accessed May 2007. 52

[AS00] M. Arnold and P.F. Sweeney. Approximating the calling context tree via sampling. Research report, IBM Research Division, 2000. 107

[BDS+92] P.A. Buhr, G. Ditchfield, R.A. Stroobosscher, B.M. Younger, and C.R. Zarnke. μC++: Concurrency in the object-oriented language C++. *Software - Practice and Experience*, 22(2):137–172, 1992. 2, 20

[BH05] P.A. Buhr and A.S. Harji. Concurrent urban legends. *Concurrency and Computation: Practice and Experience*, 17(9):1133–1172, 2005. 2, 5

[Blo70] B.H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970. 72

[Bor03] Borland. *Borland Optimizeit 6 Thread Debugger 1.4 User's Guide*, 2003. http://-info.borland.com/techpubs/optimizeit/optimizeit6/index1280x1024.html. Last accessed May 2007. 52, 57

[BS07] P.A. Buhr and R.A. Stroobosscher. *μC++ Annotated Reference Manual, Version 5.4.1*. David R. Cheriton School of Computer Science, University of Waterloo, 2007. ftp://plg.uwaterloo.ca/pub/uSystem/uC++.ps.gz. Last accessed May 2007. 4, 20, 21

[Cha91] S. Chamberlain. *LIB BFD, the Binary File Descriptor Library*. Cygnus Support, first edition, 1991. 28

[CL00] S. Choi and E.C. Lewis. A study of common pitfalls in simple multi-threaded programs. In *Proceedings of the 31st SIGCSE Technical Symposium on Computer Science Education*, pages 325–329. ACM Press, 2000. 2

[CN93] P. Coad and J. Nicola. *Object-Oriented Programming*. Prentice Hall PTR, 1993. 135

[Den97] R.R. Denda. Profiling concurrent programs. Diplomarbeit, Fakultät für Mathematik und Informatik, Universität Mannheim, 1997. ftp://plg.uwaterloo.ca/pub/theses/-DendaThesis.ps.gz. Last accessed May 2007. 10, 17, 19, 22

[ejt07] ej-technologies. *JProfiler Manual*, 2007. http://resources.ej-technologies.com/-jprofiler/help/doc/help.pdf. Last accessed May 2007. 52

[FFMC03] N. Froyd, R. Fowler, and J. Mellor-Crummey. Low-overhead call path profiling of unmodified, optimized code. In *Proceedings of the 19th Annual International Conference on Supercomputing*, pages 81–90, 2003. 68, 107

[GCC] GCC. *The GNU Compiler Collection*. http://gcc.gnu.org. Last accessed May 2007. 24

[Gen81] W.M. Gentleman. Message passing between sequential processes: The reply primitive and the administrator concept. *Software - Practice and Experience*, 11(5):435–466, 1981. 26

[GKM82] S. Graham, P. Kessler, and M. McKusick. gprof: A call graph execution profiler. In *Proceedings of the 1982 ACM SIGPLAN Symposium on Compiler Construction*, pages 120–126. ACM Press, 1982. 8, 90, 92, 98

[GR89] N. Gehani and W.D. Roome. *The Concurrent C Programming Language*. Silicon Press, 1989. 2

[gra96] University of Glasgow, Functional Programming Group. *GranSim User's Guide*, 1996. http://www.dcs.gla.ac.uk/fp/software/gransim/user_7.html#SEC47. Last accessed May 2007. 52

[HF94] D. Heller and P.M. Ferguson. *Motif Programming Manual for OSF/Motif Release 1.2*. O'Reilly & Associates, Inc., second edition, 1994. 23

[HH04] C. Hughes and T. Hughes. *The Joys of Concurrent Programming*. Addison-Wesley, 2004. 1

[HLM95] J.K. Hollingsworth, J.E. Lumpp, and B.P. Miller. Techniques for performance measurement of parallel programs. *Parallel Computers: Theory and Practice*, pages 225–240, 1995. 2, 3

[HM93] J.K. Hollingsworth and B.P. Miller. Dynamic control of performance monitoring on large scale parallel systems. In *Proceedings of the 7th International Conference on Supercomputing*, pages 185–194. ACM Press, 1993. 9

[Hol94] J.K. Hollingsworth. *Finding Bottlenecks in Large Scale Parallel Programs*. PhD thesis, Computer Sciences Department, University of Wisconsin - Madison, 1994. ftp://ftp.cs.wisc.edu/paradyn/papers/Hollingsworth94Dissertation.ps. Last accessed May 2007. 10

[HP04] HP. *HP Visual Threads Online Help*, 2004. http://h21007.www2.hp.com/dspp/-files/unprotected/visualthreads/doc/help2004/htmlhelp/vt.html. Last accessed May 2007. 52

[HP06] HP. *HPjmeter 2.1 User's Guide*, 2006. http://www.hp.com/products1/unix/java/-hpjmeter/infolibrary/user_guide.pdf. Last accessed May 2007. 52

[HP07] HP. *HP Caliper User Guide*, 2007. http://h21007.www2.hp.com/dspp/files/-unprotected/caliper/caliper-user-guide.html. Last accessed May 2007. 120

[HWG03] A. Hejlsberg, S. Wiltamuth, and P. Golde. *The C# Programming Language*. Addison-Wesley, 2003. 1

[Int07] Intel. *VTune Performance Environment User's Guide*, 2007. http://www.intel.com/-software/products/documentation/vlin. Last accessed May 2007. 94

[JFL98] M. Ji, E.W. Felten, and K. Li. Performance measurements for multithreaded programs. In *Proceedings of the 1998 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, pages 161–170. ACM Press, 1998. 2

[KR88] B.W. Kernighan and D. Ritchie. *The C Programming Language*. Prentice Hall, 1988. 1

[Les05] J. Lessard. Profiling concurrent programs using hardware counters. Master's thesis, School of Computer Science, University of Waterloo, 2005. ftp://plg.uwaterloo.ca/-pub/theses/LessardThesis.ps.gz. Last accessed May 2007. 19, 31, 66, 79, 103, 118

[LP85] C.H. LeDoux and D.S. Parker. Saving traces for Ada debugging. In *Proceedings of the 1985 Annual ACM SIGAda International Conference on Ada*, pages 97–108. Cambridge University Press, 1985. 8

[MCC+95] B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R.B. Irvin, K.L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn parallel performance measurement tools. *IEEE Computer*, 28(11):37–46, 1995. 8

[MH89] C.E. McDowell and D.P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, 21(4):593–622, 1989. 8

[MMB+94] A. Malony, B. Mohr, P. Beckman, D. Gannon, S. Yang, and F. Bodin. Performance Analysis of pC++: A portable data-parallel programming system for scalable parallel computers. In *Proceedings of the 8th International Parallel Processing Symposium (IPPS)*, Cancun, Mexico, 1994. 2

[MR82] M.F. Morris and P.F. Roth. *Computer Performance Evaluation: Tools and Techniques for Effective Analysis*. Van Nostrand Reinhold, New York, 1982. 16, 33

[Net] NetBeans. *NetBeans IDE Profiler Online Documentation*. http://-profiler.netbeans.org/docs/help/index.html. Last accessed May 2007. 52, 55

[PN93] C.M. Pancake and R.H.B. Netzer. A bibliography of parallel debuggers. In *Proceedings of the 1993 ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 169–186. ACM Press, 1993. 4

[She99] S. Shende. Profiling and tracing in Linux. In *Proceedings of the Extreme Linux Workshop #2*, Monterey, CA, 1999. 10

[Str97] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1997. 1, 20

[Sun04] Sun Microsystems. *UltraSPARC III Cu User's Manual, Version 2.2.1*, 2004. ftp://-www.sun.com/processors/manuals/USIIIv2.pdf. Last accessed May 2007. 31

[Sun05] Sun Microsystems. *Sun Studio 11: Performance Analyzer*, 2005. http://-docs.sun.com/app/docs/doc/819-3687. Last accessed May 2007. 123

[Tuf83] E.R. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, 1983. 16

[Uni95] United States Government. *Ada Reference Manual*, 1995. http://-www.adapower.com/rm95.php. Last accessed May 2007. 1

[XMN99] Z. Xu, B.P. Miller, and O. Naim. Dynamic instrumentation of threaded applications. In *Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Program*, pages 49–59. ACM Press, 1999. 1

[Zak00] D. Zak. Analyzing multi-threaded program performance with μProfiler. Master's thesis, School of Computer Science, University of Waterloo, 2000. ftp://-plg.uwaterloo.ca/pub/theses/ZakThesis.ps.gz. Last accessed May 2007. 10, 19, 34, 65, 136