Profiling Concurrent Programs Using Hardware Counters

by

Josh Lessard

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, 2005

Abstract

Concurrency is a programming tool that is widely used in applications. Concurrent user-level threads can be used to structure the execution of a program in a uniprocessor environment and/or speed up its execution in a multiprocessor setting. Unfortunately, threads may interact with each other in unpredictable ways, often leading to performance problems that are nonexistent in the sequential domain.

A profiler can be used to help locate performance problems in sequential and concurrent programs. A profiler is a tool that monitors, analyzes, and visualizes the execution performance of a program to help users verify its expected behaviour, and locate its bottlenecks and hotspots. One of the important tools a profiler has at its disposal is a set of hardware counters, which are specialized CPU registers that count the occurrences of hardware events as a program executes. Hardware-event counts provide extremely precise insight into the execution behaviour of a program, and can be used to pinpoint portions of code where performance is suboptimal.

This thesis describes the design and implementation of μProfiler, which is a profiler for sequential and concurrent programs written in a concurrent dialect of the C++ programming language called μC++. μC++ offers user-level concurrency in a uniprocessor or multiprocessor shared-memory environment. A new architecture-abstraction layer is developed, which allows μProfiler to access hardware counters on multiple CPU types. As well, two new profiling metrics are presented, which use the architecture-abstraction layer to gather hardware-event counts for μC++ programs. These metrics offer performance information about μC++ programs that is unavailable by any other means.

Acknowledgements

First of all, I would like to thank my supervisor Dr. Peter Buhr for his time, guidance, availability and understanding. He taught me more than I can put on paper, and opened my mind to a world of new possibilities, both in and out of the lab. I would like to thank my readers, Dr. Stephen Mann 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 to my lab mates: Richard Bilson, for helping me work through numerous bugs, and for the extra time and effort he spent editing my thesis; Ashif Harji, for his technical assistance with μC++; and Roy Krischer for his interest and suggestions. They all made me feel at home in a new environment, which made my stay at the University of Waterloo an enjoyable one.

Much love and gratitude goes to my family for always encouraging me to reach for the stars and be the best that I can be. I could not have reached this point without their love and support.

Last, but far from least, I would like to thank Jennifer Lasenby for being the most caring and supportive person I know. While I was chained to my desk writing this thesis, she ensured that I did the things I would never take the time to do on my own, such as eat, sleep, and once in a while, even take a break. In short, she kept me sane and healthy during a long period of raging workaholism, and for that I will always be grateful.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Concurrency has been an integral part of computer science since its beginning [Sno92]. Although concurrency has its roots in multitasking/multiprocessing operating systems, it has since evolved into a user-level programming tool that offers solutions to problems in a variety of application domains [Ous96]. User applications currently making use of concurrency include database and web servers, Internet search engines, Java interpreters, web applications, numerical computations, and graphical user interfaces [XMN99].

Many programming languages, such as Ada [U.S00], Java [AGH00], and C# [HWG03], offer native support for user-level concurrency via built-in language constructs. Other (originally sequential) programming languages such as C [KR88] and C++ [Str97] have been extended to introduce support for user-level concurrency, resulting in new dialects like Concurrent C [GR89], μC++ [BDS$^+$92] and pC++ [BBG$^+$93]. Other concurrent extensions of sequential programming languages include Concurrent Pascal [Han75], Multilisp [Hal85] and Concurrent ML [Rep91].

1.1 Performance of Concurrent Programs

While concurrency is a powerful and useful programming tool, writing correct, high-performance concurrent code is extremely difficult because concurrent programs suffer from a variety of potential pitfalls that are not present in sequential programs [JFL98]. These pitfalls include nondeterminism, synchronization, mutual exclusion, context switching, race conditions and deadlock, which affect program performance, correctness, or both.

This thesis is only slightly concerned with the correctness of concurrent programs; instead it focuses on those pitfalls affecting performance. While it is true that understanding performance can aid in establishing correctness, many performance enhancements occur *after* correctness is established. For information on debugging concurrent programs for correctness, the reader can peruse a list of concurrent debuggers [PN93].

Issues affecting concurrent program performance include:

- **Nondeterminism:** Concurrent programs are inherently nondeterministic; threads interact with one another in unpredictable ways [CL00]. While this is mainly a correctness issue, it also indirectly affects performance because its solutions, synchronization and mutual exclusion, can cause bottlenecks (see below).

- **Synchronization:** Synchronization is used when thread interactions need to be made predictable, i.e., operations need to happen in a certain temporal order. This effect is accomplished by blocking one or more threads until the contraints on their execution order have been satisfied. If threads block too often and/or for too long, the overall program may suffer a noticeable degradation in performance.

- **Mutual exclusion:** Threads accessing shared information must protect this information in critical sections. Mutual exclusion is used to restrict the number and kinds of threads occupying a critical section at any given time. Any threads arriving at an occupied critical section must block until the number or kind of threads in that critical section drops below the maximum threshold. If a piece of code protected by a critical section is large and frequently executed, performance can suffer dramatically as threads queue up and await entry.

- **Context switching:** Each time a thread blocks or is preempted, a certain amount of overhead is incurred to save its state and schedule another thread. Though this overhead is typically small for user threads, superfluous context switching due to unnecessary synchronization, excessive mutual exclusion, a poor scheduling algorithm or an inappropriate time-slice value can cause performance degradations.

1.1.1 Locating Performance Problems

Performance tuning effort is often wasted because programmers spend time improving code that minimally affects the overall program performance [AL90]. This problem is especially true for a concurrent program because its additional pitfalls generally make locating problematic sections of code non-intuitive, e.g., context switching does not appear in a program's source code. Thus, the key to alleviating the slowdowns caused by these pitfalls is actually locating them. This process is the primary task of a *profiler*, which is a tool that monitors, analyzes and visualizes the execution performance of a program to help users verify its expected behaviour, and locate its bottlenecks and hotspots.

Expected Behaviour

When a programmer writes software, s/he generally has a mental model of how the completed program will behave at run time. If the program deviates from this expected behaviour, a profiler can be used to help figure out where and why. By studying the execution profile of a program, a programmer may be able to pinpoint areas where the program does not behave according to the expected model.

Bottlenecks

Programs that perform suboptimally often do so because of bottlenecks, which are specific areas where performance degradations occur. A profiler can help isolate these bottlenecks, allowing a programmer to focus performance-tuning effort in areas where it is needed, thus bringing the target program closer to its optimal performance.

Hotspots

Hotspots are areas of a program that are executed frequently in relation to the rest of the program. While they do not necessarily suffer from performance problems, the sheer amount of time spent executing them makes them candidates for optimization. A profiler can help identify hotspots in a program, which are good places for programmers to focus their performance tuning efforts.

1.1.2 Hardware Counters

One of the latest tools that a profiler has at its disposal is a set of *hardware counters*, which are specialized registers in the CPU that have recently been made accessible at the user level by many modern operating systems. Hardware counters count the occurrences of different hardware events such as completed instructions, elapsed CPU cycles, branch mispredictions and cache misses. Hardware-event counts provide extremely precise insight into the run-time behaviour of a program, and can be used to pinpoint portions of code where performance is suboptimal.

1.2 Objectives

The goal of this thesis is to profile concurrent programs using hardware counters. The target environment for this effort is μProfiler, which is a concurrent profiler written in, and tightly integrated with, a concurrent dialect of C++ called μC++. μC++ implements an M:N user-to-kernel thread model, and a run-time library (called the μC++ kernel) to support its threads.

The first objective of this thesis is to extend μProfiler by adding an architecture-abstraction layer that provides a consistent interface for accessing hardware counters across CPU types. This layer must fit the existing μProfiler interface, and define a useful common subset of features, allowing μProfiler to extract information from hardware counters on multiple platforms.

The second objective of this thesis is to use the architecture-abstraction layer to implement a concurrent profiling system that makes use of hardware counters to effectively profile μC++ programs. This profiling system must fit the existing μProfiler framework so that no changes to the existing infrastructure are required. Because of the tight coupling between μC++ and μProfiler, the profiling system must also gather performance information from the μC++ run-time kernel and express results in terms of the μC++ concurrent execution model, i.e., performance data must be expressed on a per-thread basis and be related back to μC++ concurrency constructs and source code.

1.3 Definitions

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

- A **thread** is an independent sequential execution path through a program.
- A **process** is a program in execution [SG98]. It is encapsulated in a separate memory that contains at least one thread and an execution state. Generally, a process is an operating system construct, so its threads are often referred to as *kernel threads*. Kernel threads are scheduled independently by an operating system.
- A **task** is the user-level counterpart of a process. It contains at least one thread and an execution state, but it is generally a programming language construct that shares a common memory with other tasks in the same process. A language's threads are often referred to as *user threads*. User threads are scheduled independently by the language run-time across the kernel threads within a process.

- **Concurrency** is when the executions of multiple threads are rapidly interleaved on a processor so they *appear* to be running at the same time. Concurrency can be achieved on a single CPU, and is thus the logical notion of threads executing simultaneously [BH05].
- **Parallelism** is when multiple threads are *actually* executing at the same time. Since only one thread can be executing on a processor at any given time, true parallelism can only be achieved on multiprocessor systems. Parallelism is thus the physical notion of threads executing simultaneously [BH05].

Note that given the above definitions, there is no such thing as a parallel program. A program is a logical entity, so it cannot exhibit physical behaviour. Therefore, throughout this thesis, multithreaded programs are called concurrent programs with the understanding that they have the *potential* for parallelism if run on a multiprocessor system.

1.4 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 provides a detailed description of profiling.

Chapter 3 presents related work in the field of profiling, introducing one hardware-counter library and seven profiling tools.

Chapter 4 briefly describes the μC++ programming language, which is μProfiler's target environment.

Chapter 5 presents the design and implementation of μProfiler, which is the profiling tool that provides the basis for the contributions of this thesis.

The next three chapters cover the major contributions of this thesis. Chapter 6 explains the hardware-counter related functionality added to μProfiler. Chapter 7 discusses the Exact Hardware Metric, which provides exact hardware-event counts for a profiled program. Chapter 8 explains the Statistical Profiling Metric, which provides a statistical call-graph of a profiled program, based on hardware-counter samples.

Finally, Chapter 9 summarizes the contributions of this thesis and presents possible directions for future work.

Chapter 2

Profiling

A profiler is a tool that monitors, analyzes and visualizes the execution performance of a program to verify expected behaviour, and help programmers locate bottlenecks and hotspots. Ideally, a profiler relates any such findings back to the program's original source code in a high-level way that closely matches the programmer's mental model. This feedback guides the programmer in making changes to the high-level algorithms and data structures of the application [Int04].

Profiling consists of three main steps (see Figure 2.1):

- **Instrumentation insertion:** instrumentation is inserted into a program to allow its run-time behaviour to be monitored.
- **Execution and monitoring of instrumented program:** the instrumented program is run and performance data is gathered.
- **Analysis and visualization:** the performance data is analyzed to extract useful information, which is then visually presented to the user.

The above steps form a crucial "feedback cycle". Based on the visualized performance data, the user may make changes to the instrumentation to refine their understanding of the program's execution, and subsequently change the problematic areas of the program. This cycle repeats until the program is behaving within acceptable performance parameters.

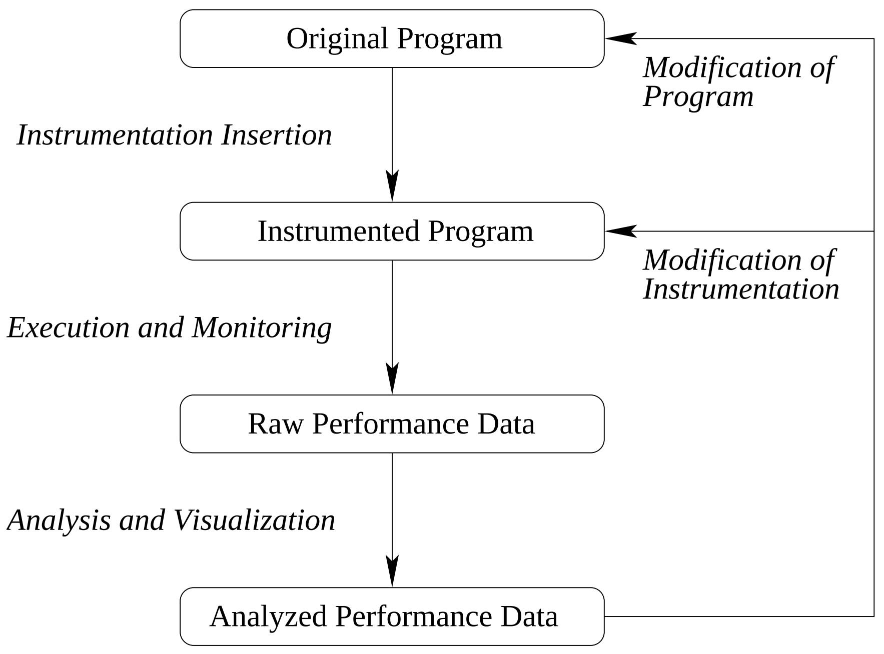

Original Program
Instrumentation Insertion
Instrumented Program
Execution and Monitoring
Raw Performance Data
Analysis and Visualization
Analyzed Performance Data
Modification of Program
Modification of Instrumentation

Figure 2.1: Steps in the profiling process.

2.1 Instrumentation

At the heart of most profilers is the instrumentation insertion phase, where instructions are added to a program to generate run-time performance data. Instrumentation can be broken down into *points*, *primitives* and *predicates* [MCC+95]:

- An **instrumentation point** is a location in a program's code where instrumentation is inserted.
- An **instrumentation primitive** is an operation that gathers performance data for a metric (see Section 2.1.1).
- An **instrumentation predicate** is a boolean expression that guards the execution of an instrumentation primitive (essentially, an if statement).

The combination of an instrumentation predicate and an instrumentation primitive is often referred to as a *hook*.

Probe Effect

The insertion of instrumentation into a program causes an anomaly called the *probe effect*, which can ultimately change the run-time behaviour of that program [LP85, MH89]. In a sequential program, such changes are limited to extraneous delays caused by the execution of instrumentation primitives. Provided the instrumentation itself contains no logic errors, a sequential program still behaves as it did before instrumentation, albeit somewhat slower.

The situation is far more complex for a concurrent program. Because of the inherent nondeterminism that is present in a concurrent program, introducing delays in one thread can affect the behaviour of others, in terms of both correctness and performance. As mentioned, the discussion in this thesis focuses on performance; see [Gai86] for a discussion of the correctness issues caused by the probe effect in concurrent programs.

The probe effect can cause a concurrent program to deviate from its expected performance in a number of ways. Bottlenecks and hotspots in one thread may move to different locations or even disappear altogether as a result of delays experienced by other threads during the execution of instrumentation primitives. In extreme cases, adding instrumentation, which is supposed to help track down bottlenecks, may actually introduce new ones [HM93]. However, a profiler strives to minimize its impact on the program it is profiling, so in practice, probe effects are generally small and program execution is only slightly disturbed.

2.1.1 Instrumentation Primitives and Metrics

There are two basic types of instrumentation primitives: *counters* and *timers* [GKM82, MCC$^+$95]:

- A **counter** keeps track of the number of occurrences of a certain event, such as the number of times a routine is called, or the total number of cache misses. A counter can be implemented in software or hardware.

- A **timer** tracks the amount of time spent performing a certain event, or the amount of time spent in a certain state, such as the time spent executing a routine, or time spent in the blocked state. A timer is implemented in hardware.

Given the nondeterminism of concurrent programs, counts can vary over time, even if the final totals are fixed. Consider a concurrent program that is run multiple times, and during each run, event counts are sampled every three seconds. Almost certainly, the interleaving of the program's threads differs for each run. Thus, event counts at the end of the first time interval are most likely different every time the program is run, as are the event counts for subsequent intervals. However, the totals at the end of each program run may or may not be the same, depending on the type of event being counted. For example, the number of calls to a routine may be constant for each task given the same data, but when the calls occur can vary during the execution of the program.

Metrics

A *metric* is a measurement of some aspect of program performance. It consists of data from one or more instrumentation primitives, and possibly some additional information. For example, a routine-call metric consists of data from two instrumentation primitives (a counter for the number of calls to each routine, and a timer for the total time spent in each routine), as well as additional information such as the routine's name, each routine called, each routine's caller, source-code file location, etc.

2.1.2 Direct and Indirect Instrumentation

Instrumentation is either direct or indirect. *Direct instrumentation* is code placed directly at an instrumentation point (see Figure 2.2). This method entails a lower probe effect than indirect instrumentation, but it also has drawbacks such as code duplication; if the same instrumentation primitive is used multiple times in an

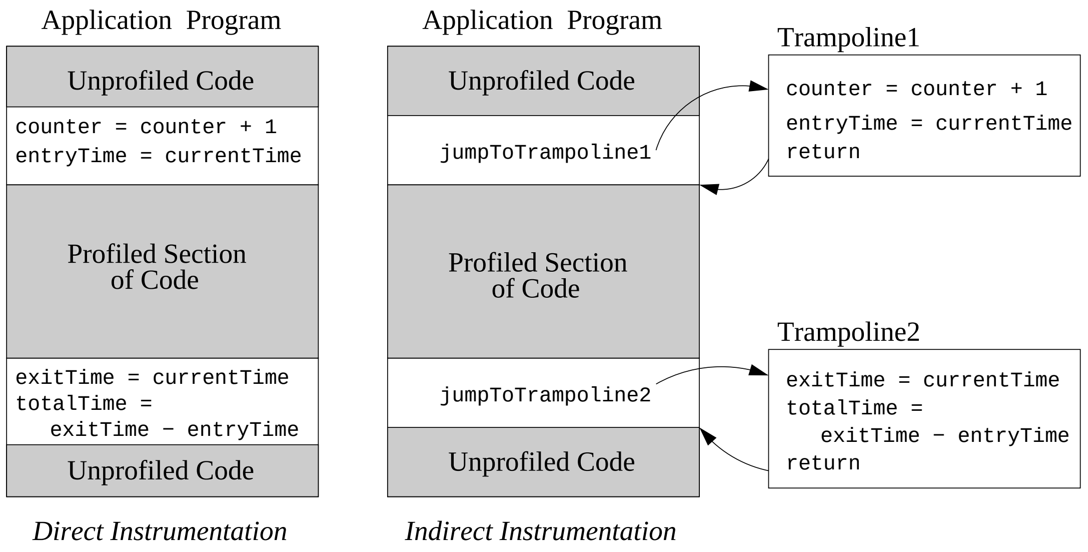


Figure 2.2: Direct and indirect instrumentation.

application, it must be duplicated at every desired instrumentation point. When instrumentation code is substantial, this method can cause a significant increase in code size (called *code bloat*).

Indirect instrumentation replaces the direct code at the instrumentation point with a jump to a *trampoline*. The instrumentation code is placed in the trampoline, which executes and then returns to the instruction following the jump, much like a routine call (see Figure 2.2). The difference from a normal routine call is that the trampoline can often be specially simplified and optimized to reduce the probe effect of the indirection. These trampolines allow jumps to the same instrumentation to be inserted through a program without significant code bloat. As well, this method modularizes the instrumentation, which allows for techniques like static and dynamic insertion and modification.

2.1.3 Instrumentation Insertion

Instrumentation insertion can be done at almost any point during the writing, compilation or execution of a program (see Figure 2.3). As the insertion point moves

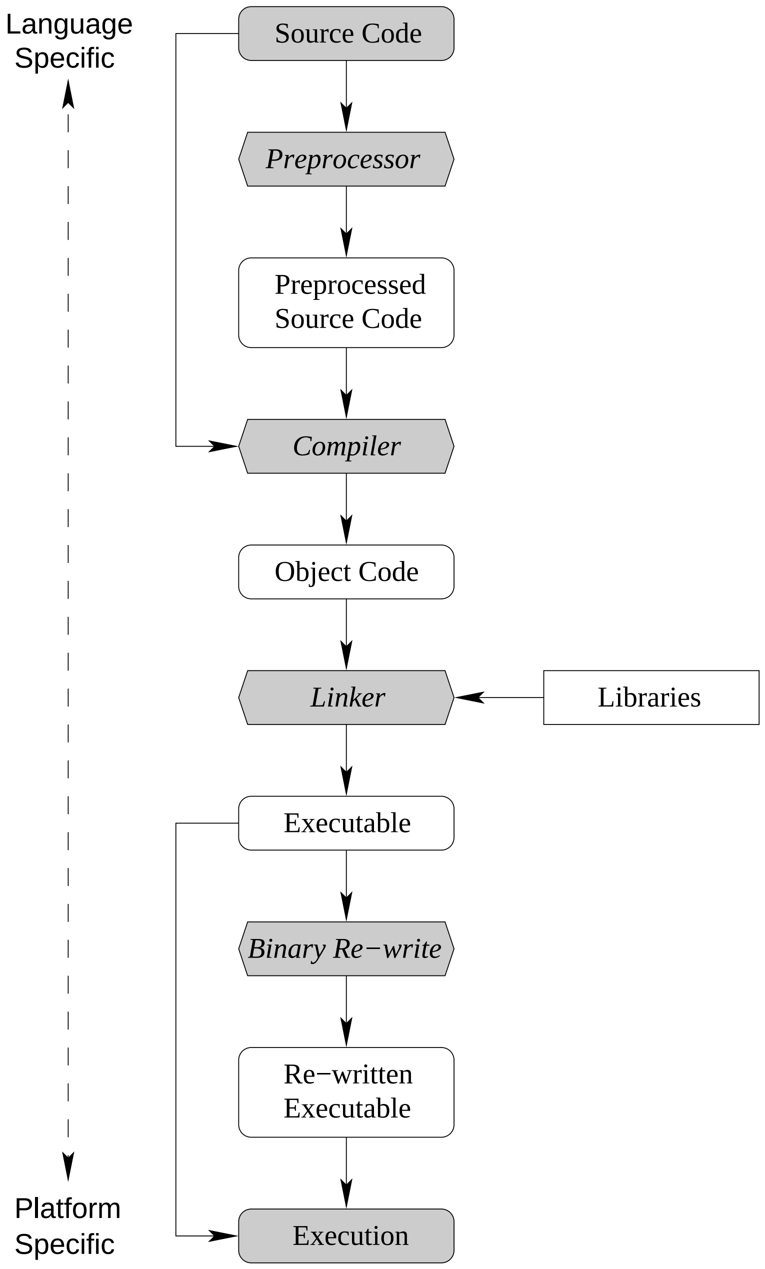


Figure 2.3: Compilation/execution chain (possible insertion points are shaded).

down the compilation/execution chain, the instrumentation goes from language-specific to platform-specific [She99]. For example, instrumentation that is inserted at the source-code level may be portable across all platforms that support the programming language being used. However, the instrumentation is most likely incompatible with other languages. On the other hand, instrumentation that is inserted at the machine-code level is independent of the program's source language, but it does not work on more than one architecture.

Instrumentation insertion is usually divided into *static* and *dynamic* insertion.

Static Insertion

Static instrumentation insertion is performed at any point *before* the execution of a program [Zak00]. Most often, it is done during the writing, compiling or linking stage, but it can also be done afterwards by changing the executable directly in a process called *binary-rewriting*.

The majority of profilers use static insertion in the form of indirect instrumentation via shared trampolines. The advantage of static insertion is that it provides information about a profiled program that is difficult to obtain by any other means, such as the number of traversals through a code segment or the callers of a particular routine [Den97]. It is also generally an easier task to insert instrumentation statically than it is to do it dynamically.

The disadvantage of static instrumentation is that once it is in place, it cannot be removed without a recompile or binary re-write. Thus, if an instrumented section of code is not helping to locate a bottleneck or hotspot, unnecessary performance data is generated and a higher probe effect needlessly occurs. This extra overhead can be minimized, though not eliminated, by using instrumentation predicates to disable unwanted instrumentation (see Section 2.1), although predicates also have a cost and effect.

Dynamic Insertion

Dynamic instrumentation [Hol94] is done during the execution phase of a program. Usually, the dynamic instrumentation process is handled automatically by a profiler because the speed at which computers operate makes it nearly impossible for a human to do it effectively.

While the target program is running, the profiler decides where and when to insert or remove instrumentation. If an instrumented code-section is found not to be a bottleneck or hotspot, its instrumentation can be removed. In this way, the profiler can check multiple sections of a program looking for bottlenecks and hotspots, which may move during execution, especially if the profiled program is a concurrent program.

Deciding where and when to insert or remove instrumentation is an iterative process and can be quite time-consuming. For this reason, dynamic instrumentation is usually suitable only for long-running programs.

No Instrumentation

A program can be profiled without any instrumentation at all. In this case, performance data is gathered by occasionally sampling the execution state of the program (see Section 2.2.2).

2.1.4 Hardware Counters and Instrumentation

Hardware counters are unique in that they provide useful information at virtually no cost, and hence, have a low probe effect. Once the hardware counters are configured, no further instrumentation needs to be inserted into a profiled program to count hardware events during execution. The hardware counters simply run in parallel with the executing program at the hardware level. There is still the cost of reading from or writing to the hardware counters and storing necessary information for a metric. Nevertheless, complex information can be gathered at low cost, and some of this information, such as cache misses or missed branch predictions, cannot be gathered in any other way.

2.2 Monitoring

Monitoring is the process of gathering, filtering and storing performance data during a program's execution. Filtering is an optional step in this process, which immediately discards irrelevant performance data, thus reducing the amount of memory required to store the data that is kept. A filtering decision during monitoring is generally a "local" decision, i.e., it is based solely on the value of the information being considered for rejection. For example, a metric that gathers program-counter values may only be interested in addresses that lie within a certain range. In this case, the decision to keep or discard a program-counter value is based solely on whether or not it lies within the desired range.

When monitoring a sequential program, it is usually enough to simply gather and store performance data as an aggregate. However, the situation is more complicated for a concurrent program. In this case, an aggregation of performance data is not particularly useful, as it most likely does not fit a user's task-based execution model. Therefore, when monitoring a concurrent program, performance data should either be collected and stored on a per-task basis, or the data should be marked according to the task that generates it, so that it can be separated into task-specific groups during the analysis phase.

There are two different kinds of monitoring: *exact* and *statistical*, each of which is discussed below.

2.2.1 Exact Monitoring

Exact monitoring (also called event-driven monitoring) captures *all* occurrences of events registered by active metrics. The profiling monitor acts as a passive entity in this case, waiting for instrumentation primitives to be triggered as the profiled

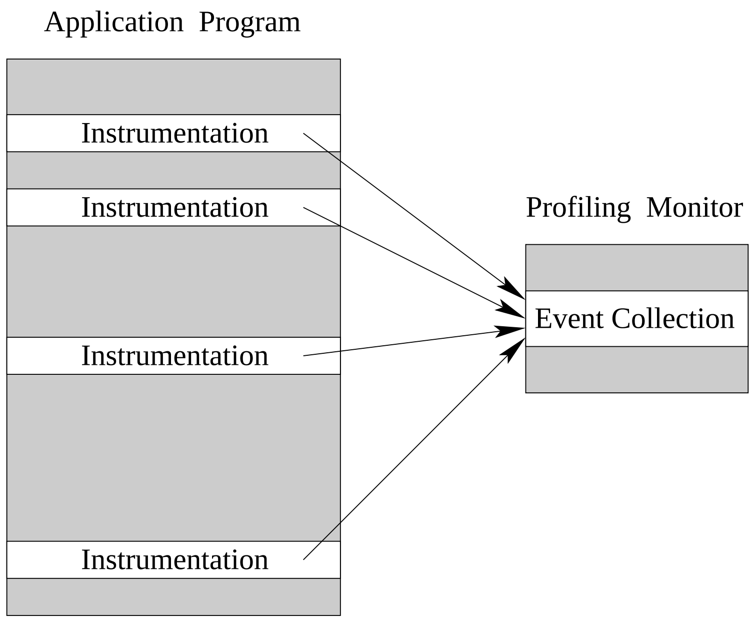


Figure 2.4: Exact monitoring.

program executes (see Figure 2.4). This type of monitoring is extremely precise and is used when a complete trace of a program's execution is needed, e.g., for a complete dynamic call-graph.

On the other hand, capturing all events incurs a penalty. While it provides a high degree of precision, it also creates a large probe effect. For each event that is triggered, a cost is introduced while it is processed, which can change the behaviour of the profiled program. Furthermore, exact monitoring has the potential to generate huge amounts of performance data. If a large number of events is being monitored, hundreds of megabytes of data can be produced within minutes. The amount of data collected can be reduced by disabling unnecessary static instrumentation through the use of predicates, or removing unwanted dynamic instrumentation. As mentioned, the latter approach takes time and is thus only suitable for long-running programs. Moreover, by the time a user or profiler makes a final decision as to what instrumentation to use, a large amount of performance data has likely been collected. Thus, exact monitoring is typically used to garner a precise event trace for a short-running application or for a short segment of a long-running

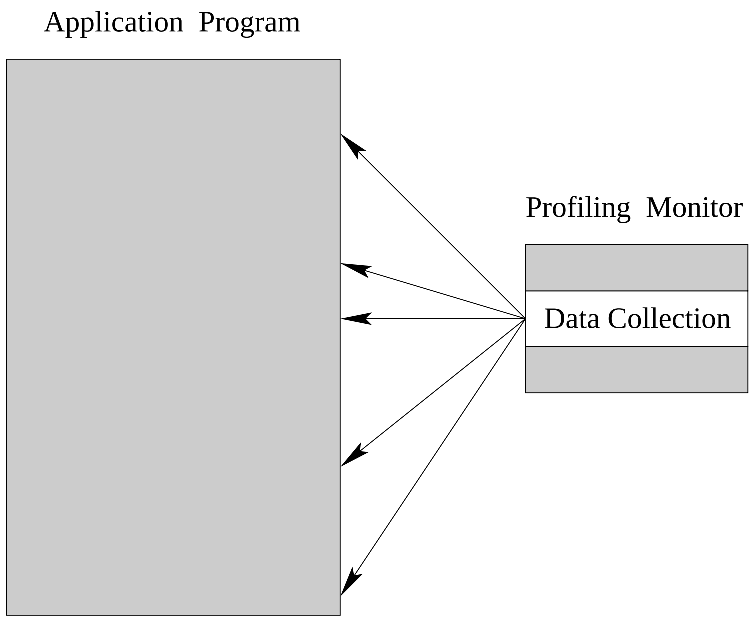


Figure 2.5: Statistical monitoring.

application.

2.2.2 Statistical Monitoring

Statistical monitoring (also called polling or sampling) provides a lower-cost alternative to exact monitoring. The performance data it produces is less precise, but the probe effect and the amount of performance data collected are also much smaller. In statistical monitoring, the profiling monitor is an active entity; it *polls* the running program at specified intervals (called sampling periods) and records various information from its execution state, such as the program counter value, the currently executing routine, and in the case of a concurrent program, the current task (see Figure 2.5). Traditionally, the sampling period is based on time, e.g., every ten milliseconds, but it can now be based on the occurrence of hardware events as well (see Section 2.2.3).

Statistical monitoring offers the user a tradeoff between precision and cost: a small sampling period generates more precise performance data, but entails a higher

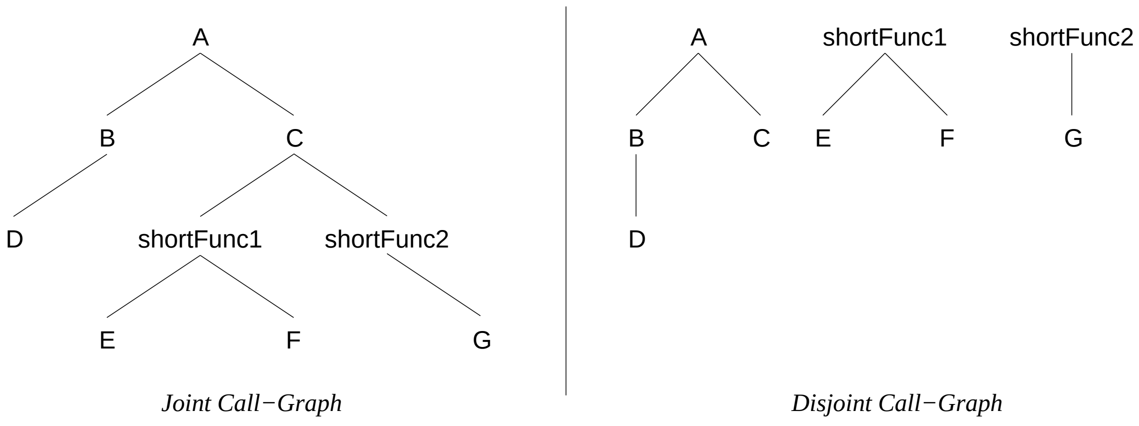


Figure 2.6: Two possible call-graph results when using statistical profiling.

probe effect. With a well-chosen sampling period, statistical monitoring provides a reasonable amount of information, and incurs only a small overhead. However, it is inappropriate in cases where precision and complete event coverage are paramount. For example, if statistical monitoring is used to build a call-graph, and each sample consists of the currently executing routine and its parent, the result could be a disjoint call-graph. In Figure 2.6, shortFunc1 and shortFunc2 are relatively short routines; if the sampling period is too large, a statistical profile may never poll the target program at times when these routines are executing, so the parents of routines shortFunc1 and shortFunc2 may not be recorded. As a result, shortFunc1 and shortFunc2 are shown as roots of their own subtrees, when in fact they should be children of C.

2.2.3 Hardware Counters and Monitoring

Hardware counters are useful for both exact and statistical monitoring. If an exact hardware-count is needed for a section of code, instrumentation to read the hardware counters is inserted at the entry and exit points of the section. Determining the number of events that occur during the execution of the section is then simply a matter of taking the difference between the entry and exit values. Note that for a concurrent program, hardware-event counts must also be saved on context switches.

In the case of statistical monitoring, the hardware counters are used to create a sampling period based on the occurrence of a chosen number of events. Hardware counters count from 0 to $2^w - 1$, where w is the architecture-dependent width of the counters, in bits. Upon overflow, a signal is delivered. To create a sampling period of n events, a hardware counter is set to a value of $2^w - n$. After the n^{th} event occurs, the counter overflows and a signal is generated. The overflow signal-handler then polls the target program and resets the counter to $2^w - n$.

2.3 Analysis

Performance data must be analyzed to extract useful information about a program's behaviour. The analysis process involves:

- Filtering to remove extraneous information and reduce the relevant data down to an appropriate and manageable subset,
- Performing calculations on the raw performance data to derive human-readable information,
- Relating the derived information back to the program's source code, if possible, and
- Preparing the derived information for visualization, which may include preparing a summary view as well as a more detailed view.

As is the case during monitoring, filtering during analysis is an optional step. A filtering decision during analysis tends to be a "global" decision, i.e., it is made based on all the available performance data. Returning to the example of the program-counter metric, assume it is interested in tallying a unique list of all the program-counter values that were gathered during program execution. In this case, the decision to keep or discard a program-counter value is based on whether or not it already exists in the list.

Analysis is more intricate if the performance data is the result of profiling a concurrent program rather than a sequential program, for three reasons. First, if it has not already been done during the monitoring phase, the data must be separated into groups according to which task was active when the performance data was generated. Second, the data from each task must be analyzed separately to relate bottlenecks and hotspots back to the high-level language constructs of which a user's execution model is comprised. Finally, the data from separate tasks should be compared so that performance problems due to their interactions can be discovered.

Analysis can be done *on-the-fly*, *post-mortem*, or with a combination thereof.

2.3.1 On-The-Fly Analysis

On-the-fly analysis is done while a target program is executing, which has several advantages. The first advantage is that it offers a second opportunity during a target program's execution to filter extraneous performance information, meaning less data has to be stored during profiling. The second advantage of on-the-fly analysis is that it can give immediate feedback to the user and/or profiler, allowing either one to adjust the instrumentation by inserting or removing it as necessary (if dynamic instrumentation insertion is being used), and/or turning instrumentation on or off via instrumentation predicates.

A disadvantage of on-the-fly analysis is that analyzing during program execution results in a higher probe effect. Also, if there is an overly large amount of performance data and/or the CPU is slow, there may be a noticeable lag between the occurrence of an event and its subsequent analysis and visualization, making adjustment of the instrumentation difficult or impossible. Finally, analysis, visualization, and any ensuing instrumentation adjustment takes time. Therefore, on-the-fly analysis of short-running programs is difficult for a profiler and next to impossible for a user. It is better suited for long-running programs.

2.3.2 Post-Mortem Analysis

Post-mortem analysis is done after a target program has terminated. Since no analysis of performance data occurs during program execution, it contributes nothing to the probe effect. However, this means that dynamic adjustment of instrumentation is impossible because there is no feedback available. Also, all performance data must be retained until execution has finished. As has already been mentioned, large amounts of profiling data can become quite a problem. Therefore, post-mortem analysis is better suited for short-running programs.

2.3.3 Combination

On-the-fly and post-mortem analysis can also be used together. For example, performance data can be analyzed and displayed on-the-fly, and also saved to disk for further post-mortem analysis. This technique is useful for replaying the execution of a nondeterministic concurrent program in a deterministic fashion [RBC+03].

2.4 Visualization

Visualization is the displaying of performance data on screen in a human-readable fashion. It is arguably the most important step in profiling, as it guides the programmer in making decisions towards improving the performance of a program. The performance data must be clear, concise, and it should convey key points at a single glace. After all, an analysis whose results cannot be understood is no better than no analysis at all [Jai91].

As is the case with monitoring and analysis, visualization is done differently when it displays performance data from a concurrent program as opposed to a sequential program. As usual, performance data should be presented in terms of a user's high-level execution model, so each group of per-task information should be visualized separately. However, because visualization should quickly convey key points at a single glance, a terse summary of each task should first be presented on a single screen. The user should then be offered a choice as to which tasks should be examined in greater detail.

Performance data can be displayed using a number of different visualization media, including tables, charts and graphs.

Tables

Tables display discrete values in a row-and-column format. They are the simplest visualization technique available, and are most often used to display a small amount of raw numerical data.

Charts

Charts also display discrete values, but in a pictorial fashion. Examples are bar charts, pie charts, histograms and Gantt charts. Histograms display the frequencies or number of occurrences of different values of a single parameter. For example, the profiling tool Tmon uses histograms to graphically depict the frequencies of different ready-queue lengths [JFL98]. See [Jai91] for an explanation and an example of Gantt charts for profiling.

Graphs

Graphs are the most complex of the three visualization media, consisting of points, lines and surfaces that represent multi-dimensional relations [Den97]. They are used to display relationships among metric data that would not be easily discernable with tables and charts.

Graphs are often customized towards specific metrics. For example, the profiling tool IPS [YM88] breaks a distributed-program's execution into non-overlapping individual jobs, which it refers to as *activities*. Many of these activities have precedence relationships, meaning some activities must finish before others can start. These relationships are displayed in the form of a *program activity graph*, which is

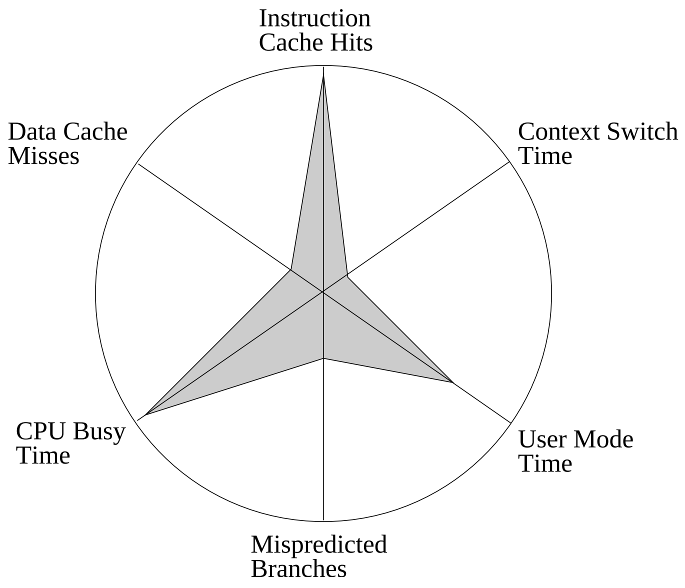


Figure 2.7: A Kiviat graph with all metrics performing well.

“a weighted, and directed multigraph, that represents program activities and their precedence relationship during a program’s execution” [YM88].

An example of a graph in common use is a Kiviat graph, which is able to display multiple metrics in one picture. A Kiviat graph is depicted as a circle with an even number of radial lines, each representing a different metric. Metrics for which higher values are considered better alternate around the circle with metrics for which lower values are better (see Figure 2.7). Each metric has a point on its radial line, where larger values are farther from the centre of the circle. When the points are connected to their neighbours, a closed polygon is formed. In an ideal situation where all metrics are performing well, the polygon is an N-pointed star, where N is the number of “higher-is-better” metrics. Any metric that is not performing well is easy to detect, as the star is deformed on that metric’s radial line.

For more information on properly displaying quantitative data, refer to [Tuf83].

Chapter 3

Related Work

Hardware counters are used extensively in today's profiling tools. This chapter introduces a number of such tools, including a portable library for accessing hardware counters on many different architectures, as well as seven profilers that support (or are in the process of adding support for) hardware counters.

3.1 PAPI

The PAPI (Performance Application Programming Interface) library specifies a standard set of hardware events and a standard interface (offered in both C and Fortran) for accessing hardware counters on multiple platforms [BDG$^+$00]. Calls to this library can be inserted into the source code of C and Fortran programs to gather performance data from the hardware counters.

3.1.1 Design and Architecture

The PAPI library consists of two layers. The first layer is a portable, machine-independent layer that exposes a high-level interface and a low-level interface to the hardware counters. These interfaces vary in terms of complexity and functionality; each targets a different type of user, as is explained below.

The second layer, called the *substrate*, contains all of the library's architecture-dependent code. One substrate per supported platform is needed to access the hardware counters, which is necessarily an architecture-dependent undertaking.

Architecture-Independent Interface Layer

PAPI's architecture-independent interface is broken into two parts: a high-level interface, and a low-level interface. The high-level interface provides basic routines to start, stop and read the underlying hardware counters. It is designed to allow users to quickly and easily obtain simple performance data.

The low-level interface is geared towards experienced application programmers and tool developers who need more control over the PAPI library and the hardware counters. For example, the low-level interface provides information about the hardware being used (such as the clock rate in MHz and the number of CPUs) and the executable being profiled (such as the addresses of the text and data sections).

The low-level interface also provides routines for multiplexing events and counter overflow notification. Multiplexing is used to count more events than there are hardware counters on the underlying CPU. To this end, PAPI defines the notion of an *event set*, which is simply a group of events to be counted at the same time. To count a large number of events, a user divides these events into event sets, each containing an amount less than or equal to the number of available counters. During execution of a target program, PAPI multiplexes these event sets by reprogramming the hardware counters every 25000 clock cycles. It does so, however, at a loss of precision. Since no one event is counted throughout the entire program run, only an estimate of the total number of occurrences of each event is generated. The more events that are multiplexed, the more statistical the final results. Finally, the constant swapping of event sets incurs some overhead, which adds to the probe effect.

As mentioned in Section 2.2.3, counter overflow is used for statistical profiling based on hardware events. PAPI supports statistical profiling by allowing users to register callbacks to be activated from within a signal context upon counter overflow. When one or more counters overflow, all registered callbacks are invoked and each one is provided with information via routine arguments, such as a reference to the event set in use, the address of the program counter when overflow occurred, and an overflow vector specifying which counters overflowed.

Architecture-Dependent Substrate Layer

For every platform supported by PAPI, an architecture-dependent substrate layer is written, which is the code that actually accesses and manipulates the machine's hardware counters. The substrate uses whatever method is most appropriate for accessing the underlying counters, whether that is system calls, calls to another library, assembly language, or some other method.

3.2 Paradyn

Paradyn [MCC+95] is the most advanced tool available for profiling large-scale concurrent and distributed programs. It is capable of profiling programs that run for hours or days on large parallel machines (consisting of thousands of nodes), manipulating large data sets. Paradyn does not require target programs to be instrumented. It inserts instrumentation dynamically and uses an automated, top-down search algorithm to isolate bottlenecks (see Section 3.2.2). Paradyn only relates to the objectives laid out at the beginning of this thesis in its ability to profile concurrent programs and visualize performance data in terms of some high-level concurrent language constructs. In fact, the majority of Paradyn's design objectives are largely the antithesis of this work, but are presented as a contrast and because Paradyn is the most pervasive profiler in the literature.

Paradyn consists of the main Paradyn process, one or more Paradyn daemons, and zero or more external visualization processes (called *visis*). The main Paradyn process is multithreaded, and consists of the following threads:

- A **Performance Consultant** that searches for bottlenecks in the target program by requesting and receiving performance data from the Data Manager.

- A **Data Manager** that is responsible for delivering performance data from the Paradyn daemons to the Performance Consultant.
- A **User Interface Manager** that displays Paradyn's main controls and performance data in a graphical fashion.
- A **Visualization Manager** that creates visis and manages their "Visi Threads".
- Zero or more **Visi Threads**, one for each visi. Each Visi Thread handles communication between its visi and the main Paradyn process.

The Paradyn daemons contain all of the architecture-dependent code, and are responsible for inserting, modifying and removing dynamic instrumentation in the executing target program, as requested by the Performance Consultant. Each daemon consists of a Metric Manager and an Instrumentation Manager, whose functions are explained in Section 3.2.1.

Visis are responsible for visualizing performance data. Paradyn provides a standard set of visis, including time-histograms, bar charts, and tables, but it is straightforward to build visis that use visualization displays from other systems, such as ParaGraph [HE91] and Pablo [RRA$^+$93].

3.2.1 Instrumentation and Monitoring

Paradyn uses dynamic instrumentation to profile a target program, meaning instrumentation insertion, modification and deletion are done at run-time. This process can be handled by the user, but is usually done automatically by Paradyn itself. In this way, only those parts of the program that are relevant to finding bottlenecks are instrumented [HMC94].

The Performance Consultant delivers instrumentation requests to the Paradyn daemons, which translate these requests into machine code during a two-step translation process. First, the daemon's Metric Manager translates the instrumentation request into an intermediate, architecture-independent representation called the

Metric Description Language (MDL) [HMG+97]. Second, the daemon's Instrumentation Manager translates the MDL code into architecture-dependent machine code, which contains the primitives and predicates needed to update the values of active metrics. This machine code is then inserted into the target program in the form of trampolines.

Two types of trampolines are used in Paradyn, *base-trampolines* and *mini-trampolines*. There is one base-trampoline for each active instrumentation point. A base-trampoline is inserted into a program by replacing one or more instructions at an instrumentation point with a jump to the trampoline, and relocating the replaced instructions to the trampoline itself. Jumps to one or more mini-trampolines are then inserted into the base-trampoline either before or after the relocated instructions. Each mini-trampoline contains machine code for one primitive or predicate.

Once instrumentation is inserted into the target program, profiling actually begins. The counter and timer primitives in the mini-trampolines are sampled periodically by the Paradyn daemons for analysis and visualization. Note that the instrumentation primitives keep precise counts and times for all active metrics, and the sampling period determines only how often Paradyn receives updated values.

3.2.2 Analysis

The Performance Consultant uses a well-defined model, called the W^3 *Search Model* [HM93] to automate its search for bottlenecks. This model defines a search space that the Performance Consultant searches in an attempt to locate *why*, *where*, and *when* performance problems arise in the target program.

The Performance Consultant searches the W^3 search space using a top-down approach, beginning with a set of high-level hypotheses, each representing a class of potential performance problems. To test these hypotheses, the Performance Consultant requests the insertion of a small amount of instrumentation, and analyzes the resulting performance data. Based on this analysis, the hypotheses are revised, and more detailed instrumentation is requested so they can be tested. No detailed instrumentation is inserted for problems that do not appear to exist in the target program. Hypotheses are tested and refined until the Performance Consultant is able to isolate the location and cause of as many bottlenecks as possible.

3.2.3 Visualization

All visualization in Paradyn is handled by visis, which are external processes that display the results of performance metrics. A visi is created when a user requests one from Paradyn's main menu. At that point, Paradyn offers a list of foci (program components) and metrics that the new visi is capable of displaying, from which the user makes a selection. The visi is then started and sent the selected foci and/or metrics.

Once running, a visi notifies the Data Manager of the performance data it requires to fulfill the user's request. If the required performance data has not already been requested by the Performance Consultant, the Data Manager adds instrumentation to the target program to generate it. With all the necessary instrumentation in place, the Data Manager begins passing the requested performance data to each visi at every sampling period. The visis visualize this data immediately, providing the user with on-the-fly feedback.

3.2.4 Hardware Counters

Some initial work with hardware counters has been done, but that code has not officially been added to Paradyn. Thus, as of this writing, none of Paradyn's metrics currently make use of hardware counters. The Paradyn team is currently experimenting with the PAPI hardware counter library, and plans to add support for it in a future release.

3.3 q-tools

q-tools [qto] is a collection of Linux-specific performance analysis tools, and the major ones are qprof, q-syscollect, q-view, and q-dot.

3.3.1 qprof

qprof [qpr] is a simple command-line-based statistical-profiling tool. It uses a polling mechanism to gather information from a target program without requiring any instrumentation. Before enabling qprof, the user chooses the sampling period, which then remains fixed throughout the execution of the target program. After each sampling period, qprof simply records the value of the program counter (PC). Upon program completion, qprof uses these PC values to give a “flat” statistical summary of where the target program spends its time. The manner in which the statistical summary is presented can be selected by the user. If the target program is compiled with debugging information, the samples can be presented in terms of instruction address, source-code line, or routine name, along with the source-code file-names. Otherwise, the samples can only be displayed in terms of instruction address or routine name.

When reporting the amount of time spent in a routine, qprof does not normally include the time spent in that routine’s callees. However, qprof is able to include information about each routine’s callees on systems that support the libunwind library [lib], which is a portable C API that determines the call-chain of a program.

When run on an Intel Itanium machine, qprof can use hardware events instead of time to determine when to poll a target program. The user selects one hardware event to count, and the number of occurrences of this event that should expire before polling. The resulting statistical summary shows a breakdown of where in the target program these events tend to occur.

qprof supports profiling of applications with multiple processes, which is accomplished by generating a separate statistical summary for each subprocess spawned by the application. qprof also supports profiling of concurrent applications at the kernel-thread level, although in this case only one aggregate summary, representing all threads, is presented. Finally, qprof supports profiling of dynamically-linked code. The statistical summary for any target program includes information on the time spent in dynamic libraries. However, Linux kernel routines are not profiled separately; time spent in the kernel is charged to the routine making the system call.

Relation to Thesis Objectives

qprof touches on a number of the objectives this thesis attempts to accomplish. It is a statistical-profiling tool that uses hardware counters on Itanium systems to statistically profile a target program, and it presents a flat, histogram-like summary of the distribution of events among its routines or instructions. qprof is also able to profile concurrent programs at the process level.

However, qprof does not make use of hardware counters on any machines other than the Itanium. It is also unable to gather and display performance data on a per-thread basis, and does not offer any exact hardware-counter metrics.

3.3.2 q-syscollect/q-view/q-dot

q-syscollect, q-view, and q-dot are different component tools of the same statistical profiler. q-syscollect does the monitoring, q-view is responsible for analysis and textual visualization, and q-dot is an optional component that handles graphical visualization.

q-syscollect is a command-line-based systemwide statistical-monitoring tool for Itanium 2 machines running Linux 2.6 kernels. It uses a general-purpose hardware counter, as well as the Itanium-specific Branch Trace Buffer (BTB), to gather histogram and call-graph information for *all* programs running on a system during a user-definable time period. The BTB is a set of eight specialized hardware counters that record the source and destination addresses of user-selectable types of branch instructions. The BTB acts like a ring buffer, so at any given time, it contains information on the four most recently executed branch instructions.

For every CPU on a system, q-syscollect programs one hardware counter to count a user-selectable hardware event (the default is CPU cycles), and to overflow based on a user-chosen sampling period. It also sets up the BTB on each CPU to record `return` branch instructions. Each time a "sampling" counter on a CPU overflows, the program counter is sampled, as are the addresses in the BTB. This sampling information is used by q-view to build a histogram and statistical call-graph.

q-syscollect creates three text files for each process it samples on each CPU. Thus, a process that executes on more than one CPU while q-syscollect monitors a system will have multiple sets of text files. The first file is a .info file, which contains general profile information in a Scheme-like syntax, readable by q-view. The second file is a .hist file, which is a simple list of program counter values and the number of times each was sampled. Finally, there is a .edge file, which contains a list of edges, i.e., source and destination addresses from a BTB, for a program's call-graph. Note that since q-syscollect is a statistical profiler, the call-graph information can be disjoint and/or incomplete.

q-syscollect does not do any visualization itself, but instead relies on q-view to analyze and display its profiles in a human-readable format. q-view is a Scheme script that processes q-syscollect's profile files and visualizes the resulting performance data in a text-based gprof-like [GKM82] format. The output is separated into a histogram section and a call-graph section. The histogram section has one line of text per sampled routine. Each line includes the name of a sampled routine, the percentage of the total time (or events, if CPU cycles are not used) occurring in that routine, the number of seconds (or events) spent in that routine, and the number of calls to that routine.

The call-graph portion of the output is split into sections (one per sampled routine). Each section has one line displaying its routine's name, the percentage of the total time/events spent in that routine *and its children*, the number of seconds/events spent only in that routine, and the number of seconds/events spent in that routine's children. Every section of the call-graph also has one line for each of its routine's parents and children. Each of these lines contains the name of a parent/child routine, the number of seconds/events spent in it, and the number of calls made to it.

q-dot is a tool for graphically visualizing call-graphs produced by q-syscollect. Much like q-view, it processes q-syscollect's profile files, but instead of displaying a text-based call-graph, it creates a graphical version of the resulting call-graph in a "dot" file [dot]. This dot file can then be converted to a common graphical file format, such as PostScript [Ado99].

Relation to Thesis Objectives

q-syscollect relates to the objectives laid out at the beginning of this thesis, as it is a hardware-counter-based, statistical profiler, capable of profiling short-running, concurrent applications. However, there is also a number of areas in which q-syscollect falls short of the objectives. For example, it does not offer an exact-profiling mode. It is also incapable of utilizing hardware counters on any systems other than the Itanium 2. Additionally, while it is capable of profiling short-running applications, its focus is on the system as a whole; it is impossible to profile only one application at a time. Finally, its concurrent profiling abilities are limited to the process level; no information is collected or presented on a per-user-thread basis. Furthermore, process-level performance data may be separated into multiple streams if a process executes on more than one CPU during a profiling session. In this case, it is up to the end user to combine these streams from multiple CPUs for a single process in a meaningful way.

3.4 Other Profiling Tools

Other profiling tools with hardware-counter support include OProfile, SvPablo, and TAU. Each of these profilers achieves some, but not all, of the objectives enumerated at the beginning of this thesis.

OProfile

OProfile [opr] is a systemwide statistical profiler for Linux systems, consisting of a kernel module, a daemon for transparently collecting performance data, and a collection of post-mortem analysis tools. Profiling is done via PC sampling, much like qprof. OProfile is able to profile *all* running code, including hardware and software interrupt handlers, kernel modules, the kernel itself, shared libraries, and regular application code, all without any instrumentation or special recompilation. However, OProfile does offer some visualization options, such as annotated source trees, that require compilation with debugging symbols enabled. On x86 machines running 2.6 Linux kernels, OProfile also provides gprof-like call-graph visualization.

On systems that support hardware counters, OProfile's sampling period is based on the occurrence of a user-configurable number and type of hardware event. Otherwise, sampling is done according to timer interrupts, with the added restriction that sections of the kernel with interrupts disabled cannot be profiled.

OProfile provides much of the same functionality as q-syscollect, although it supports hardware counters on multiple architectures, rather than just the Itanium 2. Still, it is a systemwide profiler, it does not offer any exact profiling metrics, and it offers only limited thread support.

SvPablo

SvPablo [DR99] is a statistical performance analysis and visualization system which supports programs written in C, Fortran 77/90, and High Performance Fortran (HPF) on a variety of sequential and parallel systems. Hardware counters can only be used if SvPablo is run on a MIPS R10000 system. Both automatic and interactive instrumentation of target programs is supported. The former is done via a compiler; SvPablo is integrated with Portland Group's HPF compiler, which inserts calls to SvPablo into a program as it is being compiled. The automatically-inserted instrumentation gathers statistics for each executable line in the original program, as well as routine entries and exits. Interactive instrumentation is done using SvPablo's graphical source-code browser. Instrumentation inserted in this manner is restricted to outer loops and routine calls.

SvPablo summarizes performance data as it is collected during program execution, making it suitable for long-running programs. Performance data is also written to disk in Pablo's Self-Defining Data Format (SDDF) [Ayd03], which is an architecture-independent meta-format. SDDF files can be examined post-mortem using the aforementioned source-code browser. This powerful graphical environment displays the statistical performance data in the context of the program's original source-code constructs.

SvPablo falls short of the thesis objectives as it only supports hardware counters on one architecture, it offers no exact profiling metrics, and it does not profile on a per-user-thread basis.

TAU

TAU (**T**uning and **A**nalysis **U**tilities) [TAU04] is a performance analysis environment for OpenMP and MPI concurrent programs written in C, C++, Fortran 77/90, HPF, Python and Java. TAU supports both statistical profiling (which it refers to as profiling), and exact profiling (which it calls tracing). Profiling gathers summary statistics for metrics such as CPU time in a routine or the number of calls to a routine. TAU supports the use of hardware counters on all its target platforms, so profiling can also summarize the occurrences of hardware events and relate them back to the source code. Tracing captures all the occurrences of events of interest, showing when and where they happened. Hardware counters are not used for tracing.

Instrumentation in TAU is done by adding macros directly to the source code of a target program. This step can be done manually by inserting calls to the TAU API at all desired instrumentation points, or it can be done automatically using a variety of language-specific TAU instrumentation tools. TAU provides tools for visualization, which is done post-mortem, either textually or graphically. pprof is a text-based, gprof-like visualization tool for displaying profile (statistical) information. The same data can be visualized graphically in terms of histograms and text displays with paraprof, which is simply a GUI for pprof. Trace (exact) performance data is visualized using a third-party tool called VAMPIR [Gmb98], which is a performance analysis and visualization tool for MPI concurrent programs.

TAU is the only profiler in this chapter to offer both exact and statistical profiling. However, only the statistical side of this tool makes use of hardware counters. Further, its target environment is OpenMP and MPI programs, while this thesis focuses on μC++ programs.

SBT

SBT (Stupid Barrier Tricks) [NL01, Nov02] is a library for performance monitoring of shared-memory concurrent programs written in C and C++ with POSIX threads or SGI Irix's sproc threads. It is based on the notion that concurrent programs often use barriers to delimit different phases of execution. Barriers are synchronization points between these phases; no thread is allowed to pass a barrier until all threads in the program reach it. SBT allows users to watch one barrier as a program executes, and exact per-thread performance data is visualized on-the-fly as threads reach that barrier.

SBT uses the Performance Counter Library (PCL) [BM03] to access hardware counters on multiple platforms. However, because PCL is not thread safe, hardware counts can only be displayed for one thread during a program run. For this reason, the SBT team is considering supporting PAPI.

SBT relates to the objectives of this thesis in that it provides access to hardware counters on multiple platforms, and provides exact performance data on a per-thread basis. However, it offers no statistical profiling metrics.

3.5 Summary of Related Profilers

Table 3.1 summarizes the relevant features of μProfiler and the seven profilers introduced in this chapter. From left to right, the columns represent the following features:

1. Hardware-counter support.
2. Hardware-counter support on multiple architectures.

	HW Counters	HW Counters on Multi Archs	Exact Profiling	Statistical Profiling	Per-Thread Data
μProfiler	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
Paradyn	?	?		$\surd$	
qprof	$\surd$			$\surd$	
q-syscollect	$\surd$			$\surd$	
OProfile	$\surd$	$\surd$		$\surd$	
SvPablo	$\surd$			$\surd$	
TAU	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
SBT	$\surd$	$\surd$	$\surd$		$\surd$

Table 3.1: Summary of related profilers.

3. Exact profiling metrics.
4. Statistical profiling metrics.
5. Per-thread performance data.

Paradyn has question marks under the first and second columns because it does not currently support hardware counters, but it will in a future release. Also, note that while TAU supports the same features as μProfiler, it does not use hardware counters for exact profiling.

Chapter 4

μProfiler's Target Environment

Since a programmer thinks in terms of a specific execution model, and implements programs using that model's given components, it is crucial for a profiler to gather and display performance data in the same way. Having profiling results presented in this fashion allows the programmer to easily map performance data back to a target program's high-level components, which in turn facilitates locating bottlenecks and other performance related issues.

μProfiler (Chapter 5) is a profiler that gathers and presents performance data in terms of its execution environment. To understand μProfiler's data gathering and presentation approach, the reader must first become familiar with the environment in which it operates. To that end, this chapter describes μProfiler's execution environment and its components.

4.1 μC++

μProfiler's target environment is a concurrent dialect of the C++ programming language [Str97] called μC++ [BDS+92, BS05]. μC++ extends C++ by introducing new language constructs that afford lightweight concurrency on uniprocessor and multiprocessor shared-memory computers. On uniprocessor systems, concurrency is achieved by interleaving the executions of tasks, while on multiprocessor systems, concurrency is achieved by using a combination of interleaving and true parallel execution.

μC++ is implemented using a translator and a run-time library. The translator reads a μC++ program and translates each language extension into one or more C++ statements, which are then compiled by a C++ compiler and linked with the μC++ run-time concurrency library, also known as the μC++ kernel. The μC++ kernel is responsible for managing and scheduling all user threads, and interacting with the operating system threads, while a program is running.

4.2 μC++ Language Constructs

Like other concurrent environments, such as those supplied by Java [AGH00] and C# [HWG03], μC++ provides its own execution model, composed of multiple components. μC++ introduces six new language constructs that provide execution properties such as advanced flow control, synchronization, mutual exclusion, and concurrency. These constructs are coroutines, monitors, coroutine-monitors, tasks, virtual processors, and clusters.

4.2.1 Coroutine

A coroutine has all the properties of a C++ class, as well as its own execution state. Thus, execution of a coroutine can be suspended as control leaves it, and resumed at the same point in the same state, i.e., with all local variables preserved, when control returns. In contrast, a normal routine is restarted at the beginning each time it is called, always executes to completion before returning, and no execution state is preserved, i.e., its local variables do not persist across invocations.

A coroutine has one distinguished member routine called `main`, which is either private or protected, i.e., it cannot be called from outside the coroutine. Instead, a coroutine provides an interface for resuming execution of its `main` routine at the point where it was last suspended, via a public member routine that executes a uResume statement. Thus, a coroutine is activated by making a call to this public member routine, which then explicitly resumes main. At that point, the caller context switches from its own execution state to the coroutine's execution state, and main continues execution from the point where it was last suspended.

A coroutine can suspend its execution in one of two ways. It can implicitly reactivate the coroutine that activated it by suspending execution of its own main routine, or it can explicitly activate another coroutine by calling one of that coroutine's public member routines, which in turn resumes that coroutine's main routine.

4.2.2 Monitor

A monitor has all the properties of a regular C++ class. In addition, it also provides mutual exclusion through the use of *mutex routines*, which are a set of routines that allow only one task to execute them at any given time. For example, if a task T_1 is active within a monitor's mutex set and a second task T_2 calls a routine in that monitor's mutex set, T_2 blocks and is added to an entry queue, where it waits until T_1 returns or blocks on a condition variable. Entry order into a monitor depends on that particular monitor's scheduling algorithm, which may include accepting mutex routines and/or unblocking tasks that are waiting on condition variables; more details are available in [BS05].

Not all of a monitor's member routines are mutex routines. Some member routines provide read-only access, or provide complex interactions (protocols) with the monitor's mutex routines. These routines are called *non-mutex routines*, and they can be executed simultaneously by an unlimited number of tasks.

4.2.3 Coroutine-Monitor

A coroutine-monitor has a combination of the properties of both a coroutine and a monitor. It is simply a coroutine with mutual exclusion, meaning only one task can be active inside its mutex set at any given time.

4.2.4 Task

A task is a coroutine-monitor with its own thread of control. Like a coroutine, a task has a distinguised member routine called `main`, but instead of using an existing thread to execute it, a new thread is created and starts execution in `main`. `main` is either private or protected, and thus cannot be called from outside the task. A task's thread runs concurrently with all other task threads in the same program.

4.2.5 Virtual Processor

A virtual processor is a "software processor" upon which user threads are scheduled for execution. Virtual processors are implemented by kernel threads, which are scheduled for execution on physical processors by the underlying operating system. In uniprocessor mode, μC++ simulates all virtual processors with one kernel thread, whereas in multiprocessor mode, each virtual processor gets a kernel thread of its own. Thus, when a μC++ program is run in multiprocessor mode on a multiprocessor system, there is the potential for true parallelism as the operating system may schedule multiple virtual processors for execution at the same time on different physical processors. A μC++ program may also be run in multiprocessor mode on a uniprocessor machine, which prevents the entire program from blocking if one virtual processor makes a blocking system call.

When a virtual processor is executing, the μC++ kernel schedules tasks to run on it. Thus, when the operating system gives a time-slice to a virtual processor, μC++ may further subdivide that time-slice among two or more tasks.

Since virtual processors are not bound to physical processors, μC++ programs can be written using more virtual processors than there are physical processors on the underlying machine. In this way, μC++ programs are kept portable across both uniprocessor and multiprocessor systems.

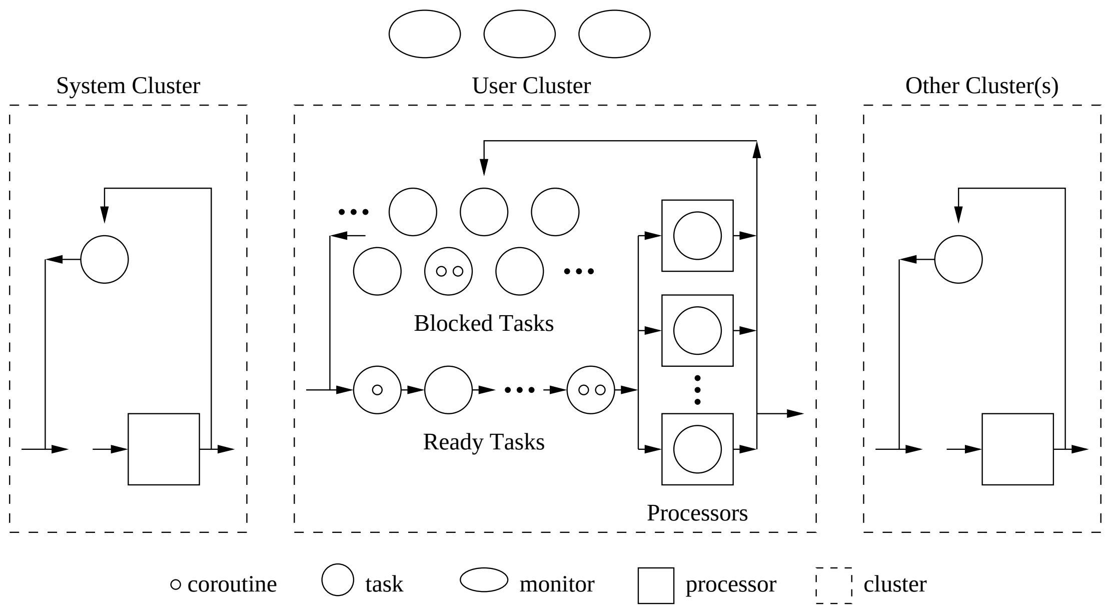


Figure 4.1: Run-time structure of μC++ language constructs.

4.2.6 Cluster

A cluster is a collection of virtual processors and tasks that execute on them. Clusters are used to control the amount of potential parallelism among tasks. A cluster requires at least one virtual processor to run tasks, and a task may only run on the virtual processors associated with its cluster. However, during the execution of a μC++ program, tasks and processors may explicitly migrate from cluster to cluster.

Each cluster has its own algorithm for scheduling its tasks for execution on its virtual processors. By default, the scheduling algorithm is round-robin using a single-queue, multi-server queueing model, which results in an automatic load balancing of tasks on virtual processors. However, users can implement their own scheduling algorithms, and several real-time schedulers are available [BS05].

Figure 4.1 shows the run-time structure of the μC++ language constructs described in this chapter.

Chapter 5

μProfiler

μProfiler is a concurrent, object-oriented profiler for concurrent, object-oriented programs written in μC++. It is part of the MVD Toolkit [Buh99], which is a set of applications for **M**onitoring, **V**isualizing and **D**ebugging μC++ programs. Other MVD tools include SMART [Sch99], which records the execution of a nondeterministic concurrent program and then replays it in a deterministic fashion, and Kalli's DeBugger (KDB) [BKS96], which is a multithreaded debugger for multithreaded programs.

The original μProfiler prototype was implemented in 1997 by Robert Denda [Den97], and was subsequently extended in 2000 by Dorota Zak [Zak00], who added a number of new metrics.

This chapter describes the design and implementation of μProfiler, taking into account the work done for both of the aforementioned theses, as well as changes and extensions I have made.

5.1 Design Objectives

μProfiler's original design is derived from the requirements laid out in [Den97]. The result is a list of six objectives, all of which are fulfilled by μProfiler's current implementation.

5.1.1 Profiling on a Per-Thread Basis

Concurrent programs are based on the notion of multiple threads of control. A profiler for concurrent programs must be aware of, and able to profile, each individual thread a programmer creates. To do this, μProfiler must be aware of how the μC++ kernel manages its threads.

5.1.2 Profiling at Different Levels of Detail

Profiling concurrent, object-oriented programs requires gathering performance data at different levels of detail. In the case of μC++ programs, μProfiler must be able to profile at the cluster, virtual processor, task, coroutine, object, and routine levels, using both exact and statistical metrics.

5.1.3 Selective Profiling

Users are not necessarily interested in profiling everything that goes on in an application. Rather, they may be interested in profiling only certain aspects of their program. This discrimination is accomplished by using a technique called selective profiling, whereby only those aspects of interest are instrumented.

μProfiler affords selective profiling of μC++ programs at both compile-time and run-time. At compile-time, profiling can be turned on for any program module by compiling with the -profile flag, while at run-time, profiling can be enabled and disabled for each task by calling the uProfileActivate and uProfileInactivate routines, respectively. Both profiled and unprofiled modules are compatible with each other, i.e., a selectively-profiled μC++ program (composed of both profiled and unprofiled modules) compiles and links just as if the program is not profiled at all.

5.1.4 Support Different Visualization Devices

Performance data can be visualized in a number of different ways, ranging from simple tables of raw numbers to graphical charts. μProfiler supports several different visualization devices and provides a custom Motif widget [HF94] for each one.

5.1.5 Extendibility

Most profilers have a built-in set of metrics that they are capable of measuring. However, no set of metrics can possibly fulfill the needs of all users. To that end, profilers should be extendible so that users can add their own metrics. μProfiler accomplishes this through the use of class hierarchies for its monitors, analyzers and visualizers. Users may customize any of μProfiler's metrics (or create entirely new ones) by deriving new monitors, analyzers and visualizers from their respective base classes, and linking the new metric into their program.

5.1.6 Portability, Interoperability, and Maintainability

μC++ is implemented on a number of different operating system/architecture pairs, such as Solaris on UltraSPARC, Linux on x86 and Linux on Itanium. Nothing in μProfiler's design or implementation precludes a port to any of the systems on which μC++ runs. In fact, μProfiler currently runs on the three operating system/architecture pairs listed above.

Since μProfiler is part of the MVD Toolkit, it is designed in such a way that it is interoperable with all other MVD tools. For example, a program profiled by μProfiler can simultaneously be debugged with KDB.

An important part of all software development is the maintainability of the end product. μProfiler is designed and implemented with maintainability in mind, making full use of high-level, object-oriented, and concurrent software development techniques.

5.2 Overview of μProfiler

This section provides an overview of μProfiler's implementation, which is explained in detail throughout the rest of this chapter. μProfiler offers two levels of instrumentation insertion, μC++ kernel instrumentation and user-code instrumentation, which are explained in Section 5.3. The object-oriented infrastructure and main functionality of μProfiler is contained in the μProfiler kernel, which is discussed in Section 5.4. μProfiler offers two types of metrics, user metrics and built-in metrics, which are presented in Section 5.5. μProfiler's execution monitors, which are objects responsible for enabling instrumentation hooks and collecting performance information, are explained in Section 5.6. Finally, Section 5.7 discusses μProfiler's analyzers and visualizers, which are objects that analyze the performance data collected by execution monitors, and display it on the screen, respectively.

5.3 Instrumentation Insertion

There are two different instrumentation insertion methods used by μProfiler. The first is insertion of instrumentation hooks into the μC++ kernel, and the second is insertion of shared trampolines into the user code of μC++ programs.

5.3.1 μC++ Kernel Instrumentation

Instrumentation hooks exist in areas of the μC++ kernel where events of potential interest occur. These hooks are present in *all* μC++ programs, but are guarded by instrumentation predicates that prevent them from being triggered unless the following conditions are true:

1. Profiling is currently enabled for the active task.
2. μProfiler is currently running, i.e., some portions of the target program are compiled with the -profile flag.

```
void uBaseTask::uSetState( uBaseTask::uTaskState state ) {
  ...
  if ( uProfileActive && uProfiler::uProfiler_RegisterTaskExecState ) {
    (*uProfiler::uProfiler_RegisterTaskExecState)( uProfiler::uProfilerInstance, *this, state );
  }
  ...
}
```

Figure 5.1: A μProfiler instrumentation hook.

3. The hook is enabled by at least one execution monitor (see Section 5.6).

Figure 5.1 is an example of a μC++ kernel hook for a task changing its execution state. The instrumentation predicate is the if statement surrounding the routine call, and it verifies that the three antecedent conditions are true. Condition 1 is verified simply by checking the uProfileActivate flag for the current task; if it is true, then profiling is enabled. Conditions 2 and 3 are verified by checking that uProfiler::uProfiler_RegisterTaskExecState is non-null. uProfiler::uProfiler_RegisterTaskExecState is a routine pointer that points to the uProfiler::RegisterTaskExecState member routine if and only if at least one execution monitor activated it. Since routine pointers are only set if μProfiler is running, conditions 2 and 3 are satisfied if the routine pointer contains a non-null value. Thus, if both conditions in the if statement are true, the hook is triggered. All other μC++ kernel hooks have a similar structure.

5.3.2 User Code Instrumentation

Instrumentation of the user code in a profiled μC++ program is done using shared trampolines, which are inserted into a target program with the help of the C++ compiler gcc [gcc]. When the -profile flag is specified, the μC++ translator activates a flag called -finstrument-functions, which causes gcc to insert calls to instrumenta-

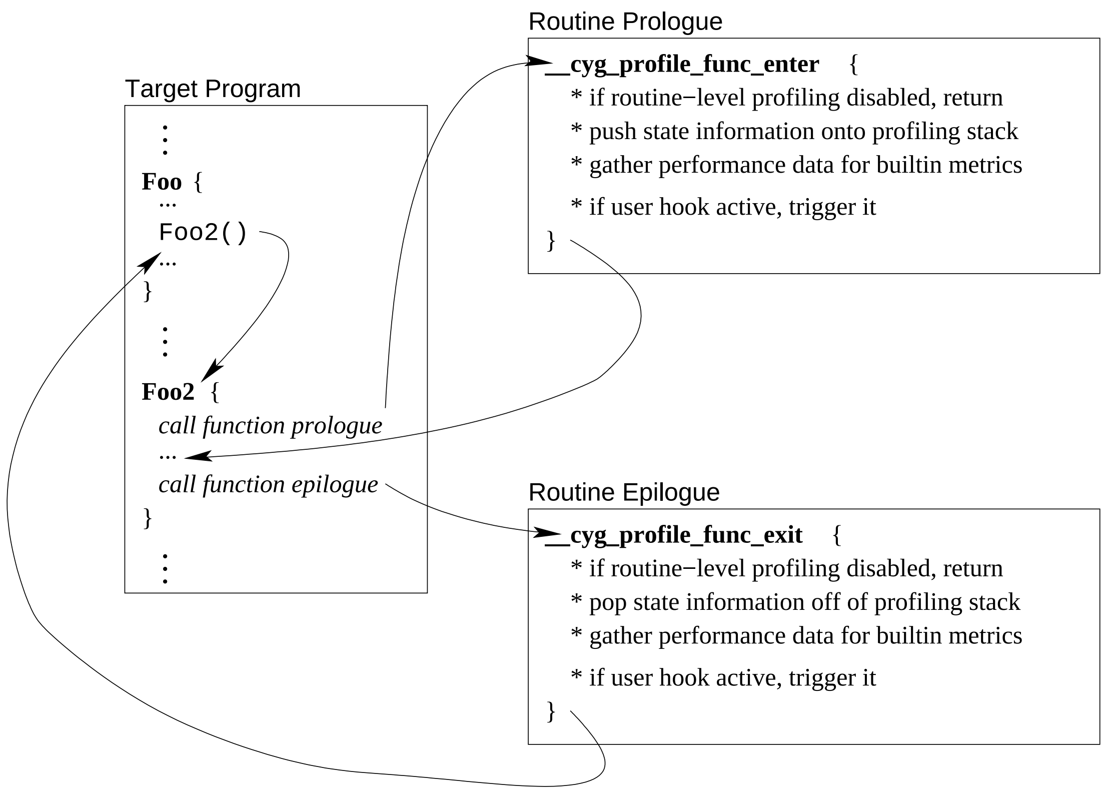

Target Program
Foo {
...
Foo2()
...
}
Foo2 {
call function prologue
...
call function epilogue
}
Routine Prologue
__cyg_profile_func_enter {
* if routine−level profiling disabled, return
* push state information onto profiling stack
* gather performance data for builtin metrics
* if user hook active, trigger it
}
Routine Epilogue
__cyg_profile_func_exit {
* if routine−level profiling disabled, return
* pop state information off of profiling stack
* gather performance data for builtin metrics
* if user hook active, trigger it
}

Figure 5.2: Flow of control for routine-level profiling in μProfiler.

tion trampolines at the entry and exit points of each routine in a program. The entry and exit trampolines are called the *routine prologue* and *routine epilogue*, respectively, and each is passed the address of the routine being entered or exited, as well as the address of the call site in its parent routine. Figure 5.2 shows the flow of control for a routine call in a μC++ program profiled with shared trampolines.

The routine prologue and epilogue first verify that at least one μProfiler metric is enabled that requires routine-level profiling; if not, then the trampolines return immediately and no data gathering is done. Otherwise, the current task or coroutine's *profiling stack* (see Section 5.4) is updated to reflect its new execution-state location; specifically, the routine prologue pushes a new frame onto the profiling stack containing information about the routine being entered, and the routine epilogue pops that frame off the stack.

Once the profiling stack is updated, the data gathering is performed. Built-in metrics (Section 5.5.2) have their monitoring code in the trampoline itself and gather their data by accessing μProfiler data structures directly. User metrics (Section 5.5.1) have no special access to μProfiler's data structures, so they must do their data gathering using hooks and execution monitors. There is a hook for routine entry in the routine prologue trampoline, and a hook for routine exit in the routine epilogue trampoline. As before, μProfiler verifies that each hook actually points to a uProfiler member routine. If so, the hook is triggered and all interested execution monitors are notified of the routine entry or exit.

5.4 μProfiler Kernel

μProfiler's main functionality lies in the *μProfiler kernel*, which is shown in Figure 5.3, using the object-oriented notation described in Appendix A. The μProfiler kernel is made up of the following tasks and objects: uProfiler, uProfilerStartWidget, uProfileTaskSampler, uExecutionMonitor, uMemoryExecutionMonitor, uMetricAnalyze, uProfileAnalyze, and uSymbolTable.

The uProfiler task acts as an administrator [Gen81] for all active metrics. All execution monitors register with uProfiler upon creation, and are managed by it from that point forward (see Section 5.6). uProfiler keeps a global list of all active execution monitors, as well as a *hook-monitor list* for each available instrumentation hook. These hook-monitor lists contain one entry for each execution monitor requesting notification of when its corresponding instrumentation hook is triggered. Depending on the type of profiling being done (exact or statistical), uProfiler either notifies execution monitors when events occur, or it polls at specified intervals. Finally, once monitoring is complete, uProfiler invokes an analyzer (see Section 5.7) for each execution monitor on its global list.

uProfilerStartWidget creates the μProfiler startup window, which is displayed before the application begins execution (see Figure 5.4). The startup window presents a list of available built-in and user metrics, from which the user makes a selection.

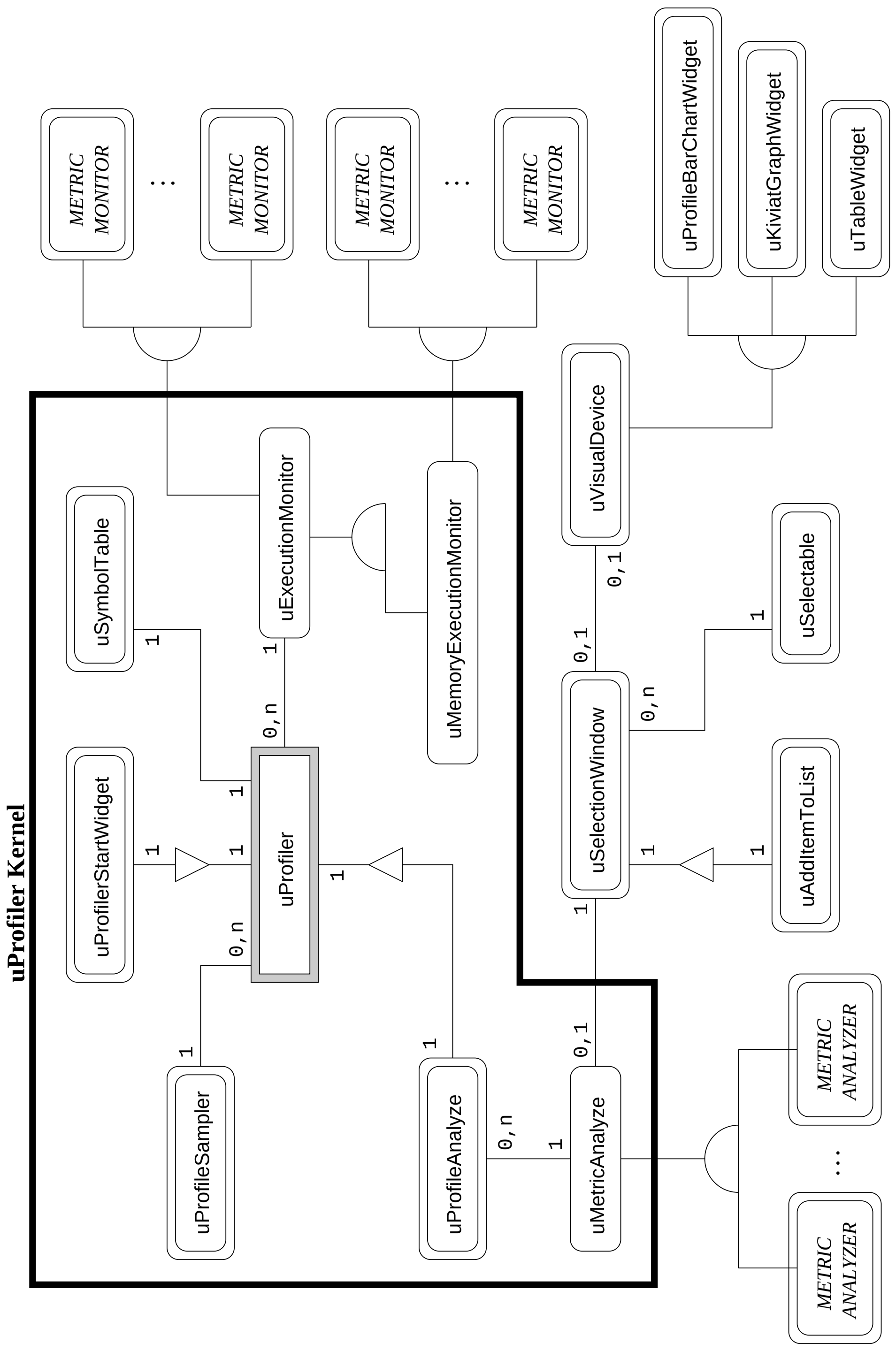


Figure 5.3: Object-oriented design of the μProfiler kernel.

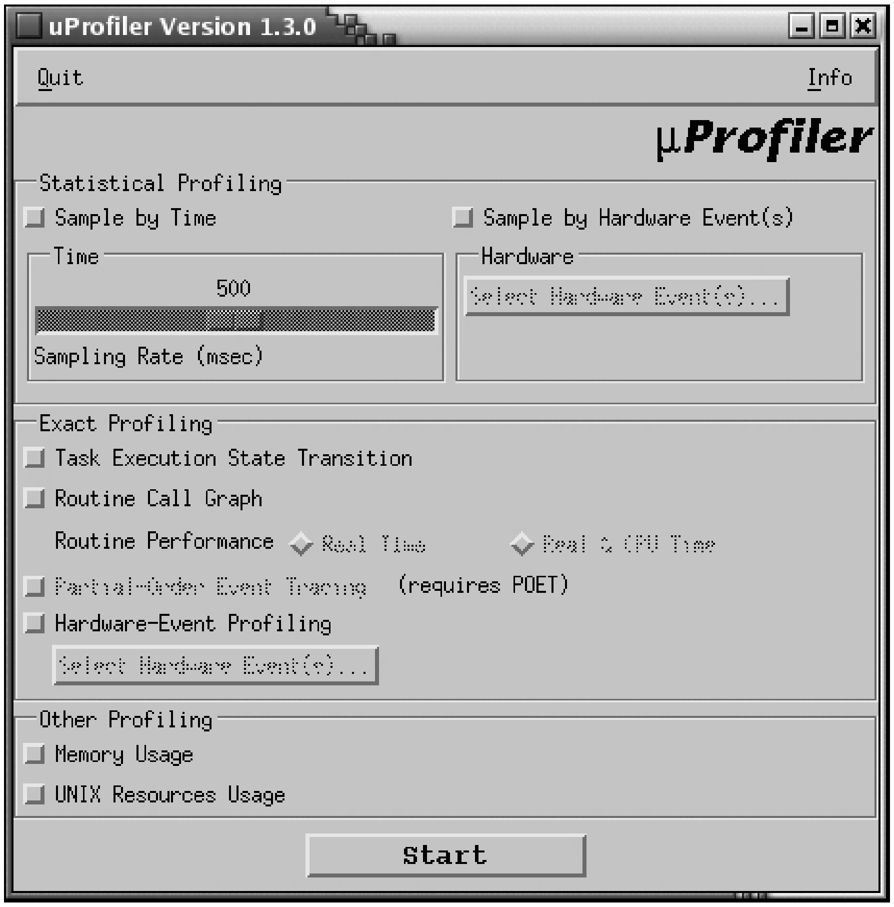


Figure 5.4: The μProfiler startup window.

Based on that selection, uProfilerStartWidget creates a subset of the available execution monitors.

Each profiled task and coroutine has one uProfileTaskSampler, which holds its pertinent performance data. If required by any active metrics, each uProfileTaskSampler also creates and maintains a per-task/coroutine *profiling stack*, which contains information about a task or coroutine's current routine-call sequence. Each stack frame contains the following information:

- The address of the current routine.
- The address of the parent routine.

- The address of the call site in the parent routine.
- An object pointer (currently unused).

This profiling stack is required by metrics needing an inexpensive method of obtaining information about a task's current routine and its parent. The profiling stack is obtained simply by dereferencing the profiling-stack pointer in a task's uProfileTaskSampler.

uExecutionMonitor, uMemoryExecutionMonitor and uMetricAnalyze are abstract bases-classes from which all metrics must derive their execution monitors and analyzers. They are explained in more detail in Sections 5.6 and 5.7.

uProfileAnalyze is invoked by uProfiler after monitoring of the target program is complete. It goes through the list of execution monitors registered with uProfiler and creates their associated analyzers.

uSymbolTable is an architecture-independent interface, built on top of the Binary File Descriptor Library [Cha91], for accessing a target program's architecture-dependent symbol table. It provides member routines for retrieving the addresses, names, and source-file names of a target program's routines.

5.5 μProfiler Metrics

A μProfiler metric is composed of an *execution monitor*, an *analyzer* and a *visualizer*. An execution monitor gathers performance data during program execution, its analyzer prepares the raw performance data for display, and its visualizer displays the performance data on the screen. This separation of duties not only makes clear what portions of a metric are responsible for performing which tasks, but it provides a simple interface for the addition of new metrics by users, and reusing components among metrics.

To create a new metric, a user simply derives an execution monitor and analyzer from the uExecutionMonitor and uMetricAnalyze base classes, respectively. For visualization, users have the choice of using one of the visualization devices provided by μProfiler (see Section 5.7), or creating a new one by deriving from the uVisualDevice base class.

5.5.1 User Metrics

A user metric is built using the μProfiler API, which provides a well-defined interface for adding execution monitors, analyzers and visualizers to μProfiler's infrastructure. The API includes abstract base-classes for all three metric components, which provide the minimum basic functionality necessary to register and interact with μProfiler. In this way, the user need not be concerned with the intrinsic details of μProfiler metrics, and can instead concentrate on the code to gather, analyze and display all required performance data.

User metrics gather performance data through the use of *user hooks*, which are instrumentation hooks in non-privileged areas of the μC++ kernel. Events that occur in privileged areas of the μC++ kernel can only be processed by built-in metrics, which are explained in the next section.

To create a trivial user metric to count the number of tasks created in a target program, a user does the following:

1. Derive an execution monitor that overrides the RegisterTaskNotify hook-notification routine (see Section 5.6) to activate the RegisterTask hook. This routine gets called every time a task is created by the μC++ kernel, and it is passed the address in memory of the task being created, as well as that of its parent task. However, since this simple metric only counts the number of created tasks, these addresses are ignored.

2. Derive an analyzer to manipulate the collected performance data as necessary. In this case, analysis is unnecessary as only one piece of performance data is collected. However an analyzer is still required because its base class provides code for creating the visualization devices.

3. Either use an existing visualization device, or derive a new one, to display the performance data on the screen.

Centralized Monitoring

All user metrics operate in *centralized monitoring mode*, which means the uProfiler task is responsible for gathering performance data. A task that triggers a user hook passes pertinent information to the uProfiler task, and then resumes normal execution. Meanwhile, the uProfiler task concurrently forwards that information to all execution monitors on the relevant hook-monitor list by calling their hook-notification routines and passing the information in as routine arguments. This system is a necessary consequence of creating and registering user metrics with μProfiler's API, but has the advantage of potential parallelism between a task that triggers a hook, and the uProfiler task that processes the resulting performance data.

5.5.2 Built-in Metrics

Built-in metrics are low-level metrics that are tightly integrated into μProfiler; they circumvent portions of the μProfiler API to provide core performance data from deep inside the μC++ kernel. This structure is essential for metrics that require notification of events that occur while a task is executing inside the μC++ kernel, such as when a task blocks or unblocks. Hooks for these types of events are called *built-in hooks* because they can only be used by built-in metrics. User hooks can be used by both user metrics and built-in metrics.

Like a user metric, a built-in metric is composed of an execution monitor, an analyzer, and a visualizer, all of which assume mainly the same responsibilities as above. The one exception to this rule for built-in metrics is the gathering of performance data, which is not handled exclusively by execution monitors, as is explained below.

Decentralized Monitoring

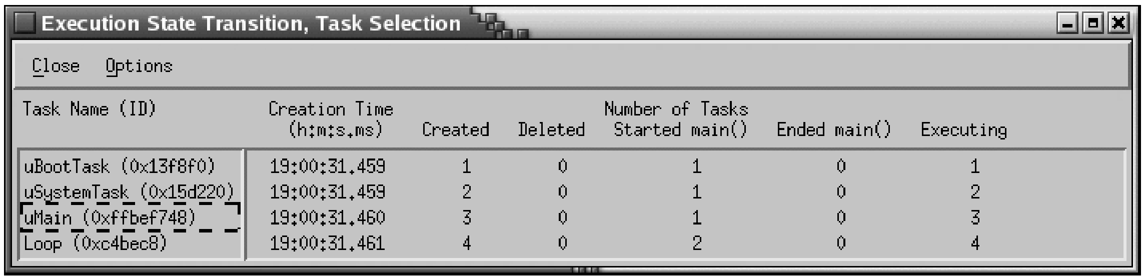
Built-in metrics operate in *decentralized monitoring mode* when processing built-in hooks, which means the task that triggers the built-in hook is responsible for gathering the resulting performance data. For example, a task that is in the midst of context-switching gathers and stores information about its own execution state, rather than requiring the uProfiler task to do it. There is no need for hook-monitor lists for built-in hooks, because execution monitors are not notified when built-in hooks are processed. This method incurs less overhead than centralized monitoring, because no communication is necessary between the task that triggers the hook and the uProfiler task. Furthermore, all performance data is placed directly into each execution monitor's data structures by the current task, rather than being passed to the execution monitors via routine calls.

5.6 Execution Monitors

μProfiler's execution monitors are passive objects that monitor a target program's execution behaviour. Each monitor registers itself with uProfiler upon creation to indicate the hooks necessary for its particular data gathering. Among other things, this allows uProfiler to keep a list of active monitors so that it can create the appropriate analyzers (which subsequently create visualizers) once monitoring is complete.

All execution monitors are derived, directly or indirectly, from uExecutionMonitor (an intermediate execution-monitor base-class called uMemoryExecutionMonitor is presented in Chapter 7). This base class has one *hook-notification routine* for each user hook in the μC++ kernel. Hook-notification routines are called by uProfiler to notify execution monitors that a user hook has been triggered. However, the hook-notification routines in uExecutionMonitor are placeholders (pure virtual routines); derived execution monitors provide functionality for these routines by defining them. This structure provides a technique for dynamically determining which hooks a derived monitor needs activated.

A user metric activates hooks using the μProfiler API by calling the Initialize routine in the uExecutionMonitor base class, which dynamically checks which hook-notification routines have been overridden, activates those hooks, and adds the monitor to the corresponding uProfiler hook-monitor lists.



Task Name (ID)	Creation Time (h:m:s.ms)	Created	Deleted	Number of Tasks Started main()	Ended main()	Executing
uBootTask (0x13f8f0)	19:00:31.459	1	0	1	0	1
uSystemTask (0x15d220)	19:00:31.459	2	0	1	0	2
uMain (0xffbef748)	19:00:31.460	3	0	1	0	3
Loop (0xc4bec8)	19:00:31.461	4	0	2	0	4

Figure 5.5: μProfiler task-selection box.

An execution monitor for a built-in metric circumvents the μProfiler API to activate any built-in hooks it requires. Besides calling the Initialize routine in the base class to activate any required user hooks, it also manually switches on the built-in hooks it needs. Since the execution monitor is not notified when built-in hooks are triggered (because of decentralized monitoring), there is no need for it to manually add itself to any hook-monitor lists.

5.7 Analyzers and Visualizers

Once monitoring is complete, i.e., once the target program terminates, the uProfiler task creates an object of type uProfileAnalyze, which creates and manages all of the required analyzers. uProfileAnalyze accomplishes this task by calling the CreateMetricAnalyze routine for each monitor in uProfiler's global execution monitor list. CreateMetricAnalyze is a virtual routine in the uExecutionMonitor base class that must be defined by all derived monitors. It returns a reference to the newly created analyzer, which uProfileAnalyze stores in a list.

Like execution monitors, all μProfiler analyzers must be derived from a common base class, called uMetricAnalyze. uMetricAnalyze provides basic routines for creating and managing *selection windows*, which are used extensively by μProfiler to display performance data in a hierarchical fashion. For example, in Figure 5.5, a list of profiled tasks is displayed in the leftmost column. Clicking on any task drills down and opens another window with information specific to that task. Selection windows are implemented by the uSelectionWindow class.

μProfiler currently provides three different types of visualization devices, all built with Motif widgets. There is a table widget (uTableWidget), a bar chart widget (uProfileBarChartWidget), and a Kiviat graph widget (uKiviatGraphWidget). These widgets allow users to create metrics that display their data in a variety of ways, without requiring them to learn Motif syntax. However, nothing precludes advanced users from creating and adding their own Motif-based visualization devices.

Chapter 6

Accessing Hardware Counters with μProfiler

This chapter describes the first of the three major contributions of this thesis, namely, integrating hardware-counter support for three different architectures into μProfiler. Currently, support is in place for accessing hardware counters on the UltraSPARC I/II/III running Solaris, the x86 (including Intel Pentium/MMX/-Pro/II/III and AMD Athlon) running Linux, and the IA-64 (Itanium I and II) running Linux.

All μProfiler hardware counter support is encapsulated in a single class called uHWCounters, which shields programmers from all low-level details. In fact, the uHWCounters API completely abstracts away the notion of the underlying hardware counters, allowing programmers to focus on choosing which events to count, rather than how to cause them to be counted.

I was unable to use an existing hardware-counter library, such as PAPI [BDG+00] (see Section 3.1), to add hardware-counter capabilities to μProfiler, for the following reasons. First, only a small subset of PAPI-like routines are necessary to program and use the hardware counters to the extent that they are required in this thesis (six routines per architecture). Most of the other routines in the uHWCounters class consist of architecture-independent code tailored towards μProfiler's requirements.

Second, μProfiler allows users to interactively select the hardware events they wish to count, which requires immediate feedback showing which events are available to be counted given a user's current event-selection. This type of feedback is unavailable from PAPI. The implementation of this feature in μProfiler is presented in Section 6.5.1.

6.1 Hardware Events

The number and types of measurable hardware events vary significantly from processor to processor. This variance is due to differences in the physical properties of each processor, such as cache hierarchies, branch predictors, and number of physical hardware counters. At present, the uHWCounters class is only capable of measuring a small subset of hardware events common to all supported platforms, where this subset represents some of the more commonly used events in program profiling. This approach has three advantages. First, it provides a proof-of-concept, showing that the uHWCounters class is capable of counting hardware events on multiple platforms. Second, it shows that the design of the uHWCounters class is flexible enough to be ported to different architectures in a straightforward manner. Finally, it keeps the user interface identical across platforms because the user is always presented with the same list of events to count. However, the design of the uHWCounters class does not preclude creating platform-specific lists of hardware events in the future.

6.1.1 User-Level vs. System-Level Hardware Events

Hardware events can be counted at two different levels: the user level and the system level. User-level hardware events are events that occur while executing user code, while system-level hardware events are events that occur while executing kernel code. The uHWCounters class is capable of measuring user-level events, system-level events, or both. Users can toggle the counting of each type of event separately (see Section 6.4.1).

6.2 Event Tables

Every CPU has a different set of events that it is capable of measuring, as well as constraints on how these events can be counted. Borrowing a concept from the PAPI library, this information is encapsulated in a set of *event tables*, one for each supported CPU. Each of the hardware events supported by μProfiler has one entry in each CPU-dependent event table. Generally, these entries contain the following information:

- Whether this event is supported by the current CPU.
- Whether this event is a component event or a composed event (see below).
- What the CPU-dependent bit-encoding of this event is, which is used to program the hardware counters when uStartCounters is called.
- Which counters can count this event.

All event tables list their events in the same order, which means that every event has a unique index, independent of which CPU is being used. These unique indices are how events are referred to when using the uHWCounters API (see Section 6.4).

6.2.1 Component Events vs. Composed Events

Oftentimes, a user wishes to measure an event that cannot be counted using only one hardware counter. For example, some architectures provide a "cache miss" event, while others provide only "cache reference" and "cache hit" events. In the latter case, the number of cache misses can be derived by subtracting the number of cache hits from the number of cache references, but such a procedure requires two counters. Another example is an event involving a ratio of two other events, such as instructions per CPU cycle.

To handle the situation described above, two hardware event types are stored in the event tables: component events and composed events. *Component events* are hardware events that can be counted using only one hardware counter, while *composed events* are hardware events obtained by adding, subtracting or dividing the counts of two or more component events.

6.3 uHWCounters State

The uHWCounters class contains a number of member variables representing a user's desired configuration of the underlying hardware counters, such as which events to count, and at what level they should be counted (e.g., user level, system level, or both). This information is stored in an architecture-independent format, and is only converted to platform-specific data to program the underlying hardware counters at a user's request. In this way, the majority of the architecture-dependent code is localized in a handful of low-level routines, which makes porting the uHWCounters class to other platforms straightforward.

Examples of the state variables contained in the uHWCounters class are the *counters* array, which has one cell for each underlying hardware counter, and the *event_availabilities* array, which has one cell for each hardware event in the event tables (see Figure 6.1). $counters[i]$ contains an index into the event table corresponding to the event that hardware counter i is currently set to count, or -1 if the counter is empty. $event\_availabilities[j]$ contains true if hardware event j is available to be counted given the current state of the *counters* array, and false otherwise. Section 6.5.1 explains in detail how the *counters* and *event_availabilities* arrays are used by the uHWCounters class.

6.4 The uHWCounters API

The uHWCounters public interface is composed of both architecture-independent and architecture-dependent member routines. The architecture-independent portion of the API consists of routines to read and modify the uHWCounters state

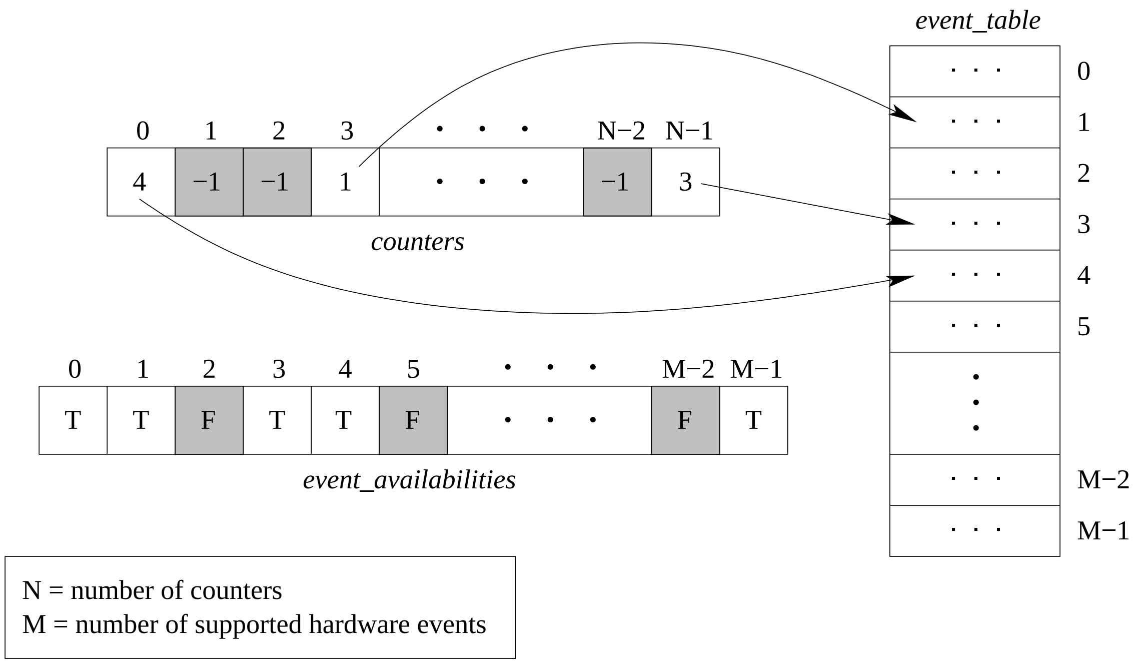


Figure 6.1: uHWCounters state.

information. The architecture-dependent routines interact with the hardware counters, converting uHWCounters state information from platform-neutral to platform-specific where necessary. All uHWCounters routines accept an index into the event tables when a specific event is required as a parameter.

6.4.1 Architecture-Independent Interface

The architecture-independent portion of the uHWCounters API is used to read and change the values that specify how the underlying hardware counters are programmed. These routines are grouped into two sections: *accessor routines*, which are used to query the current uHWCounters state, and *mutator routines*, which are used to change it.

Accessor Routines

Accessor routines provide read-only access to the current state of a uHWCounters object, such as:

- The name of a hardware event, given its index in the event table, e.g., "cache hits".
- Whether or not a hardware event is available to be counted by the hardware counters, given the current set of hardware events already chosen to be counted (i.e., given information from *counters*). This information can be used in a graphical user interface to "grey out" unavailable hardware events in a list, as is explained in Section 6.5.1.
- The total number of hardware events currently set up to be counted.
- The calling task's current event count for each hardware event being counted.

Mutator Routines

Mutator routines manipulate the current state of the uHWCounters object. The architecture-independent interface has mutator routines that:

- Toggle the counting of user-level hardware events.
- Toggle the counting of system-level hardware events.
- Add a hardware event to the current event set.
- Remove a hardware event from the current event set.

6.4.2 Architecture-Dependent Interface

The architecture-dependent portion of the interface contains the low-level routines responsible for interacting with the hardware counters. Five routines make up this part of the interface:

- uStartCounters: programs the hardware counters according to the uHWCounters object's internal state, and activates them.
- uRestartCounters: restarts the hardware counters after an overflow occurs.
- uStopCounters: Deactivates the hardware counters.
- uReadCounters: Reads the event counts currently contained in the hardware counters.
- uGetOverflowMask: Returns a bitmask indicating the counters that have overflowed most recently.

6.5 Implementation Issues

This section describes some of the interesting implementation issues encountered and solved while writing the uHWCounters class.

6.5.1 Hardware Event Availabilities

Setting up hardware counters to count a desired set of hardware events can be a difficult task. Each CPU has a different number of hardware counters, is capable of counting a different set of hardware events, and has different constraints on which events can be counted by which counters. For example, Sun Microsystems' UltraSPARC III processor [Sun04] has only two hardware counters, and most of the events it is capable of counting are constrained to one counter or the other. Many CPUs have similar restrictions.

The uHWCounters API shields users from these and all other low-level details of the hardware counters. In fact, users require absolutely no knowledge of the configuration of the underlying hardware counters to use the API. Their focus can and should be on selecting the hardware events to be counted. The uHWCounters class automatically figures out how to configure the counters for a given selection of hardware events.

The uHWCounters API is designed to be used in conjunction with the μProfiler startup window's graphical user interface, which displays available hardware events as a list of clickable buttons (see Figure 7.1 on page 78). The uEventAvailable routine accepts an event index and returns the boolean value stored in the corresponding cell in the *event_availabilities* array, which is true if that event can legally be added to the counters given the current event set to be counted, and false otherwise. The return value of this routine is used to "grey out" unavailable events in the μProfiler startup window.

The interesting part of this problem is how to properly determine the availability of all events and correctly populate the *event_availabilities* array, which must be done every time an event is added to, or removed from, the current event set. The general method for determining whether or not an event can be counted by the hardware counters is to add that event to the current event set, and then attempt to legally place the new event set into the counters. The latter step is done by exhaustively checking all permutations of the event set's events in the counters, which is the only way to guarantee that a solution is found if one exists.

The algorithms for determining the availability of a component event and a composed event differ slightly; each is explained in detail in the following sections.

Determining the Availability of a Component Event

Let S be a legal component-event set, i.e., a set of distinct component events that can be counted simultaneously by a certain CPU's hardware counters. Further, assume that that CPU's hardware counters are currently programmed to count all the component events in S. Then a component event e is available to be added to the hardware counters if and only if $S \cup \{e\}$ is a legal component-event set.

Let n be the number of physical hardware counters. The component_event_availability algorithm, which determines if e is available to be counted given the current event set S, is as follows:

1. If $e \in S$, then e is available (it is already in the counters), and stop.

2. If the *counters* array is full, then e is unavailable, and stop.
3. If e can be counted by an unoccupied $counters[i]$ for some $i \in \{0 \ldots n-1\}$, then e is available, and stop.
4. Create a *temp_counters* array with the same length as *counters*, and set each $temp\_counters[i]$ to -1, i.e., empty, for $i \in \{0 \ldots n-1\}$.
5. If legal_event_set($S \cup \{e\}$, *temp_counters*) is true, then e is available; otherwise, e is unavailable.

legal_event_set is an algorithm for determining if a given event set is legal. It accepts two parameters: an event set E, and an array c representing the hardware counters. It returns true if E is a legal event-set, and false otherwise. The algorithm is as follows:

1. If $|E|$ is 0, return true.
2. For each $c[i]$, $i \in \{0 \ldots n-1\}$, do:

   If $c[i]$ is empty and component event E_0 can be counted by $c[i]$, do:

   (a) Insert E_0 into $c[i]$.

   (b) If legal_event_set($E - \{E_0\}$, c) is true, return true.

   (c) Set $c[i]$ to -1.
3. Return false.

The component_event_availability algorithm starts out by verifying that there is room to add an event to the counters, which is an $O(1)$ operation. If that fails, it then runs through the *counters* array to determine if there are any empty counters capable of counting e, which is an $O(n)$ operation. If that also fails, a new array of length n is populated with -1's ($O(n)$ operation), and legal_event_set is called. Thus, the correctness and algorithmic complexity of component_event_available depend on legal_event_set.

legal_event_set is an exhaustive search that uses a branch-and-bound technique. At each level, component event E_0 attempts to occupy each available counter. For each successful occupancy, legal_event_set is called on the event set $E - \{E_0\}$, meaning the next event in E attempts to occupy each of the remaining $n - 1$ counters, and so on until E is the empty set. If at any point, the event being examined cannot be counted by any of the available counters, the partial solution is abandoned and the search backtracks. Thus, once a partial solution is shown to be incorrect, it is not explored further.

Since this search is exhaustive, it is guaranteed to find a solution if one exists. Furthermore, if E is found to be a legal event-set, the *temp_counters* array contains a proper configuration of E's events in the hardware counters, which can be saved and used to configure the *counters* array if e is subsequently added to the hardware counters.

Theoretically, the algorithmic complexity of legal_event_set is a combinatorial $O(\frac{n!}{(n-|E|)!}) = O(n!)$. However, in practice this is not an issue. Among the three architectures currently supported by uHWCounters, the largest number of hardware counters on any single CPU is four. Moreover, in many cases, the branch-and-bound technique shortens the search time significantly. However, if architectures with more counters are to be supported in the future, a more efficient algorithm may be needed.

Determining the Availability of a Composed Event

Let S be a legal component-event set, and assume that a certain CPU's hardware counters are currently programmed to count all the component events in S. Let K be the set of component events in a composed event k. Then k is available to be added to the CPU's hardware counters if and only if $S \cup K$ is a legal component-event set.

Let n be the number of physical hardware counters. The composed_event_availability algorithm, which determines if k is available to be counted given the current event set S, is as follows:

1. If $K \in S$, then k is available, and stop.
2. If $|S \cup K|$ is greater than the number of hardware counters, then k is unavailable, and stop.
3. Create a *temp_counters* array with the same length as *counters*, and set each *temp_counters*$[i]$ to -1, i.e., empty, for $i \in \{0 \ldots n-1\}$.
4. If legal_event_set($S \cup K$, *temp_counters*) is true, then k is available. Otherwise, k is unavailable.

This algorithm is a generalization of component_event_available, and thus shares its correctness and algorithmic complexity.

6.5.2 Per-Thread Virtual Hardware-Counter Contexts

μC++ programs are multithreaded, both at the kernel level (virtual processors) and the user level (tasks). However, since there is only one set of hardware counters per CPU, they must be shared among all the threads that want to use them. This sharing is accomplished through the use of per-thread *virtual hardware-counter contexts*. Essentially, this means that the values of the physical hardware counters become part of each thread's execution state, just like other registers; they are saved when a thread blocks, and restored when a thread resumes.

Since each virtual hardware-counter context can be set up to count a different set of events, it is possible to have different threads count different events. However, it was decided to keep the event set homogeneous across all threads of a μC++ program profiled by μProfiler. This design decision was made for two reasons. First, μProfiler currently only allows users to set profiling parameters (such as which hardware events to count) in one place: at the μProfiler startup window. Since users are not always aware of how many virtual processors or tasks are created by a program, it is impossible to select an event set for each thread ahead of time. Even if it were, selecting an event set for each thread in a program with thousands of tasks would be quite cumbersome. Second, profiling data is much easier to visualize, read, and comprehend when it is done in a uniform fashion. For example, it is easier to compare the performances of two tasks that measure retired instructions than it is to compare the performance of one task that counts CPU cycles and one task that counts cache hits.

Implementing virtual hardware-counter contexts at the task level is straightforward, as there are built-in hooks in the μC++ kernel for a task blocking and unblocking. This process is explained in Chapter 7. Using virtual hardware-counter contexts at the kernel level, however, requires operating-system support. The next three sections explain how this operating-system support is obtained and used on the three architectures supported by uHWCounters to provide a virtual hardware-counter context for each virtual processor in a μC++ program. The general procedure is as follows:

- The uStartCounters routine creates a virtual hardware-counter context based on the uHWCounters state variables and binds it to the calling kernel thread.
- The uReadCounters routine reads the calling kernel thread's virtual hardware-counter context.
- The uStopCounters routine unbinds and destroys the calling kernel thread's virtual hardware-counter context.

Solaris/UltraSPARC

Of the three operating system/architecture pairs supported by the uHWCounters class, the Solaris operating system for SPARC platforms has the simplest virtual hardware-counter context interface. It provides the cpc library [Sun] for accessing the hardware counters on the UltraSPARC I, II, and III processors. One call to the cpc_bind_event routine creates a virtual hardware-counter context and automatically binds it to the calling kernel thread. No extra work is needed or done by the uHWCounters class. The cpc library also provides routines to read and write the virtual hardware counters, as well as unbind a virtual hardware-counter context from its kernel thread. Like cpc_bind_event, these routines automatically use the calling kernel thread's virtual hardware-counter context.

The Solaris operating system for SPARC platforms provides a second library, called pctx, which can be used by a kernel thread to monitor a different kernel thread's virtual hardware-counter context. For each routine in the cpc library, there is a routine in the pctx that is directly analogous, but each one accepts two extra parameters: a pointer to the virtual hardware-counter context being monitored, and the ID of the kernel thread that the context belongs to. The pctx library is used to build performance tools that monitor processes other than themselves. However, because the μProfiler metrics in this thesis only require kernel threads to read their own virtual hardware-counter contexts, the pctx library is not used.

Linux/x86

The x86 Linux kernel has no inherent support for virtual hardware-counter contexts; it is obtained with Mikael Pettersson's perfctr Linux-kernel patch and library [Pet]. The kernel patch causes the hardware counter registers to be saved and restored when a kernel thread context switches, and the library provides routines for creating, reading and writing hardware-counter contexts, which reside in kernel space. The vperfctr_open routine creates a context, binds it to the calling kernel thread, and returns a pointer to a vperfctr structure representing that context. Unlike Solaris' cpc library, perfctr does not provide a set of functions specifically for self-monitoring kernel threads. Therefore, the context pointer must be passed to all other vperfctr routines, even though all kernel threads are self-monitoring. As a result, an extra field had to be placed in the μC++ uProcessor class to hold each kernel thread's virtual hardware-counter context pointer. The uStartCounters routine fills in this field after creating a new hardware-counter context, and the uStopCounters routine clears this field after destroying a hardware-counter context that is no longer needed. When reading the hardware counters, the uReadCounters routine retrieves the pointer from the calling kernel thread's uProcessor object and passes it to the appropriate vperfctr routine.

Linux/IA-64

The IA-64 Linux kernel [ME02] includes support for virtual hardware-counter contexts via the perfmon kernel interface [per], which contains only the perfmonctl system call. perfmonctl handles all hardware-counter related tasks by accepting a command flag as a parameter, and a file descriptor on which to operate. This file descriptor is analagous to the vperfctr pointer in the x86 case; it identifies which one of the kernel's virtual hardware-counter contexts is to be used (there are no specific routines for self-monitoring kernel threads). To create a context, the uStartCounters routine calls perfmonctl with file descriptor 0 and the command flag PFM_CREATE_CONTEXT. The return value of this routine call is a file descriptor representing the newly-created context. To bind this context to the calling kernel thread, perfmonctl is called again, and passed the new file descriptor and the command flag PFM_LOAD_CONTEXT. The file descriptor is then copied into the calling kernel thread's uProcessor object so it can easily be accessed for all subsequent calls to perfmonctl. As in the x86 case, the uReadCounters routine retrieves the calling kernel thread's file descriptor from its uProcessor object and passes it to perfmonctl with an appropriate flag when reading the hardware counters. Finally, the uStopCounters routine closes its kernel thread's file descriptor in the corresponding uProcessor object, thereby destroying its associated context, and sets it to zero.

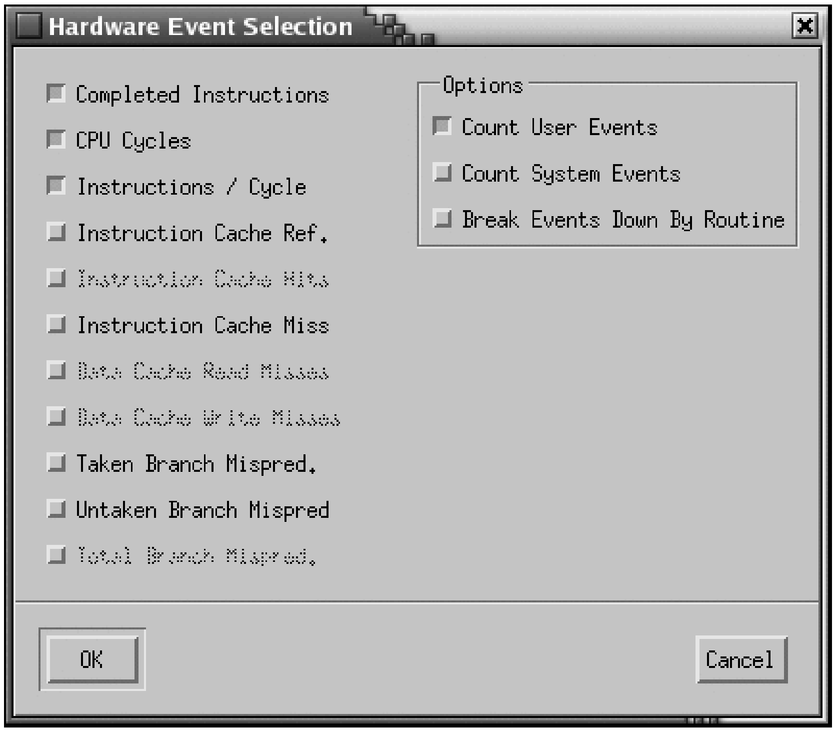
Chapter 7

Exact Hardware Metric

The two remaining contributions of this thesis are in the form of new μProfiler metrics. This chapter describes the first of these metrics, called the "Exact Hardware Metric", which is a built-in metric that produces an exact execution profile of a μC++ program using hardware counters. The objective of this metric is to provide users with two different levels of per-task exact hardware-counter profiling, each with a different cost and precision. The low level has a higher overhead, but provides exact hardware counts for each routine executed by each task. The high level is less expensive, but only provides hardware counts for each task on a time-slice basis.

7.1 Functionality

To activate the Exact Hardware Metric, the user clicks the "Hardware-Event Profiling" button in the "Exact Profiling" frame of the μProfiler startup window (refer to Figure 5.4 on page 55), which sensitizes the "Select Hardware Event(s)" button just beneath it. The user then clicks the "Select Hardware Event(s)" button, which pops up a dialog box with a list of available hardware events, as well as a list of options (Figure 7.1). A hardware event is selected by clicking the button to the left of its name. Note, as the user selects hardware events to count, other events become



- ☑ 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.

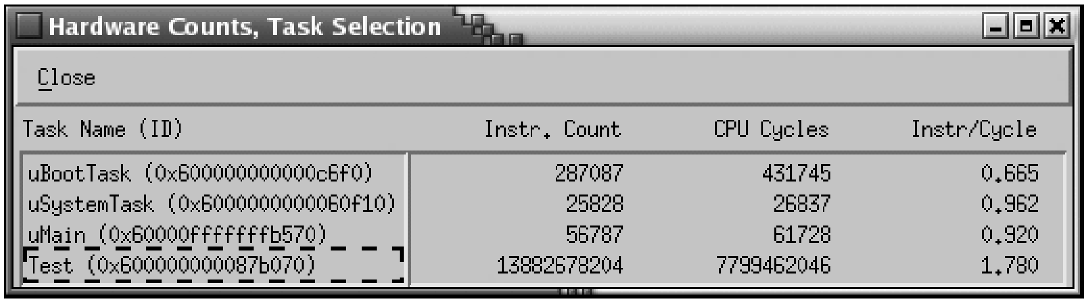
Options
- ☑ Count User Events
- ☐ Count System Events
- ☐ Break Events Down By Routine

Figure 7.1: Hardware-event selection dialog box.

unavailable and are greyed out due to hardware-counter constraints, as described in Chapter 6.

The option buttons on the right allow the user to choose among counting user-level events, system-level events, or both, as well as whether or not to break hardware-event counts down by routine. The latter option gives the user a choice between precision and overhead. If the "Break Events Down By Routine" button is pressed, then hardware-event counts are gathered and presented at the routine level for each task. This option offers an in-depth look at where each task's hardware events occur, but has a higher overhead. Conversely, if the button is not pressed, only an aggregate event count, broken down by time-slice, is gathered for each task. This option is not as expensive, but only gives an overall look at the number of hardware events during a task's execution.

Once the desired hardware events and options are chosen, the user clicks "OK"



Task Name (ID)	Instr. Count	CPU Cycles	Instr/Cycle
uBootTask (0x6000000000000c6f0)	287087	431745	0.665
uSystemTask (0x60000000000060f10)	25828	26837	0.962
uMain (0x60000fffffffb570)	56787	61728	0.920
Test (0x600000000087b070)	13882678204	7799462046	1.780

Figure 7.2: Task-selection box.

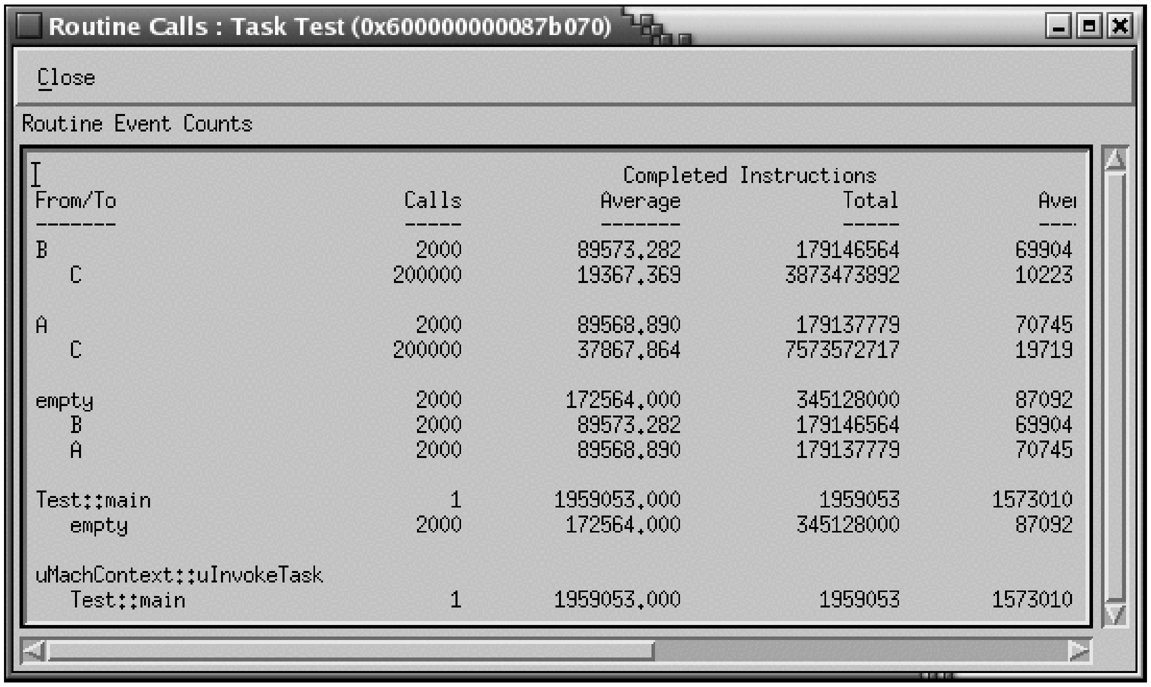
to close the dialog box and then clicks “Start” on the startup window to run the program. At program termination, a selection box appears on the screen, displaying a list of tasks created by the program, as well as the total number of hardware events incurred by each task (Figure 7.2). Clicking on a task shows one of two breakdowns of that task’s activities, depending on whether or not the user chose to break down hardware-event counts by routine.

7.1.1 Routine Breakdown

If the “Break Events Down By Routine” option was chosen, then clicking on a task in the selection box displays a list of routines called by that task, as well as call counts and hardware-event counts for each routine (Figure 7.3). The “From/To” column lists the names of all routines called by the task, as well as the caller/callee relationships between them. Caller (non-indented) routines have a list of their callee (indented) routines directly below them.

The meanings of the remaining columns depend on whether they describe a caller routine or a callee routine. In the case of a caller routine, the “Calls” column lists the total number of times a routine is called throughout a program’s execution. Similarly, the “Average” and “Total” columns list the average number of hardware events per call and the total number of hardware events for all calls to a routine.

In the case of a callee routine, the “Calls”, “Average”, and “Total” columns list the same information as above, but only for calls made to the callee by the caller



```
                                         Completed Instructions
From/To                      Calls        Average          Total        Avei
-------                      -----        -------          -----        ---.
B                             2000      89573.282      179146564       69904
   C                        200000      19367.369     3873473892       10223

A                             2000      89568.890      179137779       70745
   C                        200000      37867.864     7573572717       19719

empty                         2000     172564.000      345128000       87092
   B                          2000      89573.282      179146564       69904
   A                          2000      89568.890      179137779       70745

Test::main                       1    1959053.000        1959053     1573010
   empty                      2000     172564.000      345128000       87092

uMachContext::uInvokeTask
   Test::main                    1    1959053.000        1959053     1573010
```

Figure 7.3: Hardware-event counts by routine.

it is listed under. Breaking the event counts down in this fashion shows differences in the routines' behaviour when they are called by different parents. For example, Figure 7.3 shows the routine breakdown of the Test task from the program listed in Figure 7.4. It can be seen that routine C executes almost twice as many instructions when it is called by routine A than when it is called by routine B. This information gives the user an insightful glance into a program's run-time behaviour.

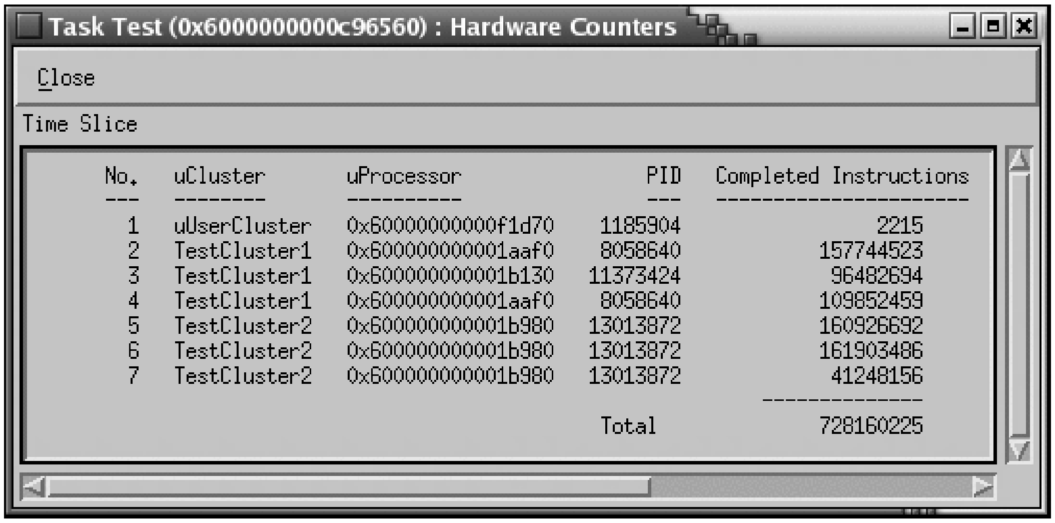
7.1.2 Non-Routine Breakdown

If the "Break Events Down By Routine" option is not chosen at program startup, then clicking on a task in the selection box displays a breakdown of hardware-event counts by time-slice (Figure 7.5 shows the non-routine breakdown of a Test task from the program listed in Figure 7.19(b) on page 109). Each line represents one time-slice, and lists the name of the task's cluster, the address of the task's virtual processor, the UNIX PID of the task's virtual processor, and the number

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

int a = 5, b = 4, c = 8, d = 2, e = 9, f;

void C( int numTimes ) {
    for ( int i = 0; i < numTimes; ++i ) {
        f -= a - b + c - d - e;
    } // for
} // C

void B() {
    for ( int i = 0; i < 100; ++i ) {
        f += a + b + c + d + e;
        C( 500 );
    } // for
} // B

void A() {
    for ( int i = 0; i < 100; ++i ) {
        f += a + b + c + d + e;
        C( 1000 );
    } // for
} // A
```

```
void empty() {
    for ( int i = 0; i < 5000; ++i ) {
        f += a - b - c + d - e;
    } // for
    A();
    B();
} // empty

uTask Test {
    void main() {
        for ( int i = 0; i < 2000; ++i ) {
            f = a + b - c + d - e;
            f += a - b - c + d - e;
            f += a + b + c + d - e;
            f += a + b - c - d - e;
            f += a + b - c + e + e;
            empty();
        } // for
    } // main
}; // Test

void uMain::main() {
    Test test;
} // main
```

Figure 7.4: Example program.



```
No.  uCluster      uProcessor          PID       Completed Instructions
---  --------      ----------          ---       ----------------------
  1  uUserCluster  0x60000000000f1d70   1185904                2215
  2  TestCluster1  0x600000000001aaf0   8058640           157744523
  3  TestCluster1  0x600000000001b130  11373424            96482694
  4  TestCluster1  0x600000000001aaf0   8058640           109852459
  5  TestCluster2  0x600000000001b980  13013872           160926692
  6  TestCluster2  0x600000000001b980  13013872           161903486
  7  TestCluster2  0x600000000001b980  13013872            41248156
                                                     --------------
                                        Total             728160225
```

Figure 7.5: Hardware-event counts by time-slice.

of occurrences of each hardware event. This non-routine breakdown provides the user with a high-level glance at a task's hardware-event activities as it moves from processor to processor and cluster to cluster between time-slices, which is useful in two ways. First, it provides a check if a hardware event has an expected value (e.g., an expected number of CPU Cycles can be determined based on a machine's clock speed and the running time of a program). If a task's event count deviates noticeably from its expected event count, this is an indication of a performance problem, which makes the program a good candidate for a routine breakdown to help pinpoint it. Second, it provides information about the performance of tasks on different processors and clusters, which is unavailable in Routine mode. For example, if a task displays an usually high number of data-cache misses during its time-slices on a particular cluster, this may be an indication of a data-locality problem with respect to the other tasks on that cluster.

7.2 Design

Figure 7.6 shows the design of the Exact Hardware Metric, using the object-oriented notation described in Appendix A. Following the metric-creation requirements

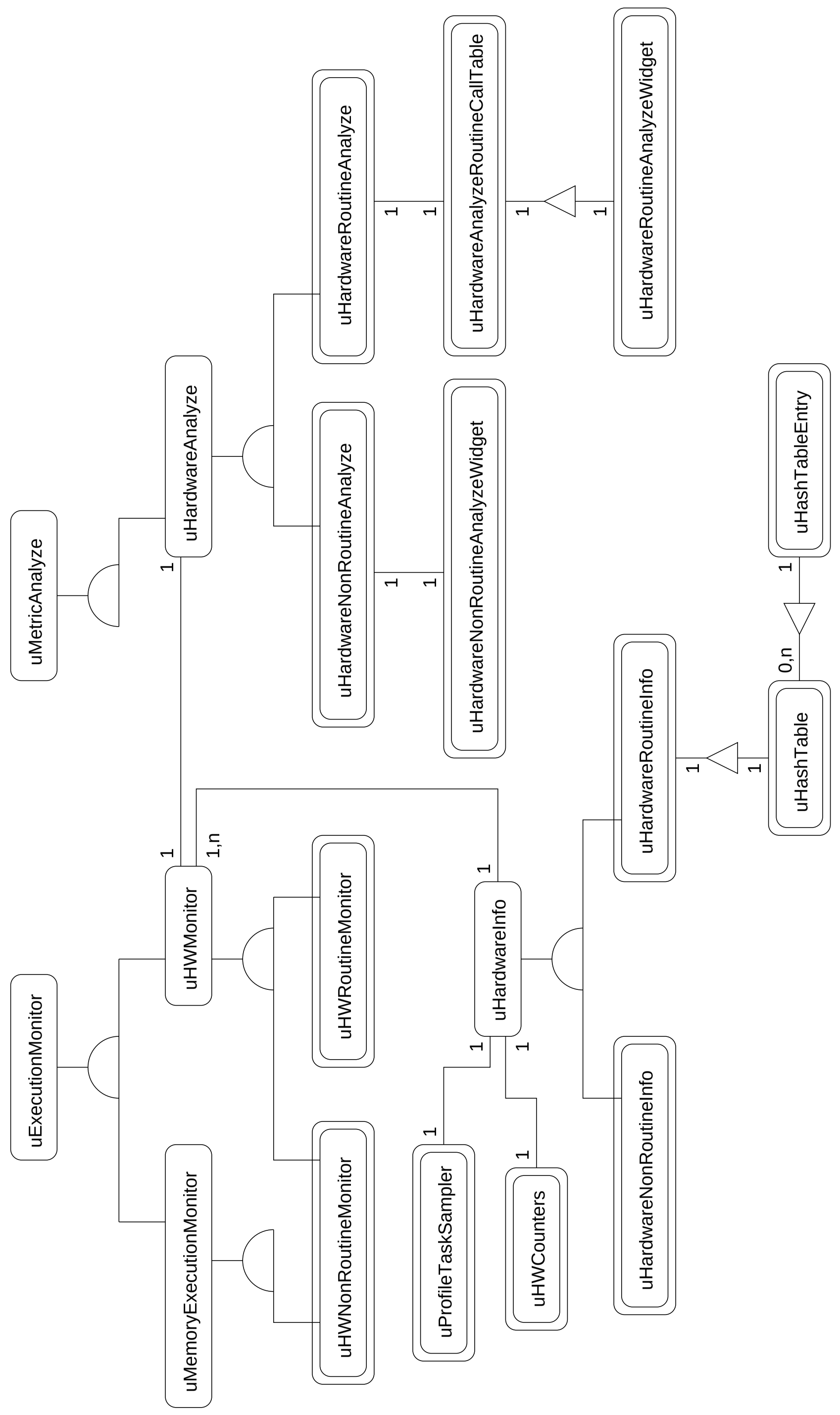


Figure 7.6: Object-oriented design of the Exact Hardware Metric.

laid out in Section 5.5, the metric's execution monitor (uHWMonitor) and analyzer (uHardwareAnalyze) are derived from μProfiler's uExecutionMonitor and uMetricAnalyze base classes, respectively. Further, because this metric is divided into "Routine" and "Non-Routine" modes of operation, its three main classes are specialized by Routine and Non-Routine derived classes, as is explained in the next three sections.

7.2.1 Common Aspects

uHWMonitor, uHardwareAnalyze, and uHardwareInfo are all abstract base-classes that encapsulate commonalities in both the Routine and Non-Routine sides of this metric. Each abstract base-class is discussed below.

uHWMonitor

Since the Exact Hardware Metric is a built-in metric, its execution monitor does not do any data gathering. Thus, uHWMonitor's main responsibility is activating the hooks needed by both the Routine and Non-Routine sides of the Exact Hardware Metric. These hooks are summarized below and represented pictorially in Figure 7.7.

- Task Creation/Destruction: Allows the metric to create/destroy a uProfileTaskSampler for a newly-created/destroyed task.
- Task Start/End Execution: Allows the metric to get a starting/ending hardware-event count for a task's execution.
- Task Block/Unblock: Allows the metric to get an ending/starting hardware-event count for a task's time-slice.
- Processor Creation/Destruction: Allows the metric to create/destroy a hardware-counter context for a processor's underlying kernel thread. These two hooks are not active in uniprocessor mode, since all processors share the same kernel thread.

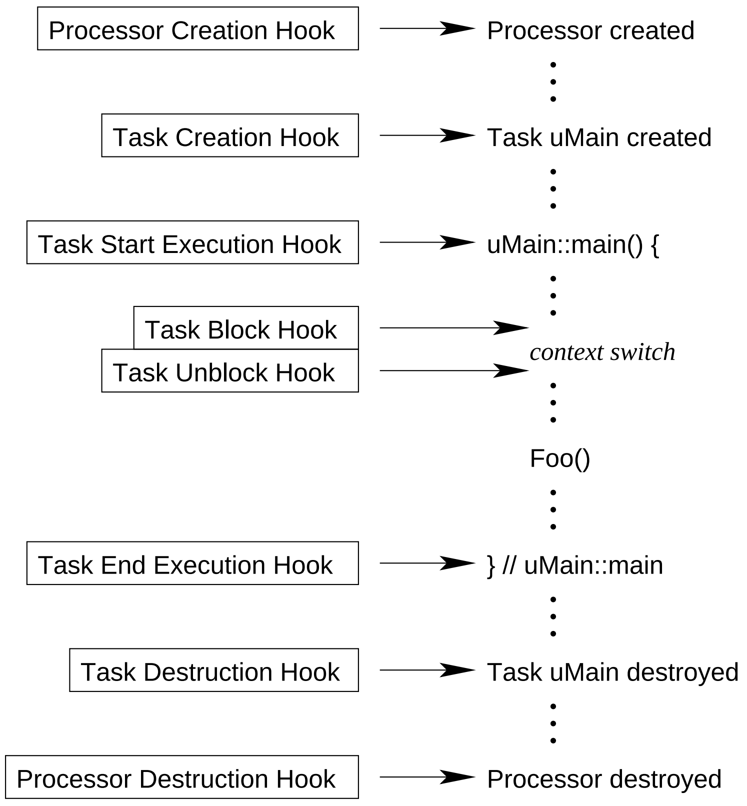


Figure 7.7: Hooks common to Routine and Non-Routine modes.

Each newly-created task also requires a uHardwareInfo object for storing task-specific performance data. However, since the type of the required uHardwareInfo object depends on what mode the metric is in (Routine or Non-Routine), its creation cannot be done in uHWMonitor, but must instead be handled by one of its derived classes. To accomplish this, uHWMonitor adds itself to the Task Creation hook-monitor list, which means it is notified when the Task Creation hook is triggered, i.e., in this one particular case, the Task Creation hook is made to behave like both a user hook and a built-in hook. uHWMonitor's hook-notification routine is a pure virtual routine, which is overridden by uHWMonitor's derived classes, allowing them to create the specialized instance of uHardwareInfo that is required (see Sections 7.2.2 and 7.2.3).

uHardwareAnalyze

uHardwareAnalyze's sole purpose is to create a task-selection box (see Figure 7.2 on page 79), which is common to both modes of operation. Routines to handle a user clicking on a specific task in the selection box are defined by uHardwareAnalyze's derived classes (see Sections 7.2.2 and 7.2.3).

uHardwareInfo

uHardwareInfo stores task-specific information common to both modes of operation, including the task's name and address in memory, and the total hardware-event count over the task's lifetime.

7.2.2 Routine-Specific Aspects

Aspects specific to the Routine mode of operation are encapsulated in the uHWRoutineMonitor, uHardwareRoutineAnalyze, and uHardwareRoutineInfo classes, which are derived from uHWMonitor, uHardwareAnalyze, and uHardwareInfo, respectively.

uHWRoutineMonitor

uHWRoutineMonitor activates the hooks needed only by the Routine side of the Exact Hardware Metric. These hooks are summarized below and represented pictorially in Figure 7.8 along with the hooks activated by uHWMonitor.

- Coroutine Creation/Destruction: Allows the metric to create/destroy a uProfileTaskSampler for a newly-created/destroyed coroutine. The sampler is created solely for the purpose of maintaining a profiling stack for the coroutine, which allows the metric to allocate hardware-event counts to the proper routines when a task is executing on a coroutine's stack rather than its own stack.

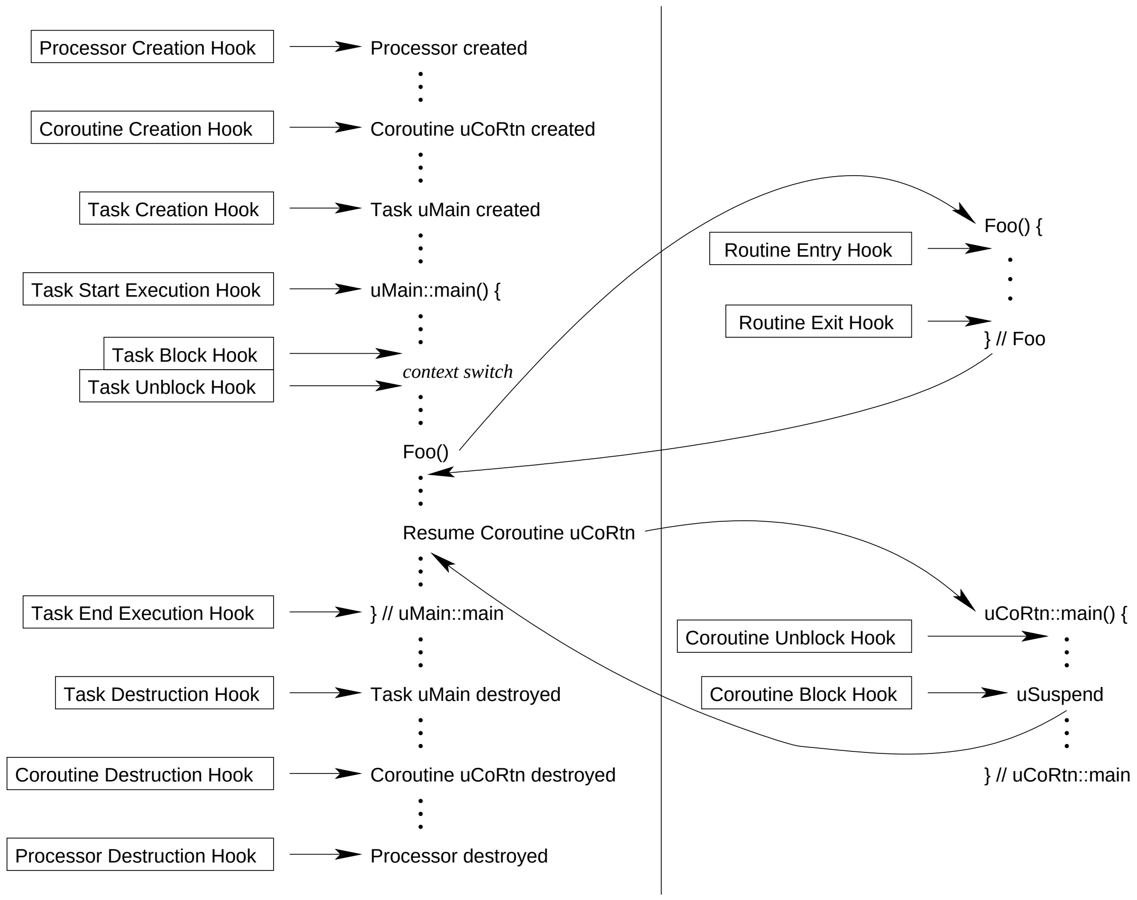


Figure 7.8: Hooks activated in Routine mode.

- Coroutine Block/Unblock: Allows the metric to get an ending/starting hardware-event count for a task's time-slice when it is executing on a coroutine's stack rather than its own stack.
- Routine Entry/Exit: Allows the metric to get a starting/ending hardware-event count for the time a task spends executing in a routine.

uHWRoutineMonitor also provides a definition for the Task Creation hook-notification routine, which is necessary because its base class adds itself to the Task Creation hook-monitor list. The routine creates two task-specific objects: a uHashTable and a uHardwareRoutineInfo (see below). uHashTable is a hash table used to keep track of the caller/callee relationships between routines, as well as the number of occurrences of hardware events in the callee when called by the caller. It was originally written by Dorota Zak for the "Call Graph and Run Time" μProfiler metric [Zak00] to hold only routine-specific timing information, but was extended for this thesis to also include routine-specific hardware-event counts.

uHardwareRoutineAnalyze

uHardwareRoutineAnalyze implements a routine to handle a user clicking on a task in the task-selection box while in Routine mode. This routine creates a uHardwareAnalyzeFuncCallTable object, which analyzes the uHashTable and assigns hardware-event counts to all routines. This object uses the same algorithm as the "Call Graph and Run Time" metric [Zak00], but is customized to analyze hardware-event counts rather than timing information. Once the analysis is complete, the uHardwareAnalyzeFuncCallTable object creates a uHardwareRoutineAnalyzeWidget, which creates a Motif display such as the one seen in Figure 7.3 on page 80.

uHardwareRoutineInfo

The uHardwareRoutineInfo object is responsible for updating its task's total hardware-event counts during time-slicing. When a task begins a new time-slice and triggers the Task Unblock hook, it notifies its uHardwareRoutineInfo object, which then reads and stores the values of the hardware counters. Similarly, when a task ends its current time-slice and triggers the Task Block hook, it notifies its uHardwareRoutineInfo object, which reads the values in the hardware counters, subtracts the values obtained at the start of the time-slice, and adds the difference to the running totals. When a task ends its execution, its uHardwareRoutineInfo object contains the total number of occurrences of each hardware event during its lifetime. These totals are displayed in the task-selection box, as shown in Figure 7.2 on page 79.

7.2.3 Non-Routine-Specific Aspects

The Non-Routine components of the Exact Hardware Metric are handled by the uH-WNonRoutineMonitor, uHardwareNonRoutineAnalyze, and uHardwareNonRoutineInfo classes, which are derived from uHWMonitor, uHardwareAnalyze, and uHardwareInfo, respectively.

uHWNonRoutineMonitor

The Non-Routine side of the Exact Hardware Metric tracks the number of hardware events that occur during each task's time-slices. All of the needed events are covered by the hooks activated by the uHWMonitor base class. However, the Non-Routine mode of this metric requires memory allocation to be done while processing a built-in hook inside the μC++ kernel, so uHWNonRoutineMonitor activates special μC++ kernel memory-allocation hooks by also inheriting from the uMemoryExecutionMonitor class. Refer to Section 7.3.1 for a more in-depth discussion of this issue.

As in Routine mode, uHWNonRoutineMonitor also defines a routine to handle the Task Creation notification message. This routine creates a uHardwareNonRoutineInfo object, which is where time-slice information for the newly-created task is stored.

uHardwareNonRoutineAnalyze

uHardwareNonRoutineAnalyze's purpose is to handle when a user clicks on a task in the task-selection box while in Non-Routine mode. It implements a routine that handles this situation by creating a uHardwareNonRoutineAnalyzeWidget, which goes through a task's list of time-slices and displays information from each one on a separate line in a Motif widget. The result is a display such as the one seen in Figure 7.5 on page 82.

uHardwareNonRoutineInfo

The uHardwareNonRoutineInfo object is responsible for gathering time-slice information when its associated task blocks and unblocks, so when it is first instantiated, it creates an empty linked list for storing this time-slice data. uHardwareNonRoutineInfo also provides routines for gathering and storing this time-slice information. When a task begins a new time-slice, it triggers the Task Unblock hook and notifies its uHardwareNonRoutineInfo object, which then allocates a new time-slice structure, stores the cluster name, current processor address, UNIX PID and hardware-event counts in it, and adds it to the tail of the time-slice linked list. Similarly, when a task ends its current time-slice, it triggers the Task Block hook and notifies its uHardwareNonRoutineInfo object, which stores the current hardware-event counts in the time-slice structure at the tail of its linked list. At the end of a task's lifetime, its uHardwareNonRoutineInfo object has a complete, ordered list of all its time-slices, including starting and ending hardware-event counts.

7.3 Implementation Issues

This section provides discussion on some of the implementation issues I encountered and solved while writing the Exact Hardware Metric.

7.3.1 Dynamic Memory Allocation in the μC++ Kernel

A metric uses hooks to gather information at appropriate times during execution of a program. When a hook is triggered, a metric usually performs three steps:

1. Allocate memory to store performance data.
2. Obtain the data.
3. Copy the data into the allocated memory and connect it to other related data.

Preallocation of all necessary storage can only be done by simple metrics, such as counting the total number of tasks created, or obtaining the creation time of each task. Typical metrics are more complex, and need to store multiple data samples during a task's lifetime, which are then analyzed post-mortem. In general, a metric has no problems accomplishing the necessary three steps.

However, a problem can occur depending on the placement of a μProfiler hook. The Task Block/Unblock hooks are conceptually placed in the μC++ kernel because their execution cannot be interrupted. For example, if a metric is storing state transitions (e.g., ready, running, and blocking states), it is essential for a task to store the blocking transition and block without interruption. If the task could be interrupted between storing the state data and it blocking, it would eventually attempt to block again, resulting in two consecutive blocking states being stored with no intervening ready or running states, which is logically inconsistent. So the placement of these hooks with respect to data gathering is crucial to generate consistent results for metrics needing this kind of information. In these special cases, the operation that triggers the hook and the subsequent metric data-gathering must be atomic.

Unfortunately, it is impossible to perform a dynamic storage-allocation from within the Task Block/Unblock hooks, which is an essential first step for a complex metric. The problem is that storage allocation may require blocking the task requesting storage. However, this task is running in the μC++ kernel, and the kernel cannot block, i.e., enter itself recursively to schedule another task. While it might be possible, in general, for a kernel to allow this complex behaviour, the μC++ kernel does not.

There are only two options to deal with this problem: either preallocate storage for the current performance data before entering the kernel to gather it, or postallocate storage after coming out of the kernel in anticipation of the next piece of performance data. However, it is now possible for a task to be interrupted with respect to either memory-allocation approach, which causes problems. The following discussion presents a solution to these problems.

Memory-Allocation Schemes

Figure 7.9 illustrates the two methods that can be used to allocate memory for performance-data gathering by hooks in the μC++ kernel. In the preallocation scheme, there exists a window between the time a task becomes aware that it is about to enter the kernel, and the time that it actually does enter the kernel. Memory is allocated while in this window, so that the Task Block/Unblock hooks have a place to store the collected peformance data. In the postallocation scheme, one initial memory allocation is done when a task is created, so a storage block is in place for use by the Task Block/Unblock hooks during the first kernel entry. When a task unblocks, there is a window between the time it exits the kernel, and the time that it resumes execution at the point where it was interrupted. During this window, another block of memory is allocated for the hooks to use during the next kernel entry.

Spinlocks

The μC++ kernel protects critical internal data structures with *spinlocks*. A task that requires access to any such data structure must first acquire its spinlock. Since entering the μC++ kernel is an expensive operation, tasks are permitted to acquire these spinlocks and access these data structures outside the kernel. However, while a task is holding a spinlock, μC++ does not permit it to enter the kernel and block, otherwise problems could occur. For example, consider what happens if a task enters the kernel and blocks while holding the spinlock for its cluster's ready queue. Since the kernel cannot acquire this spinlock to access and modify that queue, no other task can be scheduled to run in place of the blocked task, and a live-lock results. While it might be possible, in general, for a kernel to allow this complex behaviour, the μC++ kernel does not.

A task has two options if it attempts to enter the kernel while holding a spinlock. The first option is to postpone the kernel entry and set a "postponement" flag, which is checked when a task eventually releases the spinlock. Upon releasing its spinlock and finding its "postponement" flag set, a task immediately resumes

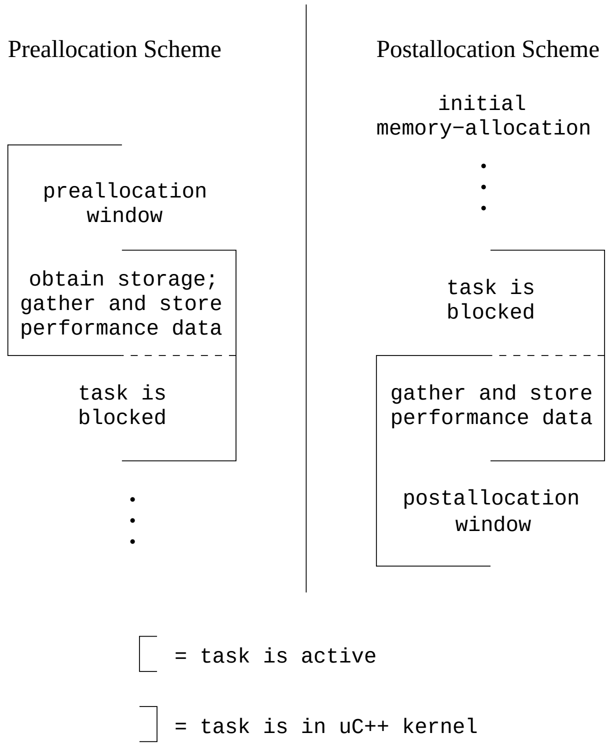
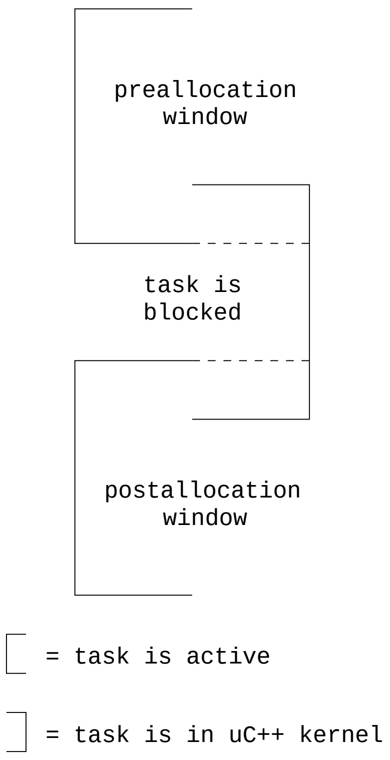


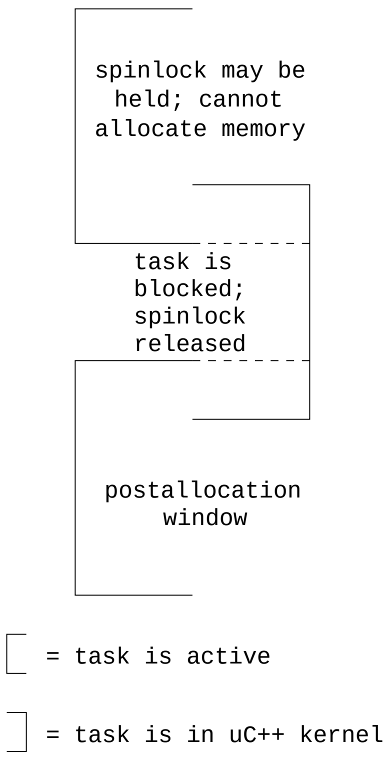
Figure 7.9: Memory-allocation schemes.

its postponed kernel entry and blocks. This option is used for *involuntary kernel entries*, which are explained in the next section. The second option is for a task to atomically release its spinlock as it blocks. This option is used for *voluntary kernel entries*, which are also covered in the next section.

In μC++, spinlocks are the source of a problem with respect to memory allocation. The μC++ memory manager is built in such a way that memory allocations and deallocations are potential blocking operations, i.e., they may cause a kernel entry. For this reason, a task may not allocate or deallocate memory while holding a spinlock, which then presents a problem during profiling.



(a) Involuntary kernel entry


spinlock may be held; cannot allocate memory
task is blocked; spinlock released
postallocation window
= task is active
= task is in uC++ kernel

(b) Voluntary kernel entry

Figure 7.10: Memory-allocation windows for kernel entries.

Involuntary vs. Voluntary Kernel Entry

When a task reaches the end of its time-slice, it is delivered a signal, which attempts to force it to *involuntarily* enter the μC++ kernel and block. However, if the task holds a spinlock when the signal is received, the kernel entry is postponed as described above. When the task finally proceeds with the postponed kernel entry (or if the kernel entry was not postponed in the first place), it is guaranteed not to be holding a spinlock. Thus, for all involuntary kernel entries, there exist two windows during which memory allocations can take place: one between receiving a preemption signal and actually entering the kernel, and one between exiting the kernel and resuming normal execution (Figure 7.10(a)).

Anytime a task enters the μC++ kernel and blocks as a direct result of its own actions, it is said to enter the kernel *voluntarily*. Voluntary kernel entries are caused by a number of different situations, including:

- A task blocking itself by calling yield.
- A task blocking on an accept queue while trying to enter a mutex routine.
- A task blocking on a monitor's internal condition variable.

Voluntary kernel entries differ from involuntary kernel entries in that a task is allowed to hold a spinlock until the moment it blocks, at which time the spinlock is atomically released. In this case, when the task eventually unblocks and exits the kernel, it is guaranteed not to be holding a spinlock. Thus, for all voluntary kernel entries, there exists a single window after exiting the kernel, but before resuming normal execution, where memory allocations can take place (Figure 7.10(b)).

7.3.2 Solution to the Memory-Allocation Problem

Each type of kernel entry provides at least one window of opportunity just outside the kernel where memory allocations can take place. To allow μProfiler metrics to allocate memory in those windows, I created a special set of hooks called *μC++ kernel memory-allocation hooks*. Metrics activate these hooks by deriving their execution monitors from the uMemoryExecutionMonitor abstract base-class. When its constructor is invoked, uMemoryExecutionMonitor arms the memory-allocation hooks, and requests a *metric-memory index*, which is a unique index used to obtain memory blocks from an array of pointers to pre/postallocated memory. The memory-allocation hooks allocate blocks of memory for each task at the appropriate time, and pass pointers to these blocks to each task's uProfileTaskSampler, where they are retrieved by the metric as needed (see Figure 7.11). This process is explained in depth below for both types of kernel entry.

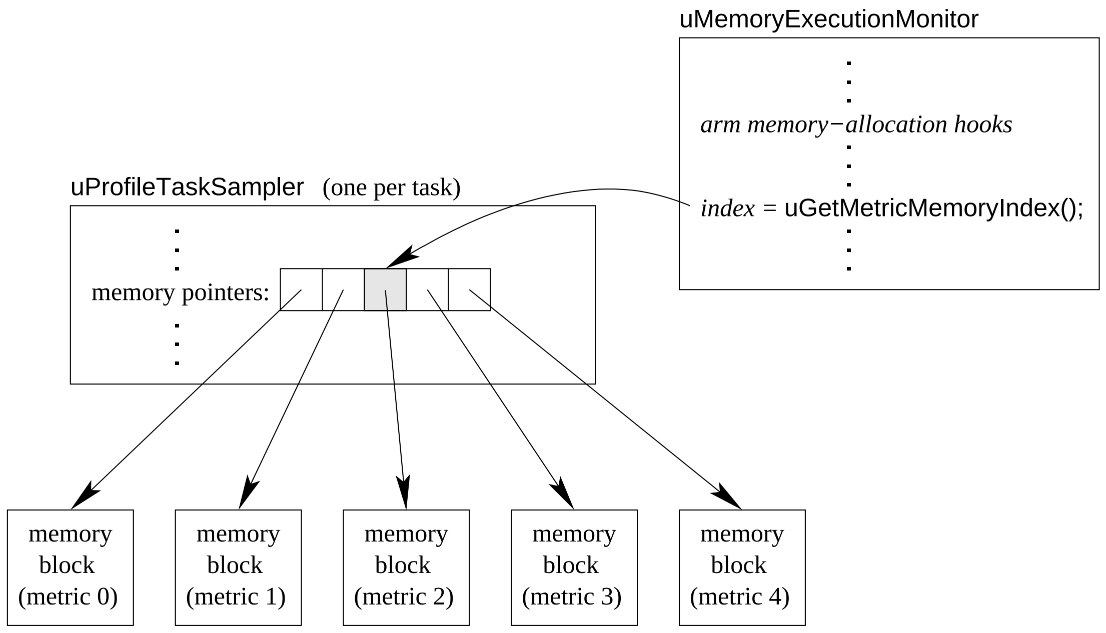


Figure 7.11: Memory blocks for μProfiler metrics.

Involuntary Kernel Entries

Involuntary kernel entries have two windows during which memory allocations can take place: one just before the kernel entry, and one just after the kernel exit. However, because of a reentrancy issue, a postallocation scheme, i.e., allocating memory after exiting the kernel, is infeasible. Assume the use of a postallocation scheme for involuntary kernel entries. An initial allocation is done when each task is created, so performance-data storage is in place for each task's first kernel entry. When a task involuntarily enters the kernel and blocks, this storage is used up. Upon exiting the kernel, the task replaces the memory it consumes by performing a postallocation. However, the moment the task returns from the kernel, it may be preempted. If preemption occurs before the task is able to replace the storage it consumes, then it has nowhere to store its performance data as it reenters the kernel (see Figure 7.12).

Reentrancy is not an issue for a preallocation scheme, because a task allocates storage for metric data before each kernel entry. So even if a task is preempted

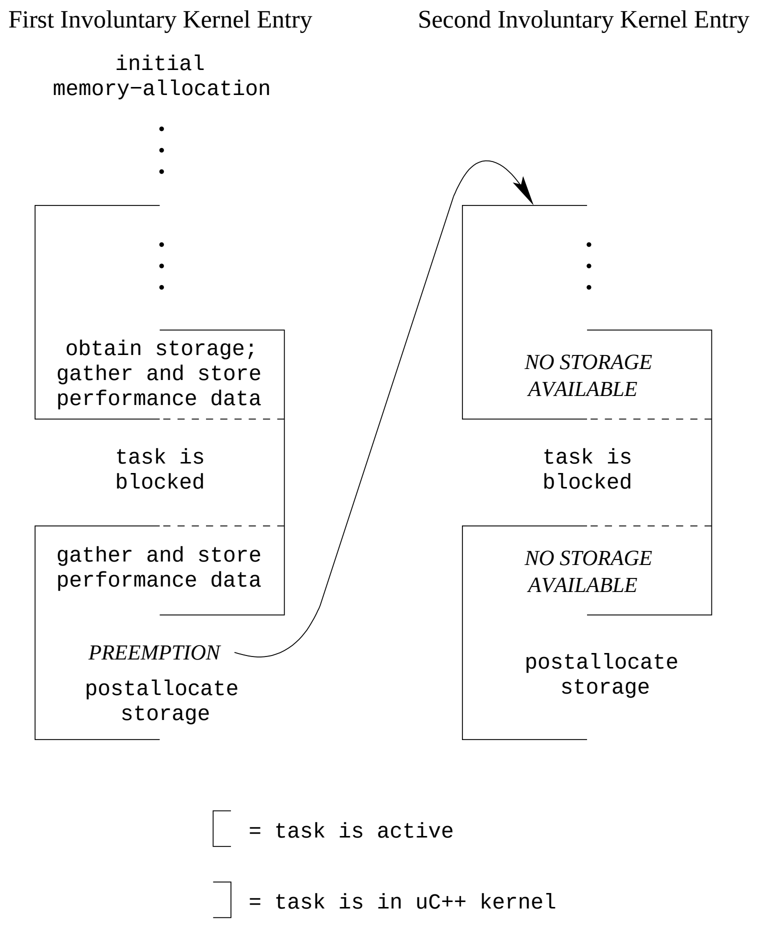

First Involuntary Kernel Entry
Second Involuntary Kernel Entry
initial memory-allocation
obtain storage; gather and store performance data
task is blocked
gather and store performance data
PREEMPTION
postallocate storage
NO STORAGE AVAILABLE
task is blocked
NO STORAGE AVAILABLE
postallocate storage
= task is active
= task is in uC++ kernel

Figure 7.12: Postallocation scheme failure for involuntary kernel entries.

before it can preallocate storage for its impending kernel entry, the preallocations are done in such a way that both kernel entries have storage for their metric data (see Figure 7.13). Therefore, a preallocation scheme is the only safe way to allocate memory for involuntary kernel entries.

To perform the preallocation, I wrote a new routine called uYieldInvoluntary, through which all involuntary μC++ kernel entries are now funneled. Any task that reaches this routine is guaranteed not to hold a spinlock, so it is safe to do memory allocations. Therefore, a *memory-preallocation hook* is placed in uYield-Involuntary just before a task enters the kernel (Figure 7.14 on page 99). If the

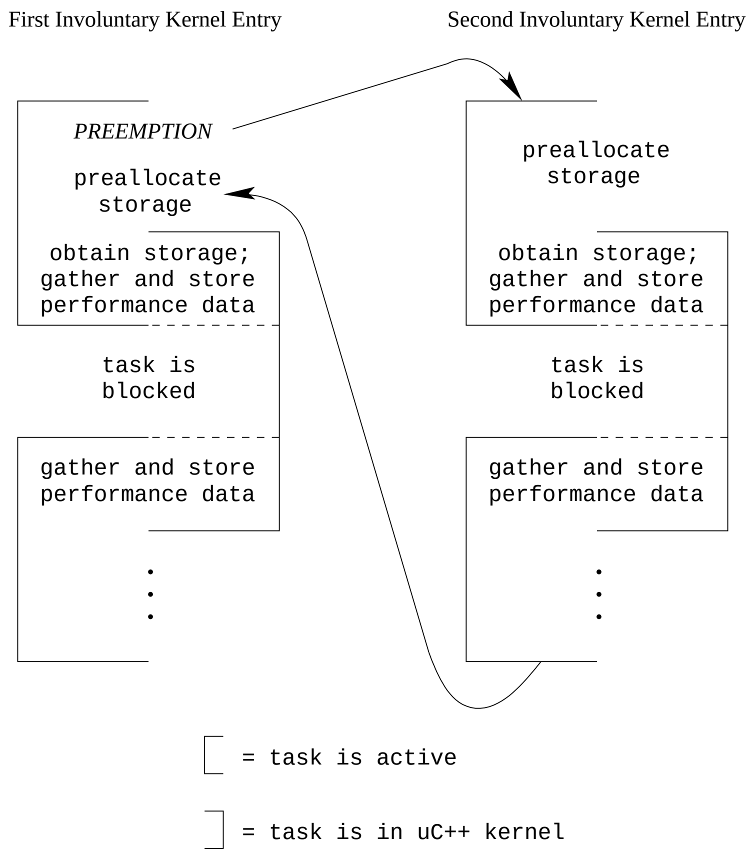

First Involuntary Kernel Entry
Second Involuntary Kernel Entry
PREEMPTION
preallocate storage
obtain storage; gather and store performance data
task is blocked
gather and store performance data
preallocate storage
obtain storage; gather and store performance data
task is blocked
gather and store performance data
= task is active
= task is in uC++ kernel

Figure 7.13: Preallocation scheme for reentrant involuntary kernel entries.

memory-allocation hooks are active, a local array of memory pointers is created for storing pointers to memory blocks (one for each metric that requires memory at time-slices). Then the memory-preallocation hook is called, which allocates one block of memory for each metric that requires it, and stores pointers to them in the local array. Finally, a pointer to the local array is inserted into the current task's uProfileTaskSampler, and the task enters the kernel and blocks. While in the kernel, the Task Block/Unblock hooks are triggered, which causes performance data to be gathered for all appropriate metrics. Any metric that requires a memory block to store profiling data, simply retrieves one from the cell corresponding to its metric-memory index in the array that was inserted into the current task's uProfileTaskSampler. The cell from which the memory block pointer is taken is set

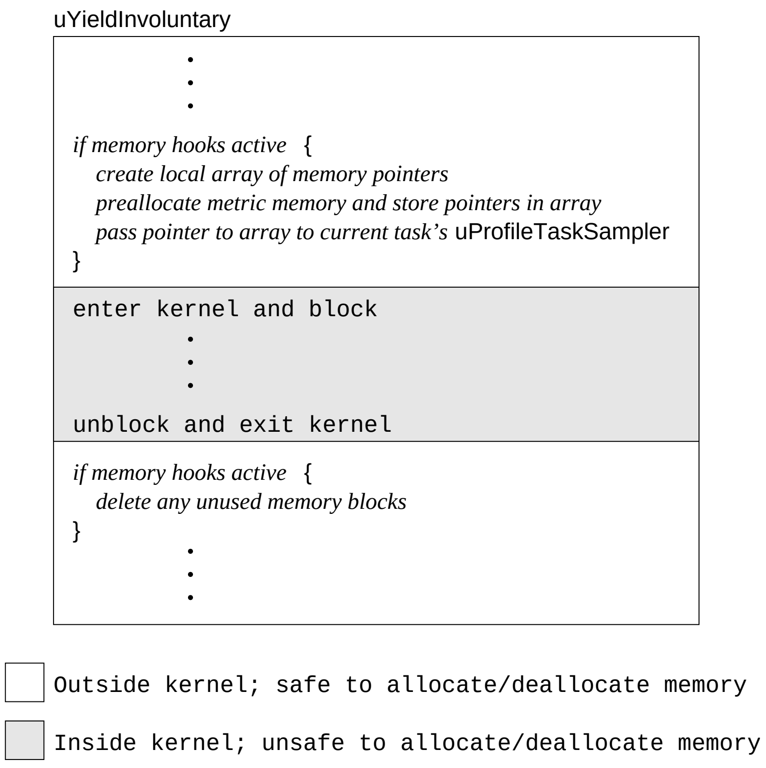

uYieldInvoluntary
if memory hooks active {
create local array of memory pointers
preallocate metric memory and store pointers in array
pass pointer to array to current task's uProfileTaskSampler
}
enter kernel and block
unblock and exit kernel
if memory hooks active {
delete any unused memory blocks
}
Outside kernel; safe to allocate/deallocate memory
Inside kernel; unsafe to allocate/deallocate memory

Figure 7.14: Memory-preallocation hook for involuntary kernel entries.

to NULL to indicate that it has been consumed. Finally, when the task exits the kernel, any of the preallocated blocks in the local memory array that were not retrieved by their metrics are deleted. This final step is in place for future metrics that may require memory-allocation in the μC++ kernel, but may not consume a block at each and every time-slice. While there is a performance inefficiency for this scenario, there does not seem to be any better alternative.

Voluntary Kernel Entries

The preallocation scheme discussed above does not work for voluntary kernel entries because in these cases, a task is allowed to hold a spinlock until the moment it enters the kernel. Therefore, the only option is a postallocation scheme. Also, the voluntary situation is much simpler, since it is not affected by any reentrancy issues.

A task cannot voluntarily enter the kernel until its previous voluntary kernel entry is complete, and if it is preempted at any point, the resulting kernel entry is now involuntary. Since reentrancy is not an issue, only one set of memory blocks per task need be available at any given time. To that end, an array of memory pointers was added to the uProfileTaskSampler class, to be used only in the case of voluntary kernel entries. This array is independent of the array of memory pointers that is passed into the uProfileTaskSampler during involuntary kernel entries.

When a task is created, it checks if the memory-allocation hooks are active. If so, it allocates a block of memory for each metric that obtained a metric-memory index, and stores pointers to them in the uProfileTaskSampler's "voluntary kernel-entry array", which guarantees that memory blocks are in place for a task's first voluntary kernel entry. When a task exits the kernel after a voluntary entry, the Task Block/Unblock hooks may have consumed some of the memory blocks stored in the task's uProfileTaskSampler, so some blocks may need to be replaced. Thus, a *memory-postallocation hook* is placed immediately after the kernel exit for every voluntary kernel entry (see Figure 7.15). When a task triggers this hook, any blocks that were used by the Task Block/Unblock hooks are replaced, so they are available for the next voluntary kernel entry.

7.3.3 Interrupts While Reading Hardware Counters

On most architectures, reading the hardware counters is not an atomic operation. For example, on the x86 architecture, it is necessary to read each hardware counter with a separate routine call. This lack of atomicity can lead to incorrect hardware-event counts if a task is interrupted while it is reading the counters.

Consider a task that is in the process of reading four hardware counters, and assume that an interrupt occurs immediately after the second counter is read. The task is context-switched out, but since hardware-counter contexts are processor-specific, hardware events continue to be counted while the task is blocked. At best, when the task unblocks, it is assigned to the same processor as its previous time-slice, and the remaining two hardware counters provide inflated event counts. At

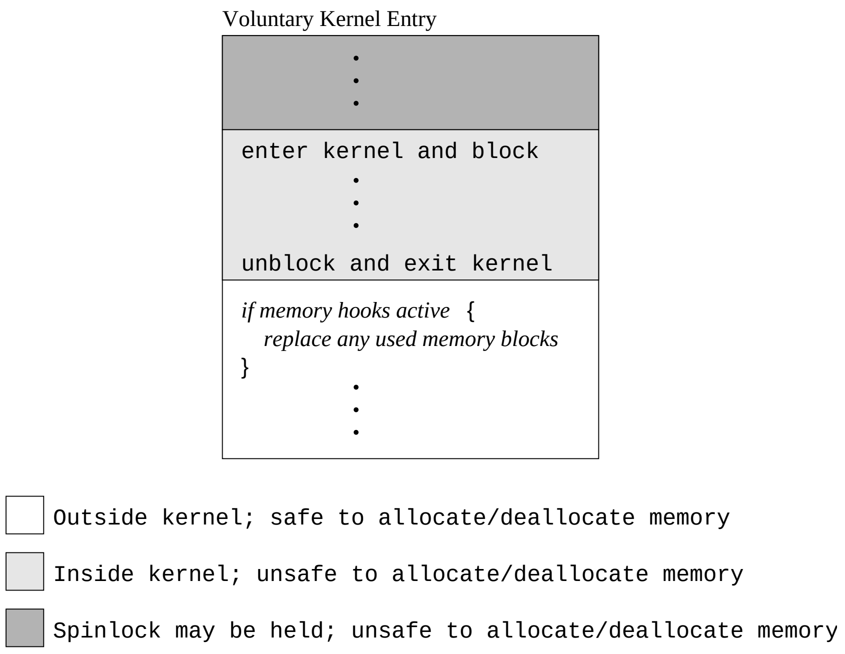

Voluntary Kernel Entry
enter kernel and block
unblock and exit kernel
if memory hooks active {
replace any used memory blocks
}
Outside kernel; safe to allocate/deallocate memory
Inside kernel; unsafe to allocate/deallocate memory
Spinlock may be held; unsafe to allocate/deallocate memory

Figure 7.15: Memory-postallocation hook for voluntary kernel entries.

worst, the task is assigned to a different processor, and the remaining two hardware counters provide completely irrelevant event counts. Either way, the results are incorrect and undesirable.

To eliminate this problem, it is necessary to prevent interrupts from occurring while the hardware counters are being read. Therefore, every call site for the uReadCounters routine was examined. Those call sites that are in areas where interrupts are allowed to occur are bracketed by calls to the uDisableInterrupts and uEnableInterrupts routines, which disable and enable interrupts, respectively, at the virtual-processor level. Finally, to guard against any future calls being made to uReadCounters with interrupts enabled, an assertion was placed in that routine, which verifies that interrupts are disabled.

7.4 Validation

This section provides validation for the Exact Hardware Metric by verifying that it produces correct results for a variety of simple μC++ programs. Since I was unable to find any profilers that use hardware counters for exact monitoring, a simple exact-monitoring test-harness from the perfmon website was used as an experimental control. The perfmon test-harness was used to profile a piece of code, and the results are used as a control for all Exact Hardware Metric validation tests.

Validation testing is performed on a dual-processor 900 MHz Itanium 2 machine, with *Completed Instructions* and *CPU Cycles* as the benchmark hardware-events. The reason these events are used is that they produce nearly identical event-counts across multiple program runs, which is important for establishing consistent results for comparison purposes. Many other hardware events tend to produce event counts that vary across program runs. For example, consider cache memory, which is a shared resource. Any program counting cache-related events can produce event counts that vary over time, depending on the number of processes actively competing with it for access to the cache.

The goal of these validation tests is to show that the Exact Hardware Metric is properly gathering hardware-event counts. I argue that if the Exact Hardware Metric produces correct counts for hardware events with a predictable value, then the data gathering is being done properly. Consequently, if the data gathering is being done properly, then the Exact Hardware Metric is able to gather proper event counts for *any* hardware event, since the selection of which hardware events to count is completely independent of the methods used to actually count them.

7.4.1 perfmon Test-Harness

The perfmon website [per] provides a general hardware-counter test-harness into which specific tests can be placed for gathering exact hardware-event counts. The harness configures the hardware counters to count Completed Instructions and CPU Cycles, and the specific test is inserted between the comments (see Appendix

B.1.1). The program's output shows the number of Completed Instructions and CPU Cycles that occur while the test code executes. I modified this program so it loops through the profiled section of code twenty times, and then displays the total number of Instructions and CPU Cycles over all twenty iterations. This addition is important for comparison with other validation tests, as some divide the workload among twenty tasks, each executing the equivalent of one iteration of the test code.

7.4.2 Testing Strategy and Hypothesis

The Exact Hardware Metric presents hardware-event counts in three different ways:

1. Per-task aggregate hardware-event counts (see Figure 7.2 on page 79).
2. Per-routine hardware-event counts (see Figure 7.3 on page 80).
3. Non-routine (or time-slice) hardware-event counts (see Figure 7.5 on page 82).

The three portions of the Exact Hardware Metric are validated with separate tests, each of which profiles the same code under different conditions. Each validation test consists of three separate experiments, and the results are compared to the perfmon control (see Section 7.4.3). The first is a purely sequential experiment, consisting of a μC++ program that uses no tasks other than the standard uMain task, which is time-sliced at regular intervals. The second experiment is concurrent, and consists of a μC++ program that divides its workload evenly among twenty tasks on five virtual processors, but the program is run in uniprocessor mode, so only one kernel thread is used with time-slicing. The third experiment is a parallel experiment, and it uses the same program as its concurrent counterpart, but runs it in multiprocessor mode, which uses a separate kernel thread with time-slicing for each virtual processor. The total event-counts for each experiment are reported, as well as the percentage difference with respect to the perfmon control. The purpose of this trio of experiments is to show that the Exact Hardware Metric gathers hardware-counter performance data correctly in all of μC++'s execution modes.

Note, five virtual processors are used, even though the underlying system has only two physical processors, which is done to increase the chances of contention among the virtual processors as they compete for the shared physical processors, which should give results that more accurately reflect the behaviour of concurrent μC++ programs.

I hypothesize that each Exact Hardware Metric validation test will produce hardware-event counts that are close to the perfmon control. However, μC++ tasks incur overhead, and hence, extra instructions and execution time, that is not present in a normal C++ program, such as entering and exiting the μC++ kernel at every time-slice. Furthermore, if multiple tasks are executing in the same program concurrently, they compete with each other for access to shared virtual processors. Similarly, in programs that run in multiprocessor mode, virtual processors' kernel threads compete for access to shared physical processors. This competition introduces overhead that does not exist in a sequential program. For these reasons, the parallel experiments should produce larger event counts than the concurrent ones, which in turn should produce larger event counts than the sequential ones. Finally, the sequential experiments should produce larger event counts than the perfmon control, due to time-slicing (while time-slicing can be turned off, it would not be representative of a normal μC++ program).

7.4.3 perfmon Control

The perfmon control code is listed in Figure 7.16, and consists of three nested loops of 1000, 1000, and 10 iterations, respectively, for a total of 10000000 iterations of the inner loop. Each loop performs one or more mathematical calculations to ensure a reasonably large number of instructions and CPU cycles occur. This code is useful as a control because it is small enough to allow many experiments to be run quickly, yet it is clearly nontrivial.

```
int w = 14, x = 99, y = 37, z = 24;

for ( int i = 0; i < 1000; ++i ) {
    w += w + x + y + z;
    x += w - x - y - z;
    for ( int j = 0; j < 1000; ++j ) {
        y += w - x + y + z;
        for ( int k = 0; k < 10; ++k ) {
            z += w + x - y - z;
        } // for
    } // for
} // for
```

Figure 7.16: perfmon control code.

7.4.4 Aggregate Event-Count Test

The sequential and concurrent programs used in the aggregate event-count test are listed in Figures 7.17(a) and 7.17(b). In the sequential version, the uMain::main routine simply executes the perfmon control code twenty times, while in the concurrent version, twenty tasks are created, each executing one iteration. Table 7.1 presents the total event-counts from each test, as well as the percentage difference relative to the perfmon control.

	Instructions		CPU Cycles	
	Total	Diff. (%)	Total	Diff. (%)
perfmon:	7281301480	0.000	4061099726	0.000
Sequential:	7281449164	0.002	4061131384	0.001
Concurrent:	7281457715	0.002	4061321703	0.005
Parallel:	7281663154	0.005	4061565277	0.011

Table 7.1: Results from the aggregate event-count test.

The results of these experiments all closely match those of the perfmon control. They also agree with my hypothesis of a slight rise as the experiments move from sequential to parallel.

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

void uMain::main() {
    for ( int iter = 0; iter < 20; ++iter ) {
        // perfmon control code
    } // for
} // uMain::main
```

(a) Sequential

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

uTask Test {
    void main() {
        // perfmon control code
    } // main
}; // Test

void uMain::main() {
    uProcessor processors[5];
    Test test[20];
} // uMain::main
```

(b) Concurrent

Figure 7.17: Aggregate event-count test programs.

7.4.5 Routine Test

One validation test was run to verify that the Routine mode of the Exact Hardware Metric properly counts hardware events on a per-routine basis. The sequential and concurrent test programs are listed in Figures 7.18(a) and 7.18(b). Both programs insert the perfmon control code into two routines called one and two. In the sequential program, the uMain::main routine invokes these two routines twenty times each, while in the concurrent program, twenty tasks are created, with each one invoking the two routines once each. If the Exact Hardware Metric is working correctly, the event counts in routines one and two should be close to each other, and close to the perfmon control as well. Also, as per my hypothesis in Section 7.4.2, the event counts should rise as the experiments move from sequential to parallel. The results from these experiments are listed in Table 7.2 on page 108. The *Diff* column is the percentage difference between the event counts of routines one and two, and is with respect to the results of routine one. The perfmon control is included in the first row of the table for comparison purposes.

The results of these experiments agree with my hypothesis. The hardware-event counts for the two routines are close, for all three experiments. Furthermore, the

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

void one() {
    // perfmon control code
} // one

void two() {
    // perfmon control code
} // two

void uMain::main() {
    for ( int iter = 0; iter < 20; ++iter ) {
        one();
        two();
    } // for
} // uMain::main
```

(a) Sequential

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

void one() {
    // perfmon control code
} // one

void two() {
    // perfmon control code
} // two

uTask Test {
    void main() {
        one();
        two();
    } // main
}; // Test

void uMain::main() {
    uProcessor processors[5];
    Test test[20];
} // uMain::main
```

(b) Concurrent

Figure 7.18: Routine test programs.

event counts all rise as the experiments move from sequential to parallel. Finally, all event counts are close to the perfmon control, with the maximum deviation being 0.016% for the CPU Cycles event-count of routine one during the parallel experiment.

7.4.6 Non-Routine Test

One test was used to verify that the Non-Routine mode of the Exact Hardware Metric properly counts events on a time-slice basis. The sequential and concurrent programs used for this test are listed in Figures 7.19(a) and 7.19(b) on page 109. Each program creates two clusters with five virtual processors each. Tasks migrate

	Instructions			CPU Cycles		
	one	two	Diff. (%)	one	two	Diff. (%)
perf:	7281301480			4061099726		
Seq:	7281478089	7281467560	0.000	4061298051	4061286374	0.000
Con:	7281479366	7281468742	0.000	4061324056	4061316322	0.000
Par:	7281664900	7281619936	-0.001	4061765548	4061729534	-0.001

Table 7.2: Results from the routine test.

to the first cluster and execute the perfmon control code, then migrate to the second cluster and execute that same code again. In the sequential program, the only task is uMain, and it executes the perfmon control code twenty times on each cluster. In the concurrent program, twenty tasks are created, each of which executes the perfmon control code only once per cluster. If the Exact Hardware Metric is functioning properly, the event counts on each cluster should be close to each other, and close to the perfmon control as well. Furthermore, in accordance with my hypothesis in Section 7.4.2, the event counts should all rise as the experiments move from sequential to parallel. Results from these experiments are listed in Table 7.3. The *Diff* column is the percentage difference between the event counts of Cluster 1 and Cluster 2, and is with respect to the results of Cluster 1. The perfmon control is included in the first row of the table for comparison purposes.

	Instructions			CPU Cycles		
	Cluster 1	Cluster 2	Diff. (%)	Cluster 1	Cluster 2	Diff. (%)
perf:	7281301480			4061099726		
Seq:	7281451736	7281454920	0.000	4061123557	4061347642	0.006
Con:	7281512396	7281476190	-0.000	4061255122	4061112094	-0.004
Par:	7281786546	7281600708	-0.003	4061886354	4061691154	-0.005

Table 7.3: Results from the non-routine test.

The hardware-event counts for the two clusters are close for all three experiments, as my hypothesis predicted. Furthermore, except for one case (which is underlined), the event counts all rise as the experiments move from sequential to parallel. The reason for this anomaly is unclear. However, all results are slightly

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

uCluster cluster1( "TestCluster1" );
uProcessor processor1( cluster1 );
uProcessor processor2( cluster1 );
uProcessor processor3( cluster1 );
uProcessor processor4( cluster1 );
uProcessor processor5( cluster1 );

uCluster cluster2( "TestCluster2" );
uProcessor processor6( cluster2 );
uProcessor processor7( cluster2 );
uProcessor processor8( cluster2 );
uProcessor processor9( cluster2 );
uProcessor processor10( cluster2 );

void uMain::main() {
    migrate( cluster1 );
    // perfmon control code
    migrate( cluster2 );
    // perfmon control code
} // uMain::main
```

(a) Sequential

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

uCluster cluster1( "TestCluster1" );
uProcessor processor1( cluster1 );
uProcessor processor2( cluster1 );
uProcessor processor3( cluster1 );
uProcessor processor4( cluster1 );
uProcessor processor5( cluster1 );

uCluster cluster2( "TestCluster2" );
uProcessor processor6( cluster2 );
uProcessor processor7( cluster2 );
uProcessor processor8( cluster2 );
uProcessor processor9( cluster2 );
uProcessor processor10( cluster2 );

uTask Test {
    void main() {
        migrate( cluster1 );
        // perfmon control code
        migrate( cluster2 );
        // perfmon control code
    } // main
}; // Test

void uMain::main() {
    Test test[20];
} // uMain::main
```

(b) Concurrent

Figure 7.19: Non-routine test programs.

higher than the perfmon control, and close enough to it to conclude that event counts are being gathered properly. The maximum deviation is 0.019% for the CPU Cycles event-count of Cluster 1 during the parallel experiment.

7.4.7 Summary

These experiments show that the Exact Hardware Metric properly counts hardware events on a per-task, per-routine, and per-time-slice basis, for programs executing sequentially, concurrently, and in parallel. Moreover, in all but one case, the hardware-event counts rose as the experiments moved from sequential to parallel, as my hypothesis predicted.

Chapter 8

Statistical Profiling Metric

This chapter discusses the final major contribution of this thesis: the Statistical Profiling Metric. This built-in metric is a complete rewrite of the original Statistical Profiling Metric by Robert Denda [Den97], with many fundamental changes in design, and of course, the addition of hardware-counter capabilities. The objective of this metric is to allow users to obtain a per-task statistical profile of a μC++ program using sampling periods based on hardware-event counts. A statistical profile for each task includes:

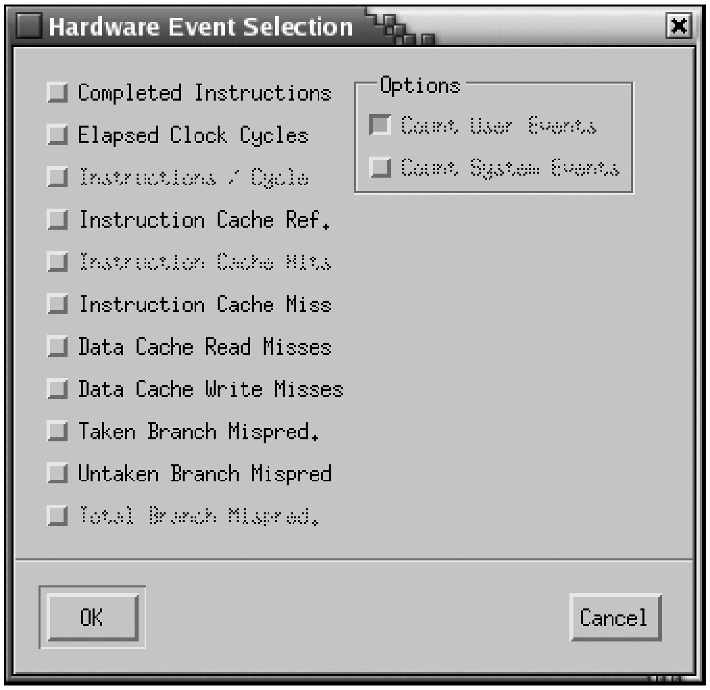
- A "flat" histogram showing the distribution of samples across the task's executed routines.
- A call-graph showing the propagation of samples up the task's call-tree.
- A list of call-cycles found in the task's call-graph.

The call-graph has two features currently unavailable in any other statistical call-graph. The first feature combines samples from multiple hardware events into one display, which gives the user the opportunity to compare the results from statistically sampling on multiple hardware events without having to run a program multiple times. The second feature is that it is constructed with complete call-stack samples, rather than single routine samples. While this does not mean the call-graph is complete (sampling may not cover all the routines executed by a task), it does guarantee that the call-graph is connected.

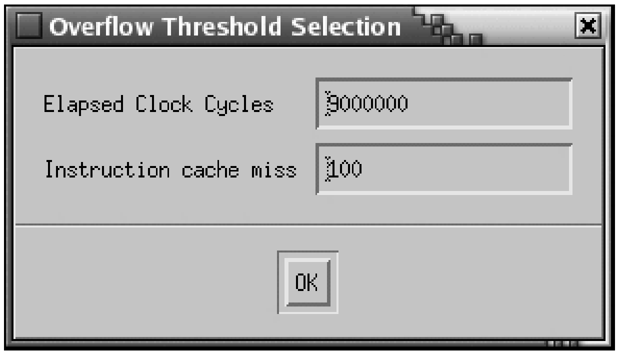
8.1 Functionality

In the “Statistical Profiling” frame of the μProfiler startup window (Figure 5.4 on page 55), the user is given two choices: “Sample by Time” and “Sample by Hardware Event(s)”. Clicking on the “Sample by Time” button sensitizes the slider underneath it, which allows the user to choose a sampling rate in milliseconds. This time-based profiling option is in place for three reasons. First, it provides a statistical-profiling option for systems that do not have a fully functioning set of hardware counters. For example, x86 machines require an Advanced Programmable Interrupt Controller (APIC) to support hardware-counter overflow monitoring [Adv02, Int05]. Machines lacking an APIC cannot use hardware counters for statistical profiling, so time-based statistical profiling is their only option. The second reason is that the x86 architecture uses a special register, called the Time-Stamp Counter (TSC), to count CPU cycles. The TSC does not generate an interrupt on overflow like other hardware counters, so it is impossible to sample timing information using hardware counters on the x86. The third reason that time-based statistical profiling is provided is for users who simply do not want to manually convert CPU cycle information to time. However, other than using a sampling period based on the expiration of a virtual timer instead of a hardware-counter overflow, the time-based option is identical in its implementation to the hardware-counter-based option. Thus, the time-based profiling option is not mentioned further.

Clicking on the “Sample by Hardware Event(s)” button sensitizes the “Select Hardware Event(s)” button underneath it, which allows the user to open an event-selection dialog box (Figure 8.1(a)). This dialog box is identical to the one shown in Figure 7.1 on page 78, except for two differences. First, all composed events are greyed out and the user is never given the opportunity to choose them, because



(a) Hardware-event selection

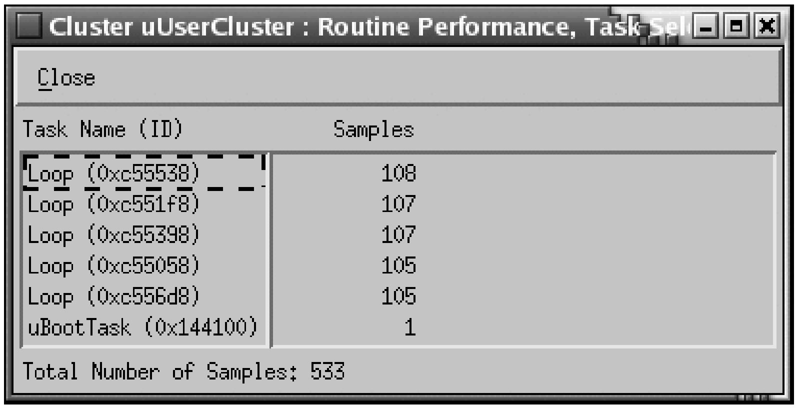


Elapsed Clock Cycles 9000000

Instruction cache miss 100

(b) Overflow-threshold selection

Figure 8.1: Selection dialog boxes.



```
Task Name (ID)           Samples
Loop (0xc55538)              108
Loop (0xc551f8)              107
Loop (0xc55398)              107
Loop (0xc55058)              105
Loop (0xc556d8)              105
uBootTask (0x144100)           1
Total Number of Samples: 533
```

Figure 8.2: Task-selection box.

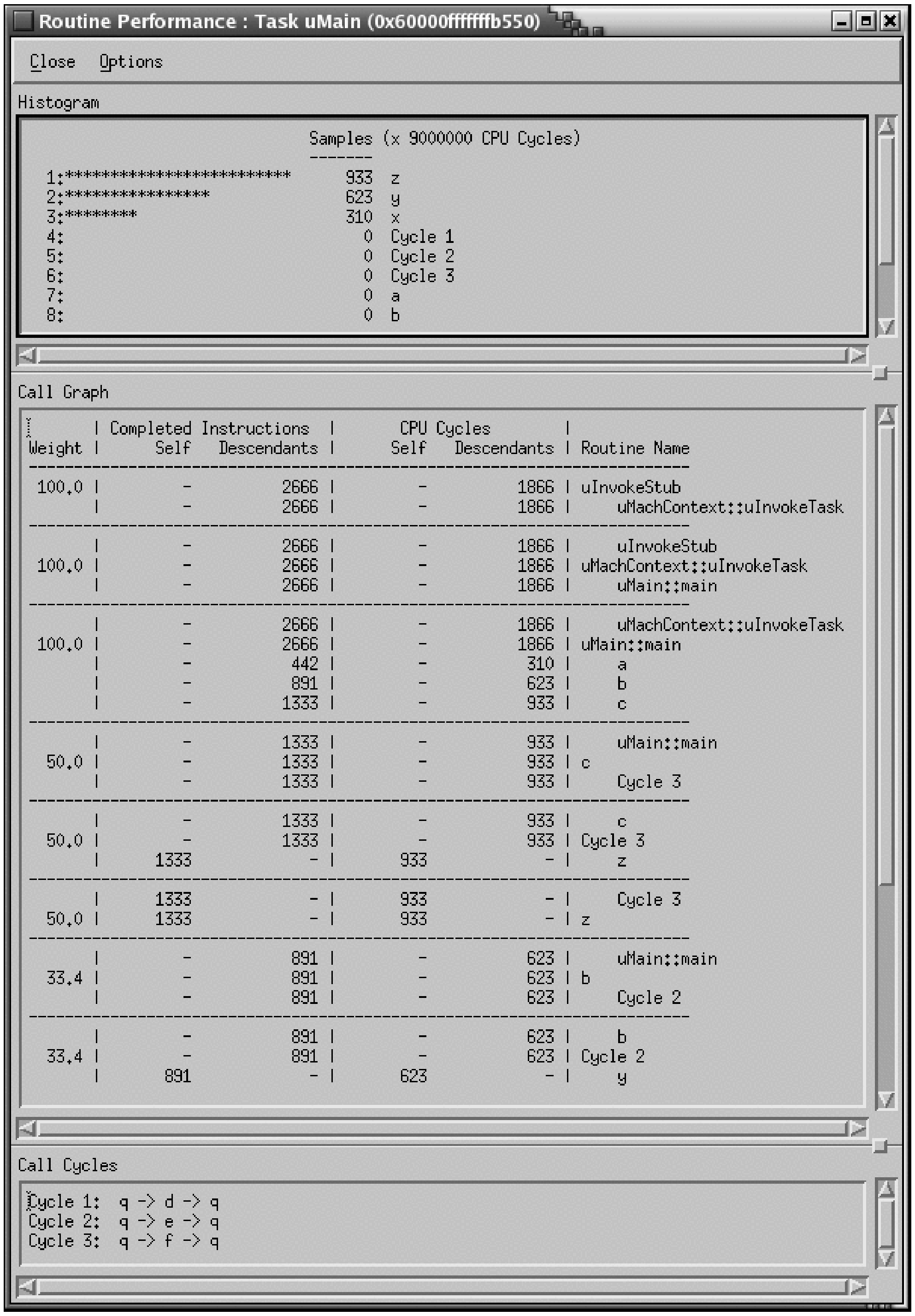
it is only possible to monitor an event for overflow if it occupies a single counter. Second, there is no "Break Events Down By Routine" option, as it has no meaning in this context.

Once the user selects which events to monitor and clicks the "OK" button, a second dialog box appears (Figure 8.1(b)). This dialog has one text box for each selected event, allowing the user to choose a non-default overflow threshold, i.e., number of occurrences before overflow and sample, for each one. Once all thresholds are chosen, the user clicks "OK" to dismiss the dialog box, and then "Start" to run the program.

At program termination, a selection list is displayed that contains the names of all tasks created by the program, along with the number of samples occurring in each task, listed in descending order by number of samples (Figure 8.2). The total number of samples across all tasks is listed at the bottom.

Clicking on a task pops up a display containing three panes. Figure 8.3 shows this display for the uMain task of the program listed in Appendix B.2.2. The first pane is a histogram showing the distribution of samples for one particular hardware event taken during a task's execution of its routines. The histogram's event can be changed by clicking "Options", then "Histogram Event", and finally, the event of choice (Figure 8.4 on page 116).

The second pane is a statistical call-graph with a format based on q-syscollect

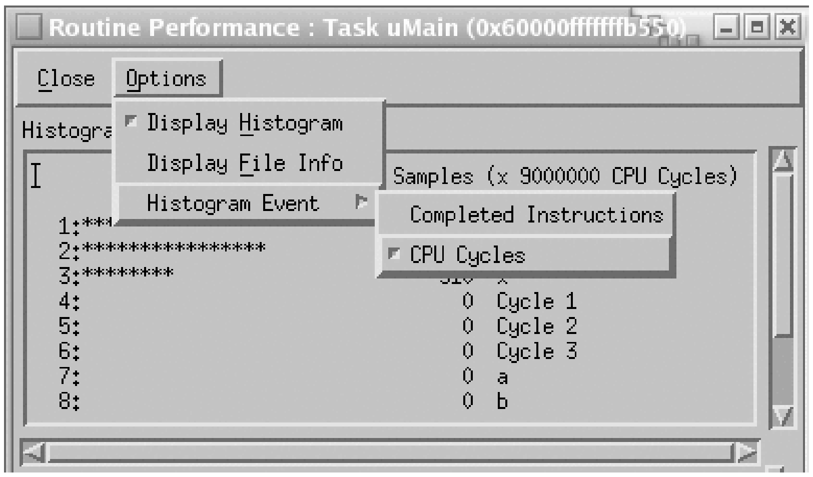


```
                                   Samples (x 9000000 CPU Cycles)
                                   -------
  1:*************************      933  z
  2:****************               623  y
  3:********                       310  x
  4:                                 0  Cycle 1
  5:                                 0  Cycle 2
  6:                                 0  Cycle 3
  7:                                 0  a
  8:                                 0  b
```

```
       | Completed Instructions  |       CPU Cycles         |
Weight |       Self  Descendants |       Self  Descendants  | Routine Name
-------------------------------------------------------------------------
 100.0 |          -         2666 |          -         1866  | uInvokeStub
       |          -         2666 |          -         1866  |     uMachContext::uInvokeTask
-------------------------------------------------------------------------
       |          -         2666 |          -         1866  |     uInvokeStub
 100.0 |          -         2666 |          -         1866  | uMachContext::uInvokeTask
       |          -         2666 |          -         1866  |     uMain::main
-------------------------------------------------------------------------
       |          -         2666 |          -         1866  |     uMachContext::uInvokeTask
 100.0 |          -         2666 |          -         1866  | uMain::main
       |          -          442 |          -          310  |     a
       |          -          891 |          -          623  |     b
       |          -         1333 |          -          933  |     c
-------------------------------------------------------------------------
       |          -         1333 |          -          933  |     uMain::main
  50.0 |          -         1333 |          -          933  | c
       |          -         1333 |          -          933  |     Cycle 3
-------------------------------------------------------------------------
       |          -         1333 |          -          933  |     c
  50.0 |          -         1333 |          -          933  | Cycle 3
       |       1333            - |        933            -  |     z
-------------------------------------------------------------------------
       |       1333            - |        933            -  |     Cycle 3
  50.0 |       1333            - |        933            -  | z
-------------------------------------------------------------------------
       |          -          891 |          -          623  |     uMain::main
  33.4 |          -          891 |          -          623  | b
       |          -          891 |          -          623  |     Cycle 2
-------------------------------------------------------------------------
       |          -          891 |          -          623  |     b
  33.4 |          -          891 |          -          623  | Cycle 2
       |        891            - |        623            -  |     y
```

```
Cycle 1:  q -> d -> q
Cycle 2:  q -> e -> q
Cycle 3:  q -> f -> q
```

Figure 8.3: Statistical histogram, call-graph, and list of call-cycles.



```
Samples (x 9000000 CPU Cycles)
1:***
2:****************
3:********
4:                                   0  Cycle 1
5:                                   0  Cycle 2
6:                                   0  Cycle 3
7:                                   0  a
8:                                   0  b
```

Figure 8.4: Selecting the histogram event.

[qto]. There is one row for each routine executed by a task. The routine corresponding to each row is outdented under the "Routine Name" column (last colum); its direct parents are indented above it, and its direct children are indented below it. If any call-cycles are found in the call-graph, each is reported as a single routine named "Cycle n", where $n \in \{1 \ldots \mathit{number\ of\ cycles}\}$. The routines in each cycle are listed in the third pane of the display.

For each row in the call-graph, there are a number of different columns. The first column is the routine's "Weight", which is the average of the percentages of each hardware event that occur in the routine and its descendants. For example, if the call-graph is based on two hardware events, and routine foo and its descendants account for 13% of the first hardware event's samples, and 24% of the second hardware event's samples, then foo's weight is (13 + 24) / 2 = 18.5. The rows in the call-graph are ordered by weight, and in the case of a tie, they are ordered by their routine's call-graph depth.

Next, there is one column for each hardware event, subdivided into "self" and "descendant" categories, the meanings of which depend on the type of routine being examined (current, parent, or child). For the current (outdented) routine, the "self" category corresponds to the number of samples that occurred in the routine, and the "descendant" category corresponds to the number of samples that occurred in its descendants. For a parent routine, “self” means the number of self-samples propagated up to the parent by the current routine, and “descendant” means the number of descendant-samples propagated up to the parent by the current routine. Finally, in the case of a child routine, “self” means the number of self-samples propagated up from the child to the current routine, and “descendant” means the number of descendant-samples propagated up from the child to the current routine. For example, refer to the CPU Cycles column in Figure 8.3 on page 115. Routine z has 933 self-samples, all of which are passed up to its only parent, Cycle 3. In the row just above, Cycle 3 is shown receiving 933 descendant-samples from routine z, all of which are passed up to routine c. Finally, routine uMain::main is shown receiving 1866 descendant-samples from three different children, and passing them up to its parent, routine uMachContext::uInvokeTask.

The third and final pane lists the call-cycles found in a task’s statistical call-graph, if any. Each cycle is listed on a separate line, which shows its exact routine-call sequence.

8.2 Design

Figure 8.5 shows the design of the Statistical Profiling Metric, using the object-oriented notation described in Appendix A. As usual, the execution monitor (uSPMonitor) and analyzer (uSPAnalyze) are derived from the uExecutionMonitor and uMetricAnalyze base classes, respectively. The following sections explain the functions of the uSPMonitor, uSPAnalyze, uSPTaskAnalyze, and uSPTaskAnalyzeWidget classes, as well as the classes that they use.

8.2.1 uSPMonitor

The Statistical Profiling Metric is a built-in metric, so its execution monitor is not responsible for gathering any data; its sole purpose is to activate the hooks needed by the metric. These hooks are summarized below.

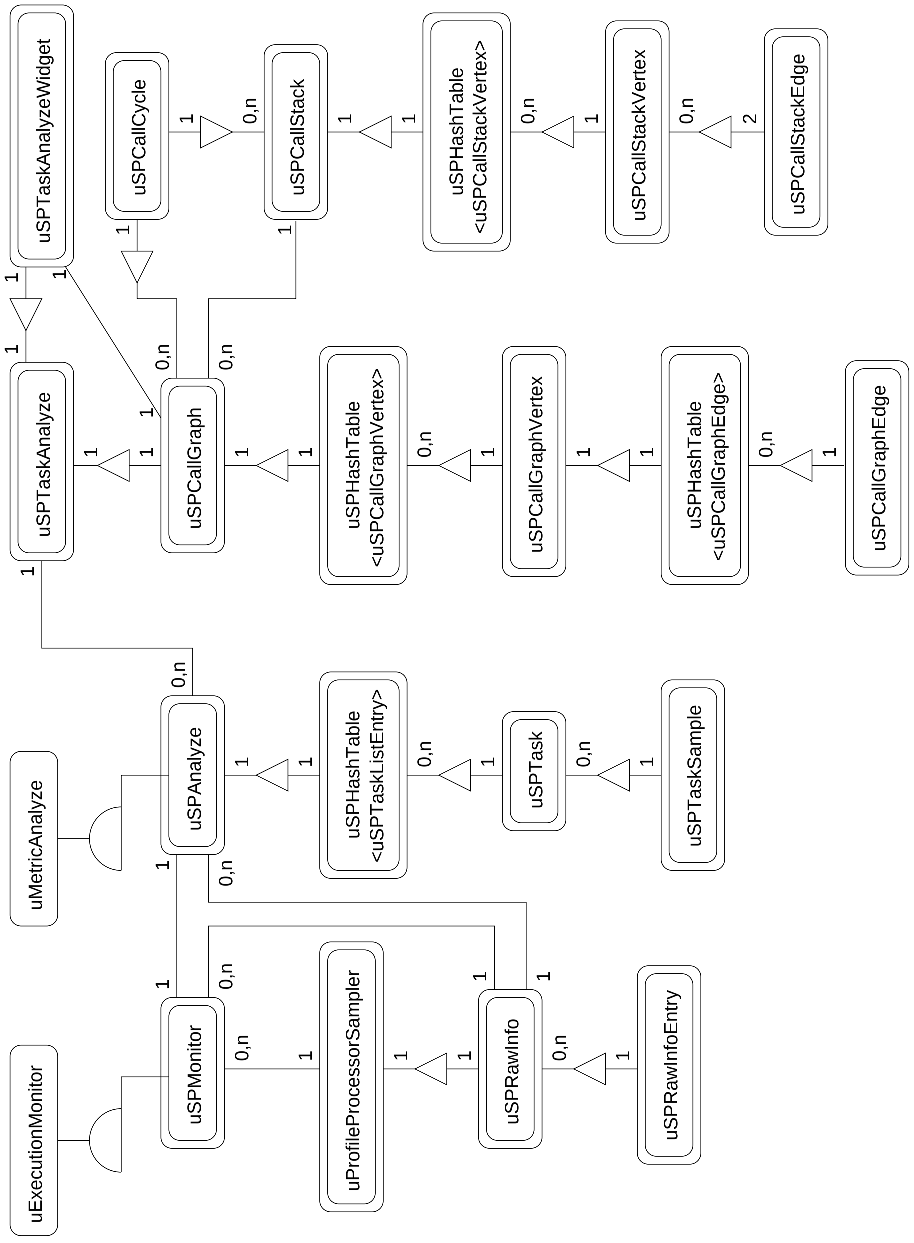


Figure 8.5: Object-oriented design of the Statistical Profiling Metric.

- Task Creation: Allows the metric to add the newly-created task's address to a list, which is stored in uSPMonitor and passed to uSPAnalyze in the analysis phase (see Section 8.2.2).

- Processor Creation/Destruction: Allows the metric to create/destroy a uProfileProcessorSampler for a newly-created/destroyed processor, and to start/stop hardware-counter overflow monitoring for its underlying kernel thread. The latter portion of these two hooks is not active in uniprocessor mode, since all processors share the same kernel thread.

When a uProfileProcessorSampler is created, it in turn creates a uSPRawInfo object to hold raw data for all samples taken on its processor. uSPRawInfo contains the address in memory of its corresponding processor, and a buffered list of uSPRawInfoEntry objects, each of which holds raw information for one sample. This information includes:

- The address in memory of the task executing on the uSPRawInfo's processor when the sample was taken.

- A bitmask indicating which hardware counters overflowed to cause the sample to be taken.

- A list of addresses representing a snapshot of the task's entire call-stack at the moment the sample is taken.

Part of the Processor Creation hook's job when it activates hardware-counter overflow monitoring is to install a signal handler to catch hardware-counter overflow signals generated by its kernel thread. This signal handler is responsible for collecting the sample information listed above and storing it in the appropriate uSPRawInfoEntry object.

Collecting Sample Information

When the signal handler is triggered, it first verifies that the current task, i.e., the task that is executing when the overflow signal is delivered, has profiling enabled. If so, it obtains the current task's address, the current overflow bitmask, and an ordered list of routine addresses from the current task's call-stack. This information is then passed to the current processor's uSPRawInfo object, which stores it in the next empty uSPRawInfoEntry object in its buffered list.

By the time a processor is destroyed, its uSPRawInfo object has a complete list of samples that occurred on it ready for analysis. When the Processor Destruction hook is triggered, it destroys the processor's uProfileProcessorSampler, but leaves its uSPRawInfo object intact. A pointer to this object is then passed to the uSPMonitor, which adds it to a linked list (see Figure 8.6).

At the beginning of the analysis phase, uSPMonitor creates a uSPAnalyze object, and passes it a list of all created tasks (addresses), as well as the list of processor samples.

8.2.2 uSPAnalyze

The purpose of the uSPAnalyze object is to separate all the processor-specific sample information into groups according to what task was active when each sample was taken, and to display a task-selection box allowing a user to choose which task's sample information to examine. Upon creation, uSPAnalyze creates a uSPHashTable object, which is a template hash-table written specifically for the Statistical Profiling Metric, to make the storing and accessing of statistical-profiling information efficient. uSPHashTable uses either a single address, or a list of addresses together with the size of the list (integer), as hash keys, and uses chaining for resolving collisions. In this case, the hash table is made to store uSPTask objects, each of which holds sample information for one task. The unique hash key for each such object is the address of the task it represents.

Once the hash table is created, uSPAnalyze runs through its list of created-task

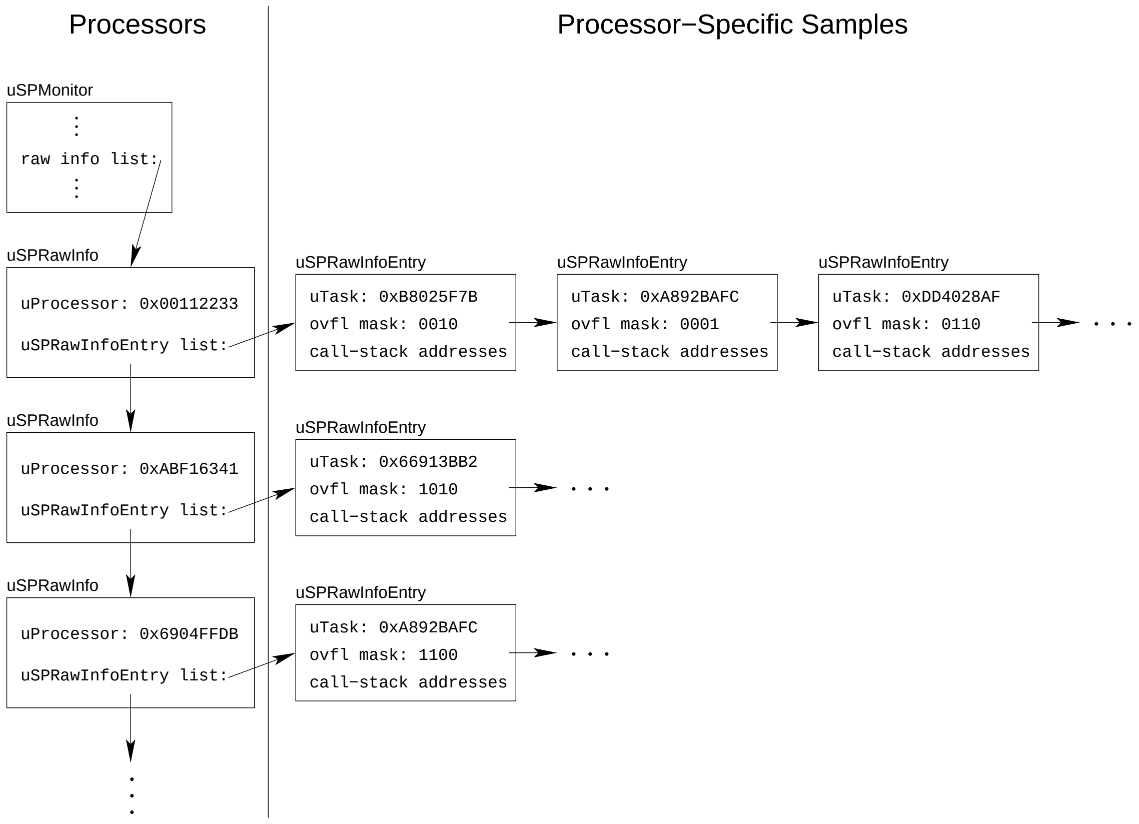


Figure 8.6: uSPMonitor's list of processor-specific uSPRawInfos.

addresses. For each address in the list, a new uSPTask object is created and added to the hash table. Each uSPTask object contains the address of its corresponding task, and a linked list for storing sample information (uSPTaskSamples).

After the hash table is populated, uSPAnalyze traverses its list of processor-specific uSPRawInfos, and for each one, traverses its list of uSPRawInfoEntrys. Each uSPRawInfoEntry is processed as follows:

- Its task address is used to retrieve the correct uSPTask object from the task hash-table.
- Its overflow bitmask and a pointer to its call-stack addresses are stored in a new uSPTaskSample, which is then added to the linked list of the uSPTask object obtained above.

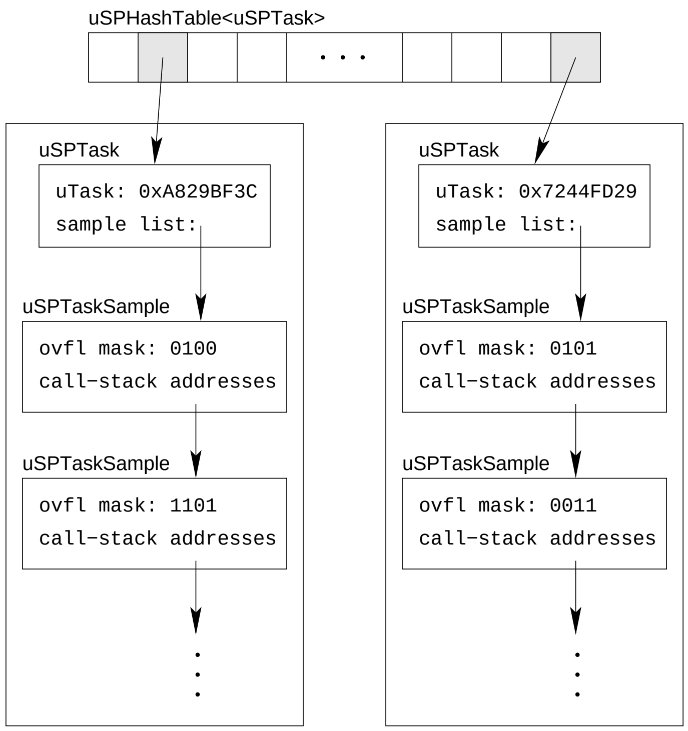


Figure 8.7: Raw sample information separated according to task.

When the list traversals are complete, all sample information is properly separated according to the task it belongs to, and is easily and efficiently retrievable from the task hash-table (see Figure 8.7). At this point, uSPAnalyze displays a task-selection box on the screen, such as the one seen in Figure 8.2 on page 114. When a user clicks on a task in this selection box, a pointer to its corresponding uSPTask object is retrieved from the hash table, and is passed to a newly-constructed object of type uSPTaskAnalyze, which performs sample-data analysis at the task level.

8.2.3 uSPTaskAnalyze

uSPTaskAnalyze simply creates an object of type uSPCallGraph, and passes it a pointer to the list of samples belonging to the task being analyzed. uSPCallGraph immediately organizes all its sample information into a call-graph format. Once this organization is complete, uSPTaskAnalyze creates a uSPTaskAnalyzeWidget object, and passes it a pointer to the uSPCallGraph. The uSPTaskAnalyzeWidget creates a three-paned Motif widget and fills it with histogram, call-graph, and call-cycle information from the uSPCallGraph, resulting in a display such as the one shown in Figure 8.3 on page 115.

The remainder of the objects in the Statistical Profiling Metric work together with uSPCallGraph to organize the sample information from one task into a call-graph format. Their complete implementations are explained below.

8.3 Implementation Issues

The most challenging task I faced while creating the Statistical Profiling Metric was the building of the statistical call-graph given one task's raw sample information. This section explains in detail the data structures and algorithms used to solve this problem.

As explained in Section 8.2.1, each sample consists of a hardware-counter overflow bitmask, and a list of routine addresses representing a task's entire call-stack at the moment the sample is taken. Each sample's call-stack is preprocessed to detect local call-cycles, and is then added to a global call-graph, where sample propagation takes place.

8.3.1 Call-Stacks

Each sample's call-stack is used to build a directed graph, encapsulated in the uSPCallStack, uSPCallStackVertex, and uSPCallStackEdge objects. uSPCallStack contains a uSPHashTable that stores instances of uSPCallStackVertex. Each uSPCallStackVertex represents one routine in the call-stack, and contains the address of the routine it represents, as well as two adjacency linked-lists of uSPCallStackEdges. One list represents directed parent edges, and the other represents directed child edges (Figure 8.8). At first glance, these adjacency lists may seem unnecessary for a call-stack, since each of its vertices should only have one parent and one

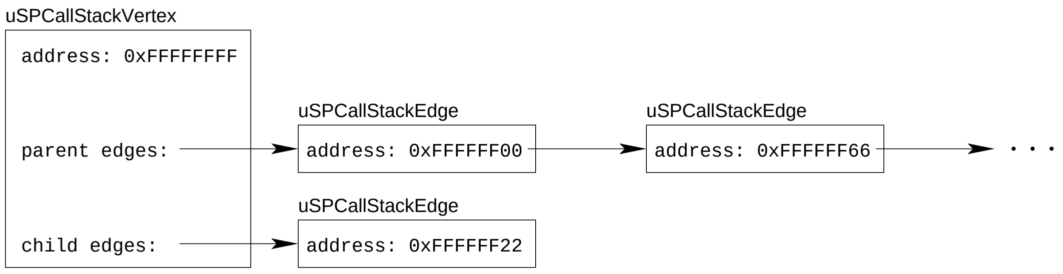


Figure 8.8: A uSPCallStackVertex with its parent and child adjacency lists.

child. However, if a call-stack contains any call-cycles, then at least one routine appears in the stack twice, and may have more than one unique parent and/or child. Therefore, one or more uSPCallStackVertexes may have multiple parent and/or child uSPCallStackEdges, which necessitates the adjacency lists.

Each uSPTaskSample is analyzed when a pointer to it is passed to a new instance of a uSPCallStack, which then creates the necessary uSPCallStackVertexes and uSPCallStackEdges. The list of routine addresses stored in the uSPTaskSample is traversed, starting with the address at the top of the call-stack.[1] For each routine address encountered, a new uSPCallStackVertex is added to the uSPCallStack's uSPHashTable (if it does not already exist in the table). Two uSPCallStackEdges are also added to the uSPCallStackVertex: one in its child adjacency-list pointing to the address immediately above it in the stack, and one in its parent adjacency-list pointing to the address immediately below it in the stack. The obvious exceptions to this rule are the routine addresses at the top and bottom of the call-stack; their corresponding uSPCallStackVertexes only receive one parent and one child edge, respectively.

At this point, the uSPCallStack is a directed, and possibly cyclic, graph representation of the uSPTaskSample's call-stack. However, before it can be merged with the global call-graph for sample propagation, all vertices that belong to call-cycles must be marked. This allows the call-graph to collapse them into single nodes

[1]Throughout this discussion, the assumption is made that all stacks grow in an upward direction. Therefore, at the top of the stack is the routine that is executing when a sample is taken.

before adding them to its own list of vertices (see Section 8.3.2).

Call-cycles are detected by finding a list of strongly-connected components in the uSPCallStack. A strongly-connected component of a directed graph $G = (V, E)$ is a maximal set of vertices $C \subseteq V$ such that for every pair of vertices $(u, v) \in C$, there exists both a directed path from $u \rightarrow v$ and from $v \rightarrow u$. Therefore, by definition, any of the following strongly-connected components in the uSPCallStack directed graph are call-cycles:

1. Those consisting of more than one vertex.

2. Those consisting of a single vertex with a self-edge.

The STRONGLY-CONNECTED-COMPONENTS algorithm [CLRS01] is run on the uSPCallStack to obtain a list of call-cycles in its directed graph. Each strongly-connected component that matches one of the above criteria is stored in a uSPCallCycle object, which is simply a linked-list of routine addresses. Marking vertices that belong to a call-cycle is then straightforward: the list of uSPCallCycles is traversed, and for each routine address found, a pointer to the uSPCallCycle is stored in the corresponding uSPCallStackVertex (retrieved from the uSPHashTable). The uSPCallCycle pointer serves as a "cycle marker" (see Figure 8.9).

At this point, the uSPCallStack has all the information it requires to be merged with the global call-graph. The merging of uSPCallStacks with the global call-graph is explained in Section 8.3.2.

Algorithmic Analysis

Building a uSPCallStack consists of two steps: creating the actual directed graph $G = (V, E)$ with the information in the provided uSPTaskSample, and discovering and marking call-cycles in that graph. Each step is analyzed separately.

As mentioned, the uSPHashTables used for this metric use chaining for resolving collisions. The expected search time of a chaining hash-table is $O(1+\alpha)$, where α is

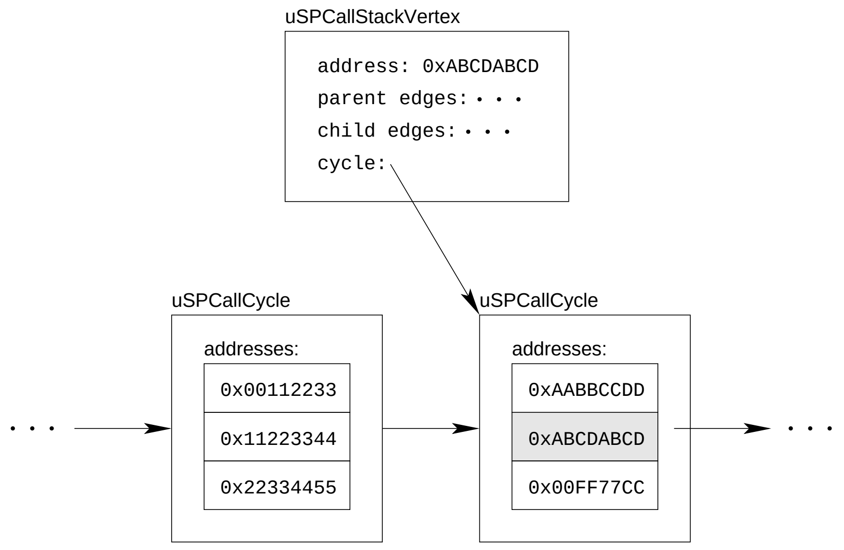


Figure 8.9: A uSPCallStackVertex that belongs to a call-cycle.

the load factor of the hash table [CLRS01]. Thus, by putting an upper bound on the load factor, the expected search time becomes bounded as well. This bounding is accomplished by sizing the hash table proportionally to the size of the symbol table of the program, which is easily obtained from its uSymbolTable (see Section 5.4). The complexity of a search then becomes independent of the number of vertices. Thus, for the purposes of this analysis, hash-table operations are assumed to be unit cost.

Building a uSPCallStack directed graph $G = (V, E)$ involves traversing a uSP-TaskSample's call-stack S, and for each routine address in S, creating up to one vertex and two edges (one parent edge and one child edge). For the vertex, a hash-table lookup is done to determine whether a vertex already exists with the current routine address. If no such vertex exists, a new vertex is created and stored in the hash table. For the edges, no hash table exists, so duplicate checking is done by running through the vertex's list of edges. However, this cost is generally negligible, for two reasons. First, when a routine address is encountered on a call-stack for the first time and a vertex is created for it, its edge lists are empty. Thus, there is zero cost for duplicate-edge checking for a reasonably large number of vertices.

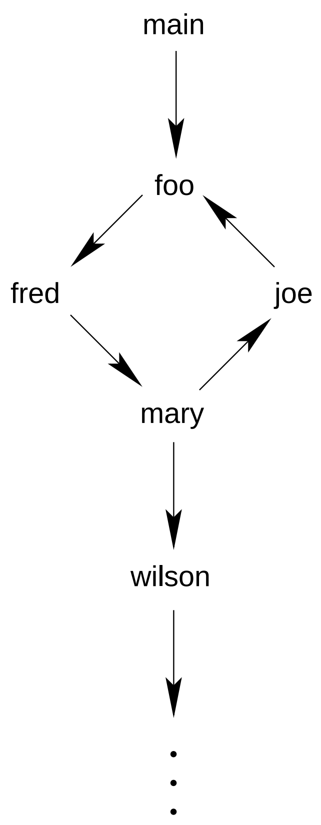


Figure 8.10: A typical call-cycle.

Second, in a typical call-cycle, such as the one shown in Figure 8.10, most vertices have only one parent edge and one child edge, and the entry and exit vertices have only two parent edges and two child edges. Note that in the worst case, the call-cycle is a clique and each vertex has $c-1$ edges, where c is the size of the clique. However, the expected case is a typical call-cycle and therefore, for the purposes of this analysis, duplicate-edge checking, and hence the creation of a vertex and its edges, are considered to have unit cost. Since this operation is done once for each routine on the call-stack, the creation of the uSPCallStack graph is $O(|S|)$.

Discovering the call-cycles in a directed graph $G = (V, E)$ is done using the STRONGLY-CONNECTED-COMPONENTS algorithm, which consists of the following steps:

1. Perform a DEPTH-FIRST-SEARCH [CLRS01] on G to obtain a topological ordering of its vertices.

2. Compute G^T, the transpose of G.

3. Perform a DEPTH-FIRST-SEARCH on G^T, considering vertices in reverse-topological order.

Step 3 returns a forest, with each tree in the forest being a strongly-connected component. Step 2 is unnecessary in my implementation of the algorithm because of the parent edges built into the uSPCallStack; traversing G^T is simply a matter of traversing G by following its parent edges rather than its child edges. Thus, finding the strongly-connected components of a uSPCallStack graph can be reduced to two depth-first searches, each of which is $O(|V| + |E|)$ [CLRS01].

The discarding of strongly-connected components that are not call-cycles is done during the depth-first search in Step 3, and it does not add any complexity to the algorithm. Strongly-connected components made up of a single vertex without a self-edge are discarded. This information is easily obtained while examining the neighbours of the root vertex each time a new tree, i.e., strongly-connected component, is searched.

The final step is to mark the vertices of the uSPCallStack that are part of a cycle, which is done simply by traversing the list of call-cycles and the list of vertex addresses in each. Since a vertex cannot be in more than one strongly-connected component [CLRS01], it cannot appear in more than one call-cycle. So the total number of vertices examined while traversing the list of call-cycles is $|V_{cycle}| \leq |V|$, which means the entire procedure is $O(|V|)$.

The aggregate complexity of creating a uSPCallStack is thus $O(|S|) + O(|V| + |E|) + O(|V|)$. The largest possible number of vertices in G occurs when no call-cycles appear in the call-stack. In this case, $|V| = |S|$. However, if any call-cycles exist, then at least one routine address is repeated in S but only gets one entry in G, and therefore $|V| < |S|$. So it is true in general that $|V| \leq |S|$, which means the complexity of this entire procedure is $O(|S|)+O(|S|+|E|)+O(|S|) = O(|S|+|E|)$.

8.3.2 Call-Graph

This section explains how a task's complete list of raw samples is converted into one global directed call-graph, encapsulated in the uSPCallGraph, uSPCallGraphVertex, and uSPCallGraphEdge classes. These classes are analogous to those used in the construction of a call-stack's directed graph, but contain extra information needed by the global call-graph, such as sample-propagation data.

uSPCallGraph's constructor accepts a pointer to a task's list of uSPTaskSamples, and immediately creates a uSPHashTable to store the uSPCallGraphVertexes used to build the directed call-graph. It then traverses the list of uSPTaskSamples, processes each one separately, and adds it to the directed call-graph.

The first step in the processing of each uSPTaskSample is to create a temporary uSPCallStack object and pass it a pointer to the uSPTaskSample. The uSPCallStack builds a directed call graph representation of the sample's call-stack, and marks all vertices that are part of call-cycles. Once the temporary uSPCallStack is built, its edge and vertex information is transferred to the uSPCallGraph, with one major change: call-cycles are represented in the uSPCallGraph as single vertices (see Figure 8.11).

To transfer the necessary information, the uSPCallStack must be traversed from the top down, and any vertices and edges not already in the uSPCallGraph must be added. However, because the uSPCallStack's directed graph may contain cycles, it is impossible to traverse it linearly from top to bottom. To solve this problem, it is actually the corresponding uSPTaskSample's call-stack that is traversed. For each routine address encountered, its counterpart uSPCallStackVertex is retrieved from the uSPCallStack, and the information contained therein is transferred to the uSP-CallGraph. How this transfer is made depends on whether the uSPCallStackVertex is marked as belonging to a call-cycle or not.

If the uSPCallStackVertex is not marked as being part of a call-cycle, then transferring its information to the uSPCallGraph is straightforward. The routine address of the uSPCallStackVertex is used as a hash-table key, and the uSPCallGraph's uS-PHashTable is queried. If no uSPCallGraphVertex with that address is found, then

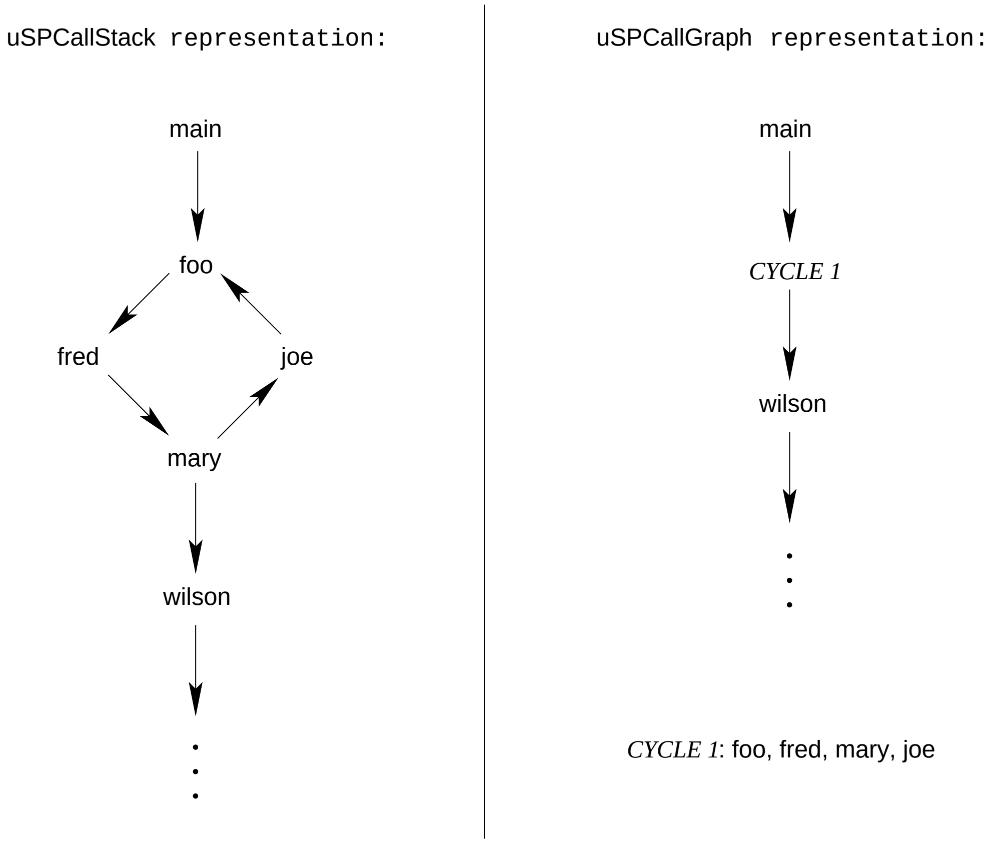
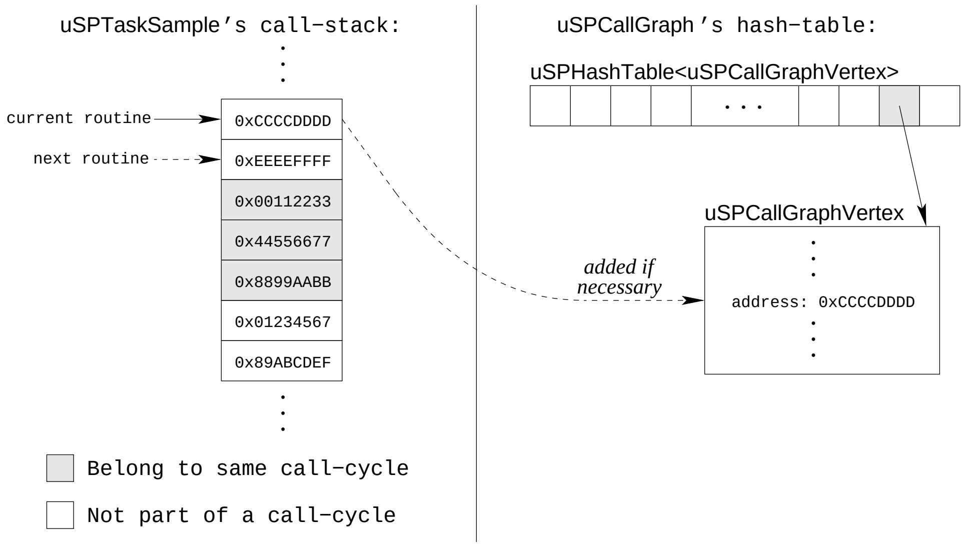

uSPCallStack representation:
main
foo
fred
joe
mary
wilson
uSPCallGraph representation:
main
CYCLE 1
wilson
CYCLE 1: foo, fred, mary, joe

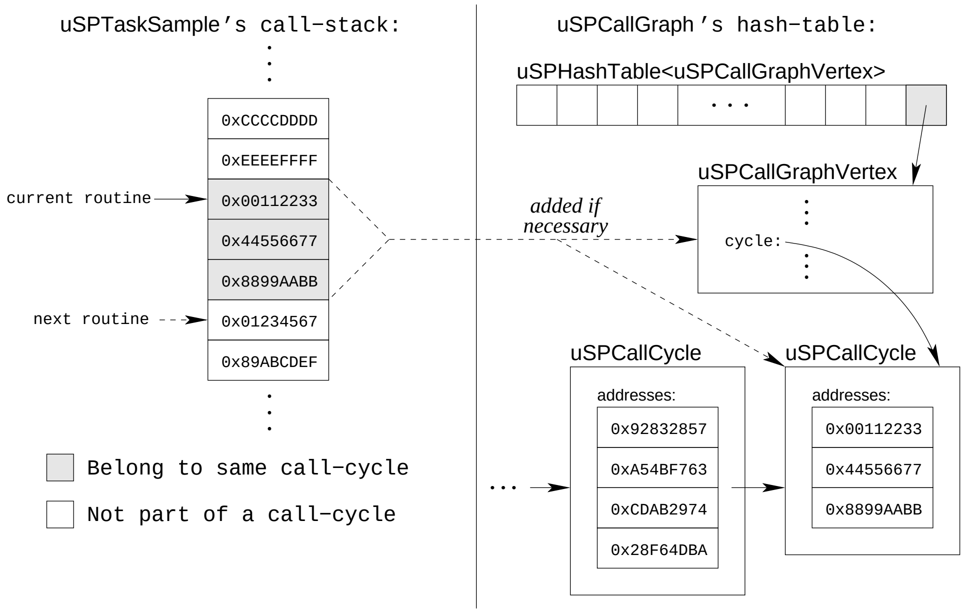
Figure 8.11: Call-cycles in uSPCallStack and uSPCallGraph.

one is created and added to the uSPHashTable. Finally, the uSPTaskSample call-stack pointer is advanced to the next routine address in the call-stack in preparation for the next iteration. This process is shown in Figure 8.12(a).

If the uSPCallStackVertex is marked as being part of a cycle, then a different procedure is used to add its information to the uSPCallGraph. In this case, the uSPCallCycle is retrieved from the uSPCallStackVertex, and its routine-address list is used as a hash-table key to query the uSPHashTable. If no uSPCallGraphVertex representing a cycle with that list of routine addresses is found, then a new call-cycle is added to the uSPCallGraph in two steps. First, a pointer to the uSPCallCycle is added to the tail of the uSPCallGraph's call-cycle list. Second, a new uSPCall-GraphVertex is created and added to the uSPHashTable, but instead of being given



(a) Singleton-uSPCallGraphVertex



(b) Cycle-uSPCallGraphVertex

Figure 8.12: Adding a uSPCallGraphVertex to the uSPCallGraph.

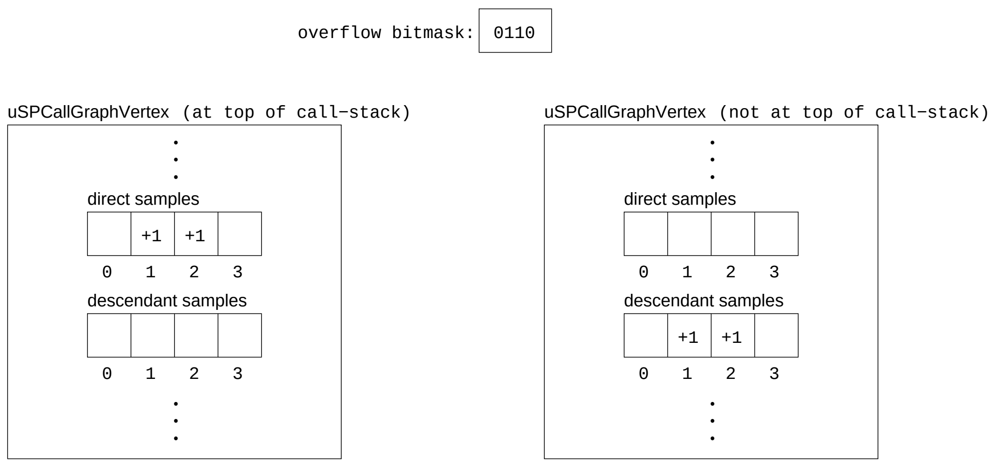


Figure 8.13: Assigning samples to a uSPCallGraphVertex.

a routine address, it is instead passed a pointer to the uSPCallCycle it represents. This procedure is similar to the one used in the uSPCallStack. Finally, in preparation for the next iteration, the uSPTaskSample call-stack pointer is advanced to the next routine address that does not belong to the uSPCallCycle that was just processed. This process is shown in Figure 8.12(b).

Once the singleton- or cycle-uSPCallGraphVertex is added to or retrieved from the uSPHashTable, one or more hardware-event samples are assigned to it, depending on the uSPTaskSample's overflow bitmask. Each uSPCallGraphVertex has two parallel arrays with one cell for each of the hardware events being monitored for overflow (see Figure 8.13). One array is used to count direct samples, i.e., samples that occur in the routine or cycle that the vertex represents, and the other array is used to count descendant samples, i.e., samples that occur in a descendant routine or cycle. If the current uSPCallGraphVertex represents the routine or cycle at the top of the current call-stack, then the sample count in every cell of the direct-sample array whose index matches a set bit in the overflow bitmask is incremented. Otherwise, the sample counts in the descendant array are incremented.

The final step in processing the current uSPCallGraphVertex is to add a child edge leading to the previous uSPCallGraphVertex, and a parent edge in the previous uSPCallGraphVertex leading to the current one. This step is obviously skipped if the current vertex represents the routine or cycle at the top of the call-stack, as it has no previous, i.e., child, vertex in that case. Each uSPCallGraphVertex has two uSPHashTables: one for parent uSPCallGraphEdges, and one for child uSPCall-GraphEdges. The current uSPCallGraphVertex's child hash-table is queried, and if it has no child edge leading to the previous uSPCallGraphVertex, an edge is created and added. A similar parent edge is added in the previous uSPCallGraphVertex if necessary. Samples are then assigned to each edge, much as in the case of the vertex samples above. The purpose of having sample counts on the uSPCallGraphEdges is to keep track of the routine-call sequence leading up to each sample, which allows the uSPCallGraph to provide the information seen in the call-graph pane in Figure 8.3 on page 115. The "self" and "descendant" columns for each non-indented routine/cycle come from the sample information in the uSPCallGraphVertexes, and the "self" and "descendant" columns for each indented parent and child routine/cycle come from the sample information in the uSPCallGraphEdges.

At this point, all the information from the current routine in the uSPTaskSample's call-stack has been added to the uSPCallGraph. The algorithm then proceeds to process the next routine pointed to by the uSPTaskSample's call-stack pointer, which was set earlier. After the entire call-stack is traversed, the algorithm moves on to the next uSPTaskSample and begins the process again, starting with the creation of a temporary uSPCallStack. Once the list of uSPTaskSamples is exhausted, the uSPCallGraph is complete, and ready to be displayed by the uSPTaskAnalyzeWidget. A high-level representation of the uSPCallGraph's final form is shown in Figure 8.14.

Algorithmic Analysis

Building a uSPCallGraph's directed graph $G' = (V', E')$ involves combining the algorithm from the previous section with this one, forming two nested loops: an outer loop to traverse a list L of uSPTaskSamples, and an inner loop to traverse a uSPTaskSample's call-stack S. Each is analyzed separately, from the outer loop to the inner.

Let s be the maximum $|S|$ over all uSPTaskSamples, that is, the number of

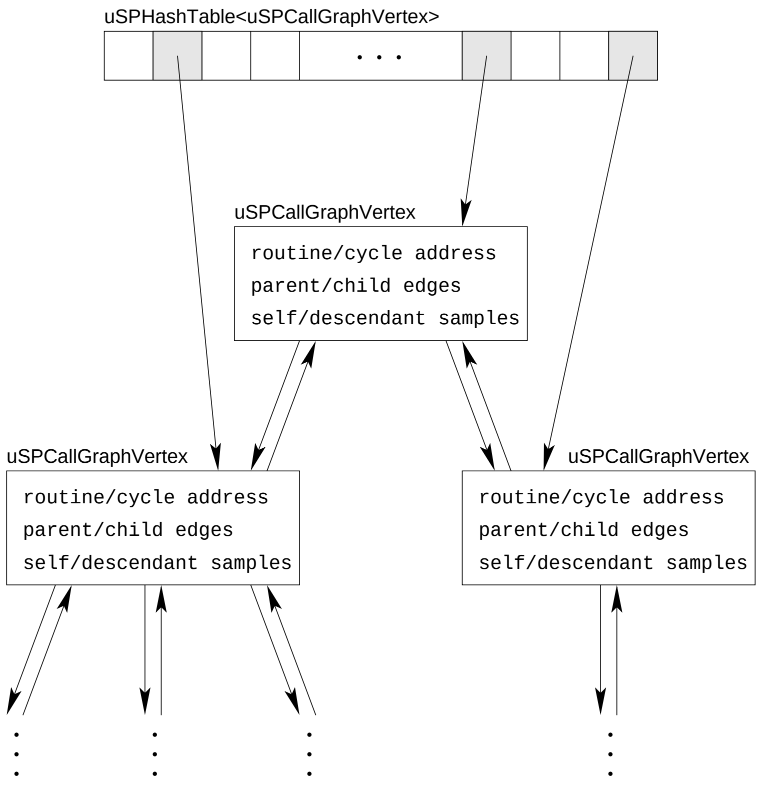


Figure 8.14: A high-level depiction of a uSPCallGraph.

routines in the largest call-stack. Let e be the maximum $|E|$ over all uSPCallStacks, that is, the number of edges in the largest uSPCallStack graph.

The outer loop traverses a list L of uSPTaskSamples, and for each uSPTaskSample, invokes the algorithm from Section 8.3.1 to create a uSPCallStack based on its call-stack S, and then invokes the inner loop. Thus, the running time for one iteration of the outer loop is $O(s + e)$ plus the running time of the inner loop.

The inner loop traverses a uSPTaskSample call-stack S, and for each routine address, creates up to one uSPCallGraphVertex and two uSPCallGraphEdges, and assigns samples to each. Each of these tasks is analyzed separately.

The first step in creating the uSPCallGraphVertex is to retrieve the uSPCallStack-Vertex corresponding to the current call-stack routine address. This step is done using a hash-table lookup with the routine address as a hash key, so it is a unit-cost operation. Next, another hash-table lookup is done to determine if a corresponding uSPCallGraphVertex already exists in the uSPCallGraph. The hash key used for this second lookup depends on whether or not the uSPCallStackVertex is part of a call-cycle. If it is not, then its routine address is used as a hash key, and the lookup is a unit-cost operation. If the uSPCallStackVertex is part of a call-cycle, then the hash-table lookup must use the entire uSPCallCycle's list of routine addresses, which is done as follows. The first routine address in the list is used as the actual key to obtain a hash bucket. Then the cardinality of the uSPCallCycle's list is checked against the cardinality of the address list of the first entry in the hash bucket (entries with a single address as a key are considered to have a cardinality of -1). Only if the cardinalities match are the address lists traversed and compared. Thus, the only situation in which an address list is traversed more than once is if two uSPCallGraphVertexes representing different call-cycles of the same size are in the same hash bucket, and the target entry is the further of the two down the collision-resolution chain, which should happen infrequently in general. Although in the worst case, a lookup of this kind is $O(|S|)$, the number of routines in a call-cycle's routine address list should, in general, be small in comparison to the number of routines in a uSPTaskSample, so a hash-table lookup of this kind is also considered to be unit cost.

In either case (cycle or non-cycle), if the hash-table lookup determines that the needed uSPCallGraphVertex does not already exist in the uSPCallGraph, it is created and added to the uSPHashTable. Then, the relevant sample counts are incremented, which is done by checking each bit of the current uSPTaskSample's bitmask, up to the number of hardware events being monitored. For each bit that is set, one integer is incremented. Since the number of active counters, and thus the number of hardware events being monitored for any given program is a constant, this operation is $O(1)$.

The final step in processing a call-stack routine address is to create up to two uSPCallGraphEdges, and assign samples to them. Samples are assigned in exactly the same way as for the uSPCallGraphVertex above, so they do not increase the complexity and are not discussed. For each uSPCallGraphEdge, a hash-table lookup is done in the originating uSPCallGraphVertex's child-edge uSPHashTable, using the destination uSPCallGraphVertex's address as a hash-table key (unit cost). If the necessary uSPCallGraphEdge does not exist, it is created and added to the proper uSPHashTable (also unit cost). Since this procedure is done a maximum of two times, it is a unit-cost operation.

Since all its operations run in constant time, the total complexity for processing one routine address on a uSPTaskSample's call-stack is $O(1)$. A maximum of s routine addresses are processed, so the inner loop has a running time $O(s)$. One iteration of the outer loop consists of the creation of a uSPCallStack, and an invocation of the inner loop, so its running time is $O(s + e) + O(s) = O(s + e)$. Since the outer loop runs $|L|$ times, its complexity is $O(|L|(s + e))$.

8.4 Validation

This section provides validation for the Statistical Profiling Metric by verifying that it produces correct results for a variety of simple μC++ programs. Validation testing is performed on the same dual-processor 900 MHz Itanium 2 machine used for validation in Section 7.4.

8.4.1 Testing Strategy

The testing strategy for the Statistical Profiling Metric is broken into two phases, each targeting a specific function. The first phase verifies sample collection is being done properly, i.e., the metric collects the correct number of samples and assigns them to the proper tasks. This validation is accomplished by profiling the perfmon control from Chapter 7. The second phase verifies the samples are being analyzed correctly, i.e., a proper call-graph and list of call-cycles is being produced. This validation is accomplished by running some new μC++ test programs, and comparing them to results obtained from the q-syscollect/q-view statistical profiling suite (see Section 3.3.2). The test programs for this phase are written to highlight key features of the sample analysis, including sample propagation in the call-graph, and call-cycle detection.

The sampling period for all validation tests is 9000000 events. This number was chosen because it is significantly smaller than the expected event-counts of all validation tests and should thus produce results with a reasonable degree of precision. Moreover, it makes the conversion of CPU Cycle samples to time (which is necessary for comparison with q-syscollect) straightforward, as each sample represents ten milliseconds on the 900 MHz CPU.

8.4.2 Sample-Collection Test

This section provides validation of the Statistical Profiling Metric's sample collecting. The sequential, concurrent, and parallel experiments from the aggregate event-count test of Section 7.4.4 are repeated for this test, and the results of the perfmon control of Section 7.4.1 provide the experimental control. The programs are profiled by the Statistical Profiling Metric, sampling on Instructions and CPU Cycles. If the Statistical Profiling Metric is working correctly, its results should be close to the perfmon control. However, since this test is being done statistically, an exact match between the two is unlikely, and unnecessary for showing correctness.

I hypothesize that the results of these experiments will all be close to, but less than the perfmon control. Since profiling is done statistically, the reported hardware-event counts are quantized in increments of the sampling period. Thus, a "round-down effect" should occur, meaning the results of the Statistical Profiling Metric compared to be perfmon control should be rounded down to the nearest multiple of the sampling period. Unlike the Exact Hardware Metric, I do not expect any difference between the results of the sequential, concurrent, and parallel tests, because the extra overhead incurred by the μC++ programs should be small relative to the sampling period.

Table 8.1 summarizes the results of the experiments and compares them to the perfmon control. Event counts for each experiment are obtained by multiplying the number of samples by the sampling period. The equivalent number of samples corresponding to each event count is listed to the right in parentheses. The *Diff* column is the percentage difference between the experiments and the perfmon control, and is relative to the perfmon control.

	Instructions		CPU Cycles	
	Total (Samples)	Diff. (%)	Total (Samples)	Diff. (%)
perfmon:	7281301480 (809)	0.000	4061099726 (451)	0.000
sequential:	7281000000 (809)	-0.004	4059000000 (451)	-0.052
concurrent:	7281000000 (809)	-0.004	4059000000 (451)	-0.052
parallel:	7263000000 (807)	-0.251	4041000000 (449)	-0.495

Table 8.1: Results of the sample-collection validation test.

The results of the experiments agree with my hypothesis insofar as all results are close to, but less than the perfmon benchmark. However, the parallel experiment underestimates the event counts by the equivalent of two samples for both Instructions and CPU Cycles, which goes against my hypothesis. The experiment was rerun a number of times, sometimes with a smaller sampling period, and the parallel experiment consistently underestimated the control by approximately the same percentage. I am unable to explain this anomaly. However the parallel experiment, as well as the sequential and concurrent experiments, produces results close enough to the perfmon control to conclude that the Statistical Profiling Metric is correctly gathering samples and assigning them to the proper tasks.

8.4.3 First Sample-Analysis Test

This test provides validation for the sample-analysis phase of the Statistical Profiling Metric through a comparison with the q-syscollect/q-view statistical profiling suite. Since sample collection has already been validated in all of μC++'s execution environments, there is no need to break this test into three experiments. Analysis of collected samples is independent of whether the program is run sequentially, concurrently, or in parallel. Thus, this new test is comprised of a single sequential

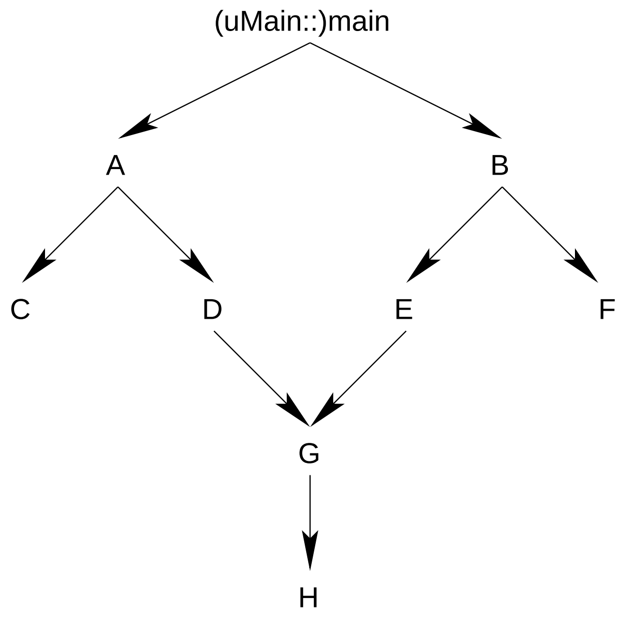


Figure 8.15: The call-tree of the first sample-analysis test program.

experiment, profiled by both the Statistical Profiling Metric and q-syscollect.

This test uses the program listed in Appendix B.2.1, and it creates the call-tree shown in Figure 8.15. Preprocessor directives allow a C++ version of the program to be compiled with a main routine, and a μC++ version to be compiled with a uMain::main routine. The C++ program is profiled by q-syscollect, while the μC++ program is profiled by the Statistical Profiling Metric.

Each routine in the program performs the same set of mathematical calculations in a loop, but each routine executes the loop a different number of times, which ensures each routine has a different running time. Besides performing the mathematical calculations, the (uMain::)main routine also makes calls to A and B to begin the routine-call sequences. It executes all of its code twenty times to ensure that the program runs for a reasonable length of time.

The test program is written in such a way that routine G executes three times longer when called by E than when called by D. Thus, in the resulting call-graph, G should pass 25% of its self-samples to D and 75% to E. Also, G calls H twice when it is called by D, but only once when it is called by E. Therefore, in the resulting call-graph, G should pass 67% of its descendant-samples to D, and 33% to E.

q-syscollect is only able to statistically sample on one hardware event at a time, therefore CPU Cycles is the only event sampled for this test. q-syscollect also automatically converts CPU Cycles into seconds, so the Statistical Profiling Metric's results are also converted into seconds for comparison, using the following equation:

$$seconds = samples \times \frac{sampling\ period}{clock\ speed} \tag{8.1}$$

where *clock speed* is the speed in Hz of the underlying CPU.

I hypothesize that the Statistical Profiling Metric's sample propagation will closely reflect the expected behaviour explained above, due to its use of complete call-stacks. I also expect the Statistical Profiling Metric's sample propagation to be more precise than q-syscollect's, because the latter profiler estimates sample propagation using only partial information from the BTB (see Section 3.3.2).

Sample-Collection Comparison

As mentioned, the sample-collection portion of the Statistical Profiling Metric has already been validated, but its results (and those of q-syscollect) are presented here for completeness. They are summarized in Table 8.2. The *Diff* column is the percentage difference between the Statistical Profiling Metric's results and q-syscollect's results, and is relative to the Statistical Profiling Metric's results.

The sample-collection results for q-syscollect and the Statistical Profiling Metric are similar, which indicates that they are both collecting sample data properly.

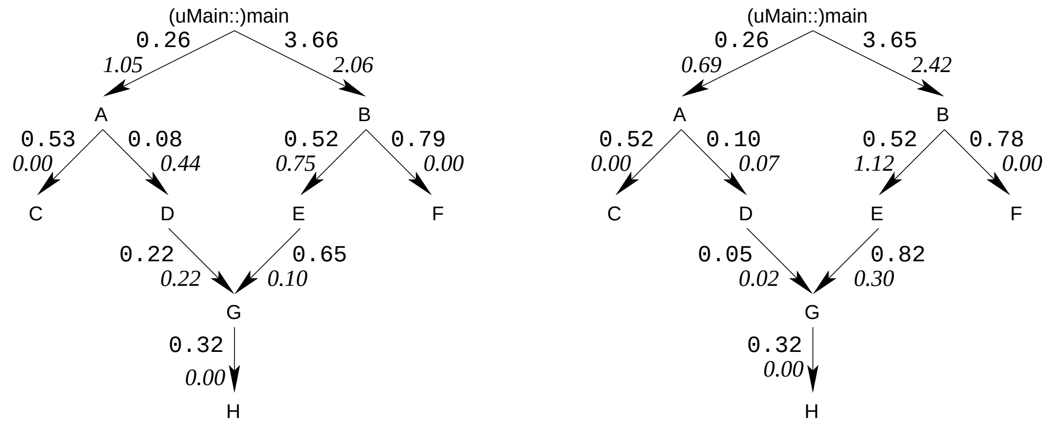
Call-Graph Comparison

The sample propagation done by the Statistical Profiling Metric is shown in Figure 8.16(a) on page 142. The numbers along each edge represent samples propagated

	SP Metric (seconds)	q-syscollect (seconds)	Diff. (%)
(uMain::)main	1.09	1.09	0.000
A	0.26	0.26	0.000
B	3.66	3.65	-0.273
C	0.53	0.52	-1.887
D	0.08	0.10	25.000
E	0.52	0.52	0.000
F	0.79	0.78	-1.266
G	0.87	0.87	0.000
H	0.32	0.32	0.000

Table 8.2: Sample collections from the first sample-analysis test.

from a child to a parent. Descendant-samples are in italics, and are always listed below self-samples. The majority of the routines have only one parent, and so they pass 100% of their self- and descendant-samples up to that parent. The interesting portion of the call-graph is routine G, which has two parents. As expected, G passes almost three times as many self-samples to E as to D (the actual split is about 25.3% to D and 74.7% to E). Also as expected, G passes about twice as many descendant-samples to D than to E (the actual split to about 68.8% to D and 31.3% to E). This indicates that sample propagation is being done correctly, using each sample's call-stack. In contrast, q-syscollect does not use the entire call-stack, and hence incorrectly propagates self- and descendant-samples in proportion to execution time. For example, referring to the q-syscollect call-graph shown in Figure 8.16(b), G passes 5.7% of its self-samples to D and 94.3% to E. Also, G passes 6.3% of its descendant-samples to D and 93.7% to E. Both propagations are clearly incorrect, given the way the test program is written.



(a) SP Metric (b) q-syscollect

Figure 8.16: Call-graphs from the first sample-analysis test.

Summary

This test shows that the Statistical Profiling Metric's sample-collecting is as accurate as that of q-syscollect. Furthermore, the test shows that its call-graph reflects the actual execution behaviour of its target program more consistently than does q-syscollect.

8.4.4 Second Sample-Analysis Test

This test provides validation for the Statistical Profiling Metric's call-cycle detection capabilities, and shows how its method of detecting "local" call-cycles in each sample's call-stack provides a better picture of the run-time behaviour of a program than traditional call-cycle detection algorithms, such as the one used by gprof [GKM82]. The test is comprised of a single sequential experiment, profiled by both the Statistical Profiling Metric and gprof. gprof is used as a substitute for q-syscollect in this test because q-syscollect does not do any call-cycle detection (it simply ignores back-edges in its sample-propagation algorithm).

The program used for this test is listed in Appendix B.2.2. Preprocessor direc-

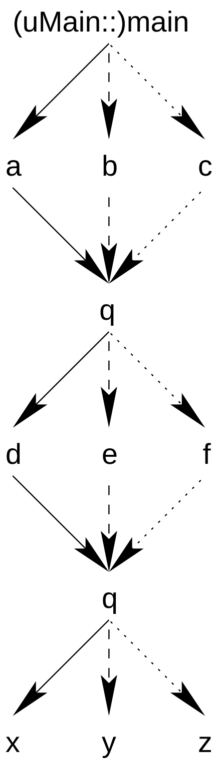


Figure 8.17: The call-tree of the second sample-analysis test program.

tives allow separate C++ and μC++ versions to be compiled as for the previous example program. Routine q accepts a routine pointer as a parameter and simply calls that routine. Routines a through f call q and pass in a pointer to the routine that q is to call. Routines x, y and z are the only routines where any significant amount of time is spent; each has a spinning loop of a different size. y's loop is twice the size of x's, while z's is thrice the size of x's. Finally, routine (uMain::)main makes calls to routines a, b and c twenty times each. This type of dynamic call-structure can be found in object-oriented programs making significant use of virtual routines; for instance in applications of the Visitor design pattern [GHJV95].

This program creates the call-tree shown in Figure 8.17. The different line styles show the routine-call sequences that occur in the program. There are three cycles in the call-tree:

1. q → d → q
2. q → e → q
3. q → f → q

Note, gprof samples only on time, so CPU Cycles is the only event sampled for this test. The CPU Cycle event-counts are converted to time for comparison, using Equation 8.1. I hypothesize that the Statistical Profiling Metric will produce sample-collection data similar to gprof, and discover the three call-cycles that occur in the program's call-tree. Also, I expect that sample propagation will only occur between routines along the call-sequences shown in Figure 8.17. In other words, x's samples should be propagated to a, y's to b, and z's to c. Finally, I expect that gprof will collapse all three call-cycles into a single call-cycle, due to the fact that it discovers call-cycles after the entire call-graph is constructed. Moreover, I expect gprof will propagate the call-cycle's samples evenly between routines a, b and c, as gprof assumes that all calls to the same routine take the same amount of time.

Sample-Collection Comparison

Sample-collection results for the Statistical Profiling Metric and gprof are provided here for completeness. They are summarized in Table 8.3. Only those routines that have any time attributed to them by either profiler are displayed. The *Diff* column is the percentage difference between the Statistical Profiling Metric's results and gprof's results, and is relative to the Statistical Profiling Metric's results.

The sample collections produced by the Statistical Profiling Metric and gprof are similar, but gprof's distribution is slightly more precise, as y has exactly twice the time that x does, while z has exactly thrice the time. In the Statistical Profiling Metric's sample collections, y has slightly more than twice the time of x, and z has slightly more than thrice the time of x.

Relative to gprof, the Statistical Profiling Metric also reports about 0.639% less total time, and did so consistently over multiple runs of the experiment. I hypothesized that the additional time reported by gprof may have been due to its probe

	SP Metric (seconds)	gprof (seconds)	Diff. (%)
x	3.10	3.13	0.968
y	6.23	6.26	0.482
z	9.33	9.39	0.643
Total:	18.66	18.78	0.643

Table 8.3: Sample collections from the second sample-analysis test.

effect, so I profiled the test program twice with q-syscollect, once after compiling with the -pg flag, and once without. The -pg flag specifies that performance data is to be collected during the program's execution, for post-mortem analysis by gprof. Table 8.4 summarizes the results of these experiments.

	with -pg flag (seconds)	without -pg flag (seconds)
x	3.11	3.11
y	6.22	6.22
z	9.33	9.32
Total:	18.66	18.65

Table 8.4: q-syscollect's samples from the second sample-analysis test.

The q-syscollect results are much closer to those of the Statistical Profiling Metric than to gprof's. The total time with the -pg matches the Statistical Profiling Metric's total time, while the total time without the -pg flag is ten milliseconds less. I am unable to explain this discrepancy between the Statistical Profiling Metric/q-syscollect's and gprof's reported times. However, the differences between them are small enough to conclude that the Statistical Profiling Metric is correctly gathering sample data.

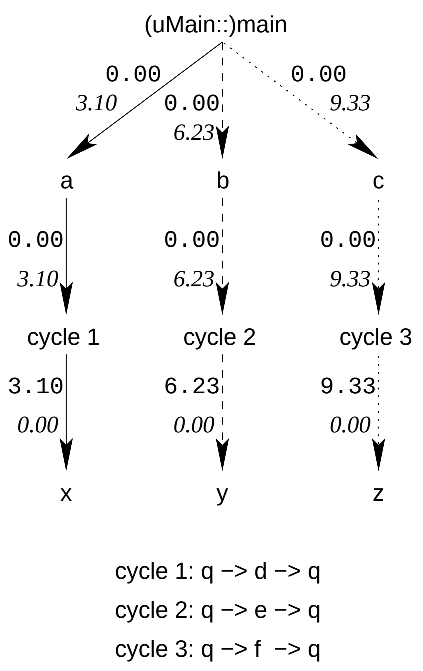
Call-Graph Comparison

The Statistical Profiling Metric finds all three call-cycles in this program, as well as the correct routine-call sequences, which are summarized in Figure 8.18(a). The numbers along each edge represent samples propagated from a child to a parent. Descendant-samples are in italics, and are always listed below self-samples. Since the cycles are discovered locally in individual call-stacks, they are reported separately, even though in the global call-graph they are part of the same strongly-connected component. This makes the routine-call behaviour of the program, i.e., which routines are parents and children of what cycles, clear. Furthermore, because cycles are discovered in individual call-stacks, their exact routine-call sequences are preserved, as can be seen at the bottom of the figure. Finally, the sample propagation is precise, as routines a, b, and c only receive descendant-samples from the routine-call sequences that they initiate, i.e., they only receive descendant-samples from x, y, and z, respectively.

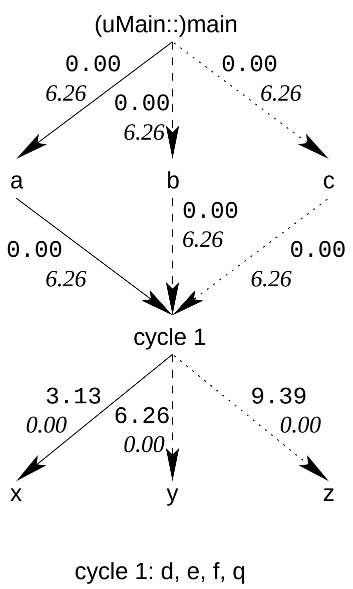
In contrast, gprof discovers call-cycles globally, i.e., after the entire call-graph has been built, which results in the graph shown in Figure 8.18(b). Since routines d, e, f, and q form one strongly-connected component in the global call-graph, they are reported as one cycle. Moreover, gprof is unable to display the different routine-call sequences that exist in the cycle. What is reported is simply a list of parent-child relationships between the routines in the cycle. In this case, gprof reports that routine q has routines a through f as parents, and routines d through f and x through z as children. Finally, the descendant-sample propagation from cycle 1 to routines a, b, and c is split evently, which is clearly incorrect.

Summary

Relative to gprof, the Statistical Profiling Metric slightly underestimates the total time spent in the test program, and its sample distribution for this test is slightly less precise. Both differences are small enough to conclude that the Statistical Profiling Metric is collecting and assigning sample data correctly.



(a) SP Metric



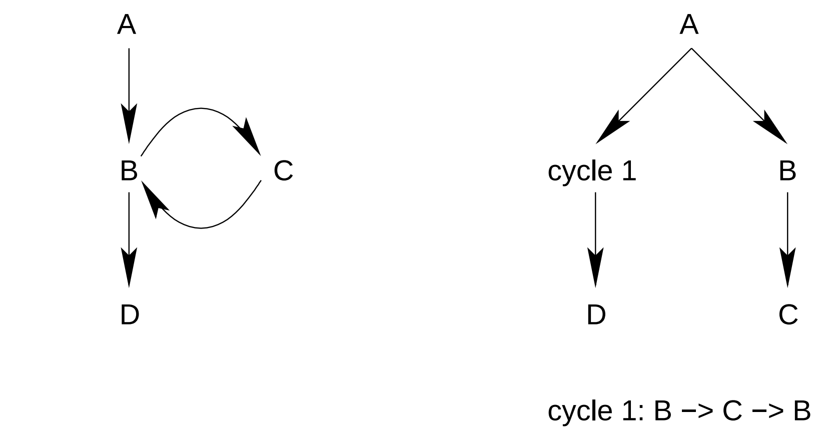
(b) gprof

Figure 8.18: Call-graphs from the second sample-analysis test.

The Statistical Profiling Metric is able to discover call-cycles at a finer level of granularity than gprof, as it finds individual cycles, based on complete call-stacks, that may be part of a larger strongly-connected component in the complete call-graph. Furthermore, because it retains the routine-call sequences of each cycle, the Statistical Profiling Metric is able to propagate samples through a cycle according to the actual execution behaviour of its target programs, rather than simply splitting the cycle's self- and descendant-samples evenly among its parents.

The Statistical Profiling Metric's method of call-cycle detection does have one major drawback: it can discover "partial cycles". For instance, consider the call-graph in Figure 8.19(a), and assume two samples are taken:

1. $\mathsf{A} \to \mathsf{B} \to \mathsf{C} \to \mathsf{B} \to \mathsf{D}$



(a) Actual call-graph

(b) Reported call-graph

Figure 8.19: An example call-graph.

2. A → B → C

The call-graph produced by the Statistical Profiling Metric looks like the one in Figure 8.19(b). Routines B and C are represented twice in this graph: once as single routines, and once as part of a cycle. Thus, the self- and descendant-samples propagated to these routines are divided among the two representations, neither of which paints a complete picture of their execution behaviour. Currently, interpretation of these results is not done by the Statistical Profiling Metric, but is left to the user.

Chapter 9

Conclusions and Future Work

The focus of this thesis is using hardware counters to profile user threads in concurrent, object-oriented programs running in a shared-memory, uni/multiprocessor environment. The target environment for this effort is μProfiler, a concurrent object-oriented profiler written in μC++, a concurrent dialect of the C++ programming language.

9.1 Contributions

The major work done for this thesis includes the following additions to the μProfiler kernel: an architecture-abstraction layer for accessing hardware counters on multiple platforms, and two new profiling metrics that make use of the new layer to help users locate bottlenecks and hotspots in programs.

To allow μProfiler to use hardware counters, an architecture-abstraction layer was written. This layer defines a useful common subset of features, allowing μProfiler to extract information from hardware counters on multiple platforms. It also interfaces with μProfiler's startup window, interactively updating the list of available hardware event buttons as users choose which events to measure, ensuring that only legal combinations of hardware events are used to profile target programs.

Two metrics were written and added to μProfiler that use the architecture-abstraction layer to profile concurrent programs using hardware counters. The first metric is the Exact Hardware Metric, which provides exact hardware event counts on a per-task basis, and the second metric is the Statistical Profiling Metric, which samples target programs at regular intervals to provide statistical approximations of hardware-event counts.

The Exact Hardware Metric uses hardware counters to generate exact performance data. It offers exact hardware-event counts in terms of μC++'s execution environment on three different levels. For each task in a target program, an aggregate hardware-event count is presented. Each task's hardware-event counts can be further subdivided into a routine and non-routine breakdown. The routine breakdown shows exact event-counts for each routine executed by a task, while the non-routine breakdown gives exact event-counts on a per-time-slice basis. Each breakdown offers insight into the run-time behaviour of each task in a program for a given cost and probe effect, and can be used to help locate bottlenecks and hotspots. The performance of this metric on all three levels has been validated by testing it against established code.

The Statistical Profiling Metric offers a lower cost and less intrusive, albeit less precise, alternative to the Exact Hardware Metric. Performance data is gathered by periodically sampling all virtual processors in a target program. Sample data is presented to the user in three ways: a flat histogram showing the distribution of samples across the routines executed by a task, a call-graph showing routine-call sequences and sample propagation, and a list of call-cycles found in a task's call-graph. The call-graph produced by the Statistical Profiling Metric is unique among statistical call-graphs because it is based on complete call-stacks, which means that while it may not be complete, the call-graph is guaranteed to be connected. Also unique to this call-graph is the ability to display event counts from multiple hardware events. The histogram, call-graph, and call-cycle detection functions of this metric have all been validated by testing it against established code.

Finally, one major implementation issue that had to be solved was allocating memory for profiling data for certain μProfiler hooks. μProfiler is able to glean information from the μC++ kernel at run-time, but μProfiler metrics had no way of dynamically allocating storage to hold this information because memory allocation is not allowed in the μC++ kernel. To solve this problem, a new class called uMemoryExecutionMonitor was added to the μProfiler kernel, which, when instantiated, activates *memory-allocation hooks*. Metrics can now use these hooks to dynamically allocate memory just outside the μC++ kernel, for storing performance data generated by tasks executing inside it.

9.2 Future Work

There are a number of possible directions for future work for μProfiler. Currently, performance data is visualized only on screen; there are no provisions in place for saving the data directly to disk. This avenue should be explored, as it would allow easy comparisons of separate program runs. In particular, the data should be saved into a well-supported profiling data file-format, such as Pablo's *Self-Defining Data Format* (SDDF).

Hardware counters are not currently supported on all of the μC++-supported architectures (e.g., the Pentium IV). Future work should include efforts to add hardware-counter support on all of the μC++ platforms.

The Exact Hardware Metric's routine breakdown provides information with a layout similar to the Call Graph and Run Time metric. Although it was developed as a standalone metric, it may be useful to examine the possibility of integrating the Exact Hardware Metric with the Call Graph and Run Time metric. Such a metric would offer a useful contrast between per-routine timing information and per-routine hardware-event counts.

A number of improvements can be made to the Statistical Profiling Metric. For example, an automatic sampling-period calibration would be a useful addition. Currently, users accept the default or choose a sampling period as a raw number of hardware events. This method is useful when sampling on events with an expected count, such as Instructions or CPU Cycles, but it is quite unintuitive for events like Branch Mispredictions or Instruction-Cache Hits. Future research should also examine ways of combining the "partial call-cycles" reported by the Statistical Profiling Metric with the complete call-cycles that they are part of. Finally, the Statistical Profiling Metric currently does not estimate the number of routine calls in its statistical call-graph, so the average event-counts per routine are not reported. Future work should include adding this capability.

Appendix A

Object-Oriented Notation

This appendix explains the object-oriented notation used in Chapters 5, 7 and 8 to depict the designs of the μProfiler kernel, Exact Hardware Metric, and Statistical Profiling Metric, respectively. It is a simplified version of the notation presented by Peter Coad and Jill Nicola [CN93], and includes a μC++-specific extension introduced by Dorota Zak [Zak00].

The notation in this appendix is broken up into two sections. Section A.1 introduces the symbols that represent classes and objects, while Section A.2 describes the notation used to represent the three different relationships between classes and objects.

A.1 Class and Object Notation

The basic building blocks of the object-oriented notation are the class and object symbols, which are shown in Figure A.1. The abstract class symbol represents a class that cannot be instantiated; it is depicted as a rectangle with rounded corners. The class/object symbol depicts a class with at least one instantiated object. The inner rectangle represents the definition of the class itself, while the outer rectangle represents its instances.

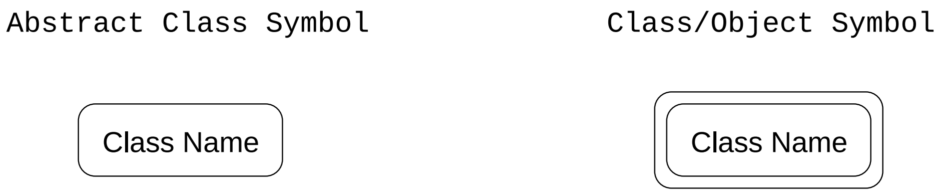


Figure A.1: Class and object notation.

Coad and Nicola's original notation does not allow for the representation of objects with a thread of control and execution state (e.g., μC++ tasks), so it was extended for this purpose by Zak, who introduced notation for an "active object" (Figure A.2). An active object is represented by regular rectangles, i.e., ones without rounded corners, with the space between the inner and outer rectangle shaded.

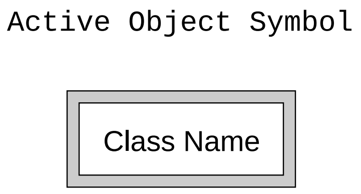
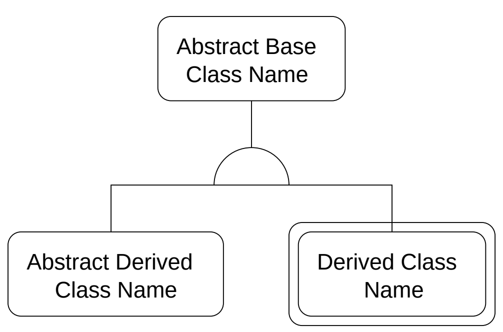


Figure A.2: Active object notation.

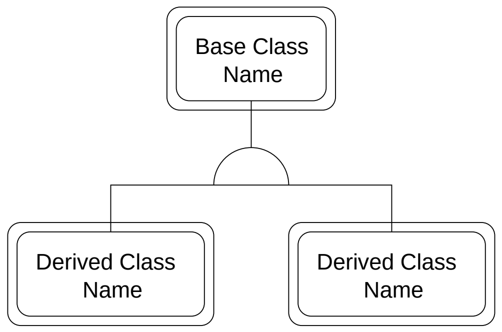
A.2 Relationship Notation

The inheritance or "is-a" relationship is depicted as a line and semicircle connecting a base class to one or more derived classes. The derived classes inherit all the appropriate attributes and member routines from the base class, and may further specialize them, and/or add new functionality. The base class is connected to the top of the semicircle, while the derived classes are connected to the bottom (Figure A.3). Inheritance takes place between classes rather than objects, which is why the lines are connected to the inner rectangles of class objects.

The aggregation or "has-a" relationship is represented by a line and triangle connecting a "member object" and a "containing object". The triangle points



(a) Abstract derivation



(b) Concrete derivation

Figure A.3: Inheritance notation.

towards the containing object, which contains the member object as an attribute (Figure A.4). The cardinality symbols next to each object represent the numeric relationship between the objects. In the figure, the whole object contains zero or more instances of the part object, and each part object is an attribute of only one whole object.

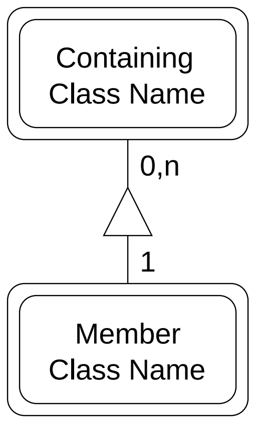


Figure A.4: Aggregation notation.

The last type of relationship between objects is association, which means the two objects are aware of each other, but neither contains the other. Association is represented by a line connecting two objects (Figure A.5). The cardinality symbols represent the numeric relationship between the objects. In the figure, object A is associated with two objects of type B, while object B is associated with one or more objects of type A.

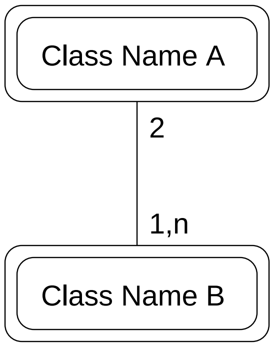


Figure A.5: Association notation.

Appendix B

Program Source Code

B.1 Exact Hardware Metric Test Programs

The following program is a modified version of the one available on the perfmon website [per].

B.1.1 perfmon Test-Harness

```
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <perfmon3/perfmon.h>
#include <perfmon3/pfmlib.h>

#de£ne MAX_EVT_NAME_LEN 256

int main(int argc, char **argv) {
    int i, ret, fd;
    pfmlib_input_param_t inp;
    pfmlib_output_param_t outp;
```

```
pfarg_reg_t pc[2];
pfarg_reg_t pd[2];
pfarg_reg_t pd_before[2];
pfarg_reg_t pd_after[2];
pfarg_load_t load_args;
pfarg_context_t ctx[1];
char name[MAX_EVT_NAME_LEN];

if (pfm_initialize() != PFMLIB_SUCCESS) {
    fprintf(stderr, "cannot initialize libpfm\n");
    exit(1);
}

memset(pc, 0, sizeof(pc));
memset(pd, 0, sizeof(pd));
memset(pd_before, 0, sizeof(pd_before));
memset(pd_after, 0, sizeof(pd_after));
memset(ctx, 0, sizeof(ctx));
memset(&inp,0, sizeof(inp));
memset(&outp,0, sizeof(outp));
memset(&load_args,0, sizeof(load_args));

ret = pfm_£nd_event("cpu_cycles", &inp.pfp_events[0].event);
if (ret != PFMLIB_SUCCESS) {
    fprintf(stderr,"cpu_cycles not found\n");
    exit(1);
}

ret = pfm_£nd_event("ia64_inst_retired", &inp.pfp_events[1].event);
if (ret != PFMLIB_SUCCESS) {
    fprintf(stderr,"ia64_inst_retired not found\n");
    exit(1);
}

inp.pfp_d¤_plm = PFM_PLM3;
inp.pfp_event_count = 2;

ret = pfm_dispatch_events(&inp, NULL, &outp, NULL);
```

```
if (ret != PFMLIB_SUCCESS) {
    fprintf(stderr, "cannot configure events: %s\n", pfm_strerror(ret));
    exit(1);
}

for (i=0; i < outp.pfp_pmc_count; i++) {
    pc[i].reg_num   = outp.pfp_pmcs[i].reg_num;
    pc[i].reg_value = outp.pfp_pmcs[i].reg_value;
}
for (i=0; i < inp.pfp_event_count; i++) {
    pd[i].reg_num          = pc[i].reg_num;
    pd_before[i].reg_num   = pc[i].reg_num;
    pd_after[i].reg_num    = pc[i].reg_num;
}

ret = perfmonctl(0, PFM_CREATE_CONTEXT, ctx, 1);
if (ret == -1) {
    fprintf(stderr, "PFM_CREATE_CONTEXT errno %d\n", errno);
    exit(1);
}

fd = ctx[0].ctx_fd;

ret = perfmonctl(fd, PFM_WRITE_PMCS, pc, outp.pfp_pmc_count);
if (ret == -1) {
    fprintf(stderr, "PFM_WRITE_PMCS errno %d\n",errno);
    exit(1);
}

ret = perfmonctl(fd, PFM_WRITE_PMDS, pd, inp.pfp_event_count);
if (ret == -1) {
    fprintf(stderr, "PFM_WRITE_PMDS errno %d\n",errno);
    exit(1);
}

load_args.load_pid = getpid();

ret = perfmonctl(fd, PFM_LOAD_CONTEXT, &load_args, 1);
```

```
if (ret  == -1) {
    fprintf(stderr, "PFM_LOAD_CONTEXT errno %d\n",errno);
    exit(1);
}

pfm_self_start(fd);

double instrAccum = 0.0, cycleAccum = 0.0;
int iter;
for ( iter = 0; iter < 20; ++iter ) {

    perfmonctl( fd, PFM_READ_PMDS, pd_before, inp.pfp_event_count );

    /*
     ******************** CODE TO BE PROFILED STARTS HERE ********************
     */




    /*
     ********************* CODE TO BE PROFILED ENDS HERE *********************
     */

    perfmonctl( fd, PFM_READ_PMDS, pd_after, inp.pfp_event_count );

    unsigned long cycle = pd_after[0].reg_value - pd_before[0].reg_value;
    unsigned long instr = pd_after[1].reg_value - pd_before[1].reg_value;

    instrAccum += instr;
    cycleAccum += cycle;
} // for

pfm_self_stop(fd);

printf( "\n" "           Instructions        CPU Cycles\n"
              "           ------------        ----------\n" );

printf( "Avg:   %20.3f %20.3f\nTotal: %20.3f %20.3f\n\n",
            instrAccum / iter, cycleAccum / iter,
            instrAccum, cycleAccum );

    close(fd);
    return 0;
}
```

B.2 Statistical Profiling Metric Test Programs

B.2.1 First Sample-Analysis Test Program

```
#ifdef __U_CPLUSPLUS__
#include <uC++.h>
#endif

int a = 4, b = 19, c = 25, d = 99, e = 34, f = 7;

void A(); void B(); void C(); void D();
void E(); void F(); void G( int ); void H();

void A() {
    for ( int i = 0; i < 250000; ++i ) {
        a += a + b + c + d + e + f;
        b += a + b + c + d + e + f;
        c += a - b - c - d - e - f;
    } // for
    C();
    D();
} // A

void B() {
    for ( int i = 0; i < 3500000; ++i ) {
        a += a + b + c + d + e + f;
        b += a + b + c + d + e + f;
        c += a - b - c - d - e - f;
    } // for
    E();
    F();
```

```
} // B

void C() {
    for ( int i = 0; i < 500000; ++i ) {
        a += a + b + c + d + e + f;
        b += a + b + c + d + e + f;
        c += a - b - c - d - e - f;
    } // for
} // C

void D() {
    for ( int i = 0; i < 100000; ++i ) {
        a += a + b + c + d + e + f;
        b += a + b + c + d + e + f;
        c += a - b - c - d - e - f;
    } // for
    G( 200000 );
} // D

void E() {
    for ( int i = 0; i < 500000; ++i ) {
        a += a + b + c + d + e + f;
        b += a + b + c + d + e + f;
        c += a - b - c - d - e - f;
    } // for
    G( 600000 );
} // E

void F() {
    for ( int i = 0; i < 750000; ++i ) {
        a += a + b + c + d + e + f;
        b += a + b + c + d + e + f;
        c += a - b - c - d - e - f;
    } // for
} // F

void G( int numTimes ) {
    for ( int i = 0; i < numTimes; ++i ) {
        a += a + b + c + d + e + f;
        b += a + b + c + d + e + f;
        c += a - b - c - d - e - f;
    } // for
    H();
    if ( numTimes == 200000 ) {  // if called by D, call H twice
        H();
    } // if
} // G

void H() {
    for ( int i = 0; i < 100000; ++i ) {
        a += a + b + c + d + e + f;
        b += a + b + c + d + e + f;
        c += a - b - c - d - e - f;
    } // for
} // H

#ifdef __U_CPLUSPLUS__
void uMain::main() {
#else
int main() {
#endif
    for ( int iter = 0; iter < 20; ++iter ) {
        for ( int i = 0; i < 1000000; ++i ) {
            a += a + b + c + d + e + f;
            b += a + b + c + d + e + f;
            c += a - b - c - d - e - f;
        } // for
        A();
        B();
    } // for
} // main
```

B.2.2 Second Sample-Analysis Test Program

```
#ifdef __U_CPLUSPLUS__
#include <uC++.h>
#endif
```

```
void a(); void b(); void c(); void d(); void e();
void f(); void x(); void y(); void z(); void q( void (*)() );

void a() {
    q( d );
} // a

void b() {
    q( e );
} // b

void c() {
    q( f );
} // c

void d() {
    q( x );
} // d

void e() {
    q( y );
} // e

void f() {
    q( z );
} // f

void x() {
    for ( int i = 0; i < 20000000; ++i ) {}
} // x

void y() {
    for ( int i = 0; i < 40000000; ++i ) {}
} // y

void z() {
    for ( int i = 0; i < 60000000; ++i ) {}
```

```
} // z

void q( void (*func)() ) {
    func();
} // q

#ifdef __U_CPLUSPLUS__
void uMain::main() {
#else
int main() {
#endif
    for ( int iter = 0; iter < 20; ++iter ) {
        a();
        b();
        c();
    } // for
} // main
```

Bibliography

[Ado99] Adobe Systems Incorporated Staff. *PostScript Language Reference*. Addison-Wesley, March 1999.

[Adv02] Advanced Micro Devices, Inc. *AMD Athlon Processor x86 Code Optimization Guide*, February 2002. http://www.amd.com/us-en/-assets/content_type/white_papers_and_tech_docs/22007.pdf. Last accessed May 2005.

[AGH00] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison-Wesley, third edition, 2000.

[AL90] Thomas E. Anderson and Edward D. Lazowska. Quartz: a tool for tuning parallel program performance. In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 115–125. ACM Press, 1990.

[Ayd03] Ruth A. Aydt. *The Pablo Self-Defining Data Format*. Pablo Research Group, Department of Computer Science, University of Illinois at Urbana-Champaign, 2003. ftp://vibes.cs.uiuc.edu/pub/-Pablo.Release.5/SDDF/Documentation/SDDF.ps.gz. Last accessed May 2005.

[BBG+93] F. Bodin, P. Beckman, D. Gannon, S. Yang, S. Kesavan, A. Malony, and B. Mohr. Implementing a parallel C++ runtime system for scalable parallel systems. In *Proceedings of the 1993 Supercomputing Conference*, pages 588–597, November 1993.

[BDG$^+$00] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*. IEEE Computer Society, November 2000.

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

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

[BKS96] Peter A. Buhr, Martin Karsten, and Jun Shih. KDB: A multi-threaded debugger for multi-threaded applications. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 80–87. ACM Press, 1996.

[BM03] Rudolf Berrendorf and Bernd Mohr. *PCL – The Performance Counter Library: A Common Interface to Access Hardware Performance Counters on Microprocessors (Version 2.2).* Computer Science Department, University of Applied Sciences Bonn-Rhein-Sieg, 53754 Sankt Augustin, Germany, January 2003. http://www.fz-juelich.de/zam/PCL/-doc/pcl/pcl.pdf. Last accessed May 2005.

[BS05] Peter A. Buhr and Richard A. Stroobosscher. *μC++ Annotated Reference Manual, Version 5.1.0.* School of Computer Science, University of Waterloo, May 2005. ftp://plg.uwaterloo.ca/pub/uSystem/-uC++.ps.gz. Last accessed May 2005.

[Buh99] Peter A. Buhr. *μC++ Monitoring and Visualization Reference Manual, Version 1.1.* School of Computer Science, University of Waterloo, February 1999. ftp://plg.uwaterloo.ca/pub/MVD/Visualization.ps.gz. Last accessed May 2005.

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

[CL00] Sung-Eun Choi and E. Christopher Lewis. A study of common pitfalls in simple multi-threaded programs. In *Proceedings of the thirty-first SIGCSE Technical Symposium on Computer Science Education*, pages 325–329. ACM Press, 2000.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* The MIT Press, second edition, 2001.

[CN93] Peter Coad and Jill Nicola. *Object-Oriented Programming.* Prentice Hall PTR, 1993.

[Den97] Robert R. Denda. Profiling concurrent programs. Diplomarbeit, Fakultät für Mathematik und Informatik, Universität Mannheim, September 1997. ftp://plg.uwaterloo.ca/pub/theses/-DendaThesis.ps.gz. Last accessed May 2005.

[dot] Graphviz. http://www.research.att.com/sw/tools/graphviz. Last accessed May 2005.

[DR99] Luiz A. De Rose and Daniel A. Reed. SvPablo: A multi-language architecture-independent performance analysis system. In *Proceedings of the 1999 International Conference on Parallel Processing*, pages 311–318, September 1999.

[Gai86] Jason Gait. A probe effect in concurrent programs. *Software – Practice and Experience*, 16(3):225–233, March 1986.

[gcc] The GNU compiler collection. http://gcc.gnu.org.

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

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GKM82] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: a call graph execution profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, pages 120–126. ACM Press, 1982.

[Gmb98] Pallas GmbH. *Vampir 2.0 User's Manual*, July 1998. http://-www.caspur.it/Files/2003/09/23/1064335667402.pdf. Last accessed May 2005.

[GR89] Narain Gehani and William D. Roome. *The Concurrent C Programming Language*. Silicon Press, 1989.

[Hal85] Robert H. Halstead, Jr. Multilisp: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

[Han75] Per Brinch Hansen. The programming language concurrent pascal. *IEEE Transactions on Software Engineering*, SE-1(2):199–207, June 1975.

[HE91] Michael T. Heath and Jennifer A. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, 8(5):29–39, September 1991.

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

[HM93] Jeffrey K. Hollingsworth and Barton 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.

[HMC94] Jeffrey K. Hollingsworth, Barton P. Miller, and Jon Cargille. Dynamic program instrumentation for scalable performance tools. In *Proceedings of the 1994 Scalable High-Performance Computing Conference*, pages 841–850, May 1994.

[HMG+97] Jeffrey K. Hollingsworth, Barton P. Miller, Marcelo J. R. Goncalves, Oscar Naim, Zhichen Xu, and Ling Zheng. MDL: A language and compiler for dynamic program instrumentation. In *Proceedings of the 1997 International Conference on Parallel Architectures and Compilation Techniques*, pages 201–212, November 1997.

[Hol94] Jeffrey Kenneth 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 2005.

[HWG03] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *The C# Programming Language.* Addison-Wesley, October 2003.

[Int04] Intel Corporation. *Intel Itanium 2 Processor Reference Manual for Software Development and Optimization*, May 2004. ftp://-download.intel.com/design/Itanium2/manuals/25111003.pdf. Last accessed May 2005.

[Int05] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual Volume 3: System Programming Guide*, April 2005. ftp://-download.intel.com/design/Pentium4/manuals/25366815.pdf. Last accessed May 2005.

[Jai91] Raj Jain. *The Art of Computer Systems Performance Analysis.* John Wiley & Sons, Inc., 1991.

[JFL98] Minwen Ji, Edward W. Felten, and Kai Li. Performance measurements for multithreaded programs. In *Proceedings of the 1998 ACM SIGMET-*

RICS Joint International Conference on Measurement and Modeling of Computer Systems, pages 161–170. ACM Press, 1998.

[KR88] Brian W. Kernighan and Dennis Ritchie. *The C Programming Language*. Prentice Hall, second edition, 1988.

[lib] The libunwind project. http://www.hpl.hp.com/research/linux/-libunwind.

[LP85] Carol H. LeDoux and D. Stott Parker, Jr. Saving traces for Ada debugging. In *Proceedings of the 1985 annual ACM SIGAda international conference on Ada*, pages 97–108. Cambridge University Press, 1985.

[MCC$^+$95] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The Paradyn parallel performance measurement tools. *IEEE Computer*, 28(11):37–46, November 1995.

[ME02] David Mosberger and Stephane Eranian. *IA-64 Linux Kernel – Design and Implementation*. Prentice Hall PTR, 2002.

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

[NL01] Ernesto Novillo and Paul Lu. On-line debugging and performance monitoring with barriers. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium*, San Francisco, California, April 2001. http://www.cs.ualberta.ca/∼ernie/SBT/download/-sbt-ipdps2001.ps.gz. Last accessed May 2005.

[Nov02] Ernesto Novillo. On-line performance monitoring of shared memory parallel programs using barriers. Master's thesis, Department of Computing Science, University of Alberta, 2002. http://www.cs.ualberta.ca/-∼ernie/SBT/download/Ernesto_Novillo-MSc_Thesis.ps.gz. Last accessed May 2005.

[opr] OProfile. http://oprofile.sourceforge.net.

[Ous96] John Ousterhout. Why threads are a bad idea (for most purposes). Invited talk at 1996 USENIX Technical Conference, January 1996. http://home.pacbell.net/ouster/threads.pdf. Last accessed May 2005.

[per] The perfmon project. http://www.hpl.hp.com/research/linux/perfmon. Last accessed May 2005.

[Pet] Mikael Pettersson. The perfctr hardware counter linux-kernel patch. http://user.it.uu.se/∼mikpe/linux/perfctr. Last accessed May 2005.

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

[qpr] The qprof project. http://www.hpl.hp.com/research/linux/qprof. Last accessed May 2005.

[qto] The q-tools project. http://www.hpl.hp.com/research/linux/q-tools. Last accessed May 2005.

[RBC$^+$03] Michiel Ronsse, Koen De Bosschere, Mark Christiaens, Jacques Chassin de Kergommeaux, and Dieter Kranzlmüller. Record/replay for nondeterministic program executions. *Communications of the ACM*, 46(9):62–67, September 2003.

[Rep91] John H. Reppy. CML: A higher-order concurrent language. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pages 293–305. ACM Press, 1991.

[RRA$^+$93] Daniel A. Reed, Phillip C. Roth, Ruth A. Aydt, Keith A. Shields, Luis F. Tavera, Roger J. Noe, and Bradley W. Schwartz. Scalable performance analysis: The Pablo performance analysis environment.

In *Proceedings of the 1993 Scalable Parallel Libraries Conference*, pages 104–113. IEEE Computer Society, October 1993.

[Sch99] Oliver Schuster. Replay of concurrent shared-memory programs. Diplomarbeit, Fakultät für Mathematik und Informatik, Universität Mannheim, April 1999. ftp://plg.uwaterloo.ca/pub/theses/-SchusterThesis.ps.gz. Last accessed May 2005.

[SG98] Abraham Silberschatz and Peter Baer Galvin. *Operating System Concepts*. Addison-Wesley, fifth edition, 1998.

[She99] Sameer Shende. Profiling and tracing in linux. In *Proceedings of the Extreme Linux Workshop #2*, Monterey, CA, June 1999. USENIX.

[Sno92] C.R. Snow. *Concurrent Programming*. Cambridge University Press, 1992.

[Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, third edition, 1997.

[Sun] Sun Microsystems. docs.sun.com: Extended library functions. http://-docs.sun.com/db/doc/817-0694/6mgfphvf2?q=cpc&a=view. Last accessed May 2005.

[Sun04] Sun Microsystems. *UltraSPARC III Cu User's Manual, Version 2.2.1*, January 2004. http://www.sun.com/processors/manuals/USIIIv2.pdf. Last accessed May 2005.

[TAU04] Department of Computer and Information Science, University of Oregon. *TAU User's Guide, version 2.13*, 2004. http://-www.cs.uoregon.edu/research/paracomp/tau/tauprofile/docs/-usersguide.pdf. Last accessed May 2005.

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

[U.S00] U.S. Government. *Ada Reference Manual*, 2000. ISO/IEC 8652:1995(E) with Technical Corrigendum 1. http://www.adapower.com/rm95/RM-TTL.html. Last accessed May 2005.

[XMN99] Zhichen Xu, Barton P. Miller, and Oscar Naim. Dynamic instrumentation of threaded applications. In *Proceedings of the seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 49–59. ACM Press, 1999.

[YM88] Cui-Qing Yang and Barton P. Miller. Critical path analysis for the execution of parallel and distributed programs. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 366–373, June 1988.

[Zak00] Dorota 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 2005.