Analyzing Multi-Threaded Program Performance with μProfiler

by

Dorota Zak

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, 2000

I hereby declare that I am the sole author of this thesis.

I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the University of Waterloo to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

Threads are widely supported by many operating systems and languages to allow concurrency in both uni-processor and multi-processor architectures. They can be used as a program structuring tool in the uni-processor environment or to accelerate the execution of an application in the multi-processor environment. Unfortunately, the actual behaviour of a multi-threaded program is often quite different from expectations and frequently does not achieve desired performance.

Since good performance is important to users and performance tuning is not easy, programmers need profiling tools to help them understand program execution and find its hot spots and bottlenecks. Profiling tools usually contain several metrics to let users select a metric or metrics that provide the best understanding of a program's run-time behaviour.

This thesis describes the design and implementation of a profiler, called μProfiler, for the μC++ user-level thread library that can execute in uni-processor and multi-processor shared-memory environments. Four new built-in metrics are presented, each characterizing various aspects of program behaviour, giving users an opportunity to view an execution from different perspectives.

Acknowledgements

First of all, I want to show my gratitude to my supervisor Dr. Peter Buhr for his time, encouragement, patience and guidance throughout development of this thesis. I would also like to thank my readers, Dr. Rick Holt and Dr. Stephen Mann, for their valuable suggestions and comments.

Furthermore, I want to thank Robert Denda for answering my numerous questions about the initial implementation of μProfiler, and Ashif S. Harji for technical information about μC++ and his help in debugging more than one problem. Also, I would like to thank my lab mates: Greg, Jiongxiong and Tom for making my work very enjoyable.

Thanks to all my friends in Waterloo and Toronto for putting up with my constant complaining and reminding me that there is life outside a computer lab.

And last but not least, "thank you" goes to my family, especially to my parents, for their love and support during my long:-) educational journey.

Trademarks

Ada is a registered trademark of the U.S.Government (Ada Joint Program Office)

AIX is a registered trademark of International Business Machines Corporation

IRIX is a registered trademark of Silicon Graphics, Inc.

Java is a registered trademark of Sun Microsystems, Inc.

Motif is a registered trademark of Open Systems Foundation, Inc.

Solaris is a registered trademark of Sun Microsystems, Inc.

SPARC is a registered trademark of Sparc International, Inc.

SunOS is a registered trademark of Sun Microsystems, Inc.

ULTRIX is a registered trademark of Digital Equipment Corporation

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited

X Window System is a trademark of The Open Group

Abbreviations

AIMS	Automated Instrumentation and Monitoring System
BFD	Binary File Descriptor Library
CG&RT	Call Graph and Run Times Metric
CPU	Central Processing Unit
EST	Execution States Transition Metric
GDB	GNU Source-Level Debugger
HLT	High Level Tracing Metric
HPF	High Performance Fortran
I/O	Input/Output
KDB	Kalli's DeBugger
MUI	Memory Usage Information Metric
MVD	Monitoring, Visualization and Debugging
POET	Partial-Order Event Tracer
SDDF	Self-Defining Data format
SMART	Shared Memory Application Replaying Tool
TAU	Tuning and Analysis Utilities

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Threads, i.e., multiple streams of execution within a single address space, are widely supported by many operating systems and languages. Some languages, such as *Ada* [U.S94] and *Java* [GJSB00], contain primitives that enable concurrent execution. Other, initially sequential, languages have developed thread libraries to allow concurrency in both uni-processors and multi-processor architectures. In the uni-processor environment, threads may be used to simplify the structure of a program. In the multi-processor environment, several threads can run simultaneously, accelerating the execution of an application.

Unfortunately, the actual behaviour of a multi-threaded program is often quite different from expectations and frequently does not achieve desired performance. The performance of a parallel application is affected by more factors than that of a sequential program. The most important of these factors are:

- competition for resources

- synchronization
- context-switching
- non-overlapping I/O

Since good performance is important to users and performance tuning is not easy, programmers need profiling tools to help them understand program run-time behaviour and find its *hot spots*, i.e., parts of code that use a disproportionately high amount of processor time, and *bottlenecks*, i.e., areas of code that restrict performance causing delays.

There are several major challenges in building a profiler for threaded programs [XMN99]:

- *associating performance data with individual threads:* Since several threads may execute the same parts of code, it must be ensured that correct counters and timers are updated. One way to accomplish this is to create separate profiling data structures for each thread and make the thread responsible for updating the structures.

- *profile thread preemption:* To gather execution-time information on a thread basis, a timer, associated with a particular thread, must be stopped and restarted every time the thread's execution is blocked and resumed. Since this involves instrumenting context switches, special care must be taken that the instrumentation code does not cause a deadlock.

- *minimizing instrumentation overhead when gathering time information for individual threads:* The profiling overhead may cause high perturbation of application execution. If it is impossible to make the perturbation insignificant, gathered data should be corrected to achieve results that more closely reflect the execution of the un-instrumented application.

Another challenge for profiling tools is the amount of performance information gathered during concurrent-program execution. Even short-running applications can produce hundreds of events that need to be presented without overwhelming a user. Profiling tools incorporate several different techniques, such as metrics, visualization and sonification, to address this problem. *Metrics* use measurements to quantify program run-time behaviour by computing numeric values for various constructs, e.g., routines, processors, mutex objects. *Visualization* and *sonification* use graphics and sound respectively to present the metric values.

For different programs and problems, different metrics provide the most useful "picture" of a program's behaviour. Therefore, profiling tools usually include several metrics to allow users to select the best one for their application.

This thesis presents the design and implementation of a profiling tool called *μProfiler*, which is used to profile concurrent programs written in μC++ [BS99], a user-level thread library for C++. μProfiler contains six built-in metrics, four of which were developed for this thesis:

- *Performance Profile Information* uses statistical monitoring to find the total time a task spends in ready, running and blocked states (not discussed in the thesis).

- *UNIX Resources Usage Information* provides information about the operating system's resource-usage (not discussed in the thesis).

- *Call Graph and Run Times* generates an execution profile that includes the number of function calls, and the real and CPU time spent in each function called during execution.
- *Execution State Transition* records characteristics of each state a thread enters during its execution.
- *High Level Tracing* traces the activities of task threads and their interactions with coroutines, monitors, coroutine-monitors and other tasks.
- *Memory Usage Information* reports memory leaks for dynamically allocated storage.

μProfiler also allows users to implement additional metrics and incorporate them into the profiler. Much of this extensibility capability was developed for this thesis.

1.1 Thesis Outline

Chapter 2 explains profiling terms and introduces a taxonomy for profiling methods and metrics.

Chapter 3 presents related work in the field. It describes four major profiling tools for concurrent execution and lists several other tools for instrumentation insertion, performance data analysis and visualization.

Chapter 4 gives an overview of μC++, which is the target environment for μProfiler, as well as other existing profiling and debugging tools supported by μC++.

Chapter 5 describes the objectives, design, instrumentation insertion and implementation of μProfiler.

The next four chapters present the four μProfiler's built-in metrics developed for this thesis. Chapter 6 presents a *Call Graph and Run Time* metric, based on a *gprof* [GKM82] sequential tool. Chapter 7 describes an *Execution State Transition* metric, Chapter 8 talks about *High Level Tracing* and Chapter 9 illustrates a *Memory Usage Information* metric.

Finally, Chapter 10 summarizes the main aspects of this thesis and presents ideas for future work.

Chapter 2

Taxonomy of Profiling

This chapter presents a classification of profiling methods and metrics. Profiling is a performance evaluation technique based on direct measurement; other performance evaluation techniques include analytical modeling and simulation.

The profiling process consists of three phases:

- **instrumentation insertion**

  Instrumentation is inserted into an application and the run-time system to allow data gathering.

- **program execution and performance data collection**

  The application is executed and its run-time behaviour is monitored through the inserted instrumentation, collecting data according to a metric specification (see Section 2.2).

- **performance data analysis and visualization**

  The data is processed according to a metric algorithm and the results are displayed on a screen or saved into a file.

Profiling is an iterative process and can be executed several times, each time using different application data and/or metric.

2.1 Instrumentation Insertion

Instrumentation is added to a program to generate performance data. It is used to identify the part of code that is executing when a specific event occurs, e.g., function entry or exit. Instrumentation consists of three parts [MCI+95]:

- *instrumentation points*: locations in an application's code where instrumentation can be inserted.
- *instrumentation primitives*: counters or timers, and operations to change their values, e.g., set, add, reset.
- *instrumentation predicates*: boolean expressions that guard the execution of primitives, e.g., if statements.

The *primitives* and associated *predicates* create *instrumentation hooks*, which can be inserted at various *instrumentation points* allowing computation of different metrics.

2.1.1 Direct and Indirect Instrumentation

Profiling instrumentation can be direct or indirect. In *direct instrumentation*, the profiling code is placed at instrumentation points. In *indirect instrumentation*,

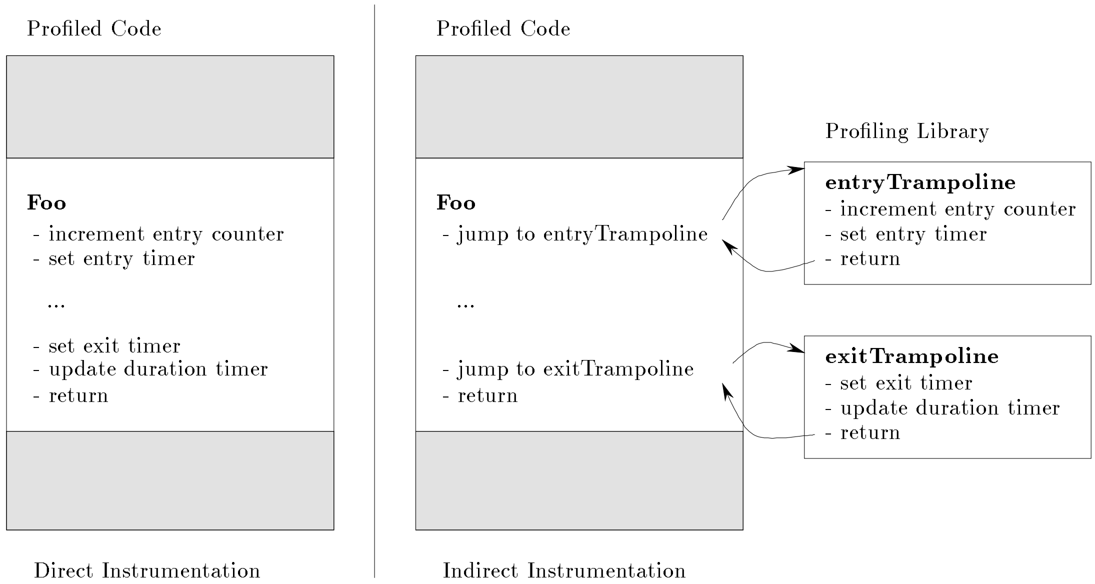


Figure 2.1: Instrumentation Types

which is more common, only a jump to a profiling routine, called a *trampoline*, is inserted at instrumentation points. Figure 2.1 presents an example of *direct* and *indirect instrumentation* for function entry and exit. The hooks gather information about the number of function calls and call duration.

2.1.2 Static and Dynamic Insertion

The instrumentation may be inserted into a program statically or dynamically.

Static insertion may be performed at any stage of the compilation-linking process before the execution stage (refer to Figure 2.2). Some profiling tools, such as *TAU* (Section 3.1), *JEWEL IS* [WRM+97] and *BRISK* [BMR99a] expect a programmer to add the profiling instrumentation when coding an application. This approach gives a user a lot of flexibility in specifying what parts of the code are

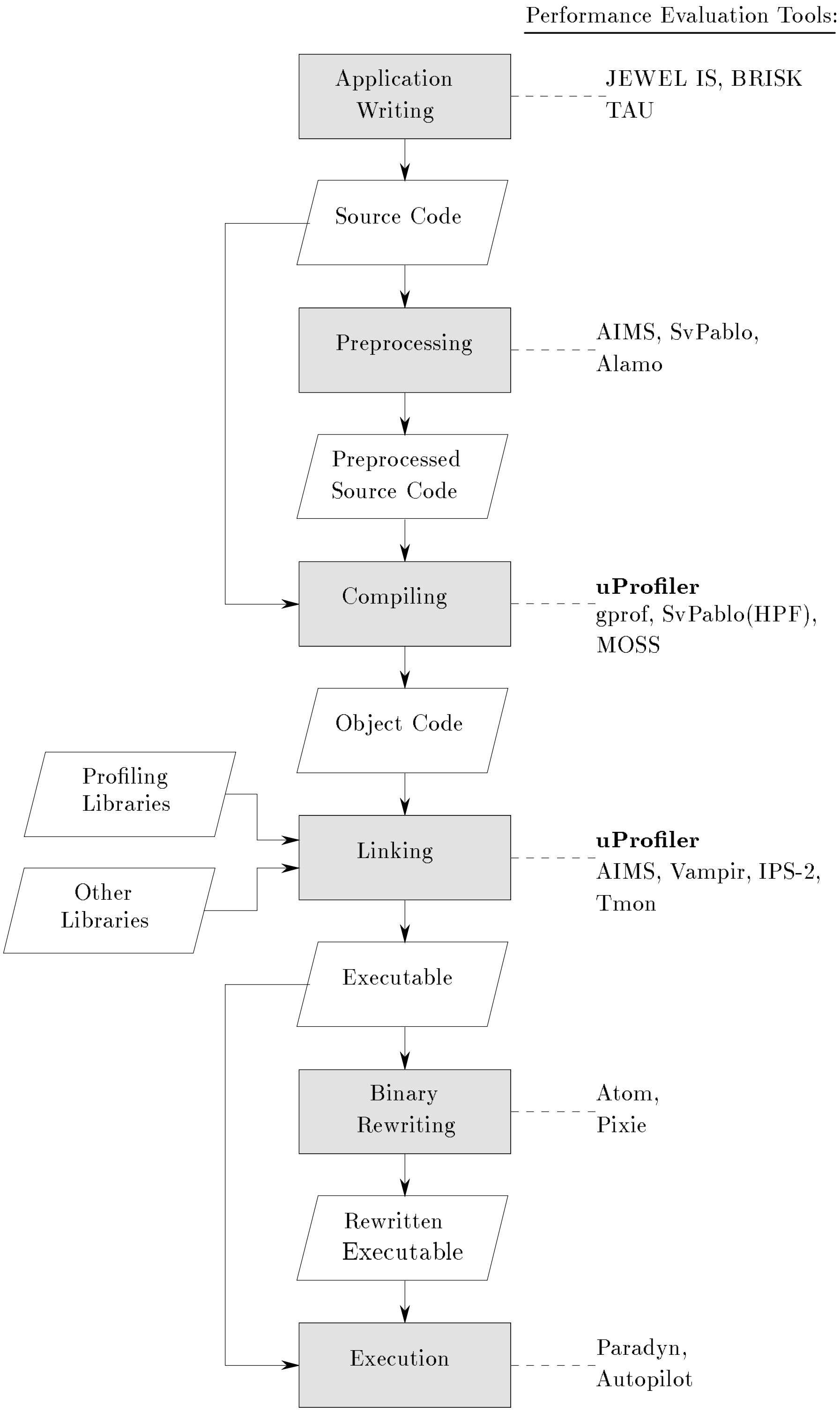


Figure 2.2: Instrumentation Insertion Options

profiled, but the process is also tedious and error-prone. As well, user insertion may be unsound because compiler optimization can move the profiling code beyond the desired instrumentation point. *Static insertion* can be also performed automatically and safely by a preprocessor or a compiler when a profiling flag is specified on the compilation command. *AIMS* (Section 3.2), *SvPablo* (Section 3.4.2) and *Alamo* [JZTB98] use a preprocessor to insert the instrumentation, and *gprof* [GKM82], *MOSS* [ES98] and *μProfiler* (Chapter 5) utilize a compiler. *Static insertion* can be completed during the linking stage, as it is done by *Vampir* [Pal98], *IPS-2* [MCH$^+$90] and *Tmon* [JWL98]), or after the linking stage by rewriting a binary executable, as done by *Atom* [ES95] and *pixie* [pix00].

In *dynamic insertion*, the executable is instrumented during application runtime. While *dynamic insertion* can be used to achieve the equivalent of *static insertion*, its cost is usually greater. Therefore, *dynamic insertion* is mainly utilized to find application bottlenecks. A user or a profiler, if the search is automated, decides dynamically where, when and what hooks are inserted, reducing the profiling overhead (*probe effect*) by not profiling parts of code that are viewed as unimportant, i.e., code not causing performance problems. After data is collected, the instrumentation may be removed and different hooks may be inserted into other parts of the application. Since it takes some time for a user or a profiler to reach a decision about where the instrumentation should be inserted to narrow the search for a bottleneck, *dynamic insertion* is preferable for long-running programs. *Paradyn* (Section 3.3) and *Autopilot* (Section 3.4.1) are examples of tools that use *dynamic instrumentation insertion*.

Moving down along the compilation-execution process, the instrumentation insertion mechanism changes from language specific to platform specific. The language specific instrumentation makes the instrumentation portable across all compilers and platforms that support the language, but the instrumentation usually does not work with other languages as features may be different or non-existent. On the other hand, the platform specific instrumentation allows profiling programs written in different languages as long as they create executables for a given platform. However, it may be difficult to map execution events to source code variables and statements when *dynamic insertion* is used.

Some simple forms of profiling can be done without inserting any instrumentation, e.g., measuring total execution time, but this approach gathers a very limited amount of performance data and is not discussed in this thesis.

2.2 Metric

A *metric* is a function (measure) characterizing some aspect of program performance, e.g., CPU time, number of function calls, thread blocking time, etc.

Since there are two basic types of instrumentation primitives, timers and counters, there are two basic types of metrics: metrics based on time information and metrics based on count information. *Time Metrics* record the time of an event and possibly its duration. *Count Metrics* show the number of times an event occurs. In many cases, these basic types are combined to create more sophisticated metrics.

2.3 Monitoring

Monitoring is the process of collecting performance data, generated by profiling instrumentation, during application run-time. There are two general types of monitoring: exact and statistical.

In *exact monitoring*, a *monitor* collects information about all events relevant to active metrics. In *statistical monitoring*, a *monitor* samples the state of a running application periodically. The *statistical monitoring* introduces lower probe-effect but the gathered data is less accurate.

When an event occurs, a monitor can update summary statistics for the event (*direct profiling*), or record the event and its characteristics into a log (*tracing*).

Summary data can be collected through both statistical and exact monitoring. For example, the distribution of execution time across routines can be performed by periodically sampling the state of a program counter, as is done in *gprof* [GKM82], or by time-stamping each function entry and exit, as is done in μProfiler (see Chapter 6). The advantage of gathering statistics is the small amount of storage required for the data.

Tracing is usually only meaningful for *exact monitoring*. It involves collecting large quantity of data that may need to be saved periodically to stable storage to reduce memory usage or for robustness.

2.4 Analysis

Data collected during run-time is retained for analysis. An *analyzer* processes the data according to an active metric algorithm and prepares the resulting information for visualization. The processing involves filtering the data, performing computations and possibly mapping the data to source code. The analyses can be done during program execution, i.e., *on-the-fly*, or after the execution is finished, i.e., *post-mortem*.

On-the-fly analyses usually reduces the amount of storage needed for data since it is processed immediately; therefore, it is useful for long-running programs. Another plus is the fact that the information is usually displayed during application run-time possibly allowing a user to modify the application behaviour or switch the profiling on and off. The drawback is higher profiling overhead caused by analyzing and visualizing the data during execution. Furthermore, if a large amount of data must be processed, there may be a significant latency between an occurrence of an event and its visualization, making dynamic control difficult or impossible.

In *post-mortem* analyses, the data is stored during execution and then analyzed after the execution finishes. This method has smaller probe effect but may be infeasible for long-running programs, as a user may not want to wait several hours or days to get performance information, and the amount of information may be extremely large.

2.5 Visualization

Visualization is the last step in the profiling cycle. Its purpose is to display gathered information in a constructive way without overwhelming a user with unnecessary detail.

Depending on the particular metric, *visualizers* may show various views of a profiled system, including [NA94]:

- *single-time system snapshot*: display of system activities at a particular point in time.
- *animation*: a sequence of *single-time system snapshots* to demonstrate the system's dynamic behaviour.
- *statistics*: summary information for the whole time under investigation.
- *time-line system view*: detailed view of system activities visualized along a time axis.

The view of the system can be represented effectively using a variety of simple visualization techniques such as tables, charts and graphs.

Tables are useful for presenting discrete two-dimensional values. Grouping the data into rows and columns allows large amounts of information to be displayed in a condensed manner.

Discrete values can be also presented graphically using *charts*. The most popular charts are: *bar charts*, *pie charts* and *Gantt charts* (see [MR82]). *Pie charts* and *bar charts* are often used to represent percentage of time a processor spends performing

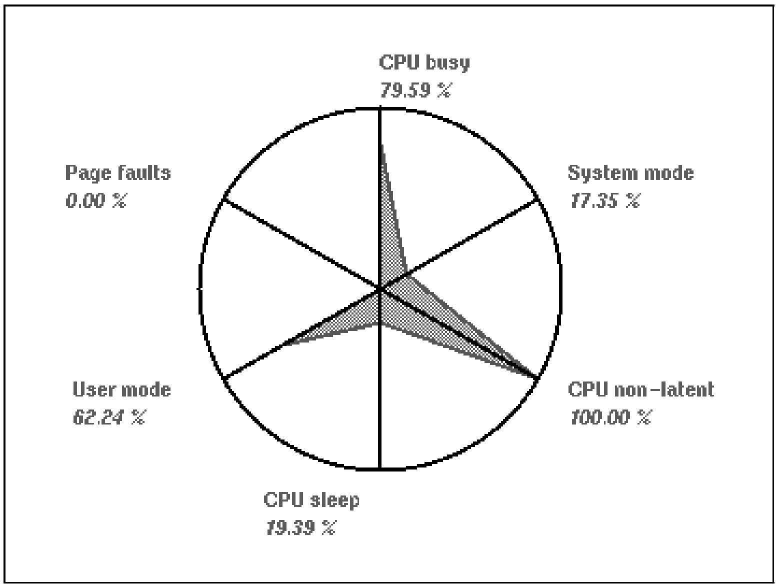


Figure 2.3: Example of a Kiviat Graph

various operations, e.g., calculation, communication, I/O operations (see [NA94]). A *Gantt chart* can be utilized, for example, to display execution of a task on different processors (see [Hea94]).

Another way to display performance data is using graphs. A *graph* is a diagram of points, lines and areas that represents multi-dimensional relations. Among frequently used graphs are *execution graphs* and *Kiviat graphs*.

An *execution graph* is a directed acyclic graph in which the arcs join consecutive events of a single process and events from different processes when it is known that one event precedes the other, e.g., the event of sending a message happens before the message is received (see [HCHP92]).

A *Kiviat graph* presents several metrics in one graph. The metrics are chosen arbitrarily, usually alternating metrics for which high values are better with metrics with preferable low values. Figure 2.3 presents an example of a *Kiviat graph*. The graph shows six metrics whose values are marked on radial lines. Each value is connected to the values of its neighbouring metrics creating the inner area of the graph (gray-shaded). Since, the metrics with "high" and "low" values are placed on the circle alternately, the resulting graph has a shape of a N-pointed star, where N is the number of "high"-value metrics. An imperfectly formed star may indicate a problem for the metric that causes the disfiguring. By observing the pattern of the star and relations among metrics, it is possible to detect some problems that would be otherwise hard to discover.

More information about design aspects of graphical data displays can be found in [Tuf84].

Chapter 3

Related Work

This chapter describes four profiling tools for concurrent programs. The tools are commercially or publicly available and are supported by their vendors or developers.

The first two projects: TAU and AIMS have been chosen because they profile programs written in a specific language or a group of languages, insert profiling instrumentation into source code, and analyze and visualize gathered data post-mortem. In other words, they are tools similar to μProfiler.

Paradyn, on the other hand, is not designed for specific languages but for specific platforms. It inserts instrumentation dynamically during program execution, and performs data analysis and visualization on-the-fly.

Pablo has been selected because it is an example of an integrated environment in which a compiler, system software and hardware cooperate to improve application performance. It contains several independent tools that can be used together to find performance characteristics of existing applications or model execution of parallel programs under development.

3.1 TAU

Tuning and Analysis Utilities (TAU) [MMC96, SMC+98] is a visual programming and performance analysis environment for pC++ [BBG+93], a language extension to C++ [Str97] that permits data-parallel operations. It uses the Sage++ preprocessor toolkit [BBG+94], which restructures a pC++ program into C++ code, for instrumentation insertion and accessing properties of program objects. TAU is also integrated with the pC++ run-time system for profiling and tracing support. It works on a variety of platforms including Linux [She99].

3.1.1 Instrumentation Insertion and Monitoring

The profiling in pC++ is done at a function level and there are three general classes of functions that can be profiled: thread-level functions, collection class methods and run-time system routines. By default, every function in a source file is profiled. However, a user can specify a subset of functions to be instrumented by using an *instrumentation command file* [Moh93].

The profiling code is inserted at function entry and exit points. The instrumentation is accomplished by inserting the declaration of an object from a Profiler class, which contains only a *constructor* and *destructor*, at the start of each function. The allocation and deallocation of the Profiler object occurs on function entry and exit, which cause implicit calls to the *constructor* on entry and the *destructor* on exit; in C++, the *destructor* is executed no matter how the function exits [MMB+94]. Various implementations of the Profiler can be easily created by providing different code for the *constructor* and *destructor*. Currently, there are two versions of the

Profiler: one based on direct profiling and the other based on tracing.

In direct profiling, the Profiler object gathers data similar to the UNIX *prof* tool [GKM82], such as the number of calls into a function, execution time spent in the function alone (*exclusive timing*), and time spent in the function and its children (*inclusive timing*). The Profiler *constructor* records the function name, its parent's name, the time-stamp of entry, and increments an entry counter. The *destructor* uses the entry time-stamp and the exit time-stamp to calculate the duration of the function call and adds the duration to the corresponding *inclusive* and *exclusive timers*. It also subtracts the time from the *exclusive timer* of the parent function. At the exit of the main() function, the *destructor* writes the profile data for the whole application into a file.

In trace-based profiling, the *constructor* and *destructor* call event logging functions from the pC++ *event tracing library* to record the type of an event, its time-stamp and the processor on which it occurred. The resulting trace files can be converted into the SDDF format used by *Pablo* performance analysis environment (Section 3.4), *ALOG* format used by *Upshot* [HL91], or *Parvis PV* format [NA94].

3.1.2 Analysis and Visualization

TAU performs the analysis and visualization of the gathered data post-mortem using several tools implemented as graphical hyper-tools for easy extendibility [MBM94]. Each tool performs a set of predefined tasks. If one tool needs a feature of another, it sends a message to the other tool requesting it. The tools also support global features, such as *select-function, select-class* and *switch-application*. If a global feature is invoked in any of the tools, it is automatically executed in all currently running tools.

3.1.2.1 Static Analysis Tools

There are three tools that enable a quick overview of a large pC++ program:

- *File And Class Display (FANCY)* is a global function and method browser that can be used to quickly locate the source code for a specific routine. It displays a list of classes and their methods as well as a list of files and global functions in each file.

- *Call Graph Extended Display (CAGEY)* shows a static call-graph of functions and methods in a user application. It uses Sage++ to determine the call-graph structure and to differentiate between global functions and class methods.

- *Class Hierarchy Browser (CLASSY)* is a class hierarchy display for programs written in object-oriented languages like C++ or pC++. It provides information about selected class members and their properties.

The static analysis tools are integrated with the dynamic analysis tools through the global features of TAU.

3.1.2.2 Dynamic Analysis Tools

The dynamic analysis tools analyze program execution behaviour using data gathered through direct profiling, event tracing or break-point debugging. There are currently five such tools:

- *Routine and Data Access Profile Display (RACY)* displays information gathered during direct profiling in a bar chart form. It provides the information on a node (processor or a group of processors) basis, i.e., what functions were accessed by the specified node, or on a function basis, i.e., which nodes executed the given function.

- *Parallel Profile Tool (PPROF)* presents the same information as *RACY* but in a text format similar to UNIX *prof* display.

- *Event and State Display (EASY)* shows states and events for individual processors on an XY graph. The X axis displays elapsed time and the Y axis represents the processors. The states and events are displayed as graphical objects, e.g., circles, arrows or lines. A user can get more information about a particular event or state by clicking on the corresponding graphical object. *EASY* is based on the *Upshot* tool.

- *Breakpoint Executive Environment for Visualization and Data Display (BREEZY)* [BHMM94] is a parallel debugger that allows user control over the execution of a pC++ program.

- *Speedup and Parallel Execution Extrapolation Display (SPEEDY)* [MMS95] predicts the performance of a n-thread pC++ program in a n-processor environment based on data gathered during uni-processor execution.

Initial work has been done to provide TAU with a run-time monitoring framework [SMS99]. At user-defined time intervals, a run-time monitor accesses the profile data and displays it on a screen during program execution.

There also exists an external performance visualization tool, called *Parallel Program and Performance Display Environment (POPEYE)* [BHMM94] that is capable of creating sophisticated, multi-dimensional displays. *POPEYE* can be used to display profile and trace data gathered by TAU.

3.2 AIMS

Automated Instrumentation and Monitoring System (AIMS) [YSM95, YS96] is a toolkit for performance measurement and analysis of message-passing programs written in FORTRAN and C using NX, PVM or MPI communication libraries. The toolkit is supported on IBM SP2 platform and clusters of workstations such as SUNSparcs, SGI, SGI/PowerChallenge and HP.

AIMS consists of four major components: *xinstrument, monitor, perturbation compensation module*, and *analysis and visualization toolkit. xinstrument* performs instrumentation of a profiled application, which is then compiled and linked with a run-time performance monitoring library, called *monitor*. After execution, a trace file, created by the *monitor*, may be put through a *perturbation compensation (pc) module*, which removes monitoring overhead and its effects on the communication patterns. At the end, the data is processed and displayed using the *analysis and visualization toolkit.*

3.2.1 Instrumentation Insertion and Monitoring

xinstrument inserts instrumentation to collect data about synchronization operations, message passing, and invocations of functions, loops and user-defined code segments. It also generates an *application database*, which stores static information about the application such as file names and line numbers of instrumented constructs. This information is used by the *analysis tools* to relate collected data to the source code.

xinstrument has a graphical interface, which allows an interactive instrumentation. A user can instrument all source files, i.e., the default option, or browse through the source code and select specific constructs for profiling. A user can also specify the data-gathering mode, choosing between *Monitor Mode*, which generates a trace file, or *Statistics Mode*, which generates summary statistics.

After the instrumentation is done, the source-code is compiled and linked with the *monitor* library containing a set of time-stamping and trace-collection routines, for gathering performance data and storing it into trace files, one per process. There is no option to turn the profiling on and off during program execution.

3.2.2 Analysis and Visualization

The trace files generated by the *monitor* are concatenated, sorted by time and may be passed through the *perturbation compensation* module to remove event delays caused by performance monitoring. *pc* creates an updated trace file with time-stamps that more closely match the execution of the un-instrumented program.

The updated or non-updated trace file is used by the *analysis and visualization toolkit* to produce performance information. The toolkit contains four post-processing kernels [YMG99]:

- *View Kernel (VK)* displays the dynamics of program execution using animations. For example, it shows the amount, type and duration of I/O operations, network topology, and the rate of execution for each basic block of code.

- *Statistics Kernel (Tally)* presents statistics about time spent by each process in each function doing computation, communication and synchronization.

- *Performance Index and Statistics Kernel (Xisk)* attempts to explain performance failures by isolating possible causes and linking them to the application's data structures and functions. It shows how much shorter (in percentage form) the application would run if the problem were completely eliminated.

- *Modeling Kernel (MK)* automates the process of building and simulating parallel-program models.

3.3 Paradyn

Paradyn [MCI+95] is a profiling tool for parallel and distributed programs. It is designed to measure performance and find bottlenecks in programs running for hours or days on massively parallel computers, workstation clusters, and heterogeneous combinations of these systems. It is able to monitor un-instrumented programs that have already started to run.

Paradyn consists of the *main Paradyn process*, one or more *Paradyn daemons*, and zero or more *external visualization processes*. The multi-threaded *main Paradyn process* includes: *Data Manager* controlling the gathering of data, *Performance*

Consultant coordinating the search for bottlenecks, *Visualization Manager* responsible for communication between *Paradyn* and external visualization tools, and *User Interface Manager* displaying the information. The *Paradyn daemon* contains the platform-dependent parts of *Paradyn* and is responsible for instrumentation insertion.

3.3.1 Instrumentation Insertion and Monitoring

Paradyn performs dynamic instrumentation insertion and performance evaluation, i.e., the instrumentation is inserted and the gathered data processed and visualized during application run-time. The instrumentation is largely automated and controlled by the *Performance Consultant* module, which inserts it only into parts of the program related to a problem under investigation. The *Performance Consultant* starts looking for high-level problems, e.g., too much total synchronization blocking, I/O blocking or memory delays for the whole program, performing only small amount of instrumentation. When one of these general problems is identified, it adds additional instrumentation to locate more specific causes of the problem.

The *Performance Consultant* sends a message to a *Paradyn daemon* requesting particular instrumentation. The message is coded in the *Metric Description Language*, which describes the instrumentation in a machine-independent format [HMG+97]. The *Paradyn daemon* translates the request into a set of machine dependent instructions, called a *base trampoline*, and inserts it into the application by replacing one or more machine instructions with a branch to the *base trampoline*, and moving the replaced instructions into that trampoline. Each *base trampoline* contains *mini-trampolines* that update profiling counters and timers.

Paradyn is also capable of profiling threaded programs [XMN99]. It utilizes a design called *Same Instrumentation Code Multiple Data* in which all threads share the same instrumentation code, but each has its own private copy of performance counters and timers. The *base trampoline* used for instrumenting threaded programs has an extra section, called *MT Preamble*, that maps a thread ID to the address of its performance data structures. In the *mini-trampoline*, additional code computes the addresses of the thread counters or timers based on the value from the *MT Preamble*. Since each thread only updates its own data structures, there is no need to provide mutual exclusion when accessing the data, which significantly reduces the probe effect.

To monitor an already running multi-threaded program, the *base-trampoline* contains extra code that detects the first time a thread executes the instrumentation code. The extra code and the *MT Preamble* increase the cost of a trampoline for a threaded application about five times in comparison with one for non-threaded program.

3.3.2 Analysis and Visualization

The *Performance Consultant* module of *Paradyn* discovers performance problems by searching through the space defined by the W^3 *Search Model*, which consists of a list of potential performance problems (hypothesis). The *Performance Consultant* tests the different hypothesis and tries to find the part of code causing a problem. It uses the application's call-graph to direct the search [CMW00]. The search starts at the top of the call-graph and proceeds only along the branches on which the *inclusive timing* value, i.e., time spent in a routine and its callees, is above an expected level.

The *Performance Consultant* is also able to apply historical performance data, gathered in previous executions of an application, to the current search [KM99]. Incorporating historical data shortens the time required to identify bottlenecks and decreases the amount of instrumentation.

Paradyn's visualization interface allows external visualization tools to display *Paradyn's* performance data in real-time [KMLM97].

3.3.3 Related Projects

This sections lists other performance measurement tools devised by the *Paradyn* Project group.

Paradyn-J [New99] is a performance tool for interpreted, just-in-time (JIT) compiled and dynamically compiled Java executions. It examines how an application's performance is affected by various interactions between the virtual machine and an application, and is designed to be used by both an application and virtual machine developers.

Process Hijacking [ZML99] is a new process checkpoint and migration technique, which uses dynamic program re-writing to add check-pointing capability to a running program. It allows to checkpoint and migrate applications that cannot be re-linked with a checkpoint library.

KernInst [TM99] is a tool that performs fine-grained dynamic instrumentation of a completely unmodified, running, commodity operating-system kernel. On top of KernInst, there is a *kernel performance profiling tool*, which collects information about kernel and application performance.

3.4 Pablo

The *Pablo performance analysis environment* consists of several independent, but related, components such as *Autopilot, SvPablo, Virtue* and *Delphi*. All these components use the *Self-Defining Data Format* (SDDF) for information interchange.

SDDF [Ayd00] is a data-description language that specifies both data-record structures and data-record instances. SDDF is viewed as a data meta-format because it describes general data-records and does not dictate semantic meaning to the stored data. SDDF files are architecture-independent, allowing data collected on one machine to be analyzed and displayed on another.

In addition to the meta-format, *Pablo* supports a library of C++ classes that perform all operations necessary to write and interpret SDDF files.

3.4.1 Autopilot

Autopilot [RVS+99] is a toolkit for closed-loop, adaptive performance-tuning and resource management of heterogenous computational grids, often spread over different geographical areas. It automatically monitors system behaviour, determines what changes are required to improve performance and implements those changes during system run-time. *Autopilot* contains the following components: distributed performance *sensors*, software *actuators*, distributed name *servers*, sensor and actuator *clients*, and a robust *decision mechanism*.

Sensors capture performance data and send the raw data for processing or compute some metrics and send the metrics. Every sensor has a set of properties that are defined during its creation. Remote clients specify a list of properties they are interested in, and *Autopilot* managers give them back a list of sensors that match those properties. A client can also be notified every time a new sensor with the required properties is created. In this way, a client can access and manage a sensor without knowledge of the sensor's physical location or its creation time. The sensors can be dynamically activated and deactivated during application execution.

Actuators allow clients to modify the values of application variables and to remotely invoke application functions. They share most of the sensor features, e.g., property lists, dynamic creation and management.

Name servers are the *Autopilot* managers. They register and deregister remote sensors and actuators on their creation and destruction, and process remote client requests for accessing those sensors and actuators. Name servers allow clients to dynamically attach themselves to geographically distributed application components, perform required changes, and then release control.

Clients can connect to remote sensors and actuators through *Autopilot* managers. After connection, sensors send data to clients that process the data and issue commands to corresponding actuators. Clients can also change sensor behaviour by modifying its variables, e.g., buffer size or sampling rate.

Decision mechanism uses the data gathered by sensors to reach optimization decisions that are then implemented by actuators. The mechanism is based on fuzzy logic [Bez93] and balances conflicting optimization goals such as minimize response time and maximize throughput.

3.4.2 SvPablo

Source View Pablo (SvPablo) [DR99] is a graphical analysis and visualization system that profiles applications written in C, Fortran 77, Fortran 90 and High Performance Fortran (HPF), running for hours or days on a variety of sequential and parallel systems.

SvPablo can be integrated with *Autopilot* and *Virtue* (see Section 3.4.3) allowing users to explore their application behaviour, display it in a three dimensional virtual environment, and modify it dynamically using *Autopilot* sensors and actuators.

3.4.2.1 Instrumentation Insertion and Monitoring

SvPablo supports automatic and interactive instrumentation of a profiled application. Currently, it provides interactive instrumentation of C, Fortran 77 and Fortran 90 applications, and automatic instrumentation of HPF programs.

In the case of automatic instrumentation, *SvPablo* is integrated with a HPF compiler that inserts calls to a *SvPablo data-capture library* into the original program. Letting the compiler insert the instrumentation code ensures correctness and decreases the risk of inhibiting the compiler's optimization, especially since the HPF compiler supports aggressive optimization, producing executable code quite different than the source code.

The interactive instrumentation is performed in the preprocessing stage. *SvPablo* parses the source code to be profiled and marks all instrumentation points chosen by a user, e.g., function entry and exit or outer loops. After the parsing is finished, *SvPablo* generates a copy of the source code with embedded calls to the *data-capture library* for all marked instrumentation points. The application is then compiled using a regular compiler.

In addition to capturing data through software instrumentation, *SvPablo* gathers data from hardware performance counters to determine the interaction between software and hardware.

3.4.2.2 Analysis and Visualization

After program execution, the *SvPablo data-capture library* records its statistical analyses in a set of *summary files*, one file per processor. All the files are then merged into one *performance file* written in the SDDF format. The *performance file* is used as an input to a graphical browser, which presents the performance data in the context of the original source code.

The *SvPablo* browser supports a hierarchy of performance displays, ranging from high-level routine profiling to detailed information on the behaviour of a single line of code on a specific processor.

3.4.3 Virtue

Virtue [SWSR99] is a general-purpose toolkit for hierarchical visualizing three-dimensional graphs, manipulating the graphs and their representations, and annotating the graph components with audio and video comments. Examples of *Virtue's* graphs include: geographic displays of network activity, displays of parallel process interactions and call-graphs of source code structure.

Virtue is integrated with *Autopilot* for data gathering, and in turn *Autopilot* uses *SvPablo* to insert sensors and actuators into distributed application source code. *Virtue* also contains a set of direct manipulation controls, i.e., *sliders*, that can be used through *Autopilot* actuators to change application or system behaviour during its execution. All data collected by *Virtue* is written in a *graph description language* based on the SDDF format.

Virtue also supports video conferencing, capturing and replaying of multimedia annotations, voice commands, and generation of audio cues for synchronous and asynchronous cooperation among geographically dispersed research or system support teams.

3.4.4 Delphi

Delphi [RPF$^+$99] is an integrated performance measurement, analysis and prediction environment for multi-lingual, object-oriented applications executing on homogeneous and heterogenous parallel systems. The goal of *Delphi* is to assist system developers in evaluating existing and proposed systems by investigating the complex interactions among an application and system software, processors, I/O, and networks.

The *Delphi* system includes *compilers* that insert profiling instrumentation for gathering execution-cost information, *performance models* of key system components such as task scheduling, memory, I/O management and networks, and flexible *measurement and analysis software* for performance evaluation. *Delphi* contains instrumented libraries, which gather quantitative data needed to understand the interaction among system components. It uses *Paradyn, SvPablo* and *Autopilot* tools for system instrumentation and monitoring. Initial work has been done to incorporate *Virtue* for information visualization.

3.5 Other Profiling Projects

The advances in technology and the growing number of parallel and distributed systems increase the demand for parallel profiling tools. The reason for that demand is that these systems quite often achieve only a small fraction of their predicted performance.

This section highlights several other profiling projects designed to gather and/or analyze performance data that can be used for parallel performance tuning, which is not an easy process as shown by Anna Hondroudak and Rob Procter [HP98].

3.5.1 Monitoring and Adaptive Control

There are several instrumentation and monitoring systems that are part of a software toolkit or are stand-alone tools that can be incorporated into other performance analysis tools. *JEWEL Instrumentation System (IS)* [WRM+97] and *Baseline Reduced Instrumentation System Kernel (BRISK)* [BMR99a] are examples of stand-alone systems that collect run-time data from distributed applications and transfer it to data collectors. *A Light-weight Architecture for Mon-*

itoring (Alamo) [JZTB98] is another monitoring framework for gathering data, which can be processed and displayed by other analysis and visualization tools. *Tmon* [JWL98] is an on-line graphical performance monitor for multi-threaded programs. It performs the monitoring as well as performance data analyzing.

OMIS Compliant Monitoring System (OCM) [WTL98] is an instrumentation system built according to the *On-line Monitoring Interface Specification (OMIS)*, which describes a universal interface between on-line tools and monitoring systems that does not target any specific kind of tools but can handle various tools at the same time. *Mirror Object Steering System* (MOSS) [ES98] is an implementation of another instrumentation and steering model, called the *Mirror Object Model*.

A software performance monitor, which gathers performance data for Shared Virtual Memory (SVM) systems implemented on the Myrinet-based cluster [LJI+98].

3.5.2 Analysis and Visualization

Event Processing and Information Rendering Architecture (EPIRA) [BMR99b] is an example of software technology that can be used to build on-line performance data analysis and visualization tools for complex parallel and distributed systems.

Parvis [NA94] is a visualization tool that translates a trace file generated by the *Portable Instrumented Communication Library (PICL)* [Wor92] or its extension, *MPICL* [Wor00], into a number of graphical views, such as state diagrams, activity charts, statistics and time-line displays.

ParaGraph [Hea94, HMR98] is a graphical display system for visualizing the behaviour and performance of parallel programs on message-passing parallel computers. It takes as input execution trace-data provided by *PICL* and *MPICL* libraries.

Scalability Analyzer (SCALA) [SPF+99] is used for performance modeling and prediction of high performance computing programs. It uses a new approach in scalability analysis to provide a user with better performance analysis.

Kanoko [Osa98] is a 3-D animation tool designed for parallel-program performance-tuning. The tool maps trace data into states of a dynamic system model, simulates the system, and visualizes and sonificates results of that simulation.

Partial Order Event Tracer (POET) [Tay95b, Tay95d] allows collection and display of event data from concurrent and distributed applications. It is utilized for visualization of data gathered by one of μProfiler's metrics (see Chapter 8).

3.5.3 Comparison of Profiling Tools

Jeffrey Hollingsworth and Barton Miller devised a technique, called *true zeroing*, that enables direct quantitative comparison of metrics for parallel programs. *True zeroing* computes the improvement in an application's execution if a single function is removed. It has been used to compare the performance quality of six tools: *gprof* [GKM82], *IPS-2* [MCH+90], *Critical Path* [YM88], *Quartz NPT* [AL90], *Logical Zeroing* [MCH+90] and *Slack* [HM94]. The comparison results are described in [HM92].

Another comparison of profiling tools was performed by Shirley Browne, Jack Dongarra and Kevin London [BDL]. They compare the tools from a user perspective using evaluation criteria such as robustness, usability, scalability, portability and versatility. They examined the following tools: AIMS(Section 3.2), *VAM-*

PIR [Pal98], *Visualization Tool (VT) for the IBM SP Parallel Environment*, *Nupshot* [GL99], *Paradyn* (Section 3.3) and *Pablo* (Section 3.4).

Chapter 4

μC++ and its Tools

μProfiler is a set of metrics and display tools designed to profile programs written in μC++. Unlike many of the general-purpose tools discussed in Chapter 3, μProfiler is tightly coupled with the μC++ run-time kernel to achieve high accuracy and efficiency. Therefore, understanding the design and implementation of μProfiler requires, at least, basic knowledge of the μC++ language itself.

This chapter describes the environment in which profiling is done and some existing tools used for profiling and debugging. The descriptions are the minimum necessary to understand the profiler and its metrics, described in the remainder of the thesis. A detailed description of the μC++ language is provided in [BS99].

4.1 μC++

The μProfiler target environment is an extension of the object-oriented C++ language [Str97], called μC++. μC++ [BS99] is a concurrent, user-level thread library that can execute in uni-processor and multi-processor shared-memory environments. On uni-processor computers, the concurrency is achieved by interleaving execution (context switching) to give the appearance of parallel execution. On multi-processor computers, the concurrency is accomplished by interleaving and true parallel execution; *parallel execution* is defined as simultaneous execution of two or more threads, and *concurrency* is execution that appears to be parallel over a period of time, but which has only one thread running at time.

Besides regular C++ objects and methods, μC++ contains several entities that support concurrent execution, e.g., monitors, tasks, processors, clusters. Part of these special objects are implemented using the μC++ translator, and part using the run-time library. The translator reads μC++ code, parses μC++ primitives and transforms them into C++ statements. A standard C++ compiler generates the object code and links it to the μC++ run-time library, called the μC++ kernel. The kernel is responsible for creating, managing and destroying coroutines, monitors, tasks, clusters and processors, as well as for processor and task migration and task scheduling. The run-time kernel also activates instrumentation for μProfiler and invokes other tools supported by μC++ (described in Section 4.2).

4.1.1 Coroutine

A coroutine is an object with its own execution-state, i.e., stack, so its execution can be suspended and then resumed at the point of suspension, preserving the state of execution and local variables. Although a coroutine executes serially so there is no concurrency involved, it is a useful construct that enables solving problems associated with finite-state machines and push-down automata.

The μC++ uCoroutine class [BS99] has one distinguished member: main(), called the *coroutine main*, that can be suspended and resumed using uSuspend and uResume statements. uCoroutine can also be declared as a mutex object (see Section 4.1.2), combining properties of a coroutine and a monitor; a mutex coroutine is called a *coroutine-monitor*.

Since a coroutine is executed by a task's thread (see Section 4.1.3) and both the coroutine and the task have their own execution stacks, special care must be taken during profiling to ensure collection of correct execution-state information, i.e., for certain metrics, the profiler must access the coroutine's stack and not the task's stack when the thread is executing a coroutine.

4.1.2 Monitor

A monitor is a mutex object that manages access to shared resources. Its public methods provide mutual exclusion, a mechanism allowing uninterrupted access to a resource. For many purposes, the mutual exclusion is all a user needs, e.g., to implement an atomic counter. However, the monitor owner, i.e., an active thread (see Section 4.1.3) in the mutex object, sometimes must synchronize with tasks calling or executing within the mutex object. The synchronization is performed through internal and external scheduling.

Internal scheduling occurs when the monitor owner synchronizes with tasks blocked on monitor condition-queues. This form of synchronization is achieved using condition variables and wait and signal statements.

External scheduling is typically achieved by the monitor owner specifying which calls to mutex methods are eligible. Only a task calling an eligible mutex-method is permitted to enter the monitor when the monitor becomes inactive and a selection decision is made. A common implementation of external scheduling, also used in μC++, is with an accept statement, but other implementations are possible.

In μC++, mutual exclusion and task synchronization is also provided through low-level mechanisms such as counting semaphores, locks and barriers. These additional mechanisms are provided for efficiency or teaching purposes.

More information about the taxonomy of monitors can be found in [BFC95]. The implementation details about μC++ mutual exclusion mechanisms and synchronization methods is presented in [BS99].

4.1.3 Task

A μC++ task is a mutex object with its own thread of control and execution-state. Like a monitor, its public methods provide mutual exclusion. Tasks are executed by μC++ processors (see Section 4.1.4) and are managed by the μC++ kernel.

Tasks communicate with each other through routine call and parameter passing, when one task calls a member routine of another task. This technique is better than message passing because the parameter types can be checked statically. Whereas in message passing, the check is performed dynamically or not at all.

Since concurrent execution involves interaction among different threads of control, a profiling tool must be able to profile an application on a thread basis.

4.1.4 Processor

A μC++ processor is a virtual, software processor that executes threads. The virtual processor is scheduled for execution on a hardware processor by the underlying operation system. In the uni-processor environment, all μC++ processors are simulated inside one UNIX process. In the multi-processor environment, each μC++ processor is implemented by a separate UNIX process.

In both uni-processor and multi-processor environments, the μC++ kernel controls the scheduling of tasks on virtual processors. Thus, when the operating system assigns an execution period to a virtual processor, μC++ may provide additional time-slicing of that period to allow execution of multiple tasks.

Because a virtual processor is not bound to a hardware processor, programs can be coded for any number of virtual processors and then executed on a computer with a smaller number of hardware processors. This makes μC++ programs portable among machines with different number of processors and various architectures.

4.1.5 Cluster

A μC++ cluster is a collection of virtual processors and tasks executed by those processors. The purpose of a cluster is to control the amount of parallelism among tasks. A cluster must have at least one processor to be able to execute tasks.

Each cluster contains its own scheduler that determines the order in which tasks are executed on its processors. The default scheduler is round-robin but users can implement their own scheduling algorithms.

Users can also create additional clusters, beside the default system and user clusters created by μC++. For example, μProfiler is created on its own cluster to minimize its impact on user application in the multi-processor environment.

Tasks and processors can be migrated from cluster to cluster during execution. The most common case of task migration is moving a task to a separate I/O cluster for input and output operations.

4.2 MVD Toolkit

μC++ supports a Monitoring, Visualization and Debugging (MVD) tool set [Buh95]. *Monitoring* is the process of gathering data about a program's run-time behaviour. *Visualization* is tightly coupled with monitoring and involves displaying the data collected by monitors. *Debugging* is a process of eliminating errors in an application.

The visualization in MVD is implemented using a modified versions of X Windows [MAS, Nye93] and Motif [HF94] libraries called μX. The modifications are required to assure safe access to visualization primitives by user-level threads. The package supplies a server task, called *XtShellServer*, which runs on its own cluster and handles all events generated for the X/Motif libraries. The μX library is described in detail in [Buh95] and is used by μProfiler.

MVD contains the following tools: μProfiler, Kalli's DeBugger (KDB), Shared-Memory Application Replaying Tool (SMART), real time monitoring of program's execution using Watchers and Samplers, and a μC++ tracing system.

4.2.1 KDB

KDB [BKS96] is a symbolic debugger for concurrent programs written in μC++.

It provides standard features of debuggers, such as setting break-points, stepping through code and looking up variables. A user can also control execution of individual threads and access each task's local variables.

An already running application can be attached to KDB so that a programmer can examine problems that have already occurred.

KDB also executes a subset of the GNU DeBugger (GDB) commands, and provides limited support for debugging non-shared memory applications.

Figure 4.1 presents a KDB *main window*, a *task-execution control-window* for the task highlighted in the *main window* (Girl / 0x7086f8), and a window displaying break-points for that task. Also, the program is compiled with the -profile flag (see Section 5.1.2), so the *main window* displays the uProfilerCluster cluster and uProfiler task, which show that KDB can be used to debug not only user code but also the profiler itself.

4.2.2 SMART

The main problem in debugging a concurrent program is that its execution is non-deterministic. Therefore, some problems, such as race condition, may occur rarely making the debugging more difficult.

SMART [Sch99] is a tool that records the execution of a program into a file and then uses the file to replay the execution in a deterministic manner. This way, the same problem can be recreated many times for inspection and debugging. Right now, SMART works only for applications with deterministic input data, i.e., SMART does not record data, that run in the uni-processor environment.

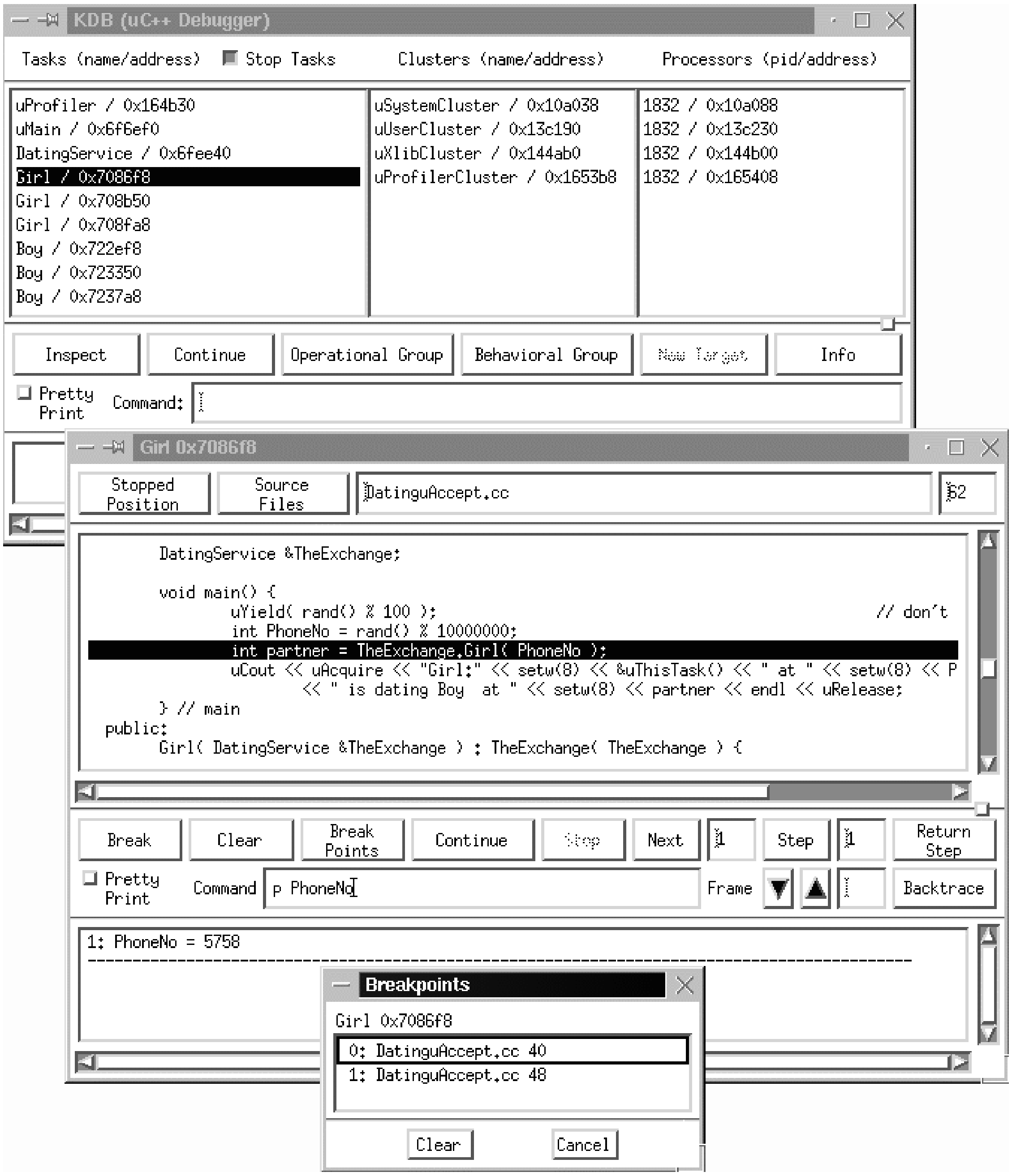


Figure 4.1: KDB Displays

4.2.3 Watchers and Samplers

Watchers and samplers [Buh95] are provided to statistically monitor the execution of a program, including the μC++ kernel, and display the information during a program's run-time.

A *watcher* is a task that manages a list of sampler objects. At specified intervals, the watcher invokes a sampler object and then the *sampler* gathers the required data, processes it, and displays it on a screen or saves it into a file. A similar monitoring technique is used for statistical profiling in μProfiler, but the data is analyzed and displayed post-mortem (see Section 5.3).

Watchers and samplers will be incorporated into μProfiler in the future, when the profiler provides run-time visualization of performance data.

4.2.4 μC++ Tracing System

The tracing system [TB96] is described in Chapter 8. It is now replaced by one of the μProfiler metrics.

Chapter 5

μProfiler

μProfiler is part of the MVD toolkit and has been developed to provide a "user-friendly" tool for profiling concurrent μC++ programs. The initial work on μProfiler was done by Robert Denda and is described in his thesis [Den97].

This chapter presents a general overview of μProfiler, covering Robert's initial work and my modifications of the profiler design and implementation.

5.1 Objectives

The main objectives of μProfiler are:

5.1.1 Profiling at Different Levels of Detail

μProfiler is able to profile UNIX processes, μC++ clusters, processors, tasks, coroutines, objects, and regular and mutex functions. It does not support profiling at the statement level because the high cost of monitoring at such a low level makes it impractical in many cases.

μProfiler performs statistical and exact monitoring, and gathers summary and tracing data.

5.1.2 Selective Profiling

Profiling is determined at two levels: at the compilation unit to ensure insertion of necessary profiling code, and within a profiled compilation unit, profiling can be dynamically enabled and disabled.

Profiling a compilation unit is specified by the `-profile` flag in a compilation command. During the compilation, necessary instrumentation is inserted into the user code and the program is linked with profiling libraries. A user controls which parts of an application are profiled by compiling only selected modules of the application with the `-profile` flag and the rest of the application without.

Within a profiled compilation unit, the profiling can be turned off and on for each task during run-time by making calls to the task's `uProfileInactivate()` and `uProfileActivate()` routines.

Note: Selective profiling requires a user to have a good understanding of the profiled application as well as the measured metrics. The user must determine whether turning the profiling off in some parts of the program affects the usefulness of the collected data. In most cases, the entire application is compiled with the `-profile` flag.

5.1.3 Extendibility

The design of μProfiler, presented in Section 5.2, allows an easy extension of the profiler for new metrics or additional data-collection hooks.

Adding a new metric into μProfiler is accomplished through inheritance. A metric monitor and analyzer are derived from abstract classes that are part of the μProfiler kernel. This mechanism ensures that the kernel structure does not need to be modified every time a new metric is built.

A user can create new metrics and attach them to μProfiler without recompiling the profiler's code. However, creating a metric requires some programming effort (see Section 5.5.4).

Implementing extra hooks is also easily achieved, since all hooks have the same interface structure and are controlled by the same mechanism in the μProfiler kernel (see Section 5.4).

5.1.4 Portability and Maintainability

Most of μProfiler code is written in μC++, which makes it easy to maintain and ensures it is up to date with new versions of the language.

μProfiler has already been ported to the SPARC architecture with SunOS and Solaris operating systems, and to architectures based on Intel's x86 processors running Linux. The long-term goal is to port the profiler to all other machines and operating systems μC++ runs on, e.g., OSF, IRIX, and AIX.

Since the instrumentation of function entry and exit (see Section 5.4.2) is implemented in machine-dependent assembler, porting μProfiler to new environments is more complicated than just recompiling the profiler's code for a new architecture. On the other hand, since only a very small portion of the profiler is written in assembly language, the porting is not an intimidating task.

5.2 Static Design

The static design of μProfiler is shown in figure 5.1. The diagram uses object-oriented analysis and design notation explained in Appendix A.

5.2.1 μProfiler Kernel

The main part of μProfiler is the *μProfiler kernel*, which consists of the following objects: uProfiler, uProfilerStartWidget, uProfilerClock, uProfileSampler, uExecutionMonitor, uProfileAnalyze, uMetricAnalyze and uSymbolTable.

uProfiler acts as an administrator for all active metrics. It registers metric monitors, accepts profiling-event calls and relays them to appropriate monitors, and invokes metric analyzers to process and display gathered data.

uProfilerClock is used in statistical profiling to allow data collecting at specified time intervals. If statistical profiling is not active, the uProfilerClock is not created.

uProfilerStartWidget is an object responsible for creating the profiler's start-up window, presented in Figure 5.2 (details in Section 5.5.4), and activating metrics chosen by a user. A user can interactively select a metric or a group of metrics to be measured from the start-up window or metrics can be activated through a *metric file*. The uProfilerStartWidget looks first for the *metric file*. If the file exists and has a correct format, uProfilerStartWidget activates metrics based on the information

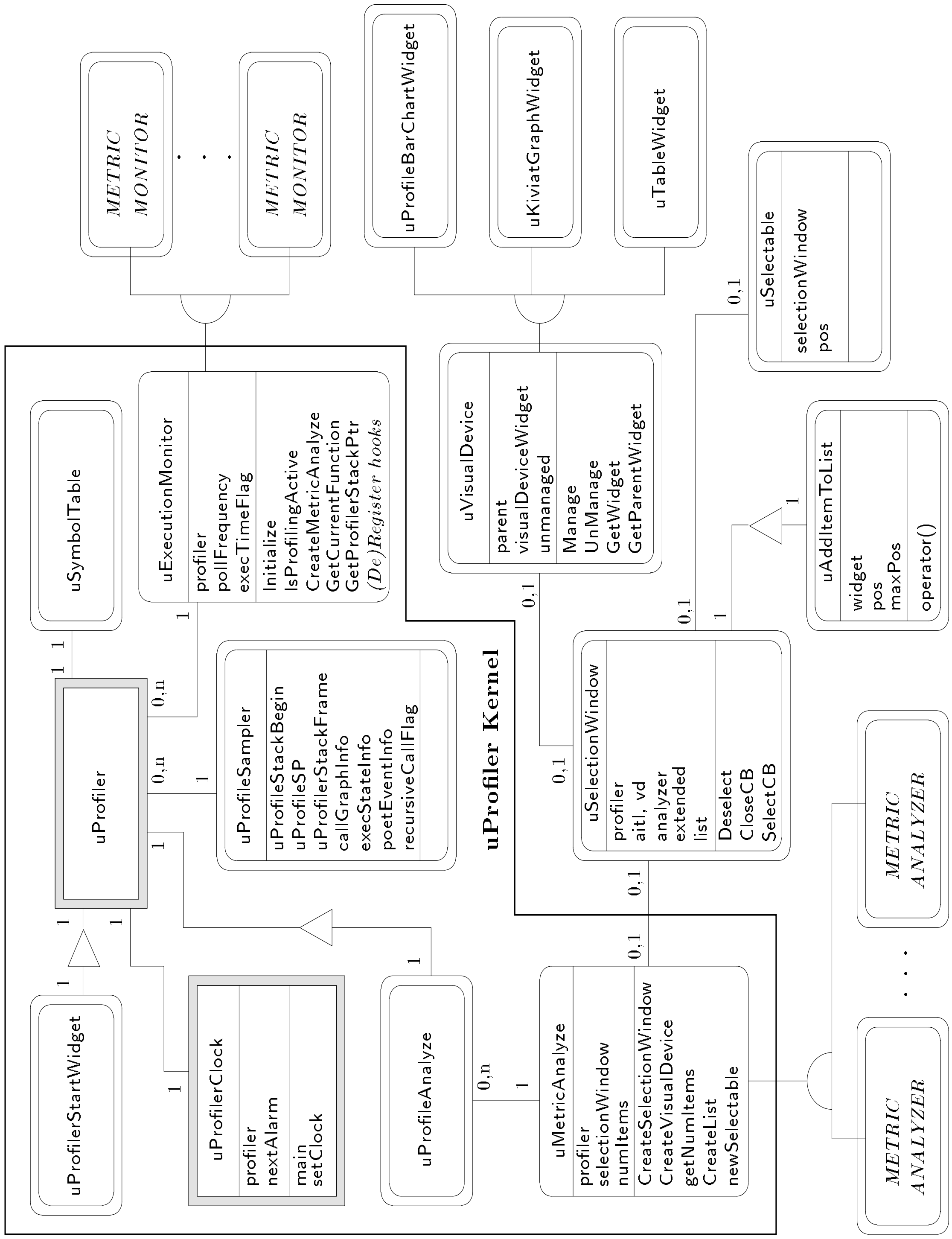


Figure 5.1: Object-Oriented Model of μProfiler

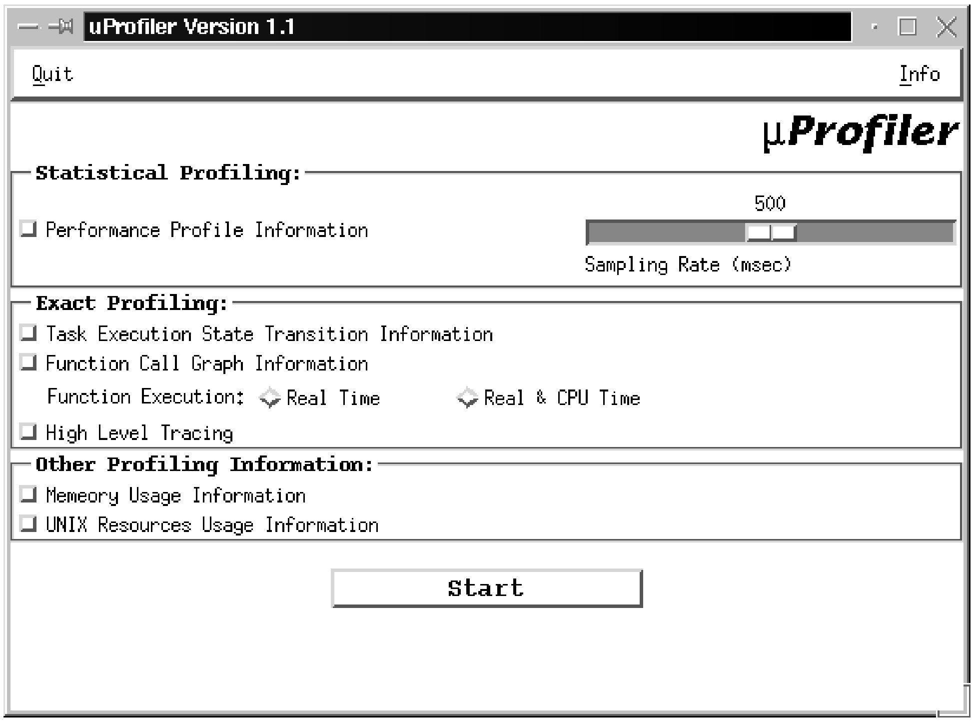


Figure 5.2: μProfiler: Start-up Window

provided in the file without creating the start-up window; if the file contains errors the application aborts. If the file does not exist, uProfilerStartWidget creates the start-up window to allow a user to make selections by pressing the button in front of a metric name and setting other parameters when required, e.g., specifying sampling rate for *Performance Profile Information*. The *metric file* is useful when a user wants to measure the same metrics multiple times, possibly on different input data, as it saves time pushing the same buttons and setting the same parameters on the start-up window each time the application is profiled.

uExecutionMonitor is an abstract class providing basic functionality for metric monitors that are derived from it. Monitors and analyzers are described in more details in Section 5.2.2.

uProfileSampler is responsible for creating and updating a *profiling stack* (see Section 5.5.3) that contains information about currently executing routines and their callers. That information is used to insert profiling instrumentation (see Section 5.4.2) and to update profiling data structures.

uProfileAnalyze is an object that invokes analyzers for all active metrics. Currently, uProfileAnalyze is only created after the profiled tasks finish their execution, i.e., the profiler performs only post-mortem analyses.

All metric analyzers must be derived from an abstract class uMetricAnalyze, which provides the interface to create and manage *selection windows*. Selection windows are used to allow hierarchical processing and displaying of gathered data. A *selection window* contains a list of selectable objects that represent the next level of detail. For example, it may contain a list of profiled tasks, then selecting a task, shows profiling information for that task.

uSymbolTable object contains a symbol table obtained through the *Binary File Descriptor (BFD) Library* [Cha91] and provides uniform access to the compiler-generated symbol table for a C++ program, encapsulating all functionality that depends on the underlying architecture and operating system.

5.2.2 μProfiler Metric

Each metric, even a built-in metric, is created as a separate entity and added to μProfiler. This design allows easy extendibility of the profiler. It also permits users to create their own metrics, using instrumentation hooks provided by the μProfiler.

Each metric consists of three parts: monitor, analyzer and visualizer. These parts reflect the stages in the profiling process. A *monitor* is responsible for monitoring program execution and gathering data during that execution. An *analyzer* processes the collected data and a *visualizer* displays the information on a screen. All parts of a metric are tightly coupled with each other. The analyzer needs to know about the data structures of the monitor and the visualizer must be aware of the format of information produced by the analyzer to be able to display it.

5.2.2.1 Monitor

All monitors are inherited from a common abstract class: uExecutionMonitor, which is a part of the μProfiler kernel (see Figure 5.1). uExecutionMonitor provides virtual member functions for all hooks inserted into the μC++ kernel and user code (see Section 5.4). A monitor derived from this base class implements only the members that activate hooks needed by its metric. The monitor's member routine Initialize(), which must be called at the end of the derived monitor's constructor, dynamically checks what virtual members are overwritten by comparing member-function pointers of the base class and the derived class, and registers all overwritten members, i.e., active hooks, with uProfiler.

On encountering an active hook, a profiled task updates the profiling data itself (*decentralized monitoring*) or it calls into the μProfiler kernel to let uProfiler update its profiling data (*centralized monitoring*). The task behaviour depends on the hook encountered and on the metric measured.

In *centralized monitoring*, uProfiler keeps a list of all active monitors registered for each hook. When a task calls into uProfiler after encountering an active hook, it blocks waiting for uProfiler to copy the data gathered by the hook into the profiler structures. After releasing the calling task, uProfiler goes through the monitor list for that hook and invokes each monitor's member function associated with that hook so that each metric can update its data structures.

The *decentralized monitoring* can be used when profiling on a thread basis. In this technique, an active monitor creates a separate profiling data structure for each profiled task, which is responsible for updating the structure during execution. However, the monitor still keeps a list of all the created structures so that the data can be accessed by an analyzer after the profiled tasks finish execution and are deleted.

5.2.2.2 Analyzer

To reduce profiling overhead, only the minimum amount of data, satisfying the metric requirements, is gathered during application execution. For example, only function addresses are stored during run-time, but they need to be mapped into function names before displaying them on a screen. That mapping is performed by an analyzer that is responsible for processing the collected data. The analyzing process consists of three steps:

1. Extract the data from the monitor's data structures.

2. Perform additional filtering of the data, if necessary. (Currently, most of the filtering is done during program monitoring to reduce the amount stored).

3. Process the data according to the metric algorithms.

A main-level analyzer is derived from the μProfiler kernel abstract class, uMetricAnalyze, and is invoked by the μProfiler kernel object, uProfileAnalyze.

5.2.2.3 Visualizer

A visualizer is responsible for displaying the information created by the analyzer. μProfiler uses Motif's widgets [HF94] to show the information on a screen. In the present implementation, μProfiler does not provide any option to store the information in a file.

There is no common class from which all the visualizers are derived. Currently, μProfiler supplies a set of general visualization routines that can be used by different metrics to display information. There are classes to draw Kiviat graphs, bar charts, tables and selection lists (see Figure 5.1). The classes are created around Motif's widgets and they enable a metric designer to display the data without the need to learn Motif syntax. The designer is free to write his own visualizing routines for complete control over the way the data is displayed.

Since displaying performance information is often done in a hierarchical manner, there can be several sets of analyzers and visualizers for a single metric.

5.3 Dynamic Design

Upon starting a profiled application, μProfiler starts first so it can determine which metric to measure, either through a *metric file* or the μProfiler start-up window. After monitors for all activated metrics are created, the profiled application is continued and its execution is monitored by the profiler. In the multi-processor environment, where μProfiler can run in parallel with the application, the impact of profiling on the application is minimized.

uProfiler and uProfilerClock are the only tasks on the profiler cluster. The reason behind not designing other objects as tasks is the fact that the cluster has only one μC++ processor. Therefore, creating more tasks would increase the tasks' scheduling and synchronization overhead for the profiling cluster. Besides, there is no need to create more tasks inside the μProfiler kernel since profiled tasks gathered the profiling data themselves in *decentralized monitoring*, reducing the amount of work done by uProfiler. Only in *centralized monitoring*, used in statistical profiling and exact profiling on levels other than a task level, does uProfiler gather the data.

In statistical profiling, the uProfilerClock task communicates with the uProfiler task to alert the uProfiler that it is time to collect samples. When the time for sampling arrives, uProfilerClock calls a uProfiler::WakeUp() routine and uProfiler gathers data for all ready sampling events. After gathering the data, uProfiler sets the alarm clock for the next sampling by calling the uProfilerClock::setClock() method (more information about μProfiler statistical sampling can be found in [Den97]). When statistical profiling is not active, the uProfilerClock task is not created at all leaving uProfiler as the only task on the profiler cluster.

5.4 Instrumentation Insertion

There are two levels of instrumentation insertion in μProfiler. The first level involves *hooks* inserted in the μC++ run-time kernel and the second is instrumentation inserted into a user application.

5.4.1 μC++ Kernel Instrumentation

The kernel hooks are permanent, i.e., they are present whether a user application is profiled or not. There are hooks for creation and deletion of clusters, processors, tasks, monitors and coroutines, for processor and task migration between clusters, for entering and exiting mutex objects, for context switches performed by the μC++ kernel, and for suspending and resuming coroutines.

```
uCluster &uBaseTask::uMigrate( uCluster &toClus ) {
    ...
    // task registered for profiling ?
    if ( uProfileActive && uProfiler::uProfiler_ RegisterTaskMigrate ) {
     ( *uProfiler::uProfiler_ RegisterTaskMigrate )( uProfiler::uProfilerInstance,
                                                      *this, fromClus, toClus );
    } // if
    ...
} // uBaseTask::uMigrate
```

Figure 5.3: μProfiler: Instrumentation Insertion of μC++ Kernel

Figure 5.3 shows an example of a kernel hook for gathering information about task migration. First, a check of a uProfileActive flag is made to determine if profiling is currently enabled for this particular task. If the task is profiled, a check of uProfiler::uProfiler_RegisterTaskMigrate is made to discover if this hook is active for the current profiling session. uProfiler::uProfiler_RegisterTaskMigrate is a function pointer that points to the uProfiler::RegisterTaskMigrate() member routine only if at least one metric that uses this hook is active (i.e., a metric's monitor implements a RegisterTaskMigrateNotify() function); otherwise, the function pointer is set to NULL. If the hook is active (not NULL), the RegisterTaskMigrate() routine is called and profiling data is gathered.

All μProfiler instrumentation, i.e., μC++ kernel hooks as well as user-code hooks, have the same interface and are activated in the same way.

5.4.2 User Code Instrumentation

The instrumentation of a user code is performed using shared trampolines. All profiled tasks and coroutines share the same profiling code but each of them has its own profiling stack (see Section 5.5.3 for profiling stack details). The trampolines are inserted statically at entry and exit points of each function in the profiled program during compilation.

μProfiler uses the support from a C++ compiler to insert the trampolines. The C++ compiler performs instrumentation insertion for *prof* and *gprof* [GKM82], profiling tools for sequential programs. The compiler inserts a call to `mcount()`, which counts and records the number of entries into a routine, as the first statement in a profiled routine. μProfiler uses the same mechanism to instrument user code. However, it modifies `mcount()` to enable instrumentation at both routine entry and exit.

Figure 5.4 shows the instrumentation insertion of a user function performed by μProfiler. The call to `mcount()` is inserted at the beginning of a user function. Inside `mcount()`, a check is performed to determine whether routine-level profiling is active. If this is not the case, control returns immediately to the profiled function; otherwise, `mcount()` gathers execution-state information about a currently executing task or coroutine. The execution-state information includes:

- current function address

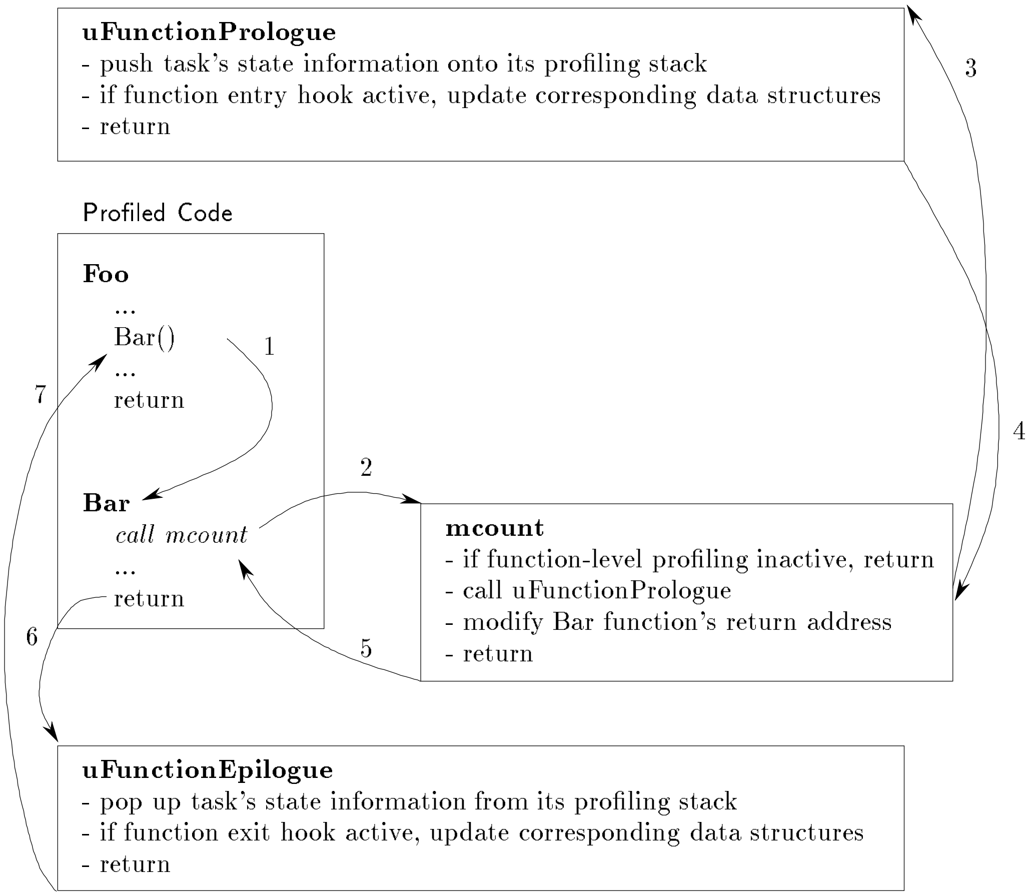


Figure 5.4: μProfiler: Instrumentation Insertion of a User Function

- start address of the parent function
- address of an instruction following the call into the current function
- object pointer

The state information is passed into uFunctionPrologue(), where it is saved on the profiling stack. If the *function-entry* hook is active, uFunctionPrologue() updates the data structures associated with this hook directly, in the *decentralized monitoring* mode, or it makes a call into μProfiler in the *centralized monitoring* mode, so uProfiler can take care of data collecting. After returning from uFunctionPrologue(), mcount() uses the address of the instruction following the call-instruction to modify the return address of the current function. On reaching its end, the current function does not return directly to its caller, but calls instead uFunctionEpilogue(), which pops the execution-state information from the profiling stack. If the *function-exit* hook is active, the corresponding data structures are updated in a similar way to the *function-entry* hook. Then, uFunctionEpilogue() returns control to the calling function.

In the *decentralized monitoring* mode when the *function-entry* and/or *function-exit* hooks are active, uFunctionPrologue() and/or uFunctionEpilogue() update the profiling data structures directly, without calling uProfiler (see Section 5.5.1), because each function-call performed by the profiling code increases the probe effect substantially, which has been verified empirically. Since, these two hooks may be processed hundreds of times during application execution, it is important to reduce the profiling overhead as much as possible. However, the *decentralized monitoring* can be used only by the built-in metrics (see Section 5.5.1), so μProfiler provides a separate set of *function-entry* and *function-exit* hooks for user-designed metrics. The user hooks can be processed only in the *centralized monitoring* mode, which is significantly more expensive.

Retrieving the execution-state information and manipulating the return address of a profiled function depend on the underlying machine architecture. Therefore, parts of mcount() must be written in assembly language.

5.5 Implementation

μProfiler is invoked by the μC++ run-time kernel before a user application starts execution, which allows for profiling the entire application including global constructors and destructors. Since the global constructors and destructors are executed by the μC++ system task, called uBootTask, this task is registered for profiling as soon as the profiler is created.

5.5.1 Monitoring

Upon reaching an active hook, a profiled task calls a uProfiler *non-mutex* member associated with that hook (see Section 5.4.1). In the *decentralized monitoring* mode, profiling data is updated by the calling task inside the *non-mutex* routine, independently of the uProfiler thread. Since each task accesses only its own data structures, there is no need for mutual exclusion and several tasks can execute the function at the same time. In the *centralized monitoring* mode, the *non-mutex* member calls a corresponding *mutex* routine. uProfiler accepts these calls in a mutually-exclusive manner, which may result in blocking of the calling task. To minimize the blocking time and to increase the concurrency, the work performed inside the *mutex* member is reduced to a minimum, i.e., the necessary information is copied into uProfiler data structures and the caller is released. Afterwards, uProfiler invokes the corresponding member routine of each active monitor registered for the hook that generated the call.

The *decentralized monitoring* is desirable because of its low profiling overhead. The profiling cost is lower than in *centralized monitoring* because a calling task does not have to synchronize, a potentially blocking operation, and communicate with another task. Nevertheless, μProfiler must support the *centralized monitoring* for two reasons. First, since user metrics are incorporated into μProfiler without recompiling the profiler's code, a user cannot modify uProfiler *non-mutex* members associated with hooks activated by a new metric to allow updating of the metric's data structures. The data structures can be updated only through the metric's monitor routines, which are executed by the uProfiler's thread. Second, the *centralized monitoring* must be used in metrics that do not profile on a task level because it provides a mechanism for mutual exclusion when several tasks update the same profiling data structures.

Note: Since μC++ and μProfiler source code is publicly available, a user can incorporate his metric into the profiler as a built-in metric (this requires modifying and recompiling the μProfiler code), which allows the metric to use *decentralized monitoring.*

5.5.2 Analysis and Visualization

Each monitor must implement a CreateMetricAnalyze() member, which creates a main-level analyzer. After the profiled application finishes execution, the uProfiler task creates a uProfileAnalyze object, which is part of the μProfiler kernel (see Section 5.2). uProfileAnalyze goes through the list of active monitors, and for every monitor on the list invokes its CreateMetricAnalyze() routine. The analyzer processes the data according to its metric algorithms and then invokes its visualizer to display the information. Depending on user interactions with the visualizer, e.g., selecting lower levels of details, additional analyzers and visualizers may be invoked.

5.5.3 Profiling Stack

Execution-state information, gathered by `mcount()` (see Section 5.4.2), needs to be stored at a memory location easily accessible to the running thread. To minimize the cost of creating and accessing the profiling stack, and to eliminate costly dynamic memory allocations, μProfiler uses the memory that is already allocated for the execution stack. The profiling stack is located at the other end of the execution stack and grows in the opposite direction. Only a small amount of information, four pointers, are added to the profiling stack on profiled function entry and removed on function exit. Therefore, in general, these two stacks should not interfere with each other. Only highly recursive function-calls can cause the stacks to intersect each other. To ensure the stacks' integrity, μProfiler checks for the profiling-stack overflow, and when it happens the application is terminated with an error message.

Since both tasks and coroutines possess their own execution stacks, the profiling stack is created for both of them.

5.5.4 Adding Metrics into μProfiler

The process of adding a metric into μProfiler consists of two steps. The first step involves offering the metric to the user in the μProfiler start-up window (see Figure 5.2) and the second, creating a metric monitor when the metric is activated by the user.

Each metric or a group of metrics is put into a metric initialization class that

```
class uExactProfiling {
    static bool ES_INFO;
    static bool CG_INFO;
    static int   CGT_INFO;
    static bool HLT_INFO;
    static bool POET;
  public:
    static void CreateWidget( Widget mainForm );
     // execution state transition metric
    static void ESToggleCB( Widget, void *, XmToggleButtonCallbackStruct *);
     // call graph and run times metric
    static void CGToggleCB( Widget, void *, XmToggleButtonCallbackStruct *);
    static void CGT_ToggleCB( Widget button, void *status,
                              XmToggleButtonCallbackStruct *call_data );
     // high-level tracing
    static void HLToggleCB( Widget, void *, XmToggleButtonCallbackStruct *);
    static  void CreateMonitor( uProfiler &profiler );
}; //uExactProfiling
```

Figure 5.5: Initialization Class for a Group of Exact Metrics

must contain two member functions: CreateWidget() and CreateMonitor(). The first function creates a Motif widget [HF94] that advertises the metric(s) and provides some way to activate it (them), e.g., push buttons, sliders, etc. The second function, as the name implies, creates monitors for the activated metrics. No objects of the initialization class are created; the class acts as a module for abstraction purposes.

Figure 5.5 shows the declaration of an initialization class for a group of metrics that perform exact profiling. The uExactProfiling::CreateWidget() method creates the *Exact Profiling* box in Figure 5.2 with push buttons for three metrics: *Task Execution State Transition Information, Function Call Graph Information* and its options, and *High Level Tracing*. The uExactProfiling class also contains Motif call-back (CB) routines, invoked when buttons are pressed, and flags that keep track of which buttons are pressed. The flags are used inside uExactProfiling::CreateMonitor() to activate the corresponding monitors.

During μProfiler start-up, the profiler looks for routines named *className*::CreateWidget and *className*::CreateMonitor in the symbol table and adds these function addresses into a metric startup list, i.e., a simple form of dynamic linking. These addresses are then converted into function calls by the uProfilerStartWidget object to create the μProfiler start-up window and invoke monitors for metrics selected by a user.

5.6 Limitations

The current implementation of μProfiler has some limitations due to other software it uses:

- **Hard-Coded File Name Access**

  The profiler needs to know the name of the executable to be profiled. Due to the problem of accessing command line arguments during the μProfiler start-up process, that name is supplied to the profiler during the compilation stage. As a result, the executable name cannot be changed after compilation. Changing the name of the executable requires the program to be recompiled.

- **Executable Size**

  Integrating the profiler into the user application increases the size of the instrumented executable considerably due to the size of X-Windows and Motif.

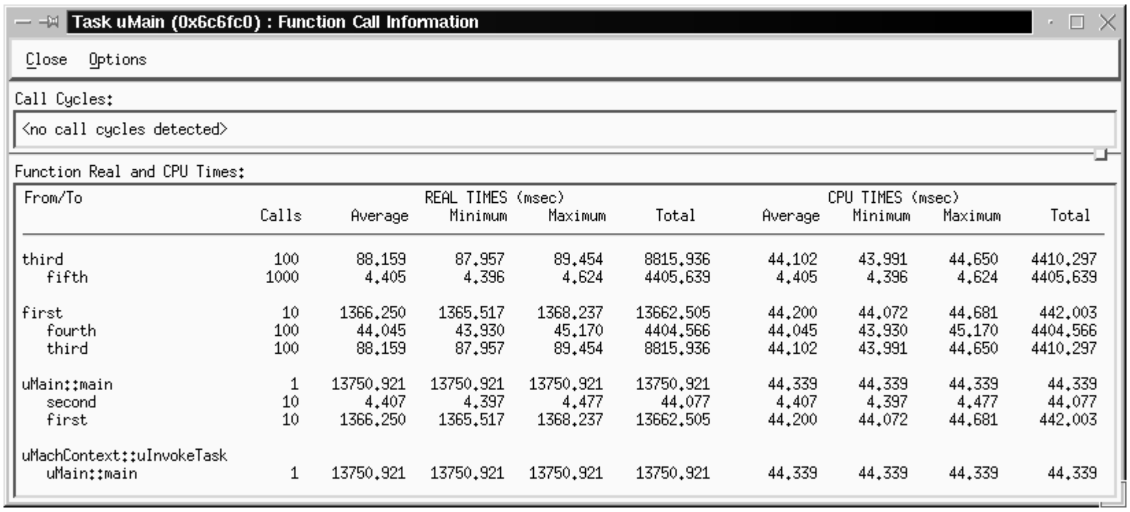
Chapter 6

Function Call Graph and Run Time Metric

This chapter discusses the μProfiler's built-in *Function Call Graph and Run Time* (CG&RT) metric.

CG&RT is based on the profiling tool called *gprof* [GKM82], which generates an execution profile of a sequential program. *gprof* records the number of function calls, the time spent in the function itself (called *exclusive time* or *CPU time*), and the time spent in the function and its descendants (*inclusive time* or *real time*). It also shows function call-cycles with a listing of the cycle members. The CG&RT metric provides the same information as *gprof*, plus the minimum and maximum real and CPU times spent in each function, a dynamic function call-graph for each task, and the source-code file and line information for presented routines.

The main difference between *gprof* and CG&RT lies in a way they gather the timing data. *gprof* uses statistical sampling of the program counter on each clock



From/To	Calls	REAL TIMES (msec) Average	Minimum	Maximum	Total	CPU TIMES (msec) Average	Minimum	Maximum	Total
third	100	88.159	87.957	89.454	8815.936	44.102	43.991	44.650	4410.297
fifth	1000	4.405	4.396	4.624	4405.639	4.405	4.396	4.624	4405.639
first	10	1366.250	1365.517	1368.237	13662.505	44.200	44.072	44.681	442.003
fourth	100	44.045	43.930	45.170	4404.566	44.045	43.930	45.170	4404.566
third	100	88.159	87.957	89.454	8815.936	44.102	43.991	44.650	4410.297
uMain::main	1	13750.921	13750.921	13750.921	13750.921	44.339	44.339	44.339	44.339
second	10	4.407	4.397	4.477	44.077	4.407	4.397	4.477	44.077
first	10	1366.250	1365.517	1368.237	13662.505	44.200	44.072	44.681	442.003
uMachContext::uInvokeTask									
uMain::main	1	13750.921	13750.921	13750.921	13750.921	44.339	44.339	44.339	44.339

Figure 6.1: CG&RT Metric: Main Display

tick, and derives the execution times from the distribution of the samples. CG&RT gathers the data through exact profiling, i.e., it actually measures the time spent in each function. Therefore, CG&RT is more accurate but has a higher overhead than *gprof*.

6.1 Design

Figure 6.1 displays information gathered by the CG&RT metric for a uMain task of the test program used in Section 6.4. For each routine invoked by the task, CG&RT reports the name of the called routine (From/To column), the number of calls made to it (Calls column), the Average, Minimum and Maximum real and CPU time of a single call, and the Total real and CPU time for all calls. The information is presented in a single-level, call-graph format that shows a relationship between a routine and its callees. In the first column, under each From routine, there is

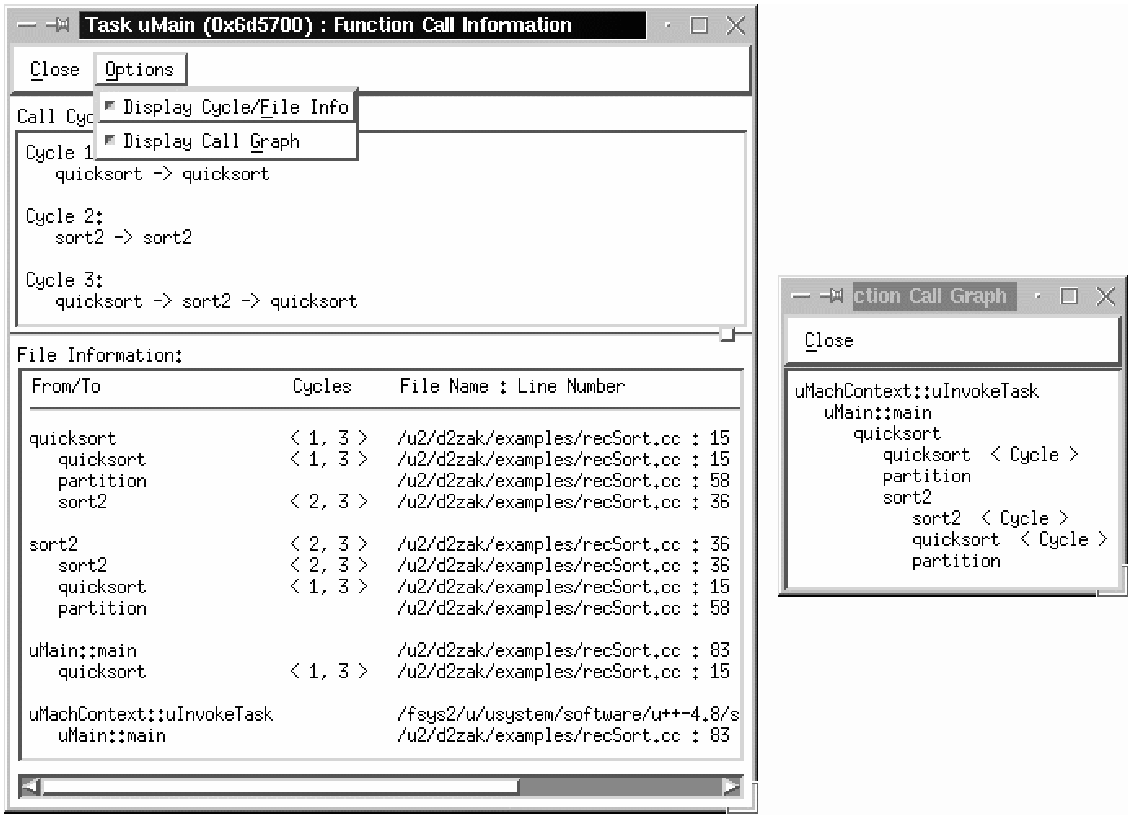


Figure 6.2: CG&RT Metric: Options Display

an indented list of its direct descendants, i.e., To routines (callees). For each To routine, the columns following its name represent the number of calls and the time spent in the routine when called only from the above parent. On the other hand, the columns associated with the From function show summary information for all the calls to that function regardless their origin. The call-graph format allows a user to notice variations in function behaviour when called from different parents. This feature is not supported by *gprof*, which assumes that all calls to a specific routine take the same amount of time to execute.

Figure 6.2 demonstrates additional features of the CG&RT metric, using information collected for a uMain task of a program containing recursive function calls. A function is *recursive* if it calls itself either directly or indirectly, i.e., creates a call-cycle. CG&RT is able to detect both the direct (Cycle 1 and Cycle 2 in the example) and indirect recursion (Cycle 3 in the example).

Another feature of CG&RTthat is not available in *gprof* is finding file and line-number information for all routines invoked by a task. Pressing the Display Cycle/File Information button in the Options menu (Figure 6.2) replaces the call and timing data (Figure 6.1) with information about cycles to which a routine belongs, the location of the routine's source code (File Name), and the line number within the file that marks the beginning of the routine.

The Display Call Graph option creates a dynamic call-graph for a task in a separate window (Figure 6.2). In both the CG&RT main window and the call-graph display, the callees of each routine are listed in an arbitrary order that does not necessarily corresponds to the order in which the routines are called by the parent function.

To reduce the execution overhead, CG&RT gives a user some flexibility in deciding the amount of data to be collected by providing the user with three options (see μProfiler start-up window in Figure 5.2):

1. *Function Call Graph Information*

   reports the number of function calls only (the first two columns in Figure 6.1)

2. *Function Execution: Real Time*

   reports the number of function calls and real time information (the first six columns in Figure 6.1)

3. *Function Execution: Real & CPU Time*

   reports the number of function calls, real time and CPU time information (all of Figure 6.1)

Since the options are really sub-options of one another, they are mutually exclusive, i.e., only one option can be active at any given time.

The design of the CG&RT metric is presented in Figure 6.3. The notation used in this object-oriented analysis model is explained in Appendix A.

As can be seen from the diagram, CG&RT follows the metric-design rules outlined in Section 5.2.2. It consists of a monitor and a main-level analyzer derived from μProfiler-kernel abstract-classes, uExecutionMonitor and uMetricAnalyze respectively. And because the metric uses the decentralized monitoring mode, uProfileSampler is responsible for updating profiling data structures for the calling task.

The data structures and their methods are encapsulated inside a uCallGraphInfo object. To maximize the speed of accessing and querying the data, a hash table in the form of uHashTable object is used (more information about the hash table and hashing technique is provided in Section 6.3.1).

Since all three options require different hooks to be activated, there are separate monitors for each of them: uCGOnlyMonitor is created for the first option, uCGIMonitor for the second and uCGEMonitor for the last one. All these monitors are derived from the common abstract class uCGMonitor that contains attributes and member routines that are generic to all three options. For example, all of them use the same structures for storing data during execution, and the same analyzers

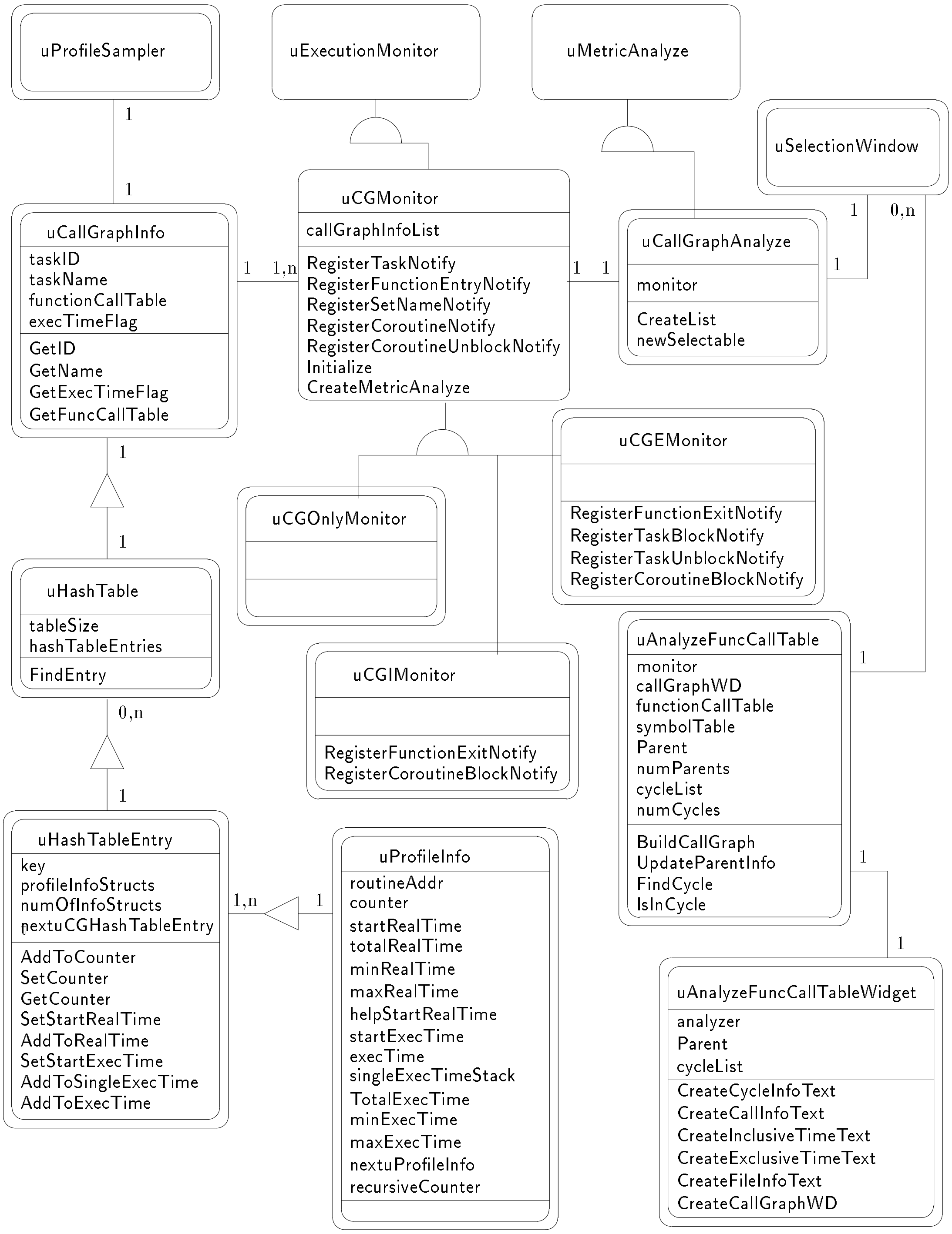


Figure 6.3: Object-Oriented Model of CG&RT Metric

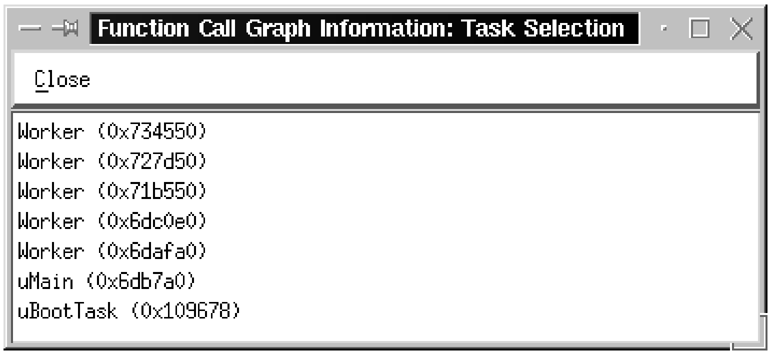


Figure 6.4: CG&RT Metric: Task Selection Display

and visualizers for processing and displaying the data. The analyzers and visualizers determine what option is active and what data has been collected by checking the uExecutionMonitor::execTimeFlag attribute (see Figure 5.1), set by the metric monitor, using information provided by the uExactProfiling::CGT_INFO flag from the metric initialization class (see Section 5.5.4).

Since the data is gathered and analyzed on a thread basis, the CG&RT metric first displays a list of profiled tasks (see Figure 6.4). The uCallGraphAnalyze object assembles the list of task names, and the common visualization class uSelectionWindow (see Section 5.2.2.3) provides the widget to display and manage the list. When a user chooses one of the listed tasks, by clicking on it, another analyzer, uAnalyzeFuncCallTable, and a visualizer, uAnalyzeFuncCallTableWD, are invoked to process and display information for the selected task. The uAnalyzeFuncCallTable analyzer finds the function call-cycles, calculates the timing information, assembles file and line-number information, and creates a dynamic call-graph when a user activates Display Call Graph in the Options menu. The uAnalyzeFuncCallTableWD creates the displays in Figures 6.1 and 6.2.

6.2 Instrumentation Insertion

The CG&RT metric's three options each requires a different instrumentation to be activated. Figure 6.5 shows the instrumentation for each option.

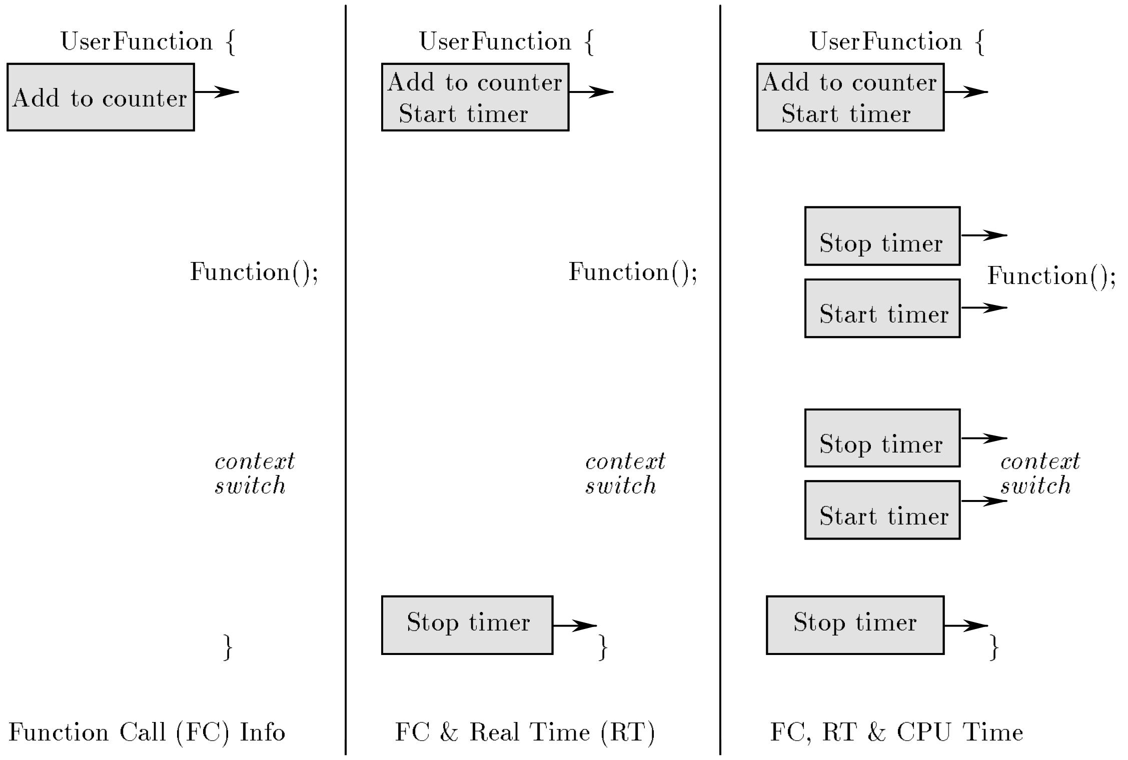


Figure 6.5: Instrumentation Insertion of CG&RT Metric

To find out the number of function calls (first option), it is enough to activate the *function-entry* hook only; instrumentation is there similar to the mcount() function used by *gprof*. For information about *real time* (second option), i.e., the clock time measured between entering a function and returning to the caller, the metric needs to get a time-stamp not only on function entry but also on function exit. By subtracting the entry-time from the exit-time, the metric calculates the time spent in the function and its descendants. The last option is the most expensive to profile. *CPU time* records the time spent in a function alone excluding its callees. Therefore, the timer must be stopped every time execution leaves the function, e.g., when the function blocks explicitly, e.g., a wait statement, or implicitly, e.g., a time slice, or when it calls another function. To gather all this data, the last option needs active instrumentation not only inside user code, i.e., function entry and exit points, but also inside the μC++ kernel context-switch routines to stop the timer when a thread blocks and resume the timer when it starts running again.

Besides the instrumentation shown in Figure 6.5, the metric activates *register-task* and *register-coroutine* hooks, inside a task's and a coroutine's constructor, which create a uProfileSampler object for each task and coroutine. The uProfile-Sampler is responsible for creating and updating the profiling stack, and collecting data during execution.

The CG&RT metric also records the change of a task's name performed by a call to the task's uSetName() routine.

The member functions that activate hooks common to all three options, such as RegisterTaskNotify(), RegisterCoroutineNotify(), RegisterSetNameNotify(), Register-FunctionEntryNotify() and RegisterCoroutineUnblockNotify(), belong to the abstract class uCGMonitor. The derived monitors implement only member routines needed to activate hooks specific to their option (see Figure 6.3).

6.3 Implementation

This sections discusses several implementation problems and solutions.

6.3.1 Hash Table

CG&RT uses a hash table for each task to allow quick access to profiling data during execution. Two designs were considered. The first one is a hash table with an entry for each executed routine. A drawback of this design is that there is no information about a routine's caller. For example, function A may use 20 milliseconds of CPU time every time it is called by function B but only 4 milliseconds when called from function C. Having that kind of information may be important to a program analyst but impossible to provide in this model. The second design allows keeping track of the caller-callee relationship. In this design, the hash table has entries only for non-leaf routines, i.e., routines that have at least one callee, and each entry keeps track of all function-calls performed from that non-leaf routine. When updating a routine's information, its caller's address is used as the hash key.

CG&RT utilizes the second design which, besides showing variations in function behaviour when called from different parents, also enables the creation of a dynamic call-graph. The implemented hash table is encapsulated inside a uHashTable object (refer to Figure 6.3) that contains an array of pointers to uHashTableEntry objects. Each uHashTableEntry consists of a key, i.e., function address, and a separate uProfileInfo structure for each callee of the key function. If a key has more than one callee, the resulting uProfileInfo structures are chained together (see Figure 6.6). A uHashTableEntry is created only for functions that have callees; a leaf function possesses only a uProfileInfo entry associated with the uHashTableEntry of its caller. If a routine is called from two different parents, it has two separate uProfileInfo entries, each linked to the respective parent's uHashTableEntry.

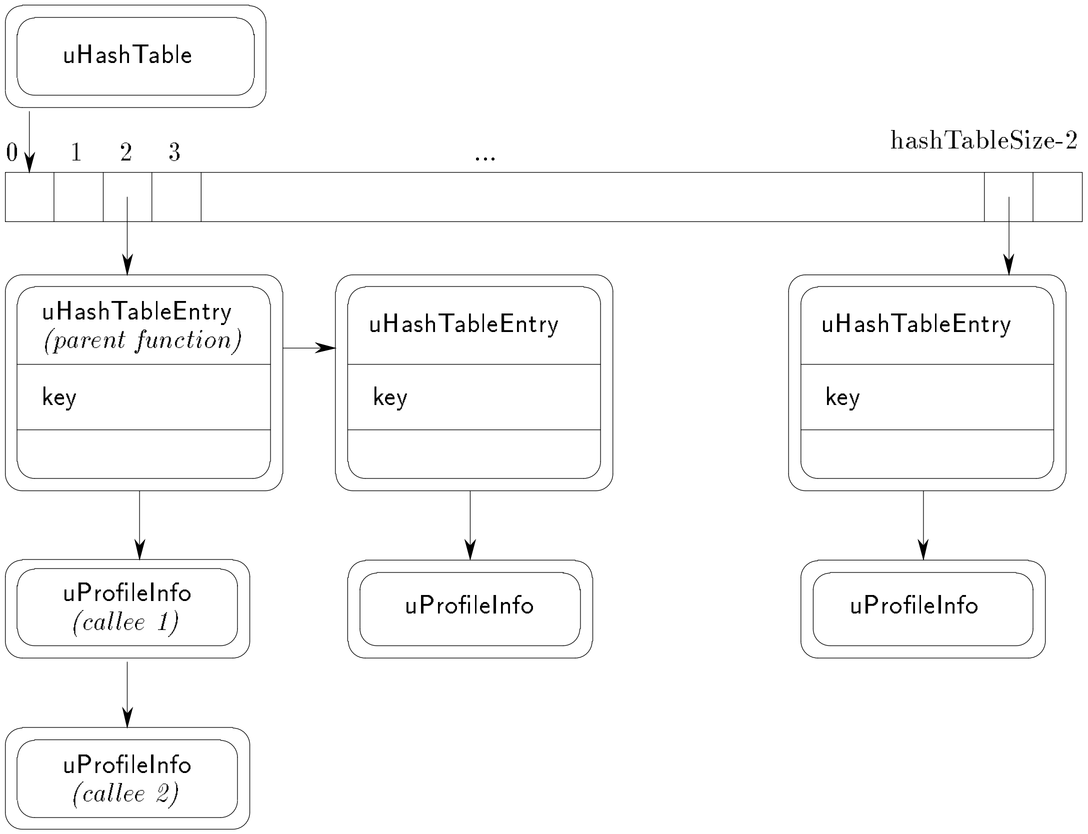


Figure 6.6: Hash Table

The hash table uses the standard hashing formula:

$$HashFunction(\mathsf{key}) = \mathsf{key} \bmod \mathsf{hashTableSize}$$

Where key is a function address, which is divisible by the address alignment of the underlying computer architecture. For example, on SPARC machines, the adresses are divisible by four. Therefore, to provide better utilization of the hash table on these machines, the key value is divided by four.

If more than one key hashes to the same array entry (hashing collision), the uHashTableEntry objects are chained together (see Figure 6.6). Since the key is stored as a part of the uHashTableEntry the chained objects are easily distinguished.

The size of the hash table is equal to the size of the μProfiler symbol table, read from the executable, that contains information about all user and system functions in a profiled application (to decrease the number of collisions, the size is rounded to a prime number). Choosing a greater size for the hash table would result in a sparse distribution of entries, because it is impossible to have more entries in the hash table than there are in the symbol table. Since many routines in a program, especially system routines, are never called, the size of the hash table should produce few collisions. In the development environment, where the μC++ kernel and the X-Window library are compiled with the -g flag for debugging, the symbol table contains between 3,500 to 4,500 entries. In the user environment, the size of the symbol table should normally be less than 1,000 entries. For simple test programs, 10 to 20 entries of the hash table are occupied.

6.3.2 Monitoring Function Calls

Figure 6.7 shows the implementation for gathering data for the CG&RT metric. It shows two cases the metric needs to deal with. The general case covers the situation when a function invokes another one without creating a call-cycle. The second case deals with recursive function invocation. Operations common to both cases are shown in regular font and the statements in bold font represent calculations specific to each case.

To keep track of the number of function calls, a profiled function's counter variable is incremented on each entry. To get the *real time* information, CG&RT generates time-stamps on function entry and exit and then subtracts the values.

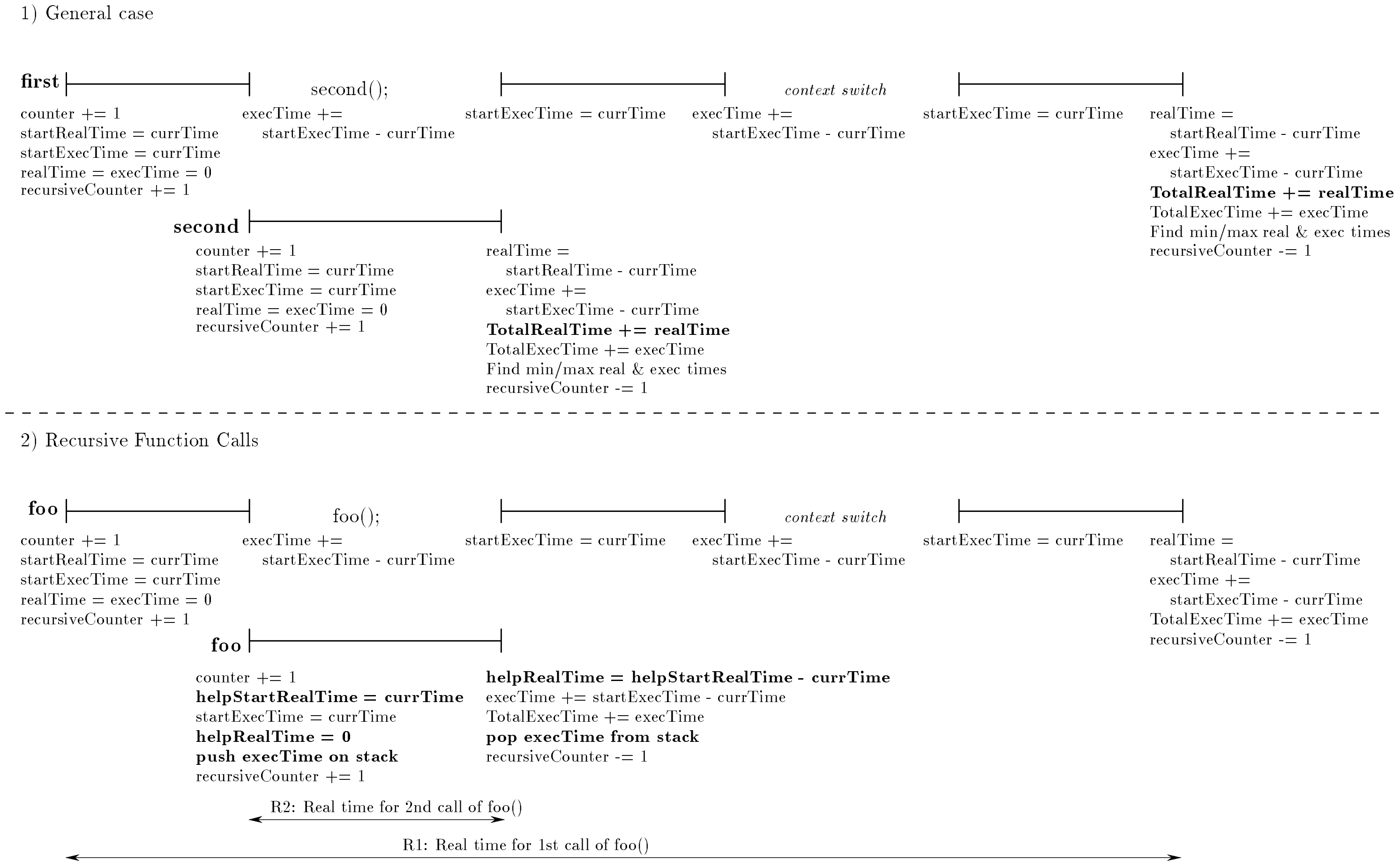


Figure 6.7: Implementation of CG&RT Metric

The greatest amount of work is done to record CPU time. The execTime variable of a current function must be updated when the function calls another routine and when a task executing the function blocks or is time-sliced.

Updating the data structures becomes more complicated when a profiled application contains recursive function calls. The recursion problems are explained using, for simplicity, a function calling itself but the solution applies to more elaborate call-cycles, such as Cycle 3 in Figure 6.2.

The first problem with recursive calls involves reporting total *real time* for a function. In the general case, the total time is calculated by summing *real time* values for each call to the function. When using the same principle in the recursion example in Figure 6.7, the total *real time* for function foo() would be calculated in the following manner:

$$totalRealTime(foo()) = R2 + R1$$

Since the time reported by $R1$ also contains the $R2$ time, it may lead to a situation where the total *real time* for foo() is larger than the execution time of the whole application. To avoid that absurd situation, CG&RT sets the total *real time* to the *real time* of the first recursive call; in the example:

$$totalRealTime(foo()) = R1$$

If a call-cycle is invoked more than once by the same parent, then the total time is a sum of the total *real times* of each cycle.

The maximum *real time* for a recursive function is equal to the longest total *real time* of a single cycle, and the minimum *real time* is equal to the shortest *real time* spent in the most nested call. Unfortunately, finding these values introduces another problem. In the general case, the startRealTime variable is set to the clock's current value on function entry, then, when exiting the function, it is subtracted from the exit time to find the *real time* of a single call. In the case of recursive calls, the startRealTime variable set in one call to a function gets overridden when processing the next call into the same function, i.e., the second call occurs before the first one is completed. One solution to this problem is to push the value of startRealTime of a previous call on a stack when entering a function the second time, and then pop it from the stack back into the variable on function exit. Pushing and popping values on a stack requires creating and deleting objects dynamically from the heap memory, which is a shared resource using locking to prevent access contention. As a result, updating a stack would increase the profiling overhead considerably. To avoid that extra cost, another approach was devised. startRealTime is set only on the first recursive call into a function; there is a recursiveCounter, incremented on function entry and decremented on exit, that is checked to determine the first call. On all subsequent calls, helpStartRealTime is set and is used to find the minimum *real time*.

The last problem with recursion involves keeping track of *CPU time* for a single function call. In the general case, when entering a new function, CG&RT stops the CPU timer of the calling function ($execTime \mathrel{+}= startExecTime - currTime$) and then the callee proceeds with updating its own execTime variable. However, in the case of recursive calls, the caller and callee share the same entry in the hash table. Therefore, by setting up its own execTime variable, the callee would modify the execTime of its caller. To prevent this situation, the execTime of a caller is pushed on a stack before entering the callee. There is no way to avoid creating the stack in this case; however, since it is expensive and is needed only for recursive calls, CG&RT uses the recursiveCounter to determine whether it is dealing with recursion and creates the stack only when necessary. As a result, gathering *CPU time* information for an application that contains call-cycles is more expensive than for one without recursive calls.

6.3.3 Monitoring Coroutines

The implementation of the CG&RT metric's data gathering for a coroutine is shown in Figure 6.8. The bold font is used to represent calculations specific to a coroutine.

One problem with a coroutine is that it may end its execution without reaching the end of the *coroutine main*, i.e., a suspended coroutine may be explicitly deleted by its creator, and as a result, not reach the *function-exit* hook to gather the required profiling data. Another problem is keeping track of the number of calls into a *coroutine main*. While the main() routine can be called in the regular way, it is also started and suspended through uResume and uSuspend statements that can be invoked by various coroutine member functions.

To solve these problems, each block of code between resuming and suspending of the *coroutine main* is treated as a single function call to main(). Therefore, the counter variable is incremented not only when starting main(), i.e., on the first uResume statement, but each subsequent time main() is resumed. Thus, suspending the execution of main() is treated as encountering the *function-exit* hook, and resuming it is equivalent to processing the *function-entry* hook. However, since the

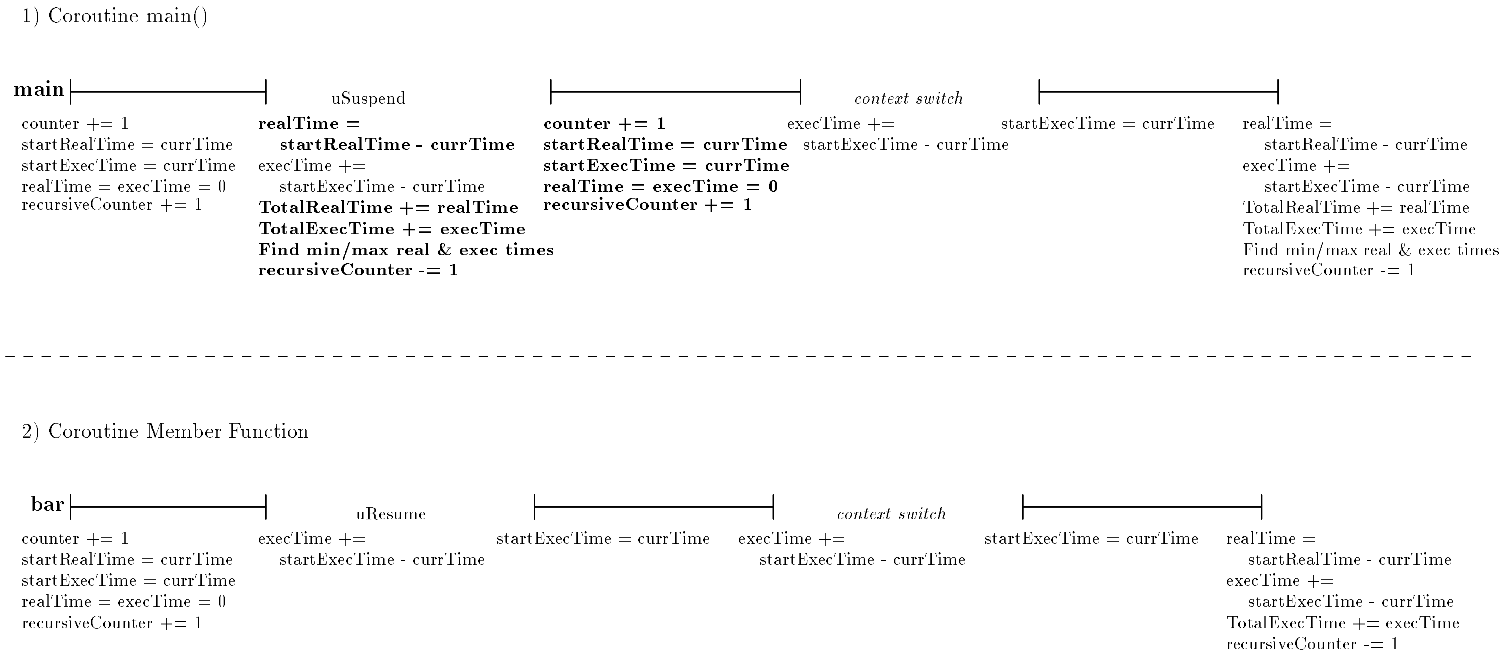


Figure 6.8: Implementation of CG&RT Metric for Coroutine

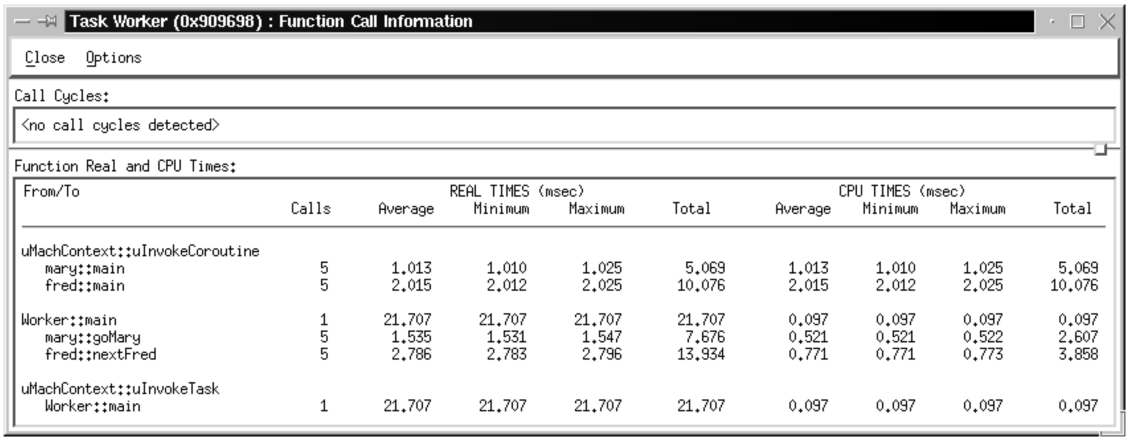


Figure 6.9: CG&RT Metric: Main Display for Task Accessing Coroutine

suspending and resuming of a coroutine is done by the μC++ kernel, the kernel's hooks are responsible for collecting the profiling data. The RegisterCoroutineBlockNotify() and RegisterCoroutineUnblockNotify() members of the CG&RT monitors activate these hooks (see Figure 6.3).

As with other functions in a profiled application, CG&RT stores the timing information for a coroutine with the task whose thread accesses the coroutine. Since a *coroutine main* is usually not called in a regular way, it is not shown as invoked by any function, but it has a separate entry in the main display of the metric. Figure 6.9 shows the display for a Worker task from the test program used in Section 6.4.1. The task accesses two coroutines: mary and fred, and the main() functions for both of them are shown under the μC++ kernel's routine uMachContext::uInvokeCoroutine(). The columns associated with the mary::main and fred::main show summary information for all "calls" made by the task into the coroutine. Also in the dynamic call-graph, created by the CG&RT metric, a *coroutine main* is displayed separately, resulting in a graph with two roots (see Figure 6.10).

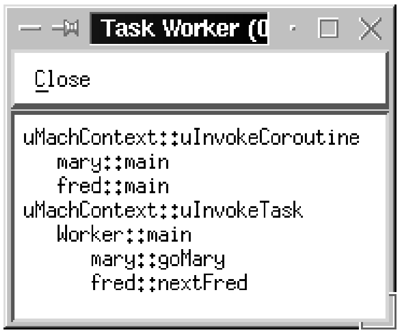


Figure 6.10: CG&RT: Call Graph Display for Task Accessing Coroutine

All other coroutine member functions, besides `main()`, are entered and exited in a regular way. Therefore, there is nothing special in processing their entry and exit hooks (see second case in Figure 6.8). Suspending the execution of the member routine, i.e., `uResume` statement, is equivalent to calling another function from the routine (which affects only the *CPU time*) and is processed by the μC++ kernel hook.

6.4 Validation

It is important to validate the CG&RT metric to verify that it provides correct information. Since the metric gathers the same information as the *gprof* tool, both should report similar results for the same program. There can be some differences because *gprof* is a statistical tool and CG&RT performs exact profiling. *gprof* provides accurate information for routines whose run-time is considerably bigger than the sampling period [?]. Therefore, the *gprof* and CG&RT results should be comparable for a reasonably large test.

Since *gprof* works only with sequential programs, a sequential C program (source code in Appendix B.1.1) is used for testing, with an adaptation for μC++ that requires the main() routine to be named uMain::main(), which creates a task to execute the routine. Since the program runs sufficiently long, the cost of the the additional task creation is insignificant. The work performed by each function is simulated by using spinning loops. Each function runs at least several milliseconds to allow *gprof* to gather timing information. Since *gprof* is a statistical tool, it usually does not report times for short-lived functions. Also in very short-lived functions, the μProfiler overhead may be the dominant factor in the results reported by CG&RT.

The C and μC++ versions of the program were run ten times each and average times were calculated for each function. The results are presented in Table 6.1.

Function Name	μProfiler			gprof		
	No. of Calls	Real T. (msec/call)	CPU T.	No. of Calls	Real T. (msec/call)	CPU T.
(uMain::)main	1	13,793.55	44.31	1	13,778.00	43.00
first	10	1,370.50	44.30	10	1,369.40	46.40
second	10	4.43	4.43	10	4.10	4.10
third	100	88.45	44.23	100	88.37	44.04
fourth	100	44.19	44.19	100	43.93	43.93
fifth	1,000	4.42	4.42	1,000	4.43	4.43

Table 6.1: CG&RT: Comparison of Results between μProfiler and gprof

As can be observed from the table, the results given by μProfiler and *gprof* are very similar. Most of the times reported by CG&RT are slightly higher than those reported by *gprof*. This increase is likely caused by μProfiler's higher overhead generated by calls to a timer routine to get the time of each function entry and exit.

6.4.1 Thread and Profiling Stack Testing

A test to verify that the CG&RT metric correctly handles multiple threads and profiling stacks associated with tasks and coroutines was constructed. The test is designed for two reasons. First, since μProfiler is intended for profiling concurrent applications, it should be tested with a program that creates several threads of control. Second, because a coroutine is executed by a task's thread and both of them have profiling stacks, μProfiler must ensure that the correct stack is accessed when gathering metric data.

The program used for testing (source code in Appendix B.1.2) creates ten Worker tasks and each Worker accesses two mutex coroutines: mary and fred. The main CG&RT display for a single Worker task is shown in Figure 6.9 and the dynamic call-graph for that task is presented in Figure 6.10.

The loops inside each coroutine member function, including main(), are calibrated by a separate program so that the *CPU time* spent in each of these functions can be estimated. The calibration program, which reports the time spent in a spinning loop, is used to adjust the number of loop iterations to obtain the desired delay.

The expected and the actual results produced by CG&RT are displayed in Table 6.2. The actual results are averages obtained from all ten Worker tasks of a single program run.

Function Name	No. of Calls	CPU Time (msec/call)	
		Expected	CG&RT
fred::main	5	2.000	2.026
fred::nextFred	5	0.750	0.776
mary::main	5	1.000	1.032
mary::goMary	5	0.500	0.526

Table 6.2: CG&RT: Test Results for Program with a Coroutine

The results are similar, indicating that the CG&RT metric correctly monitors programs with multiple threads and properly handles the profiling stack for both tasks and coroutines.

Chapter 7

Execution States Transition Metric

This chapter discusses the μProfiler's built-in *Execution States Transition* (EST) metric.

There are five execution states a thread can be in. In μC++, the states are:

- start: task created but has not started its execution yet
- ready: task able to execute but not scheduled for execution
- running: task executed by a μC++ processor
- blocked: task waiting for some event to happen
- terminate: task finished its execution but has not been deleted yet

EST collects state-data through tracing, i.e., it records each state a task enters and the duration of the state. Knowing the number of state transitions a thread makes during its life-time and the time spent in each state allows a user to observe thread's blocking patterns as well as evaluate a thread's behaviour in relation with other threads, e.g., check when several tasks run at the same time in a multi-processor environment or verify expected execution patterns in a uni-processor environment.

7.1 Design

The design of the EST metric is presented in Figure 7.1. The notation used in this object-oriented analysis model is explained in Appendix A.

Since EST utilizes decentralized monitoring (see Section 5.2.2), a uProfileSampler, one for each profiled task, is responsible for recording trace data, i.e., the time of the state change and the address of the function the thread is in when the change occurs. The data is stored into a uExecStateInfoEntry structure, which is a part of an uExecStateInfo object. Detailed information about the metric's data structures is provided in Section 7.3.

uESMonitor, derived from the μProfiler kernel's abstract-class: uExecutionMonitor (see Figure 5.1), activates the necessary hooks, creates the uExecStateInfo object for each task, and keeps a list of the created data structures to allow a metric analyzer to access the data after the end of program execution.

EST has only one analyzer, uExecStateAnalyze, derived from the μProfiler kernel's abstract-class, uMetricAnalyze. The analyzer calculates the duration of each state by subtracting the start time of a given state from the start time of the following state. It also finds the minimum and maximum state duration for each task,

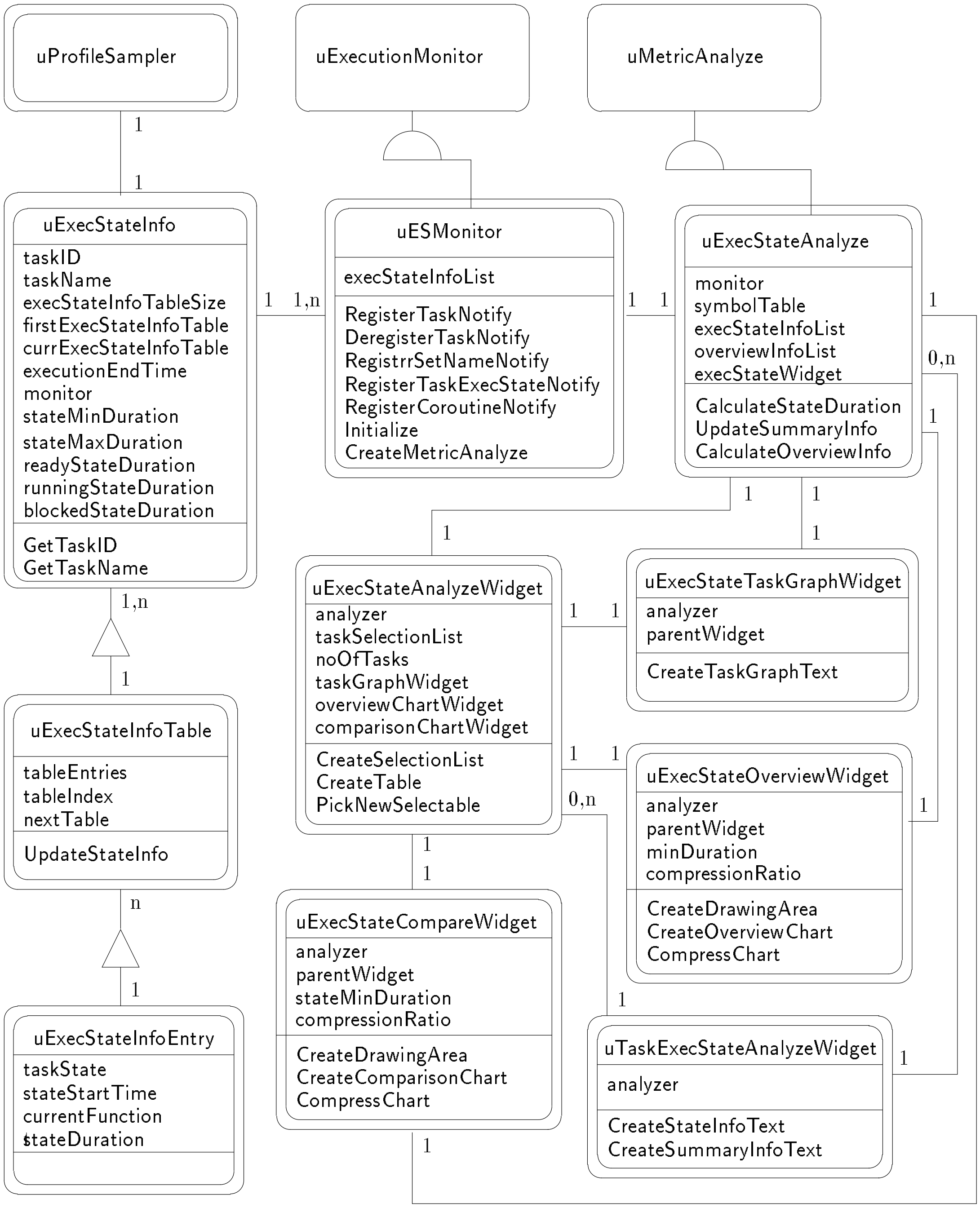
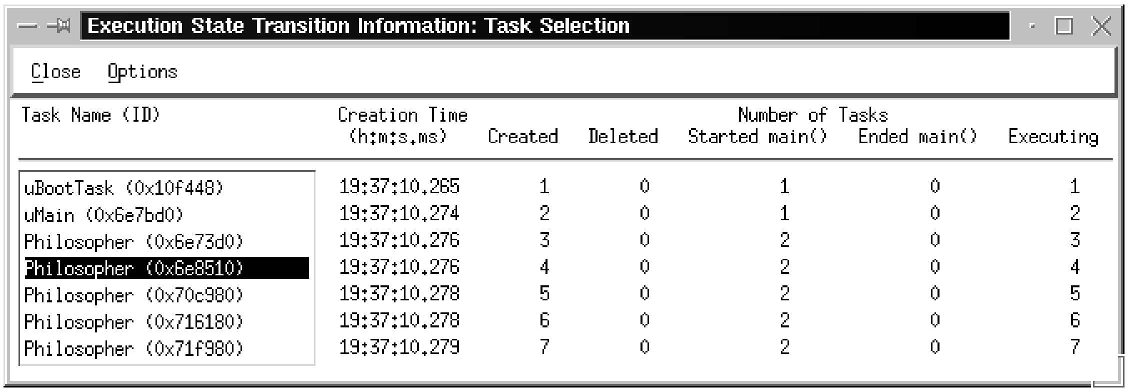


Figure 7.1: Object-Oriented Model of EST Metric



Task Name (ID)	Creation Time (h:m:s.ms)	Created	Deleted	Number of Tasks Started main()	Ended main()	Executing
uBootTask (0x10f448)	19:37:10.265	1	0	1	0	1
uMain (0x6e7bd0)	19:37:10.274	2	0	1	0	2
Philosopher (0x6e73d0)	19:37:10.276	3	0	2	0	3
Philosopher (0x6e8510)	19:37:10.276	4	0	2	0	4
Philosopher (0x70c980)	19:37:10.278	5	0	2	0	5
Philosopher (0x716180)	19:37:10.278	6	0	2	0	6
Philosopher (0x71f980)	19:37:10.279	7	0	2	0	7

Figure 7.2: EST Metric: Summary Information Table Display

the total time a task spends in ready, running and blocked states (a task is only once in the start and terminate states), and the real time for the task's main(), i.e., the duration between the first and the last running state, inclusive. uExecStateAnalyze also maps function addresses, recorded during execution, into function names.

EST has five visualizers: uExecStateAnalyzeWidget, uExecStateTaskGraphWidget, uExecStateOverviewWidget, uExecStateCompareWidget and uTaskExecStateAnalyzeWidget. All of them use information provided by the uExecStateAnalyze object. The uExecStateAnalyzeWidget is invoked by the analyzer and the other visualizers are created when a user makes a choice from the Options menu or selects a task from the list (see Figure 7.3).

uExecStateAnalyzeWidget displays a table of summary information for all profiled tasks (see Figure 7.2). The table was not a part of the original design, but has been added on users' request. Its first two columns contain a name and ID of a task, and its creation time. The next four show the total number of tasks: created, deleted, that started their main() and ended their main() from the beginning of program execution up to the specified creation time. The last column (Executing) is derived

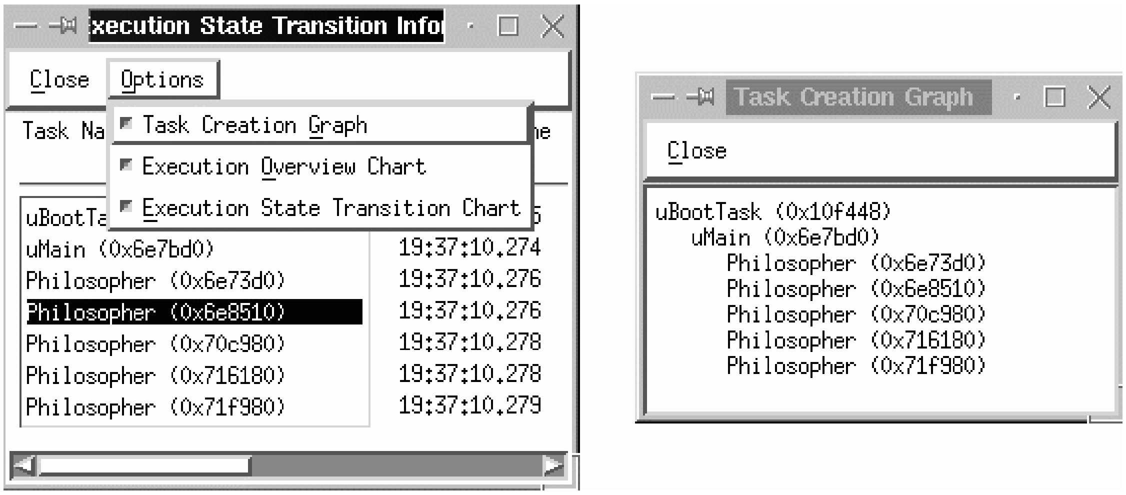


Figure 7.3: EST Metric: Options Menu and Task Creation Graph Displays

by subtracting the number of tasks that ended their main() (column six) from the number of created tasks (column three). A user can get even more information from the table by subtracting other values. For example, subtracting the number of deleted tasks (column four) from the number of created tasks (column three), gives the total number of tasks in the system at a particular time. Similarly, subtracting tasks that ended their main() (column six) from tasks that started main() (column five) produces the number of tasks still in that function.

Pressing the Options button on the menu bar reveals a list of three choices (see Figure 7.3), which can be used to obtain more summary information about all profiled tasks.

Selecting the first option, Task Creation Graph, invokes a uExecStateTaskGraphWidget visualizer, which generates a graph showing parent-child relationships for task creation process (see Figure 7.3). Under each task name, there is an indented list of its direct descendants, i.e., tasks created by that particular task.

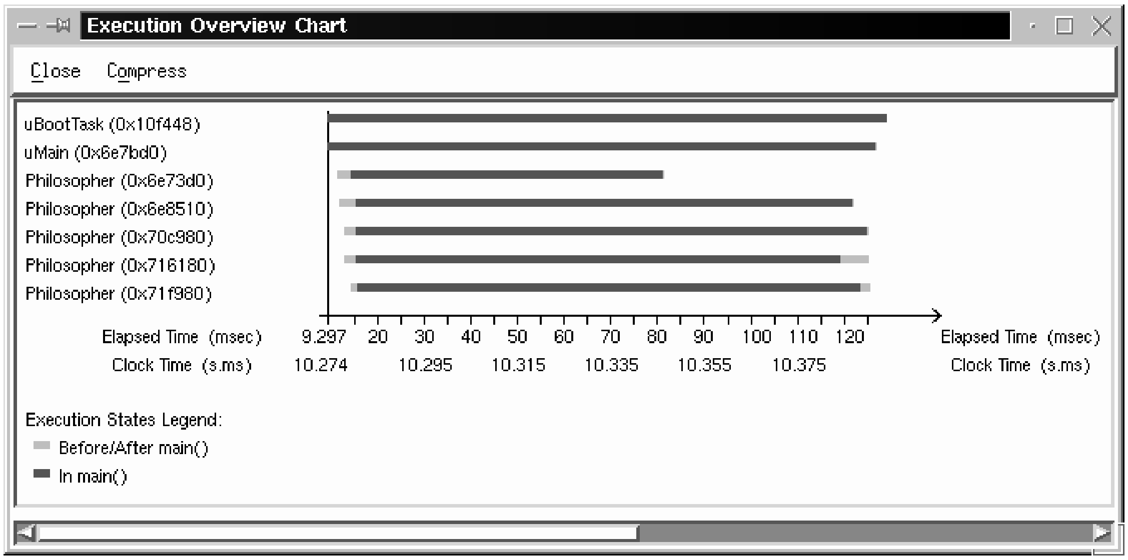


Figure 7.4: EST Metric: Execution Overview Chart Display

The second option is another feature of the EST metric that has been added on users' request. Pressing on Execution Overview Chart invokes a uExecStateOverviewWidget visualizer, which creates a Gantt chart [MR82]. The chart graphically displays, for each task, three pieces of information (see Figure 7.4): time between task creation and starting its main(), real time spent in main(), and time between ending main() and the task's deletion (terminate state). The Y axis represents the profiled tasks and the X axis shows elapsed and clock time of the execution. The time spent in main() is portrayed in purple colour, and the time outside main() in gray. To shorten the diagram, information is presented not from the beginning of program execution but from the time there are at least two tasks created. As a result, it omits information about the initial execution of uBootTask. The omitted information can be found by selecting a detailed display for uBootTask (explained below in the discussion of uTaskExecStateAnalyzeWidget).

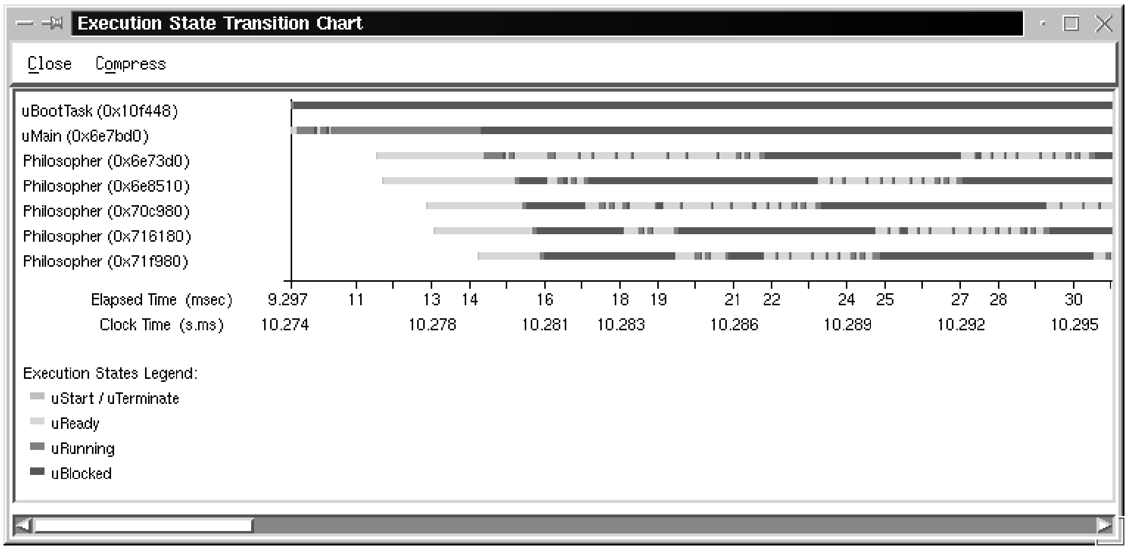


Figure 7.5: EST Metric: Execution State Transition Chart Display

Since users want to apply this chart to search for execution patterns, they like to see the whole line for a task without scrolling. To make this possible, the chart can be compressed horizontally. Pressing the Compress button on the menu bar produces a dialog box asking for the compression ratio; the box also shows the last ratio applied. After specifying the desired ratio, the chart is scaled accordingly and redrawn; some information may be lost during the compression. The scaling is always done from the original size and the chart cannot be enlarged beyond that size. If the original chart is very large, i.e., it spans more than 32,000 pixels horizontally, the chart is automatically compressed and it cannot be enlarged beyond that ratio, but it can be further compressed. The chart presented in Figure 7.4 is compressed by 2 to show the whole display.

Selecting the last option, Execution State Transition Chart, invokes a uExecStateCompareWidget visualizer that creates another Gantt chart displaying, this time, all state transitions (see Figure 7.5). The states are portrayed by different colors: start and terminate are shown in gray, ready in yellow, running in green and blocked in red. The chart axes are labeled in the same way as in the previous chart and the widget also contains a Compress button. It should be noted, however, that compressing this chart loses more information, i.e., many states are represented by very short lines that are not displayed after compression. As a result, a user may end up with a confusing diagram that, for example, shows two tasks running at the same time in a uni-processor environment. Since the compression has significant repercussions, even long charts are initially displayed using their original sizes.

The first column in the summary information table, presented in Figures 7.2 and 7.3, is a task selection list. A user can click on a task name to obtain detailed information about that task's execution.

Selecting a task from the list invokes a uTaskExecStateAnalyzeWidget visualizer for that task. The visualizer creates and displays a widget presented in Figure 7.6. The upper part of the widget contains summary information for the task, e.g., its life duration, the total duration of ready, running and blocked states, and the minimum and maximum state duration. The lower part shows detail information for each state the task entered, i.e., the start time of the state, its duration, the cumulative duration for the task execution and the name of a function in which the state began. A separate uTaskExecStateAnalyzeWidget object is created for each selected task.

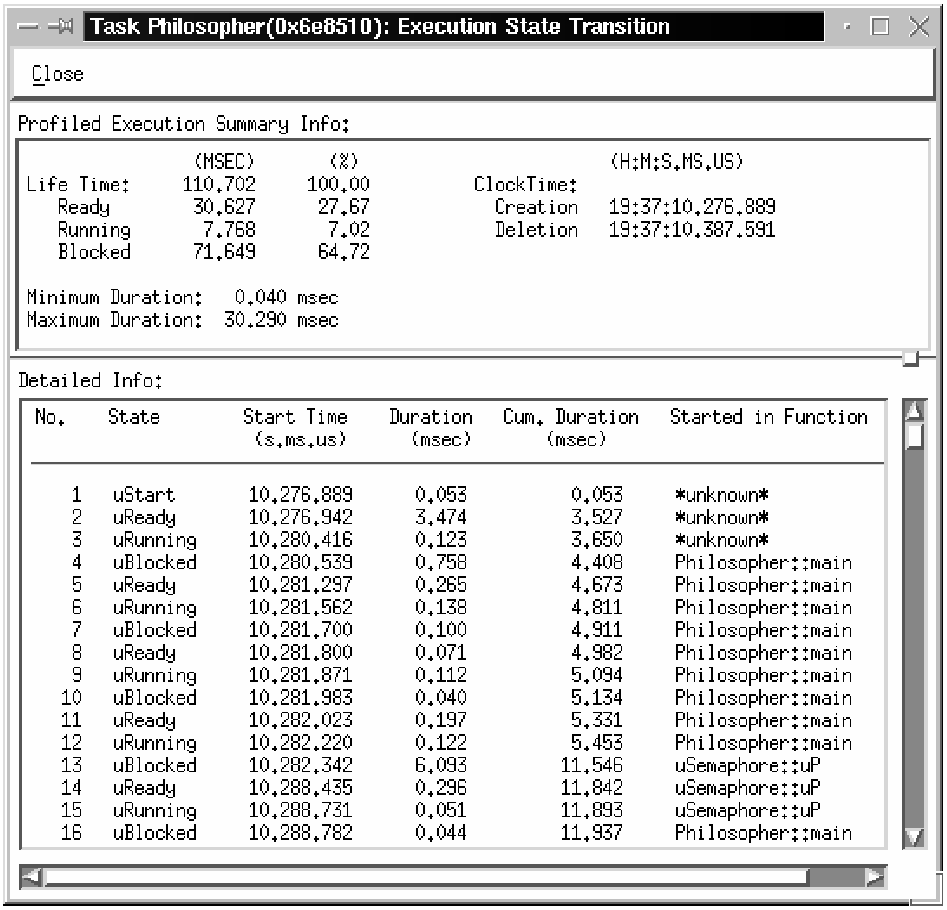


Figure 7.6: EST Metric: Single Task Execution State Transiton Display

7.2 Instrumentation Insertion

The EST metric activates five hooks inside the μC++ kernel (see Figure 7.1). The main hook, which generates data for each state, is activated by RegisterTaskExecStateNotify().

RegisterTaskNotify() and RegisterCoroutineNotify() create the uProfileSampler object for each task and coroutine registered for profiling. uProfileSampler is responsible for creating and updating the profiling stack, and gathering the data.

The start and end times of a task's execution are recorded by RegisterTaskNotify() and DeregisterTaskNotify() respectively.

RegisterTaskSetNameNotify() changes the task's name saved in the uExecStateInfo object and displayed in the selection list (see Figure 7.2) to the name given by a user through a call to a task's uSetName() routine.

7.3 Implementation

Since EST is based on tracing, which implies gathering a large amount of data, the main issue in implementing the metric is data storage. Even a short-running task can go through several hundred states during its execution. Therefore, dynamically allocating a new data structure for each state would be expensive, considering that memory is a shared resource and its allocation requires mutual exclusion. To avoid this overhead, larger chunks of memory, i.e., enough to store data for many states, are allocated at a time.

Data for a single state is stored into uExecStateInfoEntry (see Figure 7.1). The uExecStateInfoEntry objects are grouped into an array (tableEntries) that is embedded in the uExecStateInfoTable. For fast access into the array, the uExecStateInfoTable contains an index (tableIndex) that marks the last occupied spot in the array. tableIndex is also used to detected the end of the array. As soon as all entries in one array are filled out, another uExecStateInfoTable object is created and chained to the previous one through a nextTable pointer.

The uExecStateInfo object, created for each profiled task, contains two uExecStateInfoTable pointers to the first and last uExecStateInfoTable node (table) of the linked list. firstExecStateInfoTable points to the first table, enabling the metric analyzer (uExecStateAnalyze) to access the data after execution, and currExecStateInfoTable points to the currently updated table to speedup the update process during execution.

The uExecStateInfo object also includes the size (execStateInfoTableSize) of the tableEntries array. Based on empirical analysis, the size of the array has been set at one hundred entries, because most tested short-running programs went through sixty to five hundred transitions. Making the size smaller would result in a larger number of calls for memory allocation, increasing the profiling overhead. On the other hand, making the size larger could result in wasting memory, since the last array for each task usually is not completely filled. This wastage could lead to memory shortage for data analyzing and visualizing, especially when a profiled program contains many tasks.

The uExecStateInfo also contains stateMinDuration, stateMaxDuration, readyStateDuration, runningStateDuration and blockedStateDuration variables, which are computed by the uExecStateAnalyze object after program execution.

7.4 Validation

The validation of the EST metric is done using the CG&RT metric described in Chapter 6. It was shown that CG&RT generates correct data (see Section 6.4); therefore, showing that both metrics produce statistically similar results implies that the EST presents correct information.

EST finds the life time of a task, real time of its execution and duration of each state it was in. And CG&RT finds real and CPU time a task spends in each of its functions. Therefore, summing the total CPU time of each function called by a task (reported by the CG&RT metric) should be equal to the total running time for the task (reported by the EST metric). Furthermore, the real time reported by CG&RT for a task's main() should be equal to the execution real time reported by EST.

A test was performed using the program presented in Appendix B.2. The uMain task creates five User tasks that try to enter a Bathroom mutex object, which results in task's blocking. The time spent in the Bathroom is simulated by a spinning loop. Each task tries to enter the Bathroom object five times, spinning for a constant amount of time between each attempt.

Both metrics are active during the same run of the program and their results are presented in Table 7.1.

Task Name	CG&RT		EST	
	Running T. (msec)	Real Time (msec)	Running T. (msec)	Real Time (msec)
User (0x724a38)	1,132.376	5,158.284	1,131.156	5,158.316
User (0x717a38)	1,104.181	5,047.355	1,102.934	5,047.419
User (0x70aa38)	1,114.265	4,930.196	1,112.767	4,930.226
User (0x6fda38)	1,116.604	4,715.182	1,115.311	4,715.212
User (0x6f0a38)	1,123.079	4,039.161	1,121.828	4,039.190
uMain (0x6d8d48)	7.937	5,599.141	7.178	5,599.175

Table 7.1: EST: Comparison of Results between EST and CG&RT

As can be observed from the table, the results given by both metrics are very similar. The running time reported by CG&RT is slightly higher than the time given by the EST metric because of the CG&RT's higher probe effect. Since the running time in the CG&RT metric is distributed among functions, the profiling data structures must be updated not only on context switches (what is done by EST) but also on each function entry and exit. On the other hand, the real time reported by EST is slightly higher than the real time reported by CG&RT. This difference can be explained by the order in which the metrics' hooks are processed. At the beginning of execution, a task is first scheduled to run (its state changes to running) and then starts its main(), i.e., the EST *execution-state* hook is processed before the CG&RT *function-entry* hook. At the end of execution, the task first exits its main() and then it changes the state to terminate, i.e., the CG&RT *function-exit* hook is encountered before the EST hook.

7.4.1 User Experience

The EST metric has been used by students to profile algorithms for LIFO and FIFO task scheduling. Their program performs a parallel matrix-multiplication using a divide-and-conquer method. In FIFO scheduling, all tasks are created first and then the calculations are performed. Whereas, in LIFO scheduling, only one branch of tasks is created, and after they finish their work, another group of tasks is created, reducing the maximum number of tasks in the system and the amount of memory used.

The users applied μProfiler to verify that their program behaved in an expected way in a uni-processor environment to get a better understanding of the program's behaviour in a multi-processor environment where the execution is more complicated, and to improve its performance.

Figures 7.7 and 7.8 show an overview chart (compressed) for the original program with FIFO and LIFO scheduling in a uni-processor environment. It was immediately obvious that the program was not accomplishing what the users wanted. There was a long delay between the task's end of execution and its deletion. Only after multiple changes, each one examined in detail with μProfiler, did the users finally achieve their desired result. Figures 7.9 and 7.10 present execution of the improved program, which decreases the use of memory and the number of tasks in a system during execution.

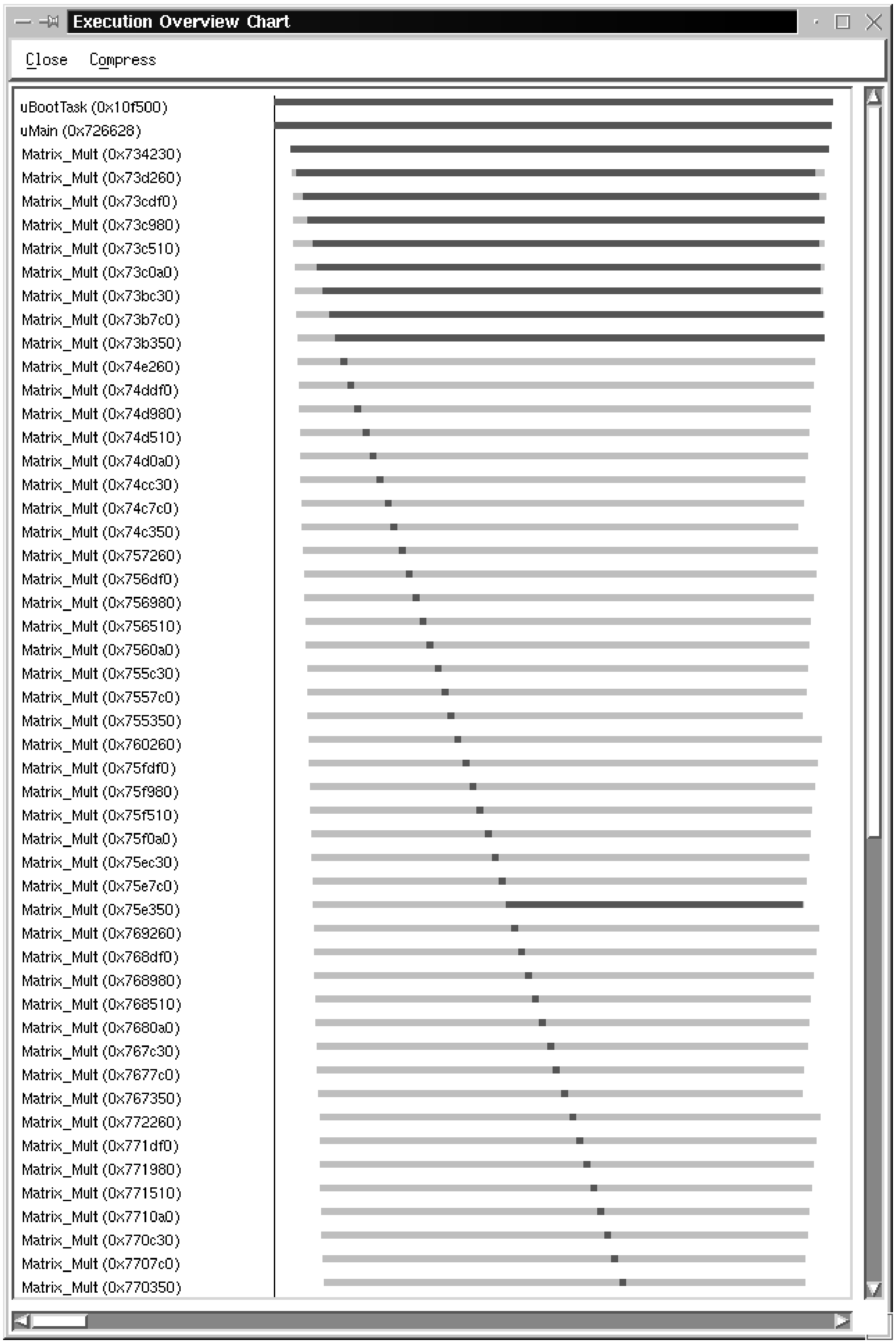


Figure 7.7: Example: Original Program - FIFO Scheduling

Figure 7.8: Example: Original Program - LIFO Scheduling

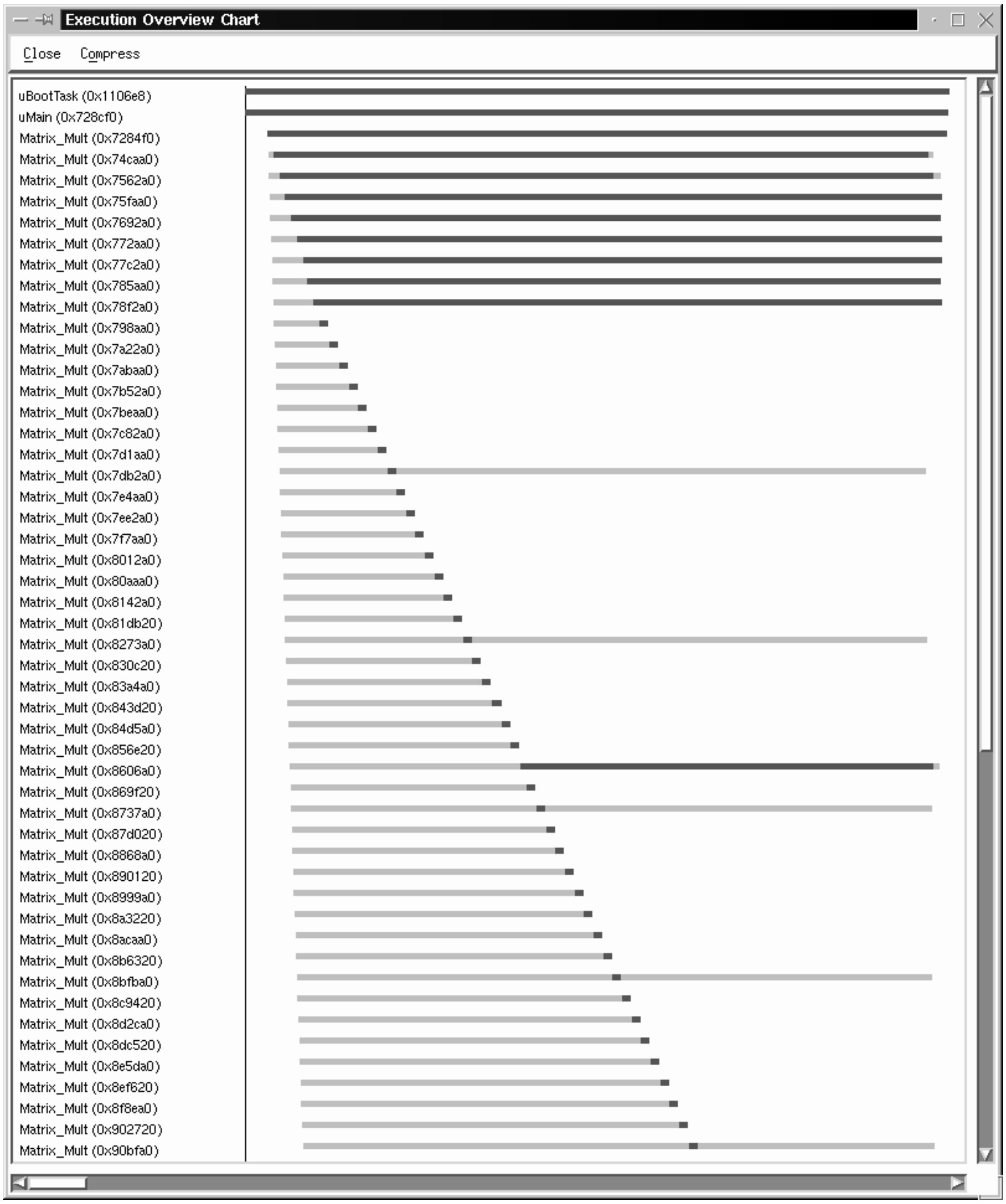


Figure 7.9: Example: Improved Program - FIFO Scheduling

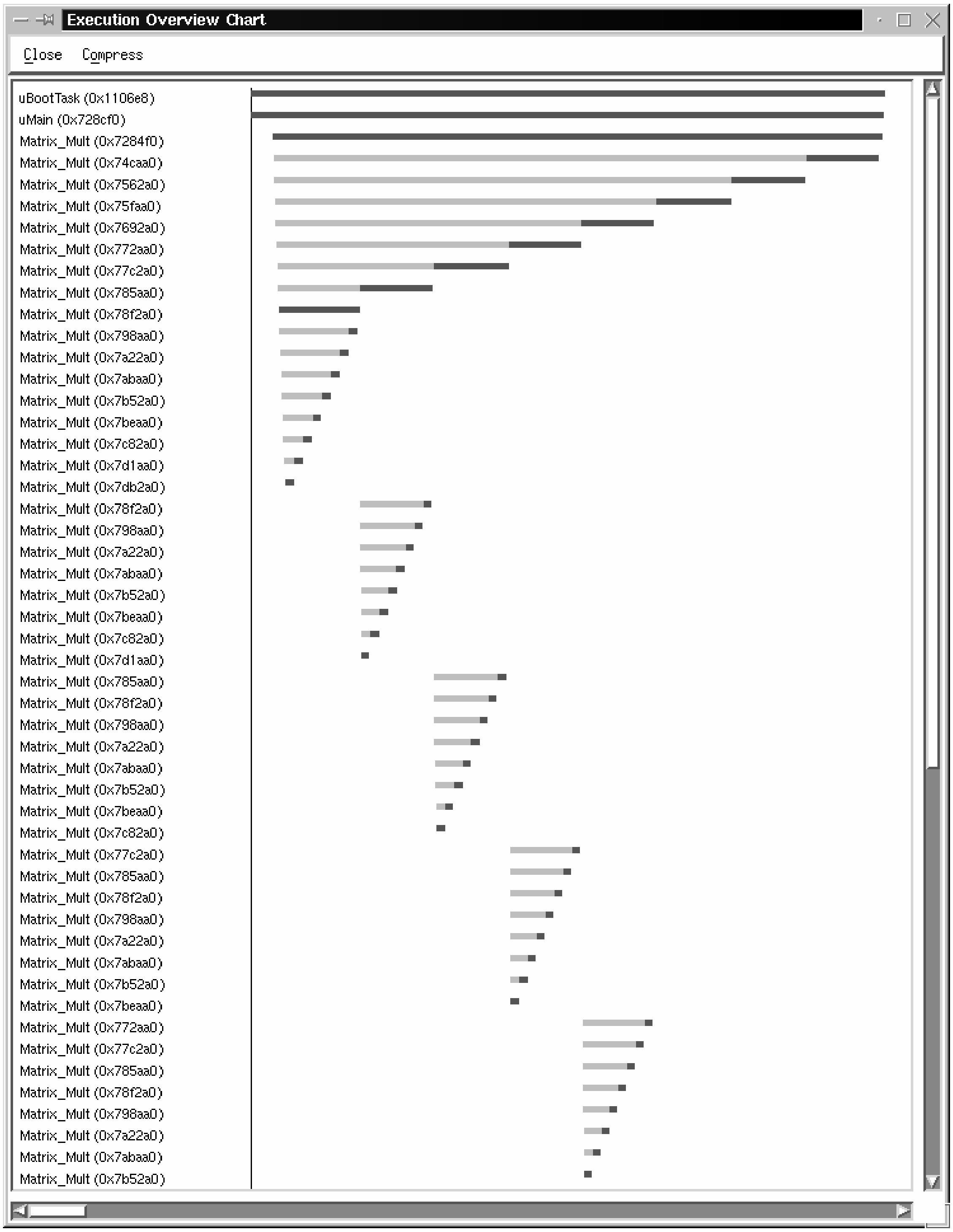


Figure 7.10: Example: Improved Program - LIFO Scheduling

Chapter 8

High Level Tracing Metric

This chapter describes the μProfiler's built-in *High Level Tracing* (HLT) metric. HLT replaces an existing tracing support, built into the μC++ translator.

In the *tracing system* [TB96], the μC++ translator inserts additional code into a user program when compiled with a -trace flag. The inserted code records synchronization and communication events and sends them immediately for visualization to an external visualization tool, POET.

Partial Order Event Tracer (POET) [Tay95b, Tay95d] is a general tool for collecting and displaying event data from concurrent and distributed applications, running in several different target environments, such as OSF DCE, ABC++, SR, PVM. To achieve target-system independence, POET places information describing each target environment in a separate configuration file. Therefore, adapting POET to a new programming environment [Tay95a] is reduced to creating a configuration file for the new environment. The old *tracing system* had a configuration file for μC++. The file is now modified to accommodate new events supported by HLT.

8.1 Design

The HLT metric traces the activities of task threads and their interactions with objects such as coroutines, monitors, coroutine-monitors and other tasks (see Section 4.1). Within these objects, HLT profiles only events that are important to concurrent execution or accessing a coroutine, i.e., interactions with mutex members and the *coroutine main*. It does not profile thread access to non-mutex objects, free routines (routines that do not belong to an object), non-mutex members of mutex objects, and coroutine members that do not resume the *coroutine main*.

The design of the HLT metric is presented in Figure 8.1. The notation used in this object-oriented analysis model is explained in Appendix A.

Since HLT utilizes POET for data analysis and visualization, its design differs from the design of other metrics; HLT has only a monitor and does not have analyzers and visualizers.

The metric's monitor, uHLTMonitor, derived from the abstract class uExecutionMonitor, activates hooks necessary to trace the specified objects and events. It also creates a uPoetInterface object, which establishes a socket connection with POET and transmits the data (called by POET *events*). POET recognizes two kinds of *events*: a *normal event*, representing an occurrence of an event, and a *text (named) event*, containing an annotation attached to a *normal event*. uPoetInterface encapsulates these two kinds of events inside EVENT and NAMED-EVENT classes.

uProfileSampler is created for each coroutine, monitor and task registered for profiling. It keeps track of accounting information required by POET's *events*, e.g., trace-line ID (explained in Section 8.1.1) or *event* number, and stores the informa-

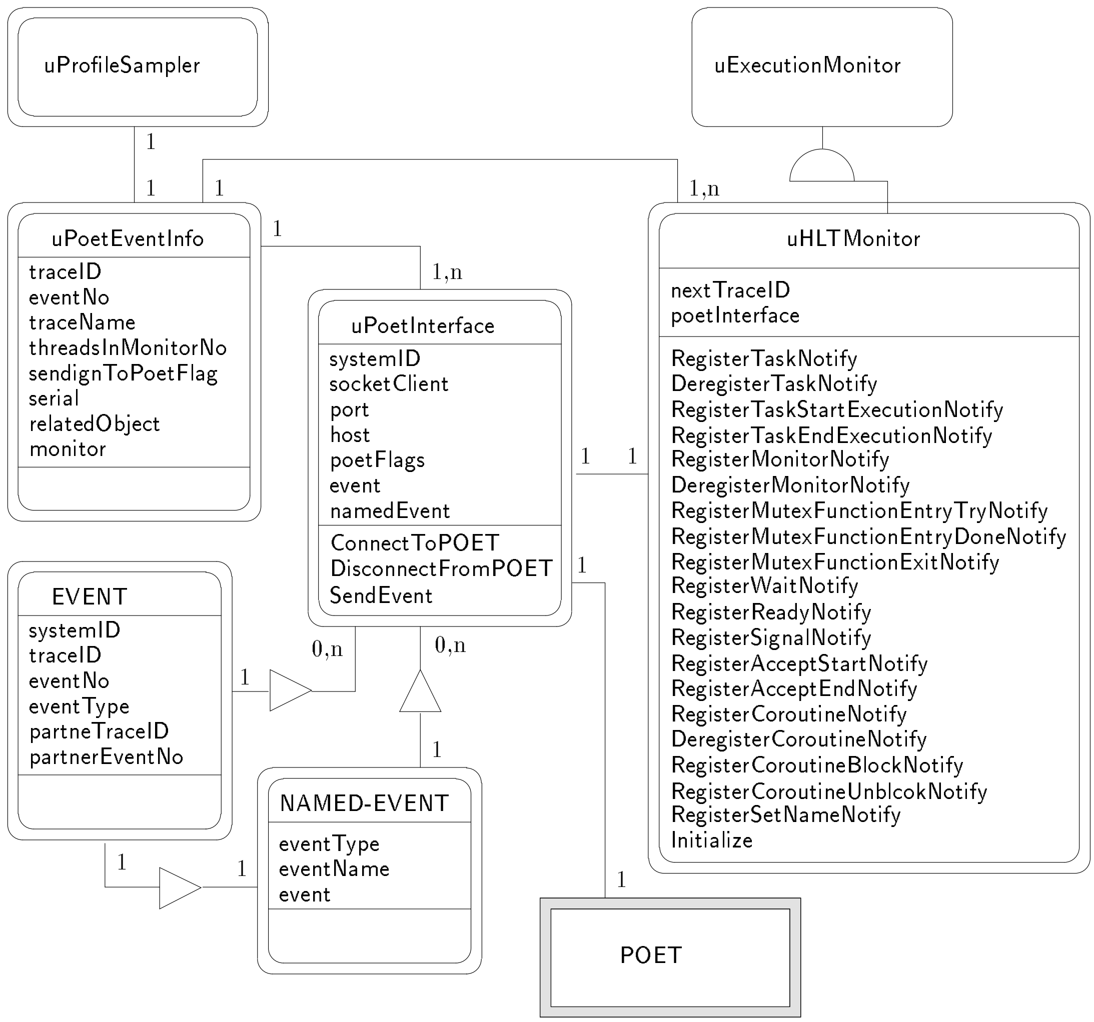


Figure 8.1: Object-Oriented Model of HLT Metric

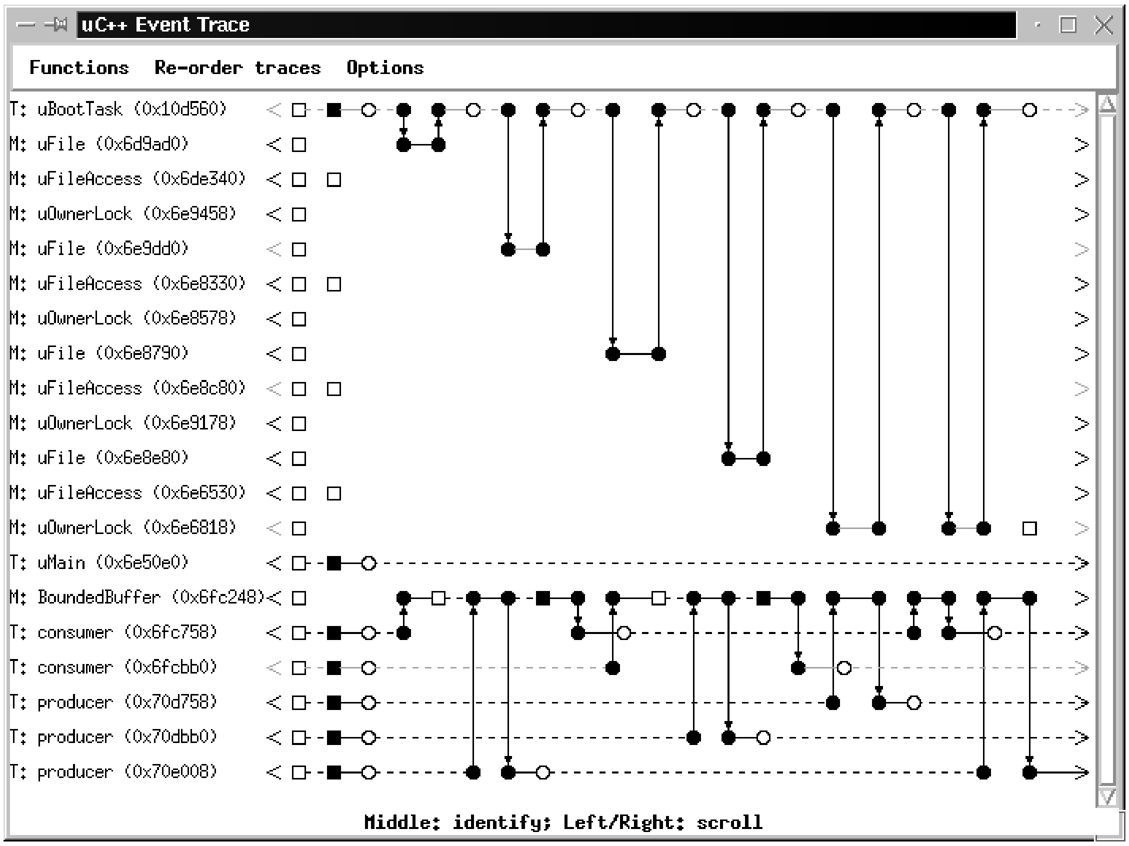


Figure 8.2: POET: Main Display for HLT Metric (Task)

tion inside a uPoetEventInfo structure.

Each recorded event is immediately sent to POET for processing and visualization; POET supports run-time visualization versus post-mortem.

8.1.1 POET Visualization

Figure 8.2 presents part of POET's display for the HLT metric, tracing execution of a producer-consumer program; the program illustrates interactions of multiple tasks with a single monitor.

POET represents each profiled object by a horizontal line, called a *trace line*. A trace line begins with a name, which consists of three parts:

- capital letter that represents the type of the object: *C* for a coroutine, *M* for a monitor, *C-M* for a coroutine-monitor and *T* for a task,
- name of a class from which the object is instantiated or a name given by a user through a call to a task's or coroutine's uSetName() routine,
- object ID, to differentiate among several objects from the same class.

Internally, a trace line has a unique ID, which must be included with each *event* transmitted to allow POET to associate the incoming event with a correct trace line.

An event is drawn as an empty or a filled circle (○, ●) or square (□, ■). Interaction between objects (*synchronization event*) is represented by a vertical line, with an arrow at one end, that connects two filled circles ● from separate trace lines. A thread's state is presented by solid (ready or running state) or dash (start, blocked or terminate state) lines. More information about specific event representation is provided in Section 8.1.1.1.

Time flows from left to right in the diagram; however, the events are placed based on relative, and not absolute, time. On a single trace line, events to the left of a particular event occurred before it and events to its right occurred after it. Events on different trace lines have no time relationship. However, there are *partial orderings* between trace lines caused by synchronization events. At a synchronization point (vertical line), all events on both connected trace lines to the left of that point occurred before the synchronization, and all events to its right occurred after the synchronization.

8.1.1.1 Event Representation

The first symbol on each trace line is an empty square □ that represents an object's creation. Similarly, each trace line ends with another empty square □ (see M:uOwnerLock (0x6e6818) or uFileAccess objects in Figure 8.2) depicting an object's deletion. For tasks, the start and end of a thread's execution are presented by filled squares ■. Since a thread is in the start state between its creation (□) and the start of its execution (■), and in the terminate state after ending its execution (■) and waiting for deletion (□), there is a dash line between the corresponding events.

In general, events that cause a thread to block are displayed with empty squares □, and events that cause a thread to become ready are shown with filled squares ■.

A call to a mutex member (*petition to enter*) is represented by an empty circle ○, followed by a dash line that indicates the task's waiting (blocking) for mutually-exclusive access to the mutex object. When a thread acquires mutual exclusion, the call to the mutex member proceeds. If the thread is not inside the mutex object already (a call from one mutex member to another is allowed in μC++), its entry is presented by a *synchronization event* and the thread is seen to transfer from one trace line to another. Leaving the mutex object is shown by another *synchronization event*. The direction of the arrow on the vertical line, represents the direction in which the thread moves.

When an object contains only blocked threads, its trace line is dashed. When there is a ready or running thread inside an object, its trace line is solid. And there

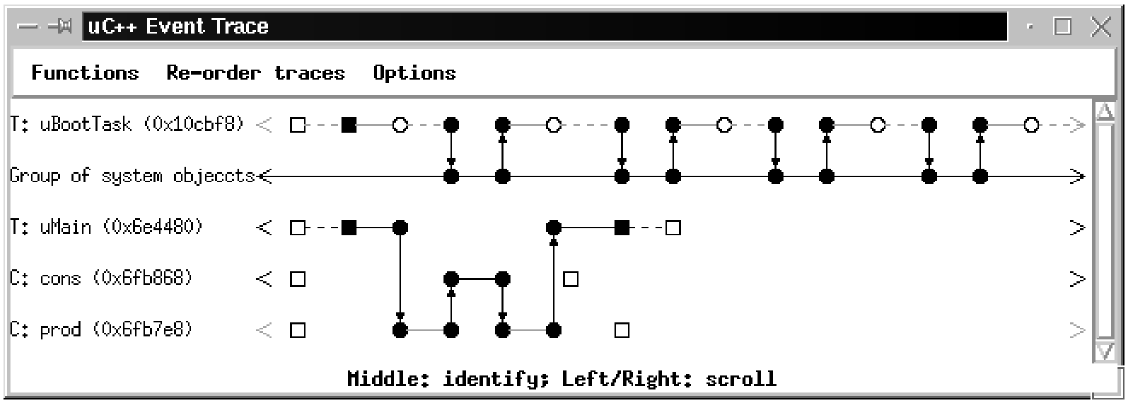


Figure 8.3: POET: Main Display for HLT Metric (Coroutine)

is no trace line if an object does not have any threads inside.

The HLT metric also traces suspending and resuming of a coroutine. When a thread executes a coroutine's member that performs a uResume statement, the thread switches from its current stack (execution state) to the coroutine's stack; it switches back after executing a uSuspend statement. These transfers are represented in POET's display by *synchronization events* (see Figure 8.3).

With a coroutine-monitor, if a coroutine's mutex member contains the uResume statement, an entry into that member is shown in the regular way, i.e., *petition to enter* followed by *synchronization event*, which transfers a thread from its current trace line into the C-M trace line (see bottom diagram of Figure 8.6). However, since the thread is already on the C-M trace line, entering *coroutine's main* is shown only by a filled square ■ followed by a solid line because the thread is executing. Similarly, suspending main() is depicted by an empty square □ with a solid line behind it (thread changes execution stack but does not block). Finally, leaving the mutex member transfers the thread back to its own trace line, represented by a *synchronization event.*

Information about POET's menu options, horizontal scrolling (which preserves the relative ordering of events), or obtaining detail information about an event and viewing its predecessor and successor events can be found in the POET user manual [Tay95c].

8.2 Instrumentation Insertion

Since there are three kinds of objects to be monitored and quite a number of events associated with them, the HLT metric activates the largest number of hooks (see Figure 8.1). All the hooks are inserted into various routines of the μC++ kernel.

The RegisterTaskNotify(), RegisterCoroutineNotify() and RegisterMonitorNotify() routines are used to inform POET that it needs to create a trace line and mark the creation event (□). The corresponding Deregister... routines are used to mark the end of a trace line by sending the deletion event (□).

RegisterSetNameNotify() generates an event instructing POET to change the name of a trace line for a coroutine or a task when a thread calls a uSetName() routine.

RegisterTaskStartExecutionNotify() and RegisterTaskEndExecutionNotify() activate hooks that mark a task's entering and exiting its main(), represented by the second and second last mark (■) on a task's trace line.

There are several hooks associated with gathering information about tasks calling a mutex member and accessing a mutex object (monitor). Since access is serialized, a task first makes a petition to enter a mutex member (RegisterMutex-

FunctionEntryTryNotify()), and when there are no tasks in the monitor, the access is granted (RegisterMutexFunctionEntryDoneNotify()); the exit from the mutex object is marked by RegisterMutexFunctionExitNotify(). On the other hand, if there are already tasks in the mutex object, the arriving task blocks and waits until its entry gets scheduled by the monitor. Since monitors support two types of scheduling, internal and external scheduling (see Section 4.1.2), both of them must be profiled. Hooks for internal scheduling are activated by RegisterWaitNotify(), RegisterReadyNotify() and RegisterSignalNotify() functions. Whereas RegisterAcceptStartNotify() and RegisterAcceptEndNotify() activate hooks for external scheduling.

Suspending and resuming of a coroutine is profiled through hooks activated by RegisterCoroutineBlockNotify() and RegisterCoroutineUnblockNotify() functions.

8.3 Implementation

Since several tasks can access the uPoetInterface object at the same time, HLT must ensure that data from different tasks does not get mixed up. To prevent the mixing, a buffer (EVENT or NAMED-EVENT structure) from which the data is transmitted to POETis declared as a local variable inside a SendEvent() function. Therefore, if a task gets context-switched when copying data into the buffer, the event structure is stored on its execution stack, which prevents overwriting the data by another task.

POET expects some events to arrive in a predetermined order, specified in a configuration file. For example, entering a monitor by a thread (marked by a *synchronization event*) really consists of two events. The first event announces that the thread is entering the monitor and puts a mark (●) on the task's trace line. The second event indicates that the monitor received the thread and draws a mark (●) on the monitor's trace line as well as connects these two dots with a vertical line and an arrow. If the events are not received in the correct order and POET is unable to straighten out the ordering, it aborts program execution. Therefore, HLT must guarantee that POET receives events in the order they are sent. To ensure this ordering, each thread reports its own execution, i.e., a task sends events for its own trace line as well as for trace lines of each object accessed by its thread. Consequently, even if a task gets context-switched inside SendEvent() and another task comes in and sends its events first, it does not cause a problem because traces from different threads are independent.

8.4 Validation

The validation of the HLT metric has been done by comparing displays generated by the metric with those created by the *tracing system* for separate executions of the same program.

Since the translator modifies only user code, the *tracing system* cannot collect data about user tasks interacting with the system's mutex objects, such as uOwnerLock, uFile or uFileAccess, created by uBootTask. As a result, it does not profile I/O operations, which use mutual exclusion in μC++ [BS99]. But I/O is profiled by the HLT metric. Therefore, to make the displays similar, the test programs did not contain I/O operations.

Also, the trace line for uBootTask, generated by the *tracing system* contains only creation, start and end of execution, and deletion events. Whereas the uBootTask trace line produced by HLT includes many more events and shows interactions with the system's objects (see Figure 8.2). Again, to make the displays similar and to shorten the HLT's display, these system's objects are grouped into a Group of system objects POET-cluster and displayed as a single line; this is done using the *Cluster Hierarchy Interface Tool (CHIT)* available under POET.

For all test cases, the upper display is generated by the *tracing system* and the lower one is done by HLT.

8.4.1 Task Interaction with Monitor

A simple producer-consumer program (source code in Appendix B.3.1) is used to test interactions among tasks and monitors. The program traces, produced by the *tracing system* and the HLT metric, are shown in Figure 8.4. The traces are identical after disregarding the uBootTask trace line.

In the program, a single "product" is generated by a producer (prod) and given to a consumer (cons). A buffer class is not a mutex object, so it does not have a trace line. However, the two semaphores (full and empty), which are implemented in μC++ as monitors [BS99], inside the class have trace lines. Also, it should be noted that the call to buffer::query() made inside prod::main() is not reported because query() is a non-mutex member.

All trace lines contain creation and deletion events (□) for each object, and the start and end of execution events (■) for tasks.

The cons task tries to get a product by accessing a full buffer slot (○ followed by

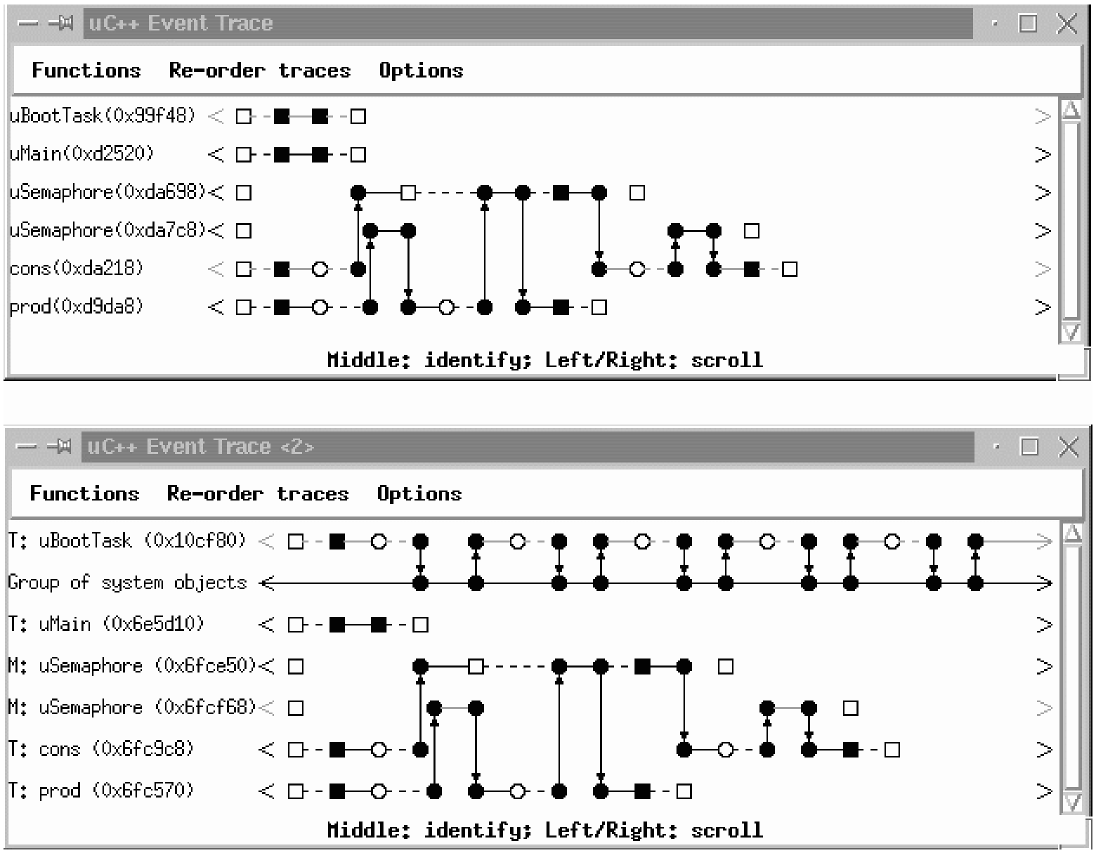


Figure 8.4: HLT testing: Task Interaction with Monitor

synchronization event between the cons task and full [first] semaphore). Because a product is not delivered yet, cons blocks inside the semaphore (□). In the meantime, the prod task drops off the product without blocking by obtaining an empty buffer slot (○ followed by two *synchronization events* between the prod and the empty [second] semaphore). The prod task then signals the waiting cons that the product has been delivered by indicating a full buffer slot (○ followed by two *synchronization events* - signal never blocks), then ends its execution and is destroyed. The cons task unblocks knowing there is a full buffer slot (■), and removes the product from the buffer (*synchronization event*). The cons task then indicates an empty buffer slot (○ followed by two *synchronization events*), ends its execution, and gets deleted.

8.4.2 Task Communication

The test program (source code in Appendix B.3.2) illustrates a simple heterosexual dating service. Girl and Boy tasks call the dating service leaving their phone numbers and block until they are given a phone number of a person of opposite sex.

A DatingService task accepts calls from Girl and Boy tasks using external scheduling. When it is time to close, the DatingService task accepts its *destructor*, which ends its execution. The result of running the program with three Girl and three Boy tasks is shown in Figure 8.5. The traces, produced by the *tracing system* (top diagram) and the HLT metric (bottom diagram), are very similar.

There is a difference in the order of matching the second pair of tasks. In the trace generated by the *tracing system*, the Girl task calls before a Boy task, so it blocks waiting for the Boy task. In the display generated by HLT, the situation is reversed. This difference is easily explained by the fact that these two diagrams come from two separate executions of the program, and in a concurrent environment the order of running tasks is unpredictable. Even though the order of some events is reversed, the logic behind them stays the same in both diagrams.

The only real difference between these two displays is at the end of the DatingService trace line. There is an extra filled square ■ between the thread's last blocking event (□) and the end-of-execution event (■) in the HLT diagram. When all Girl and Boy tasks are deleted, DatingService accepts its *destructor* (■ after the

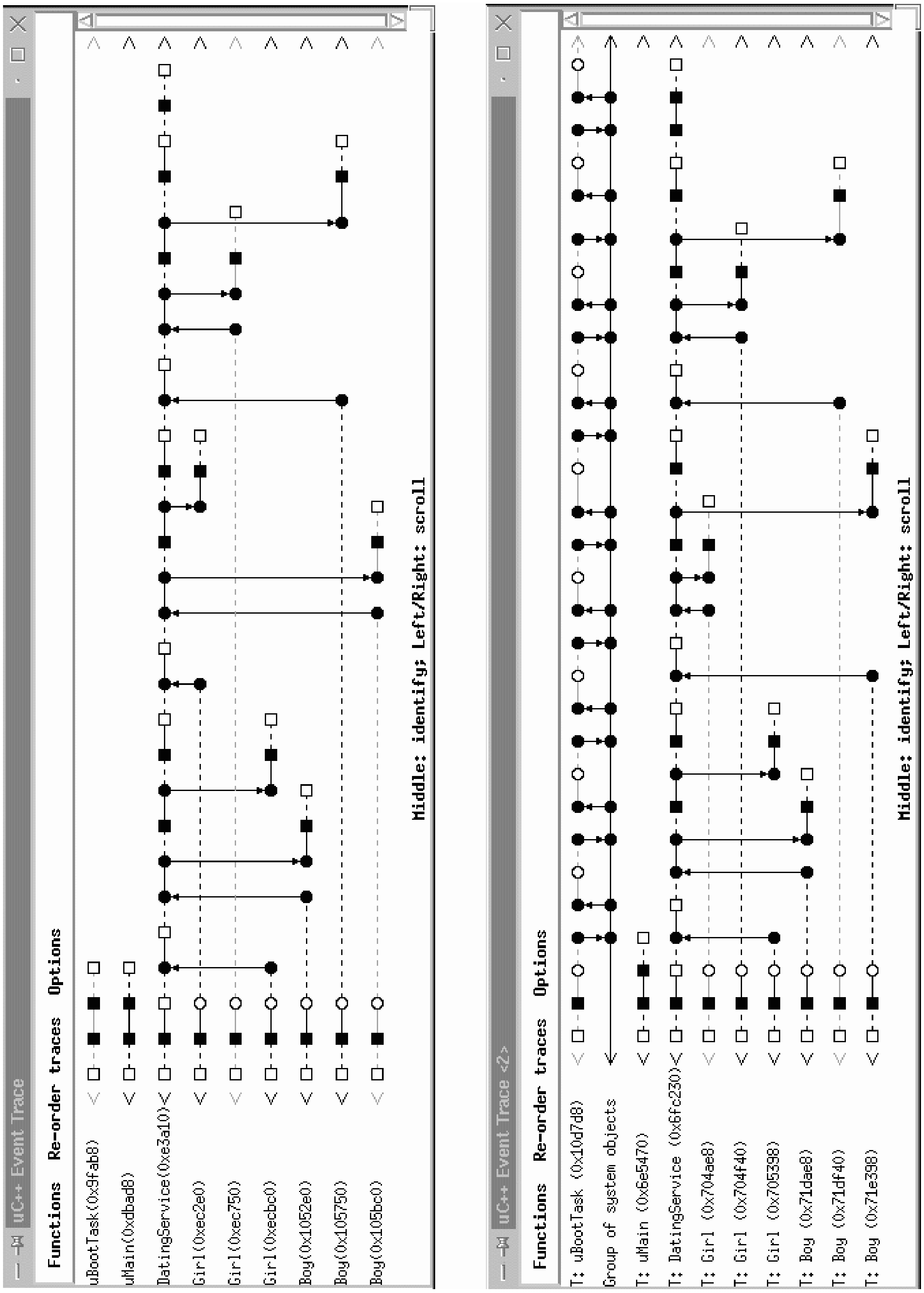


Figure 8.5: HLT Testing: Tasks Cooperation

last *synchronization event*), and blocks again (□). But the cleaning up (deleting the task's data structures) inside the *destructor* can be done only after the task finishes its execution. Therefore, the destructor unblocks the DatingService task (the extra ■), which breaks out of the infinite loop and ends its execution (last ■). Then the destructor deletes the task (last □). The *tracing system* does not record the unblocking operation because of the break statement, which moves the thread out of the loop in the middle of the uAccept statement. This omission is a bug in the *tracing system* not present in the HLT metric that has hooks inside the μC++ kernel routines, which are executed no matter how a thread leaves the uAccept statement.

8.4.3 Coroutine

A simple coroutine program (source code in Appendix B.3.3) that generated the output presented in Figure 8.3 (program compiled with the -profile flag), was also compiled with the -trace flag and run under the *tracing system*. Both created identical displays (disregarding the uBootTask).

A complex program (source code in Appendix B.3.4) using a Fibonacci number generator has been used to test task interaction with a coroutine-monitor. Two independent generators, i.e., two instances of the fibonacci coroutine, create Fibonacci numbers for tasks entering the coroutine-monitor. A task gets a single number from each generator. Since the generators are accessed by multiple Worker tasks, they require mutual exclusion to allow only one task in at a time.

Figure 8.6 contains the results of running the program under the *tracing system* and the HLT metric. The differences in showing how a thread accesses a coroutine-

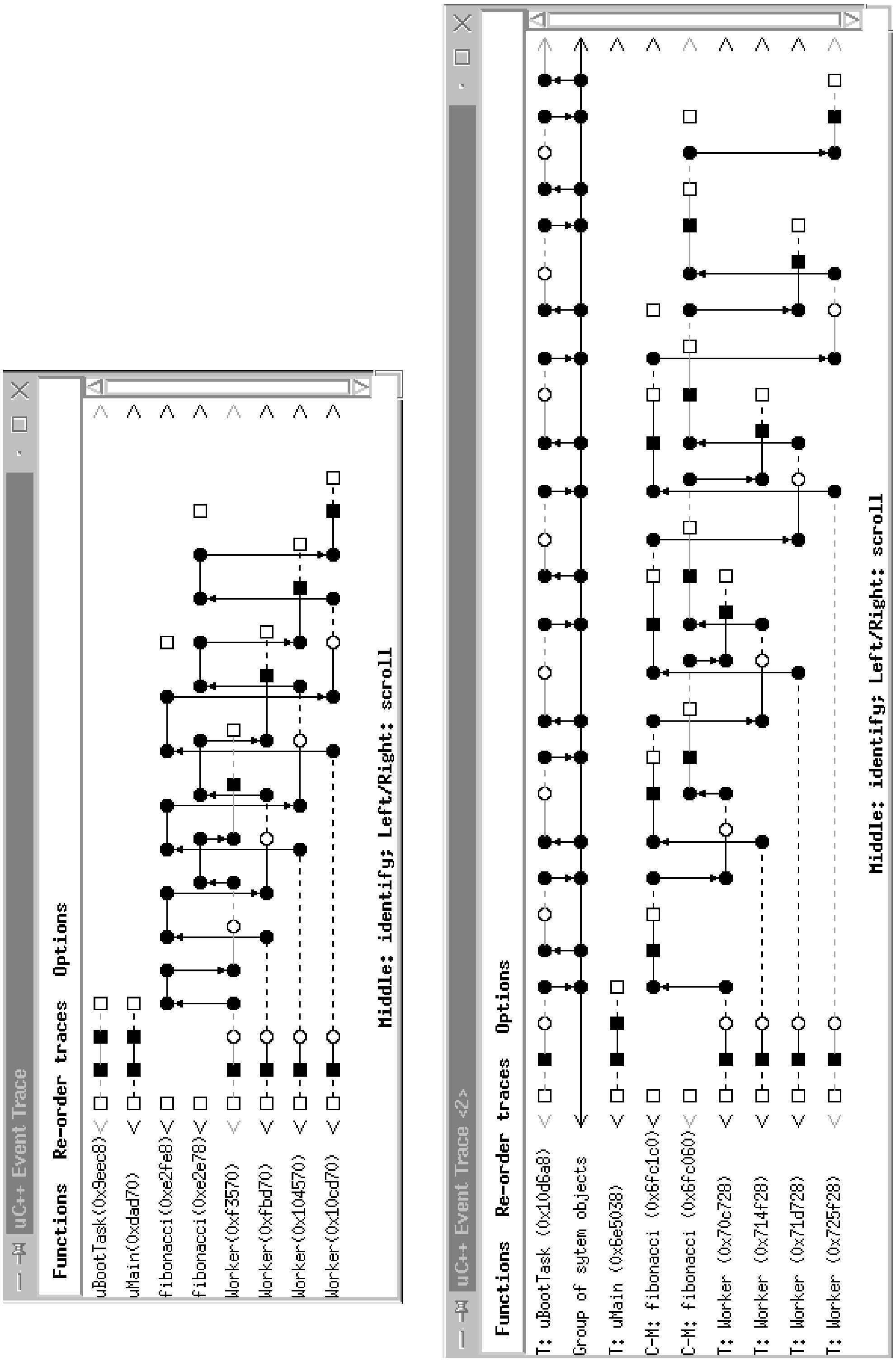


Figure 8.6: HLT Testing: Coroutine-Monitor

monitor result from extensions in the HLT metric over the original *tracing system.*

The *tracing system* shows the *petition to enter* event (○), when a task calls a coroutine's mutex member (fibonacci::next), but it does not display the actual entry into the member (*synchronization event*). The *synchronization events* in the diagram indicate only entering and leaving the *coroutine's main.* Therefore, if a coroutine's mutex member does not resume the coroutine, the trace shows *petition to enter* a mutex routine but does not show the transfer of thread to the C-M trace line for that mutex member.

On the other hand, the HLT metric traces both entering and exiting a mutex member, and resuming and suspending a coroutine. Therefore, the *synchronization events* in the HLT display represent entry and exit to/from a mutex member, and the filled (■) and empty (□) squares signify coroutine-monitor resumption and suspension.

8.4.4 HLT Extensions

This sections presents few other differences between the *tracing system* and the HLT metric.

For internal scheduling, a thread gets blocked on a condition and it is reactivated when another (active) task executes a signal statement. The *tracing system*, shows the thread's blocking, i.e., executing a wait statement, and unblocking, i.e., the result of performing a signal operation by another task, using an empty square □ and a filled square ■, respectively. However, it does not record the execution of the signal statement itself done by the signalling task. The HLT metric registers

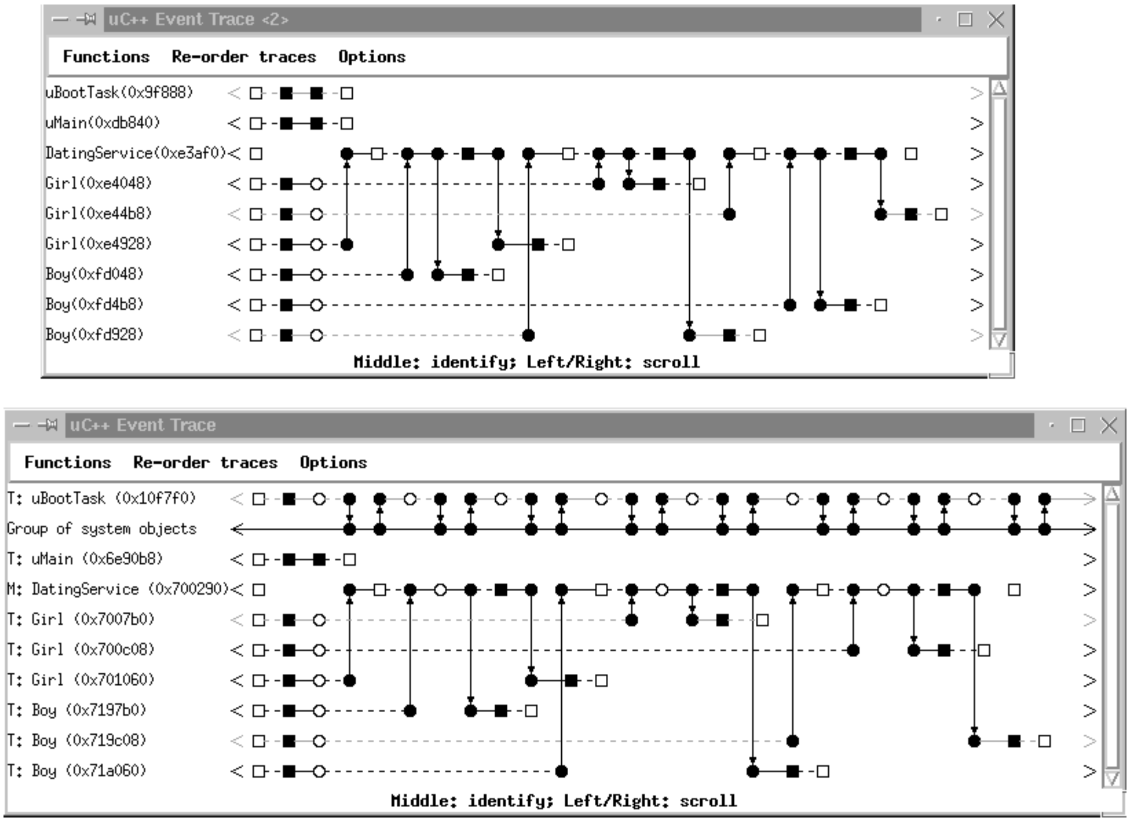


Figure 8.7: HLT testing: Internal Scheduling - Task Signalling

all three events, utilizing the same symbols for a thread's blocking and unblocking, and using an empty circle ○ to show the signal statement. Presenting the signal statement is useful especially when a task signals more than one task.

Figure 8.7 presents traces, obtained under the *tracing system* and the HLT metric, for a dating-service program, which uses internal scheduling for matching Girl and Boy tasks (source code in Appendix B.3.5). For the first pair of matched tasks, the Girl task calls first into the DatingService, which is implemented as a monitor (it is implemented as a task in the external scheduling, see Section 8.4.2), and blocks inside the monitor (□), waiting for a Boy phone number. Then, a Boy task comes

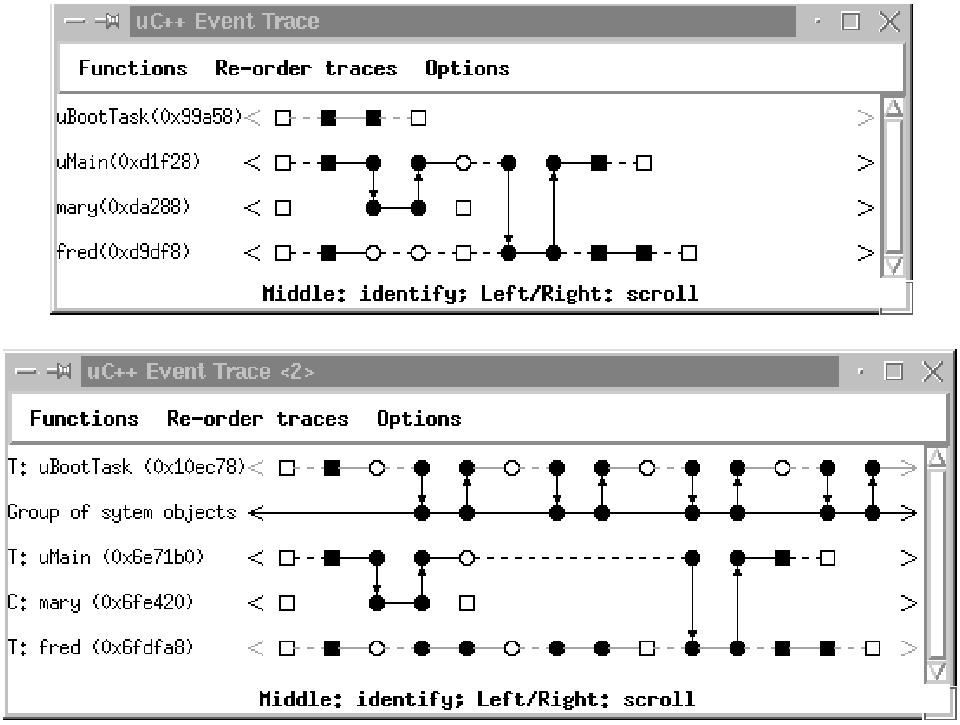


Figure 8.8: HLT testing: Task Entering its Mutex Member

in (*synchronization event*), leaves his phone number and signals the Girl task. The signalling is shown in the HLT metric with an empty circle ○, but it is not shown in the *tracing system*. After, the Boy task leaves the monitor (*synchronization event*), the Girl task unblocks (■) and continues her execution.

Another feature added into the HLT metric, involves processing calls to a mutex member in the same mutex object that has already been acquired, including calls to a task's own mutex members. In this situation, the *tracing system* indicates only *petition to enter* (○) but does not show the actual entry and exit to/from the mutex member. On the other hand, the HLT metric records all three events, i.e., the *petition to enter* is followed by two filled circles ● on the same line indicating the entering and leaving the mutex member.

Figure 8.8 shows the traces produced by the *tracing system* and the HLT metric for a simple program in Appendix B.3.6. It can be observed that the *tracing system* displays only two *petition to enter* events (two ○ on fred line) for the two calls to fred::mem() performed by fred::main(). Whereas, HLT shows the *petition to enter* as well as entering and exiting the fred::mem() routine.

This program also illustrates another property that is common for both the *tracing system* and the HLT metric. Neither system records the event of a coroutine resuming itself. Therefore, the presented traces show only the uResume performed by mary::mem() called from uMain::main(), which transfers the thread from uMain task trace-line into the corouinte's trace-line, and the uSuspend statement, which transfers the thread back to its own trace line. The diagrams do not show the uResume statement executed by mary::mem() called from mary::main() and the uResume statement in mary::main() because these two statements switch the coroutine back to itself.

Switching the coroutine to itself is not useful nor does it occur frequently in programs but it is not precluded by the language semantics. However, generating an event for this esoteric case would have complicated the implementation substantially. Therefore, it was decided to ignore this case for event-tracing.

Chapter 9

Memory Usage Information Metric

This chapter describes the μProfiler built-in *Memory Usage Information* (MUI) metric.

In languages, like C++, with explicit dynamic storage-allocation (versus implicit garbage collection), there is the potential to make a number of memory allocation and deallocation errors. Some memory managers in the language run-time system provide no error-checking for performance reasons. However, it is possible to check for double free on the same block of memory or freeing a corrupted pointer. μC++ provides this kind of error-checking in its memory manager. Detecting these errors in μC++terminates a program producing an error-message, which can be used to quickly track the problem with print-statements and/or a debugger.

However, there is one kind of allocation problem that does not lead to program termination: a memory leak. A memory leak occurs when allocated storage is never freed. Finding memory leaks is difficult because there is usually no error to work from and the point of allocation is unknown; only if the memory leak is severe, such that it exhausts storage, does an error occur.

To help with memory leaks, the MUI metric collects data about each dynamic allocation (malloc(), calloc(), realloc(), operator new) and deallocation (free(), operator delete) of memory performed by a user program or by a library function, called directly or indirectly from the user program. This data is filtered to present only information about memory that has been dynamically allocated but not freed and information about attempts to free a NULL pointer, which are just ignored by a system deallocation routine without causing an error; however, freeing a NULL pointer is sometimes an indication of a problem.

The MUI metric and the error-checking in μC++ are not meant to provide exhaustive memory checks as it is done in *Purify* [?]. Instead, they provide a low-cost, simple mechanism to locate a significant number of memory problems. A tool such as *Purify* uses a very complex compiler/architecture-dependent code-rewriting method to check every memory read and write performed by an application. Such an aggressive approach does not work for concurrent systems like μC++.

9.1 Design

Figure 9.1 presents a main display for the MUI metric. The only deallocations presented are for a NULL pointer. The reminder of the discussion only refers to the allocations without matching deallocations and is sufficient to understand the simple case of freeing a NULL pointer.

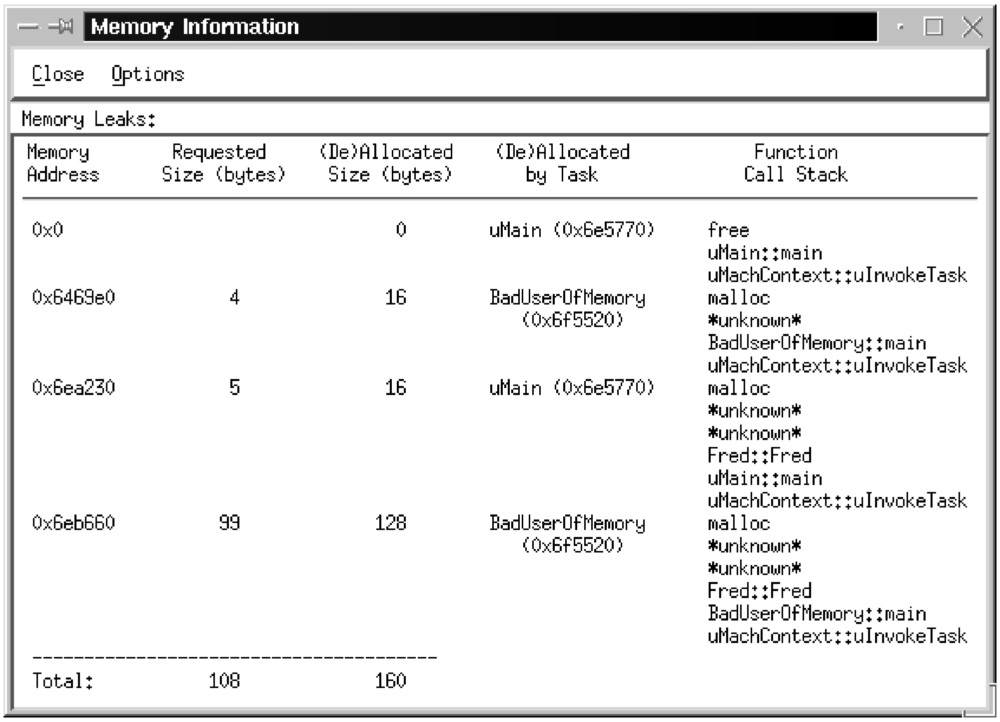


Figure 9.1: MUI Metric: Main Display

The first column in the display contains the start address of a memory allocation. The next two columns show the requested and actually allocated size; these two sizes are never identical because the allocated block includes additional data (*header*) required for memory management. The fourth column presents which task performed the allocation request. And the last column displays the call-stack at the time of the memory allocation. The call-stack is limited to sixteen functions for each memory allocation, which is adequate for most situations and bounds the cost for the metric.

The design of the MUI metric is presented in Figure 9.2. The notation used in this object-oriented analysis model is explained in Appendix A.

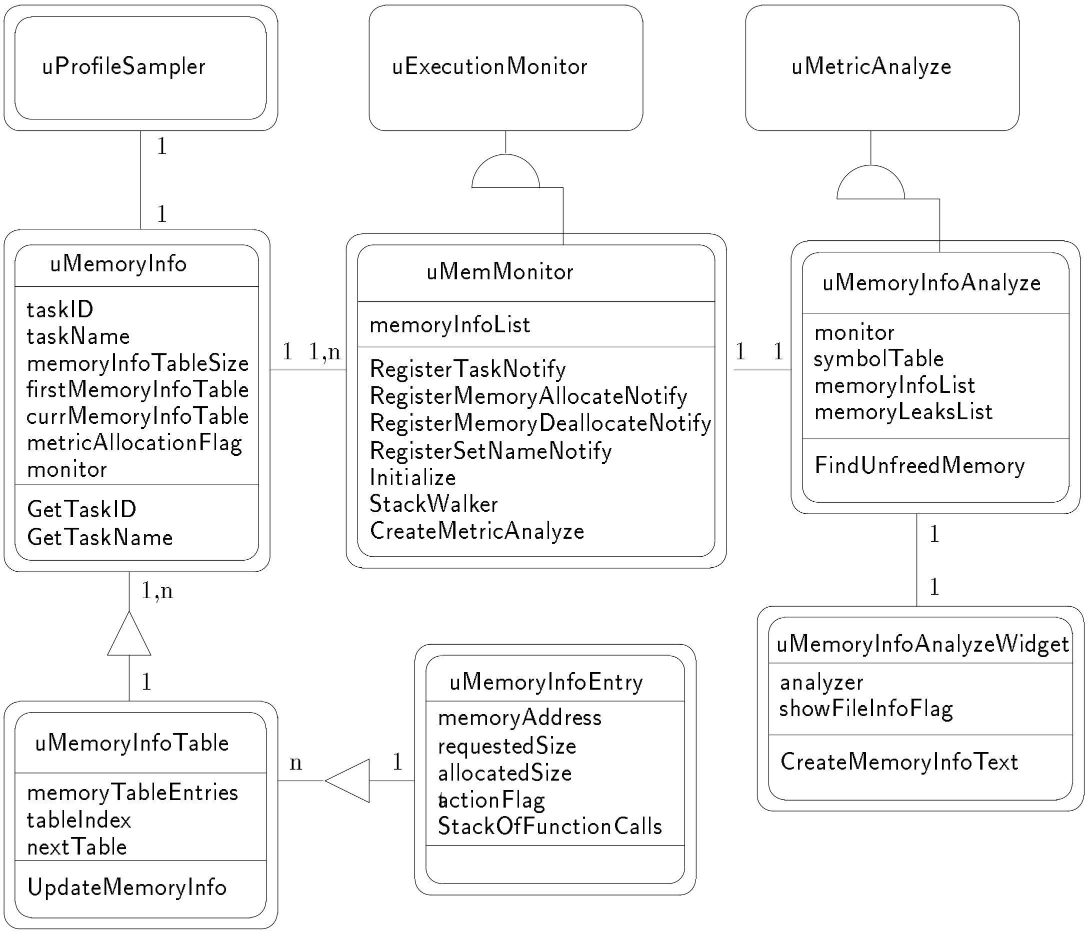


Figure 9.2: Object-Oriented Model of Memory Usage Information Metric

MUI uses decentralized monitoring; therefore, one uProfileSampler for each profiled task is responsible for recording data about each memory allocation and deallocation performed by the task, and storing it into uMemoryInfo. uMemoryInfoTable and uMemoryInfoEntry data structures inside the uMemoryInfo object are allocated in the same way as it is done in the EST metric (see Section 7.3). That is, large chunks of memory (an array of uMemoryInfoEntry), enough to store data for fifty profiled memory operations, are allocated at a time. After exhausting one array,

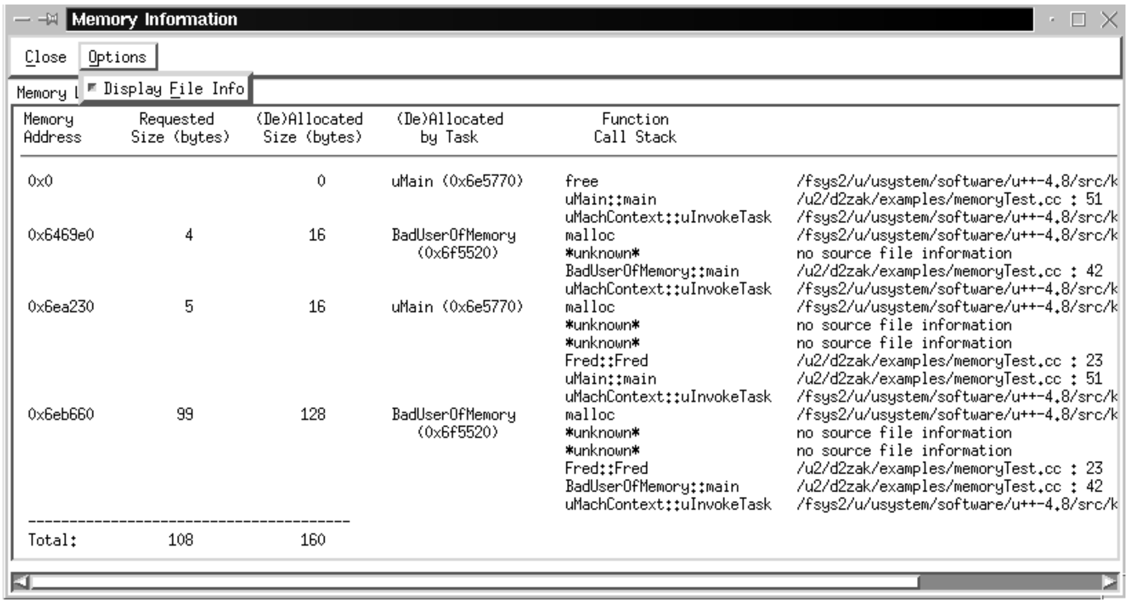


Figure 9.3: MUI Metric: Option Display

another is dynamically allocated and chained to the previous one.

uMemMonitor, derived from the μProfiler kernel's abstract-class, uExecutionMonitor, activates the necessary hooks and keeps a list of the uMemoryInfo objects to allow a metric analyzer to access the data after program execution.

A uMemoryInfoAnalyze analyzer, derived from uMetricAnalyze, goes through the data and tries to match each allocation request with a corresponding deallocation request. First, it looks for the matches within the data of a single task and transfers all unmatched entries into memoryLeaksList. Then, it makes a final sweep through the memoryLeaksList to see if some memory is allocated by one task and deallocated by another.

The memoryLeaksList is used by the uMemoryInfoAnalyzeWidget visualizer to display the memory leaks information on the screen (see Figure 9.1). The display contains an Options menu with a Display File Information button. Pressing the button adds an extra column to the display (see Figure 9.3) containing file and line number information for the presented functions.

9.2 Instrumentation Insertion

The MUI metric uses hooks inserted into the μC++ memory allocation and deallocation routines. The hooks are activated by RegisterMemoryAllocateNotify() and RegisterMemoryDeallocateNotify() routines.

RegisterTaskNotify() creates the uProfileSampler object, for each task register for profiling, and the uMemoryInfo data structures.

The job of RegisterTaskSetNameNotify() is the same as in all the previous metrics. It records the change of a task name when a user's program calls the uSetName() routine.

9.3 Implementation

There are two main issues in implementing the MUI metric. First is to avoid recording memory allocations done by the metric itself as well as by all other metrics, and the other is to get complete call-stack information.

MUI stores the memory data into an array. When a single array is full, MUI dynamically creates another one by calling operator new inside an UpdateMemoryInfo() function. Profiling this call to operator new would generate another call into UpdateMemoryInfo(). MUI would again detect that a new array must be created and would make another call to operator new, creating an infinite call-cycle. To avoid this situation, uMemoryInfo contains a metricAllocationFlag, which is turned on when MUI performs memory allocation to ensure that the operation is not registered.

The flag is also used to prevent recording of memory allocations performed by other μProfiler metrics, even though their operations do not cause recursion. These allocation are not profiled because the profiling data structures are used for analysis and are deleted when μProfiler closes (long after a user program finishes). But since the profiler collects data only during user program execution, all profiling data structures would be reported as allocated but not freed.

The other problem involves obtaining information about functions that lead to memory allocation or deallocation. Unfortunately, the profiling stack (see Section 5.5.3) contains only functions that are compiled with the -profile flag, and hence have calls to mcount() (see Section 5.4.2). Most libraries are precompiled without this flag, so their functions do not end up on the profiling stack. As a result, a dynamic (de)allocation inside a library function would get assigned the profiled function on the top of the profiling stack as the caller. Thus, users would be unsure whether the reported problem is really caused by their program or by a library routine. To avoid this confusion, MUI walks the task's execution stack to obtain the function addresses, which is architecture dependent. For example, walking the stack on SPARC machines requires flushing the register windows into a memory (trap 3), which is expensive. However, since the MUI metric is mostly used for debugging, the probe effect is usually acceptable.

An additional problem is that the MUI metric finds names only for statically-linked functions; dynamically-linked functions are reported as `*unknown*`. Finding names for dynamically-linked functions is not provided in the BFD (see Section 5.2.1) used for symbolic name lookup. However, most systems, including μC++, allow a program to be statically or dynamically linked. Therefore, programs should be statically linked to obtain maximum information from the MUI metric.

9.4 Validation

The validation has been performed by comparing expected results with the information reported by the MUI metric.

The test includes a simple program (source code in Appendix B.4) with only one task (uMain::main) that dynamically creates six objects without deleting them and also frees a NULL pointer. The objects are of the following types: char, int, float, double, Female and Male. The Female and Male objects dynamically allocate memory for their name variables without freeing it, so it also should be reported by the metric. Additionally, invoking a Male object involves creating a Female object, which in turn dynamically allocates memory for its name, generating another block of un-freed memory. Therefore, HLT should report memory leaks for nine blocks, plus it should also show the deallocation of a NULL pointer.

The estimation of the amount of a requested memory is performed by summing sizes of all variables inside the objects, using the following sizes (in bytes) for basic-types: char = 1, int, float, bool = 4, double = 8, pointer = 4.

The estimation of the allocated size is done by applying a formula used by μC++ memory management:

Object Name	Expected Sizes		MUI Sizes	
	Requested (bytes)	Allocated (bytes)	Requested (bytes)	Allocated (bytes)
letter	1	16	1	16
intNum	4	16	4	16
floatNum	4	16	4	16
doubleNum	8	16	4	16
littleMary	140	256	140	256
littleMary→name	50	64	50	64
bob	824	1024	824	1024
bob→name	18	32	18	32
bob→wife.name	18	32	18	32
nullPtr		0		0

Table 9.1: MUI: Test Results

$$allocatedSize = \lceil \log_2(requestedSize + header) \rceil$$

where *header* contains two pointers; therefore, its size is 8 bytes.

The expected and the actual results are shown in table 9.1. They are identical.

The memory addresses reported by the MUI metric are compared to the object addresses printed by the test program; they are the same.

The MUI function call-stack information is compared to the call-stack generated by the GDB debugger; both stacks are identical.

A separate program, which is not shown in this thesis, is used to verify that the metric correctly relates memory leaks to specific tasks.

Chapter 10

Conclusions and Future Work

This thesis focuses on profiling user-level threads in concurrent object-oriented programs running in shared-memory, uni-processor and multi-processor environments.

μProfiler is a performance-analysis tool implemented in μC++, a high-level concurrent object-oriented language, that makes the profiler extendible and portable. μProfiler is also intuitive and easy to use, which is important since most users are unwilling to spent much time and effort learning to use new performance tools.

10.1 Contributions

My work includes enhancements to the μProfiler kernel and implementation of four new metrics that help users to understand program run-time behaviour, and find program hot spots and bottlenecks.

This thesis contributes three major enhancements to the μProfiler kernel: decentralized monitoring, coroutine profiling and incorporating user-designed metrics.

In decentralized monitoring, each task is responsible for updating its own profiling data structures, eliminating the task's potentially blocking calls into μProfiler and, as a result, drastically reducing the profiling overhead. Furthermore, since only a single thread accesses a particular data structure, there is no need for mutual exclusion making the profiling process even less intrusive.

The initial μProfiler failed when a profiled task executed a coroutine. The failure occurred because the profiling stack was only implemented for tasks. When the task's thread switched to the coroutine's stack, it still tried to add information to the task's profiling stack, which caused a segmentation fault. To fix this problem, the profiling stack is now created for tasks and coroutines, and μProfiler has a mechanism to recognize when a thread is executing a coroutine, so it accesses the correct stack.

Users are allowed to create their own metrics using the instrumentation hooks provided by the μProfiler kernel. The metrics are incorporated into μProfiler without recompiling the profiler's code. To ensure that the process of adding a user metric works correctly, the `CG&RT` metric was originally implemented as a user metric. After testing, it has been converted into a built-in metric to take advantage of *decentralized monitoring*.

Currently, even the built-in metrics are treated by the μProfiler kernel as external metrics and are dynamically linked into the profiler. To make this possible, the μProfiler start-up process, and especially the creation of the start-up window, as well as the coupling between the μProfiler kernel and the metrics were redesigned and re-implemented.

Implementation of new metrics requires some programming effort, but it gives users the freedom to tailor μProfiler to better address their needs. For users that are not interested in creating their own metrics, μProfiler contains, at present, six built-in metrics to provide insights into program behaviour. Two of the metrics, *Performance Profile Information* and *UNIX Resources Usage Information*, were developed during initial implementation of the profiler [Den97] and altered during this work so that they function correctly with the redesigned kernel. Both metrics' visualizers are modified to better utilize the *Motif* resources, and the *UNIX Resources Usage Information* metric is updated to work with a new interface to the UNIX *process file system (proc)* [?]. The other four μProfiler's metrics, *Call Graph and Run Times, Execution State Transition, High Level Tracing* and *Memory Usage Information*, are outcomes of this thesis.

μProfiler's metrics describe various aspects of a program's run-time behaviour, giving users an opportunity to view their applications from different perspectives. It does not mean that a user must use all of the available metrics to profile an application. For one program, it may be enough to apply only one metric, whereas another program may require two or three metrics to find its problems or understand its behaviour. Since no single metric is able to answer all performance questions, a profiler needs to have several of them to successfully deal with diverse problems and behaviour in concurrent programs.

10.2 Future Work

The thesis shows that μProfiler can cooperate with an external visualization tool, i.e., POET. This idea could be taken further by allowing the profiler to save performance data into widely supported file formats, such as *Pablo SDDF*, that can be then read and visualized by other analysis tools.

Another idea, which also involves storing data into a file, is to extend μProfiler's functionality to permit comparison of performance data for several runs of the same program with different input data, changed system parameters or implemented code modifications.

Another field for future work is extending μProfiler to allow profiling of the μC++ run-time kernel. Right now, only inline kernel-functions, accessed by profiled tasks, are monitored because they contain calls to `mcount()`. It is also possible to profile the other MVD tools, KDB and SMART, or use μProfiler to profile itself, albeit with some restrictions to prevent recursive problems. Furthermore, the profiler needs to be ported to all the architectures supported by μC++.

Future work should also include: research on feasibility of dynamic instrumentation insertion, run-time analysis and visualization of performance data, a mechanism for finding names for dynamically-linked functions, and of course, the development and applicability of new metrics and suitable visualization techniques.

Appendix A

Object-Oriented Analysis & Design Notation

The notation used in this thesis is based on the object-oriented analysis and design notation provided by Peter Coad and Jill Nicola in [CN93]. It is also consistent with the notation used by Robert Denda, who did the initial work for the μProfiler [Den97]. The notation has been modified to show special objects in μC++ and to simplify the design overview. The explanation in this appendix targets people with at least a basic knowledge of the object-oriented paradigm.

Figure A.1 shows symbols used to depict classes and objects. The *Class-&-object* symbol represents a class and one or more objects of that class. The inner rounded rectangle denotes a class and the outer rectangle denotes an object or objects of that class. Below the name of the class are its attributes and member functions. Only attributes and members relevant to the design being described are shown. The *Class* symbol represents an abstract class, one that has no objects.

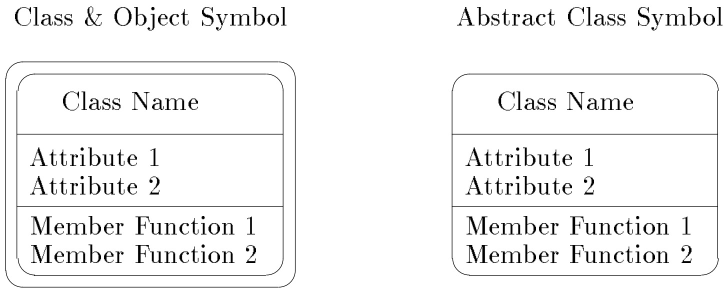


Figure A.1: Class and Object Notation

When the attributes and member functions are unimportant to the concept being explained or because they have been already listed in previous diagrams, a simplified notation containing only the class name is used. Figure A.2 shows the simplified symbols.

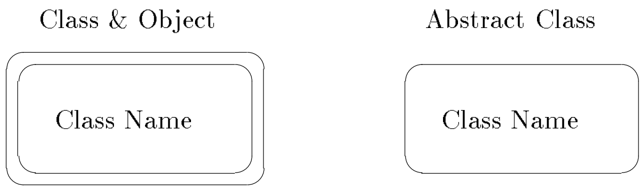


Figure A.2: Class and Object Simplified Notation

μC++ contains objects that have their own thread of control and execution state, i.e., tasks (see Section 4.1.3 for information about tasks). These "active" objects are represented by regular rectangles in which the space between the inner and outer rectangle is shaded. The *active object* symbol, which is not a part of Peter Coad's notation, is shown in Figure A.3.

Inheritance, or *generalization-specialization structure* as called in [CN93], is represented by a line with a semi-circle, drawn between classes. Inheritance is done among classes, not objects, and that is why the inner rectangles are connected in

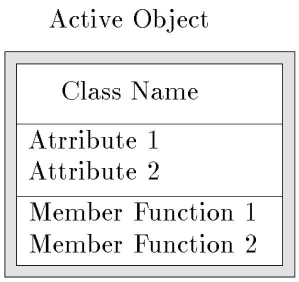


Figure A.3: Active Object Notation

Figure A.4. The derived (specialized) class inherits the attributes and member functions of the base (general) class, and it extends and modifies members to suit its own purposes. More than one specialized class can be derived from the general class.

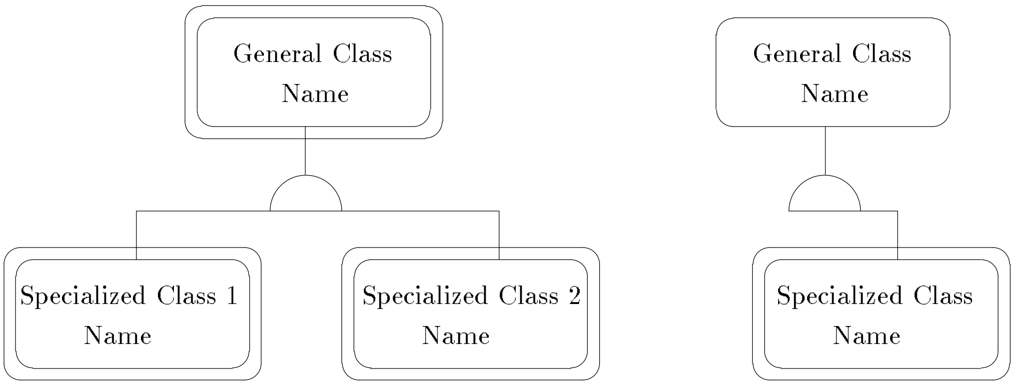


Figure A.4: Inheritance Notation

The notation in Figure A.5 represents object connection/association. The cardinality displayed on the line connecting the objects shows how many objects of one class are connected to how many objects of another class. In the given example, an object of *Class 2* knows about one or more, up to n, objects of *Class 1*. But each object of *Class 1* knows about exactly one object of *Class 2*.

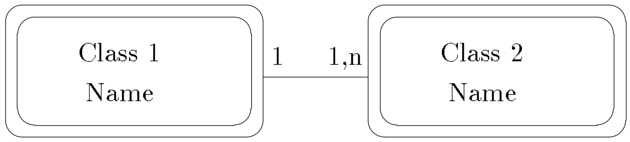


Figure A.5: Object Relationship Notation

In figure A.6, the line with a triangle on it is called a *whole-part structure* and represents the aggregation of objects. The triangle points toward the *Whole* object. The range or limit markings on the connecting line depict the relation between the objects. In the example, the *Whole* object can have between zero and n objects of *Part* class, and a *Part* object is an attribute of only one *Whole* object.

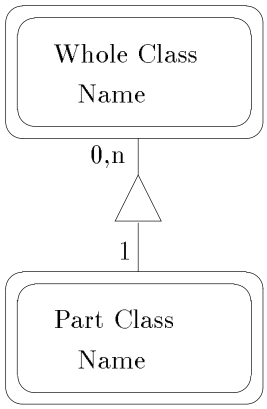


Figure A.6: Aggregation Notation

Appendix B

Test Programs

B.1 CG&RT Metric

B.1.1 C/μC++ Program

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

#define BIGLOOP     100000
#define SMALLLOOP   10

void first( void );
void second( void );
void third( void );
void fourth( void );
void fifth( void );
```

```
#ifdef __U_CPLUSPLUS__
void uMain::main ( void ) {
#else
void main ( void ) {
#endif
    for ( int i = 0; i < SMALLLOOP; i += 1 ) {
        first();
        for ( int j = 0; j < BIGLOOP; j += 1 ) {}
        second();
    } // for
} // main

void first( void ) {
    for ( int i = 0; i < SMALLLOOP; i += 1 ) {
        third();
        for ( int j = 0; j < BIGLOOP; j += 1 ) {}
        fourth();
    } // for
} // first

void second( void ) {
    for ( int j = 0; j < BIGLOOP ; j += 1 ) {}
} // second

void third( void ){
    for ( int i = 0; i < SMALLLOOP; i += 1 ) {
        fifth();
        for ( it j = 0; j < BIGLOOP; j += 1 ) {}
    } // for
} // third

void fourth( void ) {
    for ( int i = 0; i < SMALLLOOP; i += 1 ) {
        for ( int j = 0; j < BIGLOOP; j += 1 ) {}
    } // for
} // fourth

void fifth( void ) {
    for ( int j = 0; j < BIGLOOP ; j += 1 ) {}
} // fifth

// compile-command for C program: "g++ test1.c -o test1 -pg"
// compile-command for uC++ program: "setenv MVDPATH /u/usystem/software/MVD;
//                                    u++ -profile test1.cc"
```

B.1.2 μC++ Coroutine Program

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

uMutex uCoroutine fred {
    void main() {
        for ( ; ; ) {
            for ( int i = 1; i <= 45440; i += 1 ) {}            // loop for 2 msec
            uSuspend;
        } // for
    } // main
  public:
    void nextFred() {
        uResume;
        for ( int i = 1; i <= 17040; i += 1 ) {}                // loop for 0.75 msec
    } // nextFred
}; // fred

uMutex uCoroutine mary {
    void main() {
        for ( ; ; ) {
            for ( int i = 1; i <= 22720; i += 1 ) {}            // loop for 1 msec
            uSuspend;
        } // for
    } // main
  public:
    void goMary() {
        uResume;
        for ( int i = 1; i <= 11360; i += 1 ) {}                // loop for 0.5 msec
    } // mary::goMary
}; // mary

uTask Worker {
    fred &f;
    mary &m;

    void main() {
        for ( int i = 0; i < 5 ; i += 1 ) {
            f.nextFred();
            m.goMary();
        } // for
    } // main
  public:
    Worker( fred &f1, mary &f2 ) : f( f1 ), m( f2 ) {}          // Worker()
}; // Worker
```

```
void uMain::main() {
    const int NoOfWorkers = 10;
    fred f;
    mary m;                                             // create coroutines
    Worker *workers[ NoOfWorkers ];

    for ( int i = 0; i < NoOfWorkers; i += 1 ) {
        workers[ i ] = new Worker( f, m );              // create tasks
    } // for

    for ( int i = 0; i < NoOfWorkers; i += 1 ) {        // delete tasks
        delete workers[ i ];
    } // for
} // uMain::main

// compilation-comand: setenv MVDPATH /u/usystem/software/MVD;
//                          u++ -profile -multi coroutineTest.cc
```

B.2 EST Metric

```
#include <uC++.h>
#define BIGLOOP     5000000
#define SMALLLOOP     10000

uMonitor Bathroom {
    uCondition busy;
    int inBathroom;
  public:
    Bathroom() {
        inBathroom = 0;
    } // Bathroom

    void enter() {
         if ( inBathroom == 0 ) {
            inBathroom += 1;
            for( int i = 0; i < BIGLOOP; i += 1 ) {}
            inBathroom -= 1;
            uSignalBlock busy;
        } else {
            uWait busy;
         } // if
    } //enter
}; // Bathroom
```

```
uTask User {
    Bathroom &b;

    void main() {
        for ( int j = 0; j < 5; j += 1 ) {
            for ( int i = 0; i < SMALLLOOP; i += 1 ) {}
            b.enter();
        } // for
    } // main
  public:
    User( Bathroom &b ) : b( b ) {}
}; // User

void uMain::main() {
    const int NoOfUsers = 5;
    Bathroom b;
    User *users[ NoOfUsers ];

    for ( int i = 0; i < NoOfUsers; i += 1 ) {
        users[ i ] = new User( b );                    // create tasks
    } // for

    for ( int i = 0; i < NoOfUsers; i += 1 ) {         // delete tasks
        delete users[ i ];
    } // for
} // uMain::main

// compilation-comand: setenv MVDPATH /u/usystem/software/MVD;
//                     u++ -profile estTest.cc -multi
```

B.3 HLT Metric

B.3.1 Producer-Consumer Program: Semaphore Solution

Program written by Peter A. Buhr.

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

```
class buffer {
    int front, back;                  // position of front and back of queue
    int count;                        // number of used elements in the queue
    uSemaphore full, empty;           // synchronize for full and empty buffer
    int elems[ 5 ];
  public:
    buffer() : full( 0 ), empty( 5 ) { front = back = count = 0; }
    int query() { return count; }
    void insert( int elem ) {
        empty.uP();                   // wait if queue is full
        elems[ back ] = elem;
        back = ( back + 1 ) % 5;
        count += 1;
        full.uV();                    // signal a full queue space
    } // buffer::insert

    int remove() {
        int elem;

        full.uP();                    // wait if queue is empty
        elem = elems[ front ];
        front = ( front + 1 ) % 5;
        count -= 1;
        empty.uV();                   // signal empty queue space
        return( elem );
    } // buffer::remove
};

uTask prod {
    buffer &buf;                      // reference to shared buffer
    void main() {
        buf.query();                  // check status of buffer
        buf.insert( 3 );              // insert data
    }
  public:
    prod( buffer &b ) : buf( b ) {}
};

uTask cons {
    buffer &buf;                      // reference to shared buffer
    void main() {
        buf.remove();                 // remove data
    }
  public:
    cons( buffer &b ) : buf( b ) {}
};
```

```
void uMain::main() {
    buffer b;                          // create bounded buffer task
    cons c( b );                       // create consumer task
    prod p( b );                       // create producer task
}

// compile-command for tracing system: "u++ -trace prodCons.cc -o prodCons"
// compile-command for uProfiler: "setenv MVDPATH /u/usystem/software/MVD;
//                                 u++ -profile prodCons.cc"
```

B.3.2 Dating Service Program - External Scheduling

Program written by Peter A. Buhr and Richard A. Stroobosscher.

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

uTask DatingService {
    int GirlPhoneNo, BoyPhoneNo;
    void main();
  public:
    DatingService() {
        GirlPhoneNo = BoyPhoneNo = -1;
    }; // DatingService::DatingService
    int Girl( int PhoneNo );
    int Boy( int PhoneNo );
}; // DatingService

void DatingService::main() {
    for ( ; ; ) {
        uAccept( ~DatingService )
            break;
        uOr uAccept( Girl );
        uOr uAccept( Boy );
        // do other work
    } // for
} // DatingService::main

int DatingService::Girl( int PhoneNo ) {
    GirlPhoneNo = PhoneNo;
    if ( BoyPhoneNo == -1 ) uAccept( Boy );
    int temp = BoyPhoneNo;
    BoyPhoneNo = -1;
    return temp;
} // DatingService::Girl
```

```
int DatingService::Boy( int PhoneNo ) {
    BoyPhoneNo = PhoneNo;
    if ( GirlPhoneNo == -1 ) uAccept( Girl );
    int temp = GirlPhoneNo;
    GirlPhoneNo = -1;
    return temp;
} // DatingService::Boy

uTask Girl {
    DatingService &TheExchange;

    void main() {
        uYield( rand() % 100 );          // don´t all start at the same time
        int PhoneNo = rand() % 10000000;
        int partner = TheExchange.Girl( PhoneNo );
    } // main
  public:
    Girl( DatingService &TheExchange ) : TheExchange( TheExchange ) {} // Girl
}; // Girl

uTask Boy {
    DatingService &TheExchange;

    void main() {
        uYield( rand() % 100 );          // don´t all start at the same time
        int PhoneNo = rand() % 10000000;
        int partner = TheExchange.Boy( PhoneNo );
    } // main
  public:
    Boy( DatingService &TheExchange ) : TheExchange( TheExchange ) {} // Boy
}; // Boy

void uMain::main() {
    DatingService TheExchange;
    Girl *girls;
    Boy  *boys;

    girls = new Girl[3]( TheExchange );
    boys  = new Boy[3]( TheExchange );

    delete [ ] girls;
    delete [ ] boys;
} // uMain::main
```

B.3.3 Producer-Consumer Program: Coroutine Solution

Program written by Peter A. Buhr.

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

uCoroutine cons {
    int elem;

    void main() { /* consume */ }
  public:
    void delivery( int e ) {
        elem = e;
        uResume;                        // restart cons::main
    } // cons::delivery
};

uCoroutine prod {
    cons &c;

    void main() { c.delivery( 5 ); }
  public:
    prod( cons &c ) : c( c ) {}
    void start() { uResume; }           // restart prod::main
};

void uMain::main() {
    cons c;                             // create consumer
    prod p( c );                        // create producer
    p.start();
} // uMain::main
```

B.3.4 Fibonacci Number Generator Program

Program written by Peter A. Buhr and Richard A. Stroobosscher

```
#include <uC++.h>
#include <uIOStream.h>
#include <unistd.h>                     // getpid
```

```
uMutex uCoroutine fibonacci {
    int fn, fn1, fn2;

    void main() {
        fn = 1;                         // 1st case
        fn1 = fn;
        uSuspend;
        fn = 1;                         // 2nd case
        fn2 = fn1;
        fn1 = fn;
        uSuspend;
        for ( ; ; ) {                   // general case
            fn = fn1 + fn2;
            fn2 = fn1;
            fn1 = fn;
            uSuspend;
        } // for
    } // fibonacci::main
  public:
    int next() {
        uResume;
        return fn;
    } // fibonacci::next
}; // fibonacci

uTask Worker {
    fibonacci &f1, &f2;
    int n1, n2;

    void main() {
        n1 = f1.next();
        n2 = f2.next();
    } // Worker::main
  public:
    Worker( fibonacci &f1, fibonacci &f2 ) : f1( f1 ), f2( f2 ) {} // Worker::Worker
}; // Worker

void uMain::main() {
    const int NoOfWorkers = 4;
    Worker *workers[ NoOfWorkers ];
    fibonacci f1, f2;                   // create fibonacci generator

    int i;
    for ( i = 0; i < NoOfWorkers; i += 1 ) {
        workers[ i ] = new Worker( f1, f2 );
    } // for
    for ( i = 0; i < NoOfWorkers; i += 1 ) {
        delete workers[ i ];
    } // for
} // uMain::main
```

B.3.5 Dating Service Program - Internal Scheduling

Program written by Peter A. Buhr and Richard A. Stroobosscher

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

uMonitor DatingService {
    int GirlPhoneNo, BoyPhoneNo;
    uCondition GirlWaiting, BoyWaiting;
  public:
    int Girl( int PhoneNo ) {
        if ( BoyWaiting.uEmpty() ) {
            uWait GirlWaiting;
            GirlPhoneNo = PhoneNo;
        } else {
            GirlPhoneNo = PhoneNo;
            uSignal BoyWaiting;
        } // if
        return BoyPhoneNo;
    } // DatingService::Girl

    int Boy( int PhoneNo ) {
        if ( GirlWaiting.uEmpty() ) {
            uWait BoyWaiting;
            BoyPhoneNo = PhoneNo;
        } else {
            BoyPhoneNo = PhoneNo;
            uSignal GirlWaiting;
        } // if
        return GirlPhoneNo;
    } // DatingService::Boy
}; // DatingService

uTask Girl {
    DatingService &TheExchange;

    void main() {
        uYield( rand() % 100 );            // don't all start at the same time
        int PhoneNo = rand() % 10000000;
        int partner = TheExchange.Girl( PhoneNo );
    } // main
  public:
    Girl( DatingService &TheExchange ) : TheExchange( TheExchange ) {} // Girl
}; // Girl
```

```
uTask Boy {
    DatingService &TheExchange;

    void main() {
        uYield( rand() % 100 );            // don't all start at the same time
        int PhoneNo = rand() % 10000000;
        int partner = TheExchange.Boy( PhoneNo );
    } // main
  public:
   Boy( DatingService &TheExchange ) : TheExchange( TheExchange ) {} // Boy
}; // Boy

void uMain::main() {
    DatingService TheExchange;
    Girl *girls;
    Boy  *boys;

    girls = new Girl[ 3 ]( TheExchange );
    boys  = new Boy[ 3 ]( TheExchange );

    delete [ ] girls;
    delete [ ] boys;
} // uMain::main
```

B.3.6 Task Calling Its Mutex Member

Program written by Peter A. Buhr.

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

uCoroutine mary {
    void main() {
        mem();
        uResume;                           // resumes itself
        uSuspend;
    }
  public:
    void mem() {
        uResume;
    } // mary::mem
}; // mary
```

```
uTask fred {
  public:
    void mem() {
    }
  private:
    void main() {
        mem();                          // call to its member
        mem();                          // call to its member
        uAccept( mem );                 // call from another task
    } // fred::main
}; // fred

void uMain::main() {
    mary m;
    m.mem();                            // start coroutine
    fred f;
    f.mem();
} // uMain::main
```

B.4 MUI Metric

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

class Female {
    int children;
    int parents;
    char childrenSchool[ 128 ];
  public:
    char *name;

    Female( char *n ) {
        name = new char[ strlen(n) + 1 ];       // space for \0
        strcpy(name, n);
    } // Female::Female
}; // Female
```

```
class Male {
    char house[800];
    double bestScore;
  public:
    char *name;
    Female wife;

    Male( char *n, Female woman ) : wife( woman ) {
        name = new char[strlen(n) + 1];           // space for \0
        strcpy(name, n);
    } // Male::Male
}; // Male

void uMain::main() {
    char *letter = new char[1];
    int *intNum = new int(2);
    float *floatNum = new float(1.3);
    double *doubleNum = new double(5.678);
    Female *littleMary = new Female(
                "Maria Anna Smith-Tompson with very long last name");
    Male *bob = new Male("Bob Zachary Brown", Female("Stacy Tracy Black"));

    uCout << "letter " << (void *)letter << endl;
    uCout << "intNum " << (void *)intNum << endl;
    uCout << "floatNum " << (void *)floatNum << endl;
    uCout << "doubleNum " << (void *)doubleNum << endl;
    uCout << "littleMary " << (void *)littleMary << endl;
    uCout << "littleMary's name " << (void *)littleMary->name << endl;
    uCout << "bob " << (void *)bob << endl;
    uCout << "bob's name " << (void *)bob->name << endl;
    uCout << "bob's wife name " << (void *)bob->wife.name << endl;

    char *nullPtr = NULL;                         // freeing NULL pointer
    free(nulPtr);
} // uMain::main()

// compile-command for uProfiler: "setenv MVDPATH /u/usystem/software/MVD;
//                                 u++ -profile memMetricTest.cc"
```

Bibliography

[AL90] Thomas E. Anderson and Edward D. Lazawska. Quartz: A Tool for Tuning Parallel Program Performance. In *Proceedings of the 1990 SIGMETRICS Conference on Measuremnt and Modeling of Computer Systems. Boston, USA*, volume 1, pages 115–125, May 1990.

[Ayd00] Ruth A. Aydt. *The Pablo Self-Defining Data Format.* Pablo Research Group, Department of Computer Science, University of Illinois, Urbana, Illinois 61801, February 2000. URL: http://www-pablo.cs.uiuc.edu/Publications/.

[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, Portland, Oregon, USA*, pages 588–597, November 1993.

[BBG+94] F. Bodin, P. Beckman, D. Gannon, S. Srinivas, and B. Winnicka. Sage++: An Object-Oriented Toolkit and Class Library for Building Fortran and C++ Restructuring Tools. In *Proceedings OONSKI '94*, 1994.

[BDL] Shirley Browne, Jack Dongarra, and Kevin London. Review of Performance Analysis Tools for MPI Parallel Programs. Technical report, Computer Science Department, University of Tennessee, Knoxville, TN 37996-1301. URL: http://www.cs.utk.edu/ browne/perftools-review/.

[Bez93] James C. Bezdek. Fuzzy Models - What Are They, and Why? In *IEEE Transactions on Fuzzy Systems*, volume 1, pages 1–6, 1993.

[BFC95] Peter A. Buhr, Michel Fortier, and Michael H. Coffin. Monitor Classification. *ACM Computing Surveys*, 27(1):63–107, March 1995.

[BHMM94] Darryl Brown, Steven Hackstadt, Allen Malony, and Bernd Mohr. Program Analysis Environment for Parallel Language Systems: The TAU Environment. In *Proceedings of the Workshop on Environments and Tools For Parallel Scientific Computing, Townsend, Tennessee*, pages 162–171, May 1994.

[BKS96] Peter A. Buhr, Martin Karsten, and Jun Shih. KDB: A Multi-threaded Debugger for Multi-threaded Applications. In *Proceedings of SPDT'96: SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 80–87, Philadelphia, Pennsylvania, USA, May 1996. ACM Press.

[BMR99a] A. Bakic, M. W. Mutka, and D. T. Rover. BRISK: A Portable and Flexible Distributed Instrumentation System. In *Proceedings of the IEEE IPPS/SPDP '99, International Parallel Processing Symposium on Parallel and Distributed Processing*, pages 387–391, April 1999. URL: http://-www.cse.msu.edu/ mutka/research.html.

[BMR99b] A. Bakic, M. W. Mutka, and D. T. Rover. An On-Line Performance Visualization Technology. In *Proceedings of the IEEE Heterogeneous Computing Workshop*, pages 47–59, April 1999.

[BS99] Peter A. Buhr and Richard A. Stroobosscher. μC++ Annotated Reference Manual, Version 4.7. Technical report, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, December 1999. ftp://plg.uwaterloo.ca/pub/MVD/uC++.ps.gz.

[Buh95] Peter A. Buhr. μC++ Monitoring and Visualization Annotated Reference Manual, Version 1.1. Technical report, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, 1995. ftp://-plg.uwaterloo.ca/pub/MVD/Visualization.ps.gz.

[Cha91] Steve Chamberlain. *The Binary File Descriptor Library.* Cygnus Support, 1 edition, April 1991.

[CMW00] Harold W. Cain, Barton P. Miller, and Brain J.N. Wylie. A Callgraph-Based Search Strategy for Automated Performance Diagnosis. Technical report, Computer Science Department, University of Wisconsin-Madison, Dayton Street, Madison, WI 53706-1685, USA, February 2000.

[CN93] Peter Coad and Jill Nicola. *Object-Oriented Programming.* Prentice Hall Inc., Englewood Cliffs, New Jesrsey 07632, USA, second edition, February 1993.

[Den97] Robert R. Denda. Profiling Concurrent Programs. Diplomarbeit, Universität Mannheim, Mannheim, Deutschland, September 1997. ftp://-plg.uwaterloo.ca/pub/MVD/DendaThesis.ps.gz.

[DR99] Luiz DeRose and Daniel A. Reed. SvPablo: A Multi-Language Architecture-IndependentPerformance Analysis System. In *Proceedings of the Inter-*

national Conference on Parallel Processing (ICPP'99), Fukushima, Japan, September 1999.

[ES95] Alan Eustace and Amitabh Srivastava. ATOM: A Flexible Interface for Building High Performance Program Analysis Tools. In *Proceedings of the Winter 1995 USENIX Conference*, January 1995. URL: http://www.research.digital.com/wrl/projects/om/wrlpapers.html.

[ES98] Greg Eisenhauer and Karsten Schwan. An Object-based Infrastructure for Program Monitoring and Steering. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools, Welches, OR USA*, pages 10–20, August 1998.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley Pub. Co., second edition, April 2000. URL: http://java.sun.com/docs/books/jls/index.html.

[GKM82] Susan L. Graham, Peter B. Kesseler, and Marshall K. McKusick. gprof: a Call Graph Execution Profiler. In *Proceedings of the SIGPLAN'82 Symposium on Compiler Construction, Jun 23-25, Boston Massachusetts, USA*, volume 17 of *SIGPLAN Notices*, pages 120–126, June 1982.

[GL99] William Gropp and Ewing Lusk. *User's Guide for mpich, a Portable Implementation of MPIVersion 1.2.0*. Argonne National Laboratory, 9700 S. Cass Avenue, Argonne, IL 60439, December 1999. URL: http://www-unix.mcs.anl.gov/mpi/mpich/.

[HCHP92] Timothy J. Hickey, Jacques Cohen, Hirofumi Hotta, and Thierry PetitJean.

Computer-Assisted Microanalysis of Parallel Programs. *ACM Transactions on Programming Languages and Systems*, 14(1):54–106, 1992.

[Hea94] M. T. Heath. Performance Visualization with ParaGraph. In *Proceedings of the Second Workshop on Environments and Tools for Parallel Scientific Computing*, pages 221–230, 1994.

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

[HL91] Virginia Herrarte and Ewing Lusk. Studying Parallel Program behaviour with Upshot. Technical report, Argonne National Laboratory, 9700 S. Cass Avenue, Argonne, IL 60439, 1991.

[HM92] Jeffrey K. Hollingsworth and Barton P. Miller. Parallel Program Performance Metrics: A Comparison and Validation. Technical report, Computer Science Department, University of Maryland, College Park, MD 20902, 1992.

[HM94] Jeffrey K. Hollingsworth and Barton P. Miller. Slack: A Performance Metric for Parallel Programs. Technical Report CS-TR-95-1260, Computer Science Department, University of Maryland, College Park, MD 20902, December 1994.

[HMG+97] Jeffrey K. Hollingsworth, Barton P. Miller, Marcelo J. R. Gonalves, Oscar Naim, Zhichen Xu, and Ling Zheng. MDL: A Language and Compiler for Dynamic Program Instrumentation. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques. San Francisco, California, USA*, November 1997.

[HMR98] M. T. Heath, A. D. Malony, and D. T. Rover. Visualization for Parallel Performance Evaluation and Optimization. *Software Visualization*, pages 347–365, 1998.

[HP98] Anna Hondroudakis and Rob Procter. An Empirically derived Framework for Classifying Parallel Program Performance Tuning Problems. In *Proceedings of the SIGMETRICS Symposium on Parallel and distributed Tools, Welches, OR USA*, pages 112–123, August 1998.

[JWL98] Minwen Ji, Edward W.Felten, and Kai Li. Performance Measurements for Multithreaded Programs. In *Proceedings of the Joint International Conference on Measurement and Modeling of computer Systems. Madison, WI USA*, pages 161–170, June 1998.

[JZTB98] Clinton Jeffery, Wenyi Zhou, Kevin Templer, and Michael Brazell. A Lightweight Architecture for Program execution Monitoring. In *ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 67–74, 1998.

[KM99] Karen L. Karavanic and Barton P. Miller. Improving Online Performance Diagnosis by the Use of Historical Performance Data. In *SC'99, Portland, Oregon, USA*, November 1999.

[KMLM97] Karen L. Karavanic, Jussi Myllymaki, Miron Livny, and Barton P. Miller. Integrated Visualization of Parallel Program Performance Data. *Parallel Computing*, 23, 1997. Special issue on environments and tools for parallel scientific computing.

[LJI+98] Cheng Liao, Dongming Jiang, Liviu Iftode, Margaret Martonosi, and Douglas W. Clark. Monitoring Shared Virtual Memory Performance on a Myrinet-based PC Cluster. In *Proceedings of the 1998 International Conference on Supercomputing*, pages 251 – 258, 1998.

[MAS] J. McCormanck, P. Asente, and R. R. Swick. X Toolkit Intrinsics - C language X Interface. electronic document.

[MBM94] Bernd Mohr, Darryl Brown, and Allen Malony. TAU: A Portable Parallel Program Analysis Environment for pC++. In *Proceedings of CONPAR 94 - VAAP VI, Linz, Austria, Springer Verlag, LNCS 854*, pages 29–40, September 1994.

[MCH+90] B.P. Miller, M. Clark, J.K. Hollingsworth, S. Kierstead, S. Lim, and T. Torzewski. IPS-2: The Second Generation of a Parallel Program Measuremnt System. In *IEEE Transactions on Parallel and Distributed Systems*, volume 1, pages 206–217, April 1990.

[MCI+95] Barton P. Miller, Jonathan M. Cargille, R. Bruce Irvin, Krishna Kunchithapadam, Mark D. Callaghan, Jeffery K. Hollingsworth, Karen L. Karavanic, and Tia Newhall. The Paradyn Parallel Performance Measuremnt Tools. *IEEE Computer (Special issue on performance evaluation tools for parallel and distributed computer systems)*, 28(11):37–46, November 1995. URL: http://www.cs.wisc.edu/paradyn/papers/index.html.

[MMB+94] A. Malony, B. Mohr, P. Beckman, D. Gannon, S. Yang, and F. Bodin. Performance Analysis of pC++: A Portable Data-Parallel Programming System for Scalable Parallel Computers. In *Proceedings of 8th International Parallel*

Processing Symposium (IPPS), Cancun, Mexico, IEEE, pages 75–85, April 1994.

[MMC96] B. Mohr, A. Malony, and J. Cuny. *Parallel Programming using C++*, chapter 15: TAU. M.I.T. Press, Cambridge, Mass, 1996.

[MMS95] Bernd Mohr, Allen Malony, and Kesavan Shanmugam. Speedy: An Integrated Performance Extrapolation Tool for pC++ Programs. In *Proceedings of the Joint Conference PERFORMANCE TOOLS'95 and MMB'95, 20-22 September, Heidelberg, Germany*, May 1995.

[Moh93] Bernd Mohr. A Portable Dynamic Profiler for C++ based Languages. *Scientific Programming*, 2(3), September 1993.

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

[NA94] Wolfgang E. Nagel and Alfred Arnold. Performance Visualization of Parallel Programs - The PARvis Environment -. Technical report, Research Centre Jlich (KFA), Central Institute for Applied Mathematics (ZAM), D-52425 Julich, Germany, June 1994. URL: http://www.ccsf.caltech.edu/ nagel/PARvis/PARvis.html.

[New99] Tia Newhall. *Performance Measurement of Interpreted, Just-in-Time compiled, and Dynamically Compiled Executions.* PhD thesis, University of Wisconsin, Madison, August 1999.

[Nye93] Adrian Nye, editor. *Xlib Programming Manual for Version 11 of the X Window System.* O'Reilly & Associates, Inc., third edition, 1993.

[Osa98] Noritaka Osawa. An enhanced 3-D Animation Tool for Performance Tuning of Parallel Programs Based on Dynamic Models. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools, Welches, OR USA*, pages 72–80, August 1998.

[Pal98] Pallas GmbH, Hermulheimer St. 10, D-50321 Bruhl, Germany. *Vampir 2.0 Tutorial*, December 1998. URL: http://www.pallas.com.

[pix00] *SpeedShop User's Guide*, chapter 8: Using SpeedShop in Expert Mode: pixie. SGI Technical Publications, January 2000. URL: http://techpubs.sgi.com/.

[RPF+99] Daniel A. Reed, David A. Padua, Ian T. Foster, Dennis B. Gannon, and Barton P. Miller. Delphi: An Integrated, Language-Directed Performance Prediction, Measurement, and Analysis Environment. In *Frontiers '99: The 9th Symposium on the Frontiers of Massively Parallel Computation, Annapolis, MD*, February 1999.

[RVS+99] Randy L. Ribler, Jeffrey S. Vetter, Huseyin Simitci, , and Daniel A. Reed. Autopilot: Adaptive Control of Distributed Applications. In *Proceedings of the 7th IEEE Symposium on High-Performance Distributed Computing, Chicago, IL*, July 1999.

[Sch99] Oliver Schuster. Replay of Concurrent Shared-Memory Programs. Diplomarbeit, Universität Mannheim, Mannheim, Deutschland, April 1999. URL: http://www.plg.uwaterloo.ca:80/ usystem/MVD.html.

[She99] S. Shende. Profiling and Tracing in Linux. In *Proceedings of the Extreme Linux Workshop Number 2, USENIX, Monterey CA, USA*, June 1999.

[SMC+98] S. Shende, A. D. Malony, J. Cuny, K. Lindlan, P. Beckman, and S. Karmesin. Portable Profiling and Tracing for Parallel Scientific Applications using C++. In *Proceedings of SPDT'98: ACM SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 134–145, August 1998.

[SMS99] T. Sheehan, A. Malony, and S. Shende. A Runtime Monitoring Framework for the TAU Profiling System. In *Proceedings of the Third International Symposium on Computing in Object-Oriented Parallel Environments (ISCOPE'99), San Francisco, CA, USA*, December 1999.

[SPF+99] X.H. Sun, M. Pantano, T. Fahringer, , and Z. Zhan. SCALA: A Framework for Performance Evaluation of Scalable Computing. In *Proc. of the 4-th Workshop on High-Level Parallel Programming Models & Supportive Environments, in Lecture Notes in Computer Science*, volume 1586, April 1999.

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

[SWSR99] Eric Shaffer, Shannon Whitmore, Benjamin Schaeffer, and Daniel A. Reed. Virtue: Immersive Performance Visualization of Parallel and Distributed Applications,. *IEEE Computer Visualization*, pages 44–51, December 1999.

[Tay95a] David Taylor. Adapting the Partial-Order Event Tracer to a New Target Environment. Technical report, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, 1995.

[Tay95b] David Taylor. Event Displays for Debugging and Managing Distributed Systems. In *International Workshop on Network and Systems Management, Kyongju, Korea*, pages 112–124, August 1995.

[Tay95c] David Taylor. Partial-Order Event Tracer - User Documentation. Technical report, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, 1995.

[Tay95d] David Taylor. Partial-Order Event Tracer: Structure and Operation. Technical report, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, 1995.

[TB96] David Taylor and Peter A. Buhr. POET with μC++. Technical report, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, December 1996. ftp://plg.uwaterloo.ca/pub/MVD/-Poet.ps.gz.

[TM99] Ariel Tamches and Barton P. Miller. Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels. In *Third Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, USA*, February 1999.

[Tuf84] Edward R. Tufte. *The Visual Display of Quantitative Information.* Graphics Press, Box 430, Cheshire, Connecticut 06410, USA, second edition, January 1984.

[U.S94] U.S. Government. *ADA Reference Manual. Language and Standard Libraries*, December 1994. International Standard ISO/IEC 8652:1995(E).

[Wor92] P. H. Worley. A new PICL Trace File Format. Technical report, Oak Ridge National Laboratory, Mathematical Sciences Section, P.O. Box 2008, Bldg 6012, Oak Ridge, TN 37831-6367, September 1992.

[Wor00] P. H. Worley. MPICL - Portable Instrumentation Library. Web Page, URL: http://www.epm.ornl.gov/picl/, May 2000.

[WRM+97] A. Waheed, D. T. Rover, M. W. Mutka, H. Smith, and A. Bakic. Modeling, Evaluation, and Adaptive Control of an Instrumentation System. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium (RTAS'97)*, pages 100–110, June 1997.

[WTL98] Roland Wismuller, Jog Trinitis, and Thomas Ludwig. OCM - A Monitoring System for Interoperable Tools. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools, Welches, OR USA*, page 149, August 1998.

[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. May 4-6, Atlanta, GA USA*, pages 49–59, May 1999.

[YM88] C. Yang and B.P. Miller. Critical Path Analysis for the Execution of Parallel and Distributed Programs. In *Proceedings of 8th Int'l Conf. on Distributed Systems, San Jose, California, USA*, June 1988.

[YMG99] Jerry Yan, Lauren Mariani, and Ana Grady. Tracefile Animation and Analysis Web Page. URL: http://www.nas.nasa.gov/Groups/Tools/-AIMS/vis_anlLM.html, August 1999.

[YS96] C. Yan and S. R. Sarukkai. Analyzing Parallel Program Performance Using Normalized Performance Indices and Trace Transformation Techniques. *Parallel Computing*, 22(9):1215–1237, November 1996.

[YSM95] C. Yan, S. R. Sarukkai, and P. Mehra. Performance Measurement, Visualization and Modeling of Parallel and Distributed Programs using the AIMS Toolkit. *Software Practice & Experience*, 25(4):429–461, April 1995.

[ZML99] Victor C. Zandy, Barton P. Miller, and Miron Livny. Process Hijacking. In *The Eighth IEEE International Symposium on High Performance Distributed Computing (HPDC'99), Redondo Beach, California*, pages 177–184, August 1999.