Profiling Concurrent Programs

Diplomarbeit

von

Robert R. Denda

aus

Emmelshausen

vorgelegt am

Lehrstuhl für Praktische Informatik IV

Prof. Dr. Effelsberg

Fakultät für Mathematik und Informatik

Universität Mannheim

September 1997

Betreuer: Prof. Dr. Wolfgang Effelsberg

Acknowledgements

The underlying work of this thesis was done during multiple sojourns at the University of Waterloo, Canada, where I was provided with an excellent working environment.

First of all, I would like to give a special note of thanks to Professor Dr. Peter Buhr for his splendid assistance and guidance throughout my work on this thesis. It was mainly due to his efforts that my work was very enjoyable.

Furthermore, I would like to thank Professor Dr. Wolfgang Effelsberg as my supervisor at the University of Mannheim, Germany, for his patience in supporting a remotely written Diplomarbeit.

I also wish to thank Professor Dr. Stephen Mann and Professor Dr. Thomas Kunz for many valuable suggestions and helpful comments.

"Danke" goes to my parents and family for making this stay possible and being a great source of support and non-technical help.

Last but also foremost, I thank all my Canadian, German and Spanish friends for a wonderful time in Canada and for providing me with the necessary distraction from work. In particular, I wish to thank my friends Bárbara Eizaga Rebollar, Allan Baril, Marcus Michael Burth, P. Angelika Engelmann, Ian Hayes, Jürgen Heilig, Dr. José "Capullo" Medina, and Jim "Bosius" Slezak.

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Diplomarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Waterloo, den 10. September 1997 Robert Denda

Abstract

The larger and more complex a program becomes, the greater the need to understand its dynamic behaviour, both to locate problems and to optimize performance.

One of the most important tools for locating dynamic problems and performance bottlenecks is a *profiler*. A profiler monitors a program's dynamic behaviour and reveals information about the program's execution, possibly at multiple levels of abstraction. For example, one essential step in performance optimization is to detect a program's "hot spots" for potential optimization. Detecting these spots is usually non-trivial and time consuming without appropriate performance tools.

Concurrency adds substantially to the complexity of a program. Profiling concurrent programs entails many problems not present in sequential programs, such as thread communication and access to shared resources. A profiler has to deal with these multiple executing threads of control, all potentially introducing errors and performance problems.

In this thesis, the important aspects and problems of profiling are discussed, and an implementation of a prototype profiler called μProfiler for the μC++ thread library is presented, which provides multiple performance metrics on a per-thread basis in a shared-memory concurrent environment.

Trademarks

Ada is a registered trademark of 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 Micosystems, 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 of Unix System Labratories, Inc.

X Window System is a registered trademark of X Consortium, Inc.

Abbreviations

CISC	Complex Instruction Set Computer
CPU	Central Processing Unit
I/O	Input/Output
KDB	Kalli's DeBugger
MVD	μC++ Monitoring Visualization and Debugging
POET	Partial-Order Event Tracer
RISC	Reduced Instruction Set Computer

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The primary motivations for writing concurrent programs are two-fold. First, dividing a problem into multiple executing tasks may be a natural way of describing it. Second, it is often possible to reduce a program's execution-time on machines with multiple processors.

Many programming languages come with concurrent language primitives. *Ada* [76] and *Java* [21] are probably the most popular examples. Other, originally non-concurrent, programming languages invoke external facilities such as thread libraries to introduce concurrency.

Concurrent programs have a higher potential for performance problems than their sequential counterparts. Each executing thread can potentially experience all the performance problems of a sequential program. In addition, there are problems that occur because of the basic nature of concurrency among threads. Concurrent programs require mechanisms for synchronization, communication, protection of critical sections, etc. The introduction of these mechanisms causes an unavoidable overhead and can have a high impact on a program's performance.

The type of performance optimization also depends on the underlying structure of the concurrent programming language primitives. It makes a difference whether shared or distributed memory is used, and whether different threads of control are created at the operating system or user-level.

Nevertheless, given the right tools, potential bottlenecks of concurrent applications can often be located and eliminated with only little effort. This process of performance optimization encompasses different aspects and involves the application of several methodologies, the main one being profiling.

In addition to performance optimization, profiling serves many other purposes: for example, it can help understand the underlying concurrent algorithm of a program, help discover faulty execution behaviour, and thereby, provide support for debugging, or supply information for a coverage analysis.

This thesis analyzes different profiling aspects of user-level thread-libraries for shared-memory systems. Conceptual and implementation weaknesses of user-level thread-libraries as well as profiling tools are identified. The resulting profiling model lead to the design and implementation of a new profiler for the *μC++* language [8] called *μProfiler*. In addition, user-level thread profiling at a function level, dynamic filtering of profiling information, dynamic frequency calibration for statistical profiling and profiling a thread-library's run-time kernel are investigated.

Although this thesis mainly focuses on the implementation concepts of *μC++*, the underlying ideas are applicable to many kinds of concurrent systems.

Thesis Outline

Chapter 2 describes general approaches to profiling. Different profiling methods for instrumentation, monitoring, analysis and visualization are introduced and their applicability to different concurrent programming environments is discussed. This chapter also introduces various classification models for the different profiling stages and discusses the fundamental concepts and definitions for profiling.

Related work in the field of profiling concurrent programs is reviewed in Chapter 3. Different approaches and their applicability to profiling user-level threads in a shared-memory environment are examined.

An overview and some implementation details of the target system for μProfiler, i.e., the *μC++* system [8], are surveyed in Chapter 4 with regard to their consequences concerning the design and implementation of a profiler. In addition, previous work for monitoring and visualizing a concurrent μC++ program's run-time behaviour is described.

Chapter 5 discusses the relevant design aspects of the prototype profiler *μProfiler* for μC++: the design objectives are presented, static and dynamic design considerations surveyed and a validation of the design is given.

The implementation of μProfiler for *μC++* is explained in Chapter 6. Some of the main conceptual and algorithmic aspects are described in detail.

Chapter 7 concludes the main issues of this thesis and discusses how the ideas presented can be applied to future work.

Chapter 2

Profiling Methods and Metrics

The performance of concurrent using user-level thread-libraries is affected by various factors. Two abstraction levels can be identified: first, the variety of language primitives offered, thereby representing the underlying concurrent programming paradigm, and second, the actual implementation of these language primitives.

Performance optimization is composed of the following steps, which form a cycle that can be iterated to some fixed point:

1. a careful performance evaluation of the system to determine the cost of one or more operations,

2. an analysis of the component cost of these operations,

3. where possible, optimizing the operations to decrease their cost, and thereby, increase performance.

This cycle is performed multiple times to ensure an optimization is effective and does not interfere with other operations.

Selecting an evaluation technique and its associated metric are the key steps in every performance evaluation [33]. When evaluating user-level thread-libraries and programs that make use of them, *measurement* is the best and often the only applicable evaluation technique. In addition to *measurement*, techniques such as *analytical modeling* or *simulation* should be applied to decrease a system's potential for performance problems. Nevertheless, *measurement* is the only technique allowing efficient detection of performance bottlenecks at a fine-grained level, e.g., statement level. This thesis deals with the most important measuring tool for concurrent systems, namely *profilers*.

There are many reasons for profiling a concurrent program, the main ones being:

performance analysis to find and examine the "hot spots" of the program, i.e., the most frequent and most time-consuming sections,

algorithm analysis to empirically confirm the predicted algorithmic behaviour of the program,

coverage analysis to identify the executed segments of the analyzed concurrent program, thereby determining the adequacy of a test run,

tuning to determine the efficiency of optimization changes made to the program,

debugging to locate code segments with non-predicted behaviour.

After identifying "hot spots" in a concurrent program, the component costs of the evaluated operations must be analyzed in order to determine if the cost is fundamental or can be optimized. Often, this analysis cannot be entirely automated and involves some effort from the programmer. Nevertheless, modern profiling tools help in this reasoning process and give enough advice to significantly reduce the time needed for this step.

Depending on the results of the performance analysis, the program can be optimized at different levels. It is often the case that simple code re-arrangements on an inter- or intra-task level give good performance increases. The situation gets more complex when the concurrent structure of the program has to be changed. In this case, the degree of performance improvement often depends on the available concurrent language primitives and their implementation.

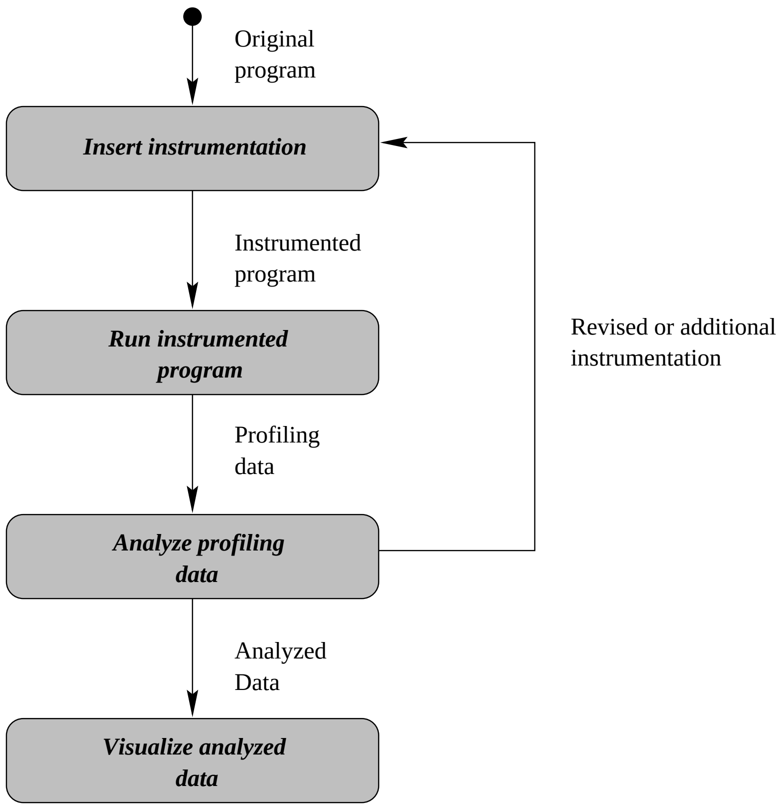

Figure 2.1: Steps in Profiling.

2.1 Profiling Methods

Profiling is typically done following the steps illustrated in Figure 2.1. First, instrumentation is added to the original program. The instrumented program is then run under the control of the execution-monitoring profiler, which generates a report of the program's run-time behaviour. This report is then analyzed and the results visualized. Often, other metrics are chosen, new instrumentation added and this *profiling cycle* is repeated several times.

Today's profiling tools perform many operations: they monitor how much time an executing task spends in each traversed block of code, collect data about how many times certain functions are called, analyze the concurrent system's load during synchronization and communication among tasks, try to detect bottlenecks due to mutually exclusive access to shared resources, etc.

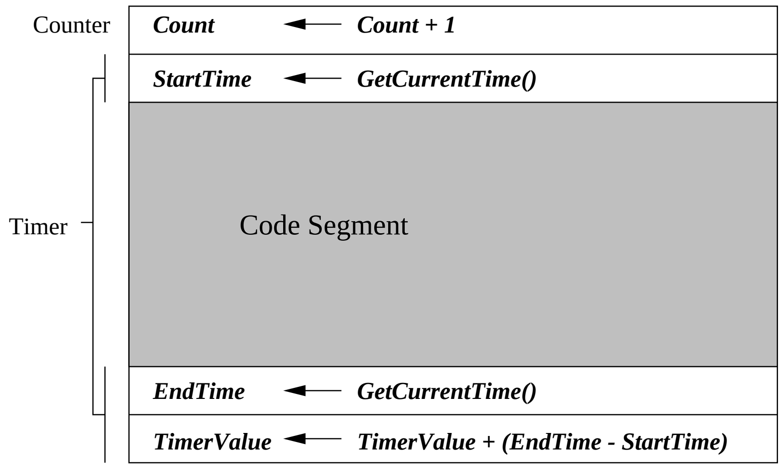

Figure 2.2: Basic Instrumentation Insertion Primitives.

2.1.1 Instrumentation Insertion

Adding *instrumentation* to a program means extending a program with additional code and data segments that allow gathering additional information about the program's execution. Multiple instrumentation insertion primitives, methods and methodologies are available and the main ones are discussed.

Instrumentation Insertion Primitives

The two basic insertion primitives are *counters* and *timers*.

Counters are utilized in various ways. An example is to check how often a certain program segment is traversed. To provide this very important information, a static counter is inserted, typically at the beginning or end of the code segment. At any point during the program's execution, the counter holds the exact number of traversals of this block of code.

Timers are used to measure the time spent in a specific block of code, or the time between two distinct traversals of the same code segment.

A simple example of how these two primitives can be implemented and used to instrument a code segment is shown in Figure 2.2. The instrumentation primitives are inserted at the original code segment's entry and exit; in Figure 2.2, the original code segment is shaded in gray. The counter is represented by a data structure named "Count", which is incremented before the thread of control traverses the code segment. The timer primitive consists of two parts: before traversing the original code segment, a data structure "StartTime" is set to the current time. After the current thread of control has finished executing the code segment, it sets "EndTime" to the now current time value. Then, the difference between "EndTime" and "StartTime" is added to the value of the data structure "TimerValue". Thus, if initialized to zero, "TimerValue" always holds the total amount of time spent inside the instrumented code segment.

Although providing very important profiling information about the examined block of code, *counters* and *timers* are insufficient for complex profiling metrics (see also Section 2.2). For instance, if the examined code segment is a function, it is useful to additionally insert instrumentation primitives that store information about the caller of the function. One possibility to improve the potential profiling effectiveness and applicability to multiple metrics is to add conditional statements inside of the basic profiling primitives. For example, a metric might require merely incrementing a value if the current thread is the only running thread.

When profiling concurrent programs in shared memory, additional issues have to be considered. If the profiler uses shared data structures for the inserted primitives, the primitives must provide mutual exclusion to avoid corruption of the data. Alternatively, the profiling data can be made independent for each executing thread, so no mutual exclusion is required.

Instrumentation Insertion Methods

Two different instrumentation insertion methods can be distinguished: *direct insertion* and *indirect insertion*.

For **direct insertion**, the instrumented code is placed at the front and/or end of the code block to be instrumented.

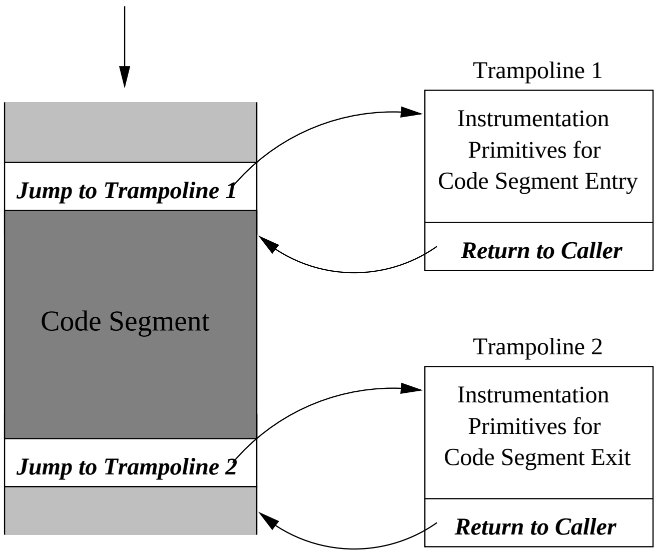

Figure 2.3: Indirect Instrumentation Insertion.

A more common and general approach is **indirect insertion** of instrumentation primitives. This approach is achieved by simply inserting a jump to a *trampoline* for each point in the instrumented code segment where instrumentation primitives are to be placed. As shown in Figure 2.3, the instrumentation primitives are moved into the trampoline and after executing them, execution jumps back to the original code segment, like a call to a subroutine.

Using indirect insertion as the instrumentation insertion method makes automated and configurable instrumentation easier, as the trampolines are located at known positions, where they can be easily modified. In addition, it is possible to insert parameterized calls to a specific trampoline from various points of the instrumented program and control the selection of the appropriate instrumentation primitive by using conditional statements inside the trampoline.

A variation of indirect insertion is copying the first few instructions of the original code segment to the end of the trampoline. Then, a jump-to-trampoline instruction is placed at the beginning of the original code segment, overwriting the instructions just copied from the code segment to the trampoline. This variant, which is similar to breakpoint insertion for debuggers [36], has as a result that, most likely, storage has to be allocated for each different trampoline to hold the relocated code. Nevertheless, this approach is the preferable one if the instrumentation has to be done after compilation.

Instrumentation Insertion Methodologies

Depending on the selected profiling metrics (see Section 2.2), monitoring technique (see Section 2.1.2), and the point of time at which instrumentation occurs, three different methodologies of instrumentation insertion can be distinguished:

1. *No instrumentation insertion*

   Even though this approach is not very popular, profiling can be performed without inserting any additional instrumentation. In this case, *statistical monitoring* (see Section 2.1.2) is the only applicable method to collect the required execution-time information.

   Nevertheless, not inserting additional instrumentation code clearly reduces the so-called *probe effect* [15] when profiling concurrent programs. The *probe effect* results from the fact that concurrent programs are non-deterministic in terms of the order of executed instructions; hence, every insertion of additional instructions potentially changes the program's behaviour. Therefore, the more instrumentation inserted, the greater the potential for disturbing the sequence of execution.

2. *Statically inserted instrumentation*

   Most profiling tools statically insert instrumentation code before compile-time, at compile-time, or by dynamically changing the executable. The latter one is also called *binary re-writing*. In most cases, the applied instrumentation method is indirect instrumentation insertion in the form of trampolines.

   Even though statically inserting instrumentation entails a higher probe effect, it reveals important information about the profiled program that is impossible or very difficult to obtain by any other methodology. Examples are the insertion of counters to measure the number of traversals through a code segment, or inserting instrumentation that gathers information about an instrumented function's callers, which makes the construction of a function call graph possible.

The drawback of statically inserted instrumentation code is that if instrumentation is placed at a point in the program that it is not a potential bottleneck, it is infeasible to remove it without stopping the program's execution. Since this problem becomes crucial for long running programs, the instrumentation primitives should be conditional and depend on the value of flags indicating if profiling of the respective block of code is activated, and thereby allowing the necessary amount of run-time control. Only then is statically inserted instrumentation suited for profiling of applications of all sizes.

3. *Dynamically inserted instrumentation*

Dynamically inserting profiling instrumentation is a relatively novel methodology [29], where the instrumentation insertion is performed at run-time, according to special profiling algorithms that decide where, when and what type of instrumentation is needed to profile a program.

Since address relocation during run-time is a non-trivial task, the preferable, and in many cases only, applicable instrumentation method is the variation of indirect instrumentation insertion that moves the first instructions of the instrumented code segment to a trampoline and executes them from there as described above.

In contrast to statically inserted instrumentation, profiling via dynamically inserted instrumentation cannot normally be performed at the same fine-grained level with acceptable accuracy, because in order to dynamically change the profiled program's code segment, the program has to be executed under control of the profiler. Therefore, the overhead cost of dynamically inserting code increases the potential for probe-effects and is not applicable to small code segments when accurate results are required.

Nevertheless, profiling with dynamically inserted instrumentation has the potential to reduce the amount of irrelevant information collected to a minimum and does not cause any overhead in "unimportant" code blocks. The instrumentation overhead can be dynamically measured and adjusted by adding and removing instrumentation

	exact monitoring	statistical monitoring
no instrumentation	$-$	$\circ$
static instrumentation	$+$	$+$
dynamic instrumentation	$\circ$	$+$ (long running programs)

Table 2.1: Combinations of Instrumentation Insertion and Monitoring.

insertion primitives.

Therefore, dynamically inserted instrumentation normally aims at performance measurement of large-scale long-running programs.

2.1.2 Monitoring

Monitoring is the process of asynchronously (or synchronously) collecting information about a program's execution behaviour [6].

During execution, the profiler monitors the program, possibly making use of inserted instrumentation, and produces an execution profile, which subsequently is passed to the profile analyzer (see Section 2.1.3).

Two different monitoring methods can be distinguished: *exact monitoring* and *statistical monitoring*. Thus, there are various possible combinations of monitoring methods and instrumentation insertion methodologies; these combinations are evaluated and illustrated in Table 2.1. The symbol "+" means sound results, "$\circ$" represents acceptable results with some limitations, and "$-$" stands for unacceptable results. In the following, all different combinations are discussed and their evaluation elaborated.

Exact Monitoring

Exact monitoring gives exact information about the profiled program's execution behaviour in regard to the chosen metric (see Section 2.2). Exact monitoring is used, for example, to provide information about the exact time when the monitored program reaches certain points in its code or about the exact number of calls to a certain function. As mentioned before, depending on the underlying instrumentation insertion methodology, there exist several different approaches to exact monitoring.

1. In the case of no instrumentation insertion, exact monitoring can be performed by running the profiled program under control of the profiler, which yields and resumes the profiled program's execution on a machine instruction level, similar to tracing and single-stepping in debugging tools. It is clear that this approach introduces the maximum probe effect, and therefore, is not applicable in most cases. In addition, exact monitoring without instrumentation on multiple processors is non-trivial, and in most cases not wanted, since unless each processor has a separate profiler, the execution controlling profiler has to schedule the order of execution among different processors, which results in a serialization of execution.

2. For exact monitoring, statically inserted instrumentation is the preferable methodology if a complete event trace is required by the selected profiling metric. Special instrumentation code is inserted at all points of interest in the monitored program. Selecting these locations for instrumentation insertion allows exact monitoring to be performed at different levels of granularity possibly at a very fine grained level, i.e., statement level. The inserted instrumentation consists of code that generates some type of event and sends it to a monitoring profiler as shown in Figure 2.4. The type of event depends on the profiling metric used and can be everything from a simple time stamp to detailed information about the state of the currently executing thread.

   According to the selected profiling metric, the event data is collected by the monitoring profiler, which then performs an analysis and visualization of the data (see Sections 2.1.3 and 2.1.4).

3. Dynamically inserting instrumentation into the monitored program entails the risk of missing events needed to gather exact information with respect to certain profiling metrics. Thus, dynamic instrumentation insertion is only applied if either the profiling metric does not depend on a complete trace of events, or if the execution path of the monitored program can be predicted and instrumentation inserted accordingly for each

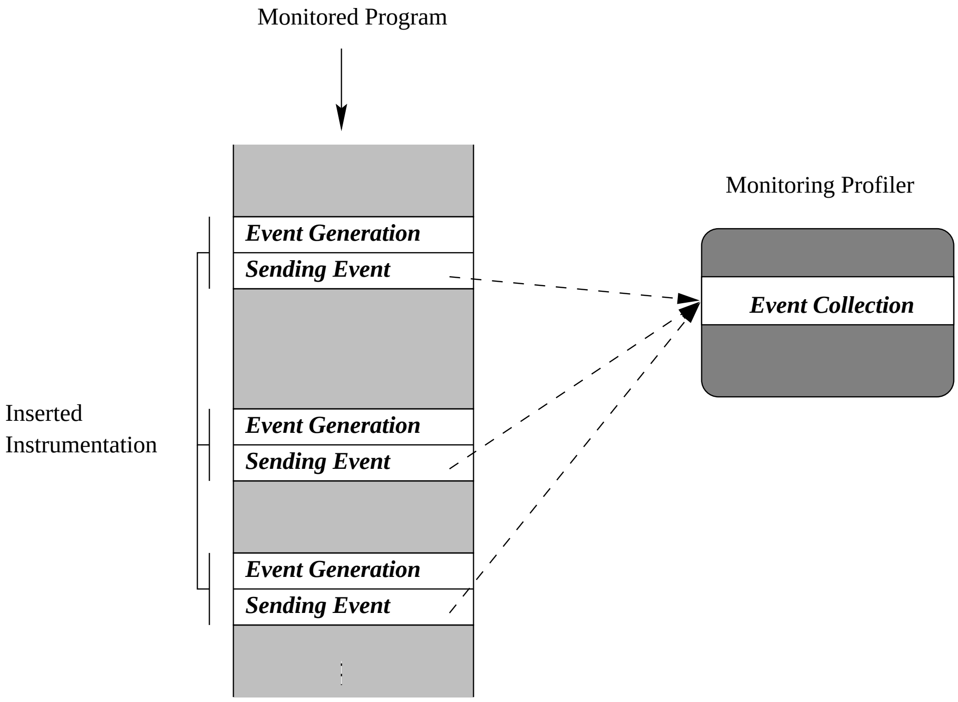

Figure 2.4: Event Collection in Exact Monitoring with Instrumentation Insertion.

executing thread *before* the monitored program's threads of control reach the points critical for exact monitoring.

Statistical Monitoring

Statistical monitoring is used to collect run-time information about the monitored program by periodically sampling the examined program's state of execution. This technique is illustrated in Figure 2.5.

In most cases, the probe effect introduced by profiling has to be minimized, which makes exact monitoring infeasible. In addition, applying exact monitoring to long-running applications can produce an enormous amount of collected data, too large for any reasonable analysis.

Statistical monitoring deals with these problems by intentionally omitting information about events that occur between samples. This technique reduces the amount of data gath-

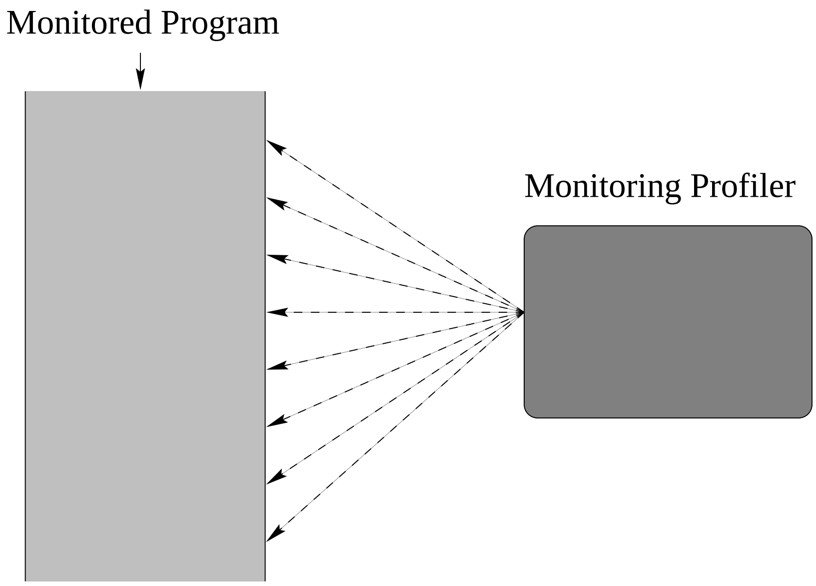

Figure 2.5: Statistical Monitoring.

ered, but still provides enough information to produce a representative image of the monitored program, which is accurate enough to detect relevant performance problems. In addition, special sampling hardware can be used, and in a multi-processor environment, the profiler performing the statistical monitoring can be executed by a dedicated processor, thereby lessening the probe effect on the monitored program.

Similar to exact monitoring, different approaches can be taken depending on the methodology of instrumentation insertion utilized (see Table 2.1).

1. Statistical monitoring can be done without any instrumentation insertion. In this case, the monitoring profiler collects the relevant information only by periodically inspecting the state of the program under examination. Although statistical monitoring with no instrumentation insertion minimizes the probe effect, getting accurate and detailed information about a concurrent program can be very difficult, unless there is support by the underlying system, i.e., the operating system or the kernel of the thread library used. For example, on most systems it is impossible to get the current value of the program counter of a process on another physical processor. Also, the sampling algo-

rithms have to be extremely robust towards potential state changes; if, for example, the monitored program changes state while the profiler is in the process of reading exactly this identical state information, the sampled value is very likely to be inconsistent.

2. Performing statistical monitoring with static instrumentation is probably the most common combination when statistically profiling a concurrent program. In this case, the inserted instrumentation consists of only basic instrumentation primitives and their corresponding profiling data structures, the locations of which are known by the monitoring profiler. According to the selected profiling metric, these primitives operate on their data structures, and the profiler periodically reads the data structures' values. This approach allows sampling data structures internal to the profiled program, because the sampled state information is available to the profiled threads, which execute the instrumentation primitives and update their corresponding profiling data structures. Hence, by introducing some cooperation between the profiled program and the profiler, additional information becomes available.

3. Dynamic instrumentation is also applicable to statistical monitoring. This combination is useful for profiling long-running applications. The sampled data structures and the primitives that operate on them are dynamically inserted and removed by the profiler. While the same profiling metrics can be applied as in the case of statistical monitoring with static instrumentation insertion, dynamic instrumentation only causes an overhead where it is unavoidable.

Summary

Inserting instrumentation often results in more accurate results although it introduces a greater probe effect; in addition, it facilitates the operation of a statistically monitoring profiler. Inserted instrumentation allows monitoring at different levels of granularity according to the programming language primitives utilized. Information about the executing program can, for instance, be collected at a function level, which in most cases suffices for profiling purposes. In this case, instrumentation is added at the monitored program's function entries and/or exits. In general, instrumentation is inserted at a level depending on the basic language primitives, and the inserted code stores an identifier for the instance of the currently executed primitive in a data structure periodically sampled by the monitoring profiler. Since these data structures only change values at the beginning and end of the monitored primitive, the probability of incorrect data due to state changes during sampling is reduced compared to statement level state changes when no instrumentation is inserted.

2.1.3 Profile Analysis

Subsequent to monitoring, the next stage in the profiling cycle is the *profile analysis*. In the following, the *time aspect* and the *purpose* of profile analysis are discussed.

Time Aspect

Depending on the profiler's architecture, the profile analysis can be performed anywhere from post-code-generation to pre-visualization. Two opposing techniques can be distinguished:

1. One possibility is that the analysis is done *on-the-fly*; i.e., while the profiled program is executing, the profiling monitor produces the profiling data and passes it along to a profile analyzer. This approach has the advantage that it reduces the amount of stored profiling data, since it is immediately processed. Another advantage of this approach is that the profiling results are available while the program is still running. This feature makes dynamic profiling possible and is very useful when long-running programs are profiled. Also, a visual representation of the profiling data can be presented while the program is running, thereby allowing possible user-interaction. For example, the user could dynamically switch profiling on and off for certain modules and possibly detect performance problems in the current context of the program. The drawbacks of analyzing the execution profiles on-the-fly are that it introduces an additional probe-effect and that it might be impossible to perform the analysis with the speed of the incoming data from the monitoring stage when sampling is done at a high frequency. In this case, relevant data has to be discarded, or the analyzer has to work with a large

delay between the time when a profiling events occurs and when it is processed. Both of these cases make an on-the-fly analysis infeasible for short-running programs. Also, when profiling programs that are highly user-interactive, additional user-interaction at the profiler-level might be undesirable.

2. The second variant is performing the analysis *post-mortem*, i.e., after the profiled program has finished execution and after all relevant profiling data is collected. This technique is used by most profiling tools and is applicable to programs of all execution durations, although it might not be preferable for long-running applications. This method entails a lower probe-effect than performing the profiling analysis on-the-fly. On the other hand, especially when profiling long-running applications, there is the problem of vast amounts of profiling data that have to be managed. Also, post-mortem analysis is inflexible in terms of user-interaction, because the user cannot influence the profiling during the program's run-time, but there are no additional burdens on the user while the program is executing.

Purpose

The purpose of the profile analysis is twofold: first, analyzing the collected profiling information, and second, preparing the analyzed information for visualization:

1. *Analysis of the profiling information*

   An analysis of the profiling information mainly consists of two parts: *filtering* and a *correlation analysis*, which can both be performed independently of the time aspect.

   Filtering aims at reducing the collected data to a manageable subset. Wrong and irrelevant data is detected and removed, and depending on the profiling metric utilized, the amount of data is reduced to a necessary minimum. Therefore, filtering is a main aspect in the profile analysis. However, it is a non-trivial task to choose a proper filtering technique that discards unnecessary information, which could potentially overwhelm the user, while still maintaining enough information to correctly represent the selected profiling metric.

Correlation analysis is the second task performed during the analysis stage. Its main goal is to cut through the profiling data along a certain dimension in order to detect correlated profiling events. For example, the profiling metric can require information about thread synchronization events; therefore, it is important to compare profiling events for different concurrently executing threads and detect the ones that are using synchronization primitives at the examined point of time.

2. *Preparation for visualization*

   The other purpose of the profile analysis is to prepare the profiling data for visualization. The preparation comprises condensing the profiling data according to the metrics and visualization technique. Condensing the profiling data is important both to provide the required level of abstraction and to avoid overwhelming the user with irrelevant information. Also, it is clear that different visualization techniques for the same data require different preparations; simple tables, for instance, might require more precise information than bar charts displaying accumulated values.

There exists a wide range of possible levels of "intelligence" for a profile analyzer: the profile analysis can be entirely automated, or at the other extreme, the profile analyzer can be a help-tool only supporting the user.

Complete automation of the profile analysis mainly aims at *performance analysis* and *tuning* of programs. It usually requires expert-system-like artificial intelligence support, which defines and evaluates hypotheses in order to locate a performance bottleneck. This approach has the advantage that the user does not have to be a profiling expert to effectively use the profiler. The drawback is that it may not be as flexible as a help-tool and that relevant information might be omitted because the hypotheses does not cover all imaginable cases. Initial research in this field has been done; examples can be found in [11], [14], [50], [56] and [78].

Help tools require the user to be familiar with profiling, but they have the advantage that, in addition to *performance analysis* and *tuning*, they can be used for other profiling reasons, namely *algorithm analysis*, *coverage analysis* and *debugging*.

It is clear that these two possibilities can be combined in a variety of ways. Automated tools can provide interaction in the selection and evaluation of hypotheses, and help tools can have a certain degree of automated profiling analysis functionality to lead the user to possible problems.

2.1.4 Visualization

Visualization is the last step in the profiling cycle. Its purpose is to present the profiling information to the user in an effective, concise and appealing manner. Different basic visualization techniques can be distinguished: *tables*, *charts*, and *graphs*.

Tables

The simplest and in many cases the best visualization technique is presenting the profiling data in a table. When utilized to visualize profiling data, a table is often sorted by decreasing order of importance according to the profiling metric. For example, if the durations a thread spends in different functions are measured, the profiler presents the functions in order from relatively longest period to shortest period in the table. An advantage of visualizing data in a table is that it is the best format to show many details at the same time and the raw data values prevent misleading the user. Nevertheless, tables are restricted to representing discrete values depending on at most two dimensional input, and therefore, can only be applied in certain situations. Also, for many profiling metrics, the information should be presented by more appealing, graphical visualization techniques.

Charts

Another visualization technique is presenting profiling data in a chart. Charts are pictures or diagrams that give information by graphically displaying discrete data values. Many different types of charts can be applied, examples of which are *bar charts*.

An example for a bar chart is presented in Figure 2.6. It shows a bar chart presenting profiling information of the total time a thread spent in the running state inside different

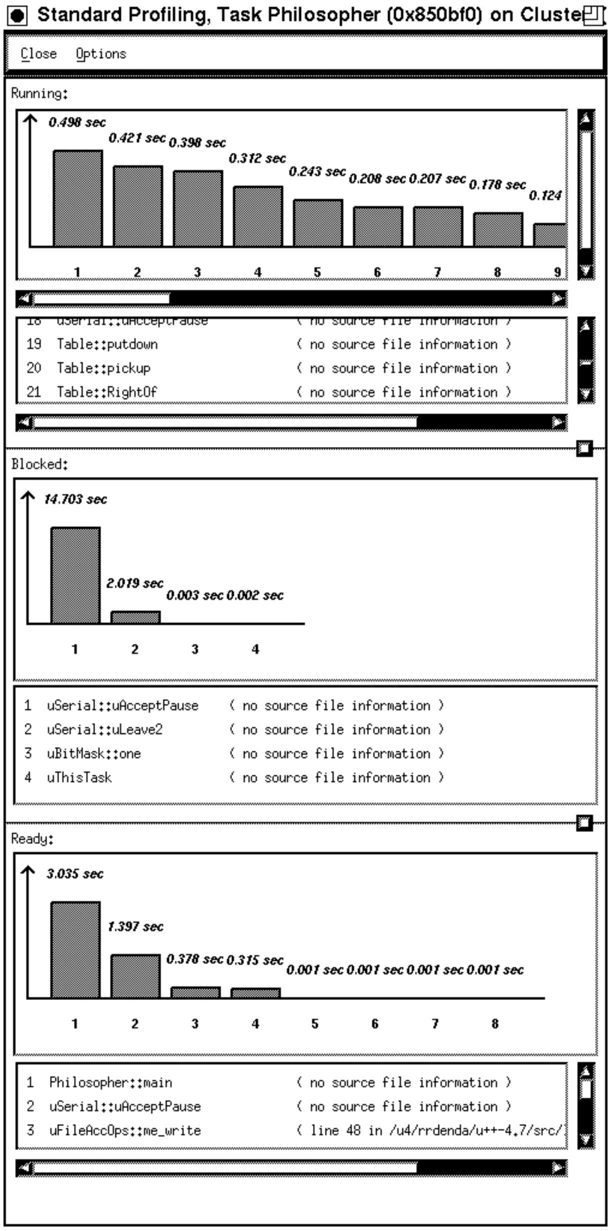

Figure 2.6: Example of a Bar Chart.

functions. Distinct functions are represented by different bars and the bars' heights indicate how much time the visualized thread spent inside the corresponding function.

In addition to bar charts, there exists a variety of chart types useful when visualizing profiling data; examples are *Gantt charts* and *Schumacher charts* (see [55]).

When designing a chart, it should be ensured that a large share of the visible points on a graphic presents data information, and that the chart contains as little redundancy as possible. For example, the width of a bar in a bar chart does not provide any further information and might even be misleading. Therefore, the bar chart presented in Figure 2.6 can be redrawn in a simpler and more concise way; such a display is presented in Figure 6.8 in Section 6.3. For a more detailed discussion on graphical excellence, graphical integrity and the theory of data graphics, see [75].

Although charts can fulfill the purpose of efficiently communicating complex quantitative ideas, they are restricted to discrete data values that depend on only one or two dimensional input. For displaying contiguous multi-dimensional data, a *graph* is used instead of a *chart*.

Graphs

The third visualization technique exposes profiling data in the form of a graph. Graphs show how quantities depend on each other; they use lines and surfaces to represent relations of possibly multi-dimensional functions. Many profiling metrics can only be effectively visualized in the form of a graph; each of these metrics might require its own graph type.

A very useful example is a *kiviat graph* [39], which is shown in Figure 2.7. This kiviat graph shows eight different metrics aligned in a circle along radial lines. For each metric, its value is marked at the corresponding radial line and connected to the neighbour metrics' marks, thereby creating the inner area of the kiviat graph, which is shaded in gray. Note that the metrics alternate in such a way that a metric for which a higher value is better is between metrics for which a lower value is better. Therefore, the ideal shape of the kiviat graph is an N-pointed star where N is the number of metrics, for which a higher value is better. In this case, the star is malformed for the metric "CPU wait" indicating that this is the metric whose value might be too large. Kiviat graphs can be applied to various sets of metrics as long as an even number of metrics is used, and the metrics are aligned in the manner described above. Further examples of kiviat graphs are discussed in Section 6.3.2.

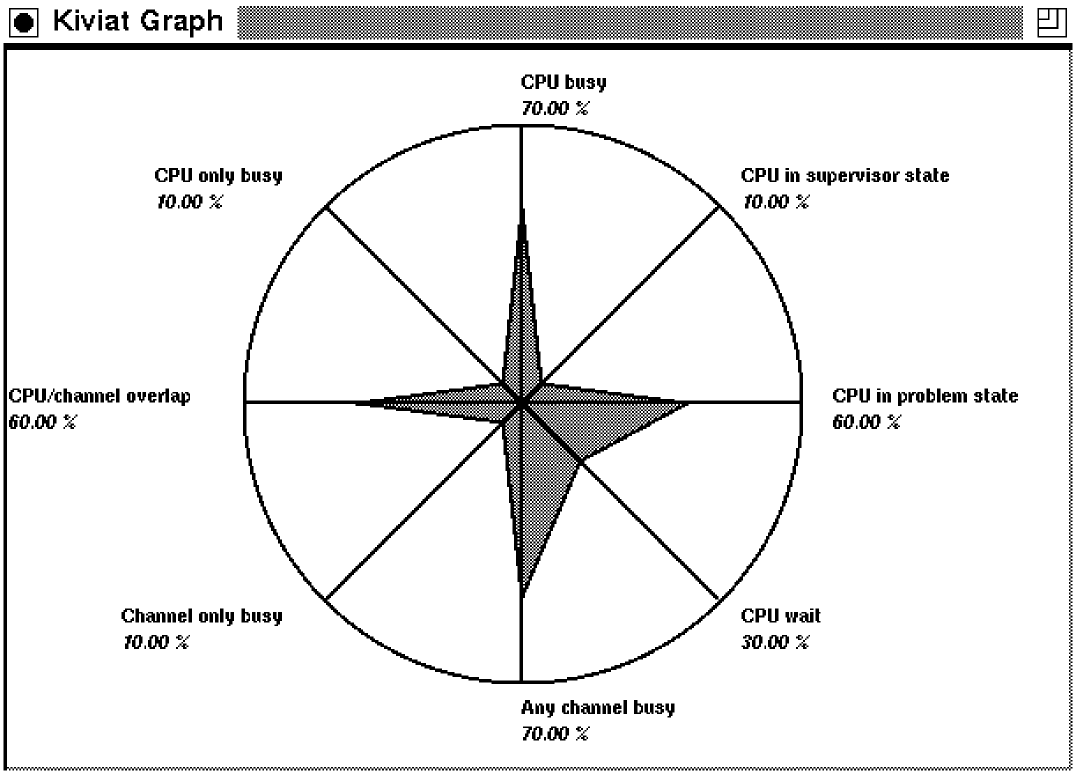

Figure 2.7: Example of a Kiviat Graph.

2.2 Profiling Metrics

The effectiveness of profiling tools mainly depends on the selection of appropriate profiling metrics. There exist a great variety of profiling metrics corresponding to the underlying profiling philosophy; many profilers create new metrics to suit their profiling environment the best. Different existing approaches are discussed in Chapter 3.

In this section, different approaches are introduced to classify profiling metrics.

2.2.1 Metrics on Time versus Metrics on Counts

As alluded to in Section 2.1.1, profiling can be done along two dimensions: *time* and *counts*. The underlying reason for profiling has a strong influence on the choice between these two dimensions, or a possible combination of them. Whereas *coverage analysis* usually requires a metric on counts, other reasons for profiling might entail a utilization of metrics on time.

Metrics on Time

Profiling metrics on time are usually used when performing a *performance analysis*, an *algorithm analysis*, or *tuning*. Both time events and time durations can be used by metrics on time. As well, time events are needed when metrics are applied that make use of the order of the profiled events, e.g., in a real-time analysis. Common examples of profiling metrics on time durations measure the total time spent in a specific function or the time spent blocking on I/O requests.

Metrics on Counts

Although profiling metrics on counts are mainly applied when performing *algorithm analyses*, *coverage analyses*, or *debugging*, they might be useful to give supplementary information for *performance analysis* or *tuning*. An example of a metric on counts is the number of times a certain function is called during the profiled program's execution.

It is clear that in many cases, metrics on counts and metrics on time can be combined to build even more effective metrics.

2.2.2 Exact versus Statistical Metrics

Depending on the monitoring technique applied, profiling metrics can be classified as *exact metrics* and *statistical metrics* and are a consequence of the chosen instrumentation insertion and monitoring methodology.

Exact Metrics

Exact metrics provide exact information about the profiled program. For example, if the metric is the number of calls to a certain function, statistical information is insufficient. Therefore, metrics on counts usually imply exact metrics. Nevertheless, a metric on time can also be an exact metric when it is built on time events. Important examples of exact metrics are trace-based metrics (see Section 3.2), or metrics that are based on a function call graph. Although exact metrics might introduce a high probe effect, modern profilers should provide support for exact metrics, since they are preferable to statistical metrics when performing *algorithm analysis*, *coverage analysis* and *debugging*.

Statistical Metrics

For *performance analysis*, statistical metrics are preferable to exact metrics, since they usually do not require exact monitoring, and therefore, introduce less of a probe effect. Statistical metrics are usually metrics on time, for example, the time spent inside a specific block of code. It is clear that in most cases, statistical metrics can be computed using profiling data collected via statistical monitoring.

2.2.3 Operating Levels

Profiling metrics can also be classified according to the level at which they operate and where the profiling data is available. Since profiling can be done at different levels of detail depending on the underlying system, corresponding metrics must be built at different levels of abstraction to correctly represent this system under examination. Profiling support by the underlying system can lead to more accurate data or reveal relevant information intrinsic to the run-time system's kernel. For example, many operating systems provide various profiling support and statistical information about application processes, normally including data concerning the resource-usage, memory and input/output behaviour, system load, active processor time, etc. Furthermore, this information is only available through the instrumentation provided at this level. A good profiling tool should present the information and take it into account when performing the profile analysis. In many cases, support by the operating system or the run-time kernel facilitates gathering profiling data needed to develop informative and accurate profiling metrics. In addition, information about the system's load during the profiling process is needed to validate the adequacy of a test-run. Thus, integrating profiling supporting instrumentation into the kernel is an important design issue and should be part of any programming development environment.

At a higher level, the programming language itself can support profiling at various levels, e.g., via the programming paradigms and constructs supplied, or by supporting profile instrumentation insertion at the pre-processor, compilation or linking level.

Optimally, a programming environment supports profiling metrics at the different operating levels discussed above to allow flexible, accurate and fine-grained metrics to be integrated into the profiling process.

Chapter 3

Related Work

In the following sections, different profiling tools and environments are briefly introduced and distinguished in terms of their instrumentation insertion, monitoring, profile analysis and visualization techniques, and the target programming environment they are aimed at.

3.1 Instrumentation Insertion

Profiling tools can be classified in various ways. One important aspect representing the underlying methodology is the stage where the instrumentation insertion is performed: instrumentation insertion can be done at various stages, anywhere between writing the program's source code and executing the program. Some previous profiling tools and their points of instrumentation insertion are presented in Figure 3.1.

Inserting profiling instrumentation at the source code stage gives the programmer complete control about when, where and possibly how the application is profiled. This approach of manually inserting the instrumentation is, for example, required by the profiling tool *JEWEL* [42]. Although this method provides a detailed level of selectivity, it has the drawback that it requires a lot of time and effort by the program instrumentor. Other profiling tools, such as *PARAVER* [57], do not require the program analyst to modify the general source code, but to only add some new statements at the beginning and end of the blocks of code to be profiled, which start and terminate the profiling event collection process. *PIE* [62]

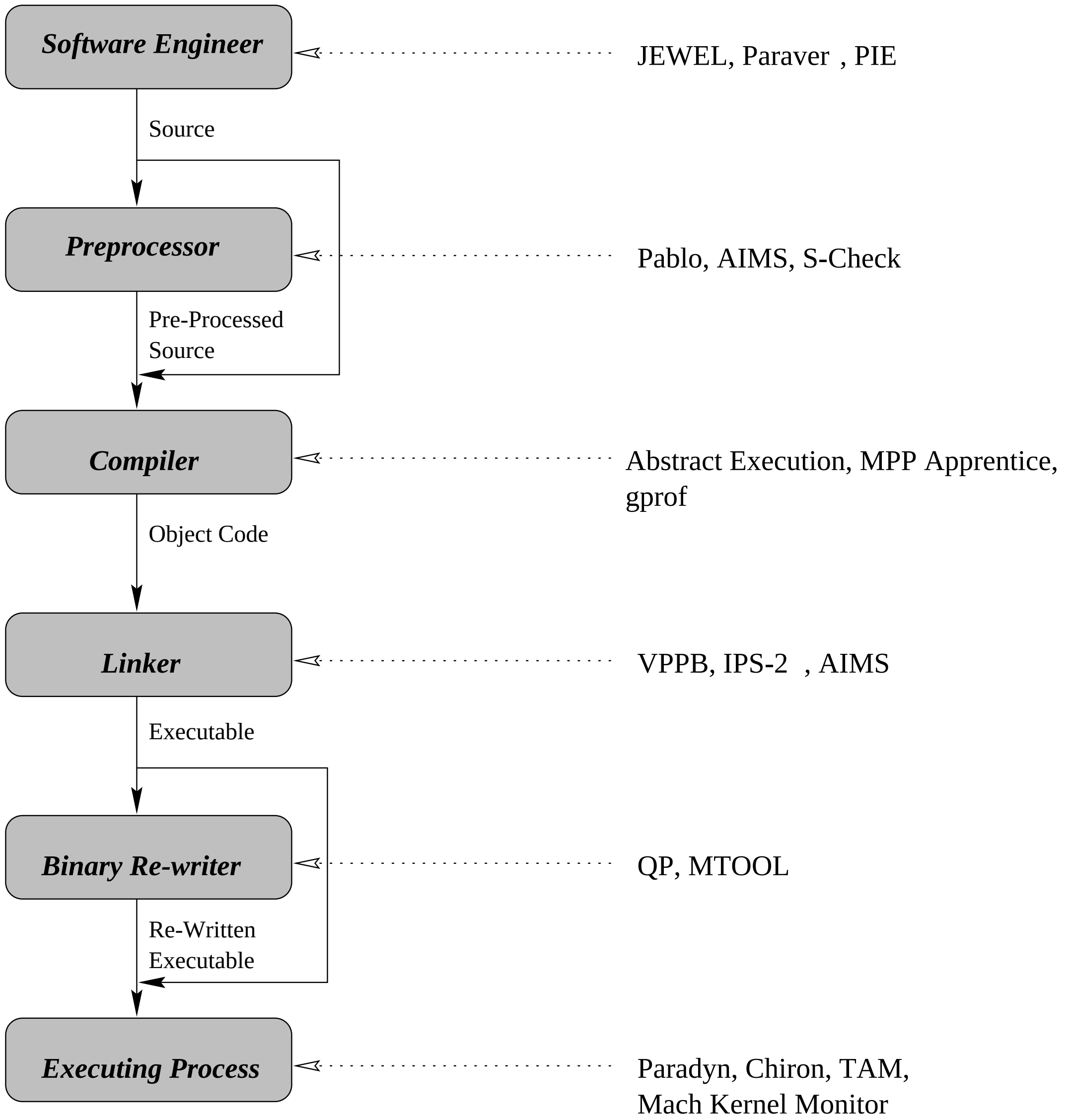

Figure 3.1: Profiling Tools and their Instrumentation Insertion Points.

is another example where parts of the instrumentation have to be inserted in the source by the programmer when fine-grained profiling is required.

Other tools perform the instrumentation insertion at a pre-processor level. *AIMS* [80] [81], *Pablo* [59], *pC++ Instrumentor* [48] and *S-Check* [63] are examples of profiling tools where the necessary instrumentation is inserted into the source code by a source code instrumentor before program compilation. In this case, instrumentation insertion can be done in an interactive manner, or can be used to perform automated *synthetic perturbation screening* [46] [64]: the synthetic perturbation screening technique inserts artificial, controllable delay points into the source code at locations of potential performance bottlenecks. These delay points can be switched on and off, thereby representing numerous different versions of the original program with a distinct run-time behaviour. The instrumented program is executed with many different delay settings and the resulting execution times are recorded. Using statistical techniques, such as variance analysis, the relative strength of the instrumented performance bottlenecks is identified and critical spots are located.

AE (Abstract Execution) [43], *MPP Apprentice* [78], and the sequential profiling tool *gprof* [22] are examples of tools that accomplish the profiling instrumentation insertion at the compilation stage. This approach has the advantage that profiling insertion can be done at a fine-grained level, even down to the assembler instruction level. An obvious drawback of inserting instrumentation at the compilation stage is that a different version of the compiler is needed for each supported platform.

At the linking stage, there exist different approaches for adding instrumentation. One possibility is to replace the standard system libraries by instrumented ones. This approach is taken by *IPS* [52] and *IPS–2* [51]. Similarly, the run-time libraries can be replaced, which is done by *VPPB* [4] and *AIMS* [80]. Also, when the instrumentation insertion is added at an earlier stage, the profiling tool usually requires linking additional libraries to the program to be profiled, in order to supply the necessary monitoring and analysis functionality.

After linking, some performance tools, e.g., *QP* [3] and *MTOOL* [19], perform binary re-writing of the object code to insert the necessary instrumentation. This approach has the advantages that it is almost independent to the programming language used and does not require changing the pre-processor, compiler, or the linked libraries. Obviously, since this method does not take the underlying programming primitives into account, it does not allow profiling on an abstract basis corresponding to the higher-level language constructs that are used to write the concurrent program.

There exist several possible techniques that can be applied to perform profiling on an executable that has not been instrumented in a previous stage. *Chiron* [20], for example, performs the profiling by using the *Mint* [77] object code multiprocessor simulator. Another technique is used by the *TAM* [61] tool for Intel Paragon architectures, which defers the instrumentation insertion to the beginning of the program's execution. *Paradyn* [50] is a profiling tool that dynamically inserts and removes the instrumentation during run-time. Dynamic instrumentation has the advantage that it reduces the amount of profiling data generated and does not slow down the program's execution for parts of the program that are considered unimportant in terms of profiling. *Paradyn* dynamically inserts the necessary instrumentation by controlling the profiled program's execution with a variation of the UNIX `ptrace` interface [71].

In addition to the instrumentation insertion techniques described above, other mechanisms can be applied on systems that either have operating system kernel instrumentation support, an example of which is the *Mach Kernel Monitor* [44], or provide a facility that allows a process to monitor the execution of other processes. Monitoring support for processes is provided by most operating systems, e.g., different versions of UNIX provide this information via system call libraries (see [69]) or process information management facilities, like the `/proc` filesystem [70].

3.2 Metrics and Monitoring Methods

Before profiling can be applied, the relevant profiling entities, i.e., the parts of the program on which profiling is performed, have to be identified. The next step is to define and select profiling metrics to appropriately represent these profiled entities. Theoretical considerations about performance metrics are, for instance, found in [72]. Once the metric has been defined, a corresponding monitoring method is selected.

Metrics and monitoring methods are two essential aspects in profiling; as discussed in the following, the monitoring method applied can be used to classify different profiling tools.

Many tools perform a profile analysis based on event-traces of the program's execution. Exact monitoring is the commonly applied technique in this case (see Section 2.1.2). Based on an event-trace, different tools calculate different metrics. The sequential tool *gprof* [22], e.g., uses trampoline insertion to collect information about the profiled application's call graph. Various tools extend this idea to profiling parallel applications and collect call graph information for different processes. The *pC++* [48] programming system and *PARAVER* [57] are examples of this technique.

There is a variety of event types that can be collected: *PARAVER* collects events representing the CPU activity, communication events and allows user-definable events. Based on an event-trace, *Monit* [35] computes metrics, such as the number of busy processors or the number of threads waiting to enter a critical section. *IPS* [52] and *IPS–2* [51] both calculate the critical path by making use of the program's event-trace, where *critical path* is the longest path through a series of sequential code fragments between communication points. *IPS–2* also provides a metric called *Logical Zeroing*, which gives, based on the critical path, an estimate of how much the execution time will improve when improving a selected function. *Slack* [28] is a third metric introduced by *IPS–2*, which is based on a critical path analysis. *PV* [31], the *Uniform Trace Environment (UTE)* [79], *Pablo* [59], *Upshot* [24] and *JEWEL* [42] are other examples of event-driven performance measurement tools. *Pablo* defines the trace log-file format *SDDF* [2], which is supported by many different profiling and visualization environments. A more general approach to event-trace monitoring and analysis is introduced with the *ZM4/SIMPLE* [53] tool set: an abstraction model is given, which applies to many different exact monitoring approaches. A comparison and validation of various, mostly trace-based, metrics is found in [30].

Statistical monitoring is the other technique used by many tools. As in the case of *gprof*, an interrupt-driven sampler monitors the program's behaviour during run-time by sampling its program counter. *Quartz* [1] is a tool that, in its original version, calculates a metric called *Normalized Processor Time (NPT)* based on statistical monitoring. *NPT* represents the time for each executed function divided by the number of currently executing processors performing useful work, which gives the effective parallelism for that time interval. Limitations of statistical monitoring are discussed in [58].

Paradyn takes a hybrid approach of exact and statistical monitoring: It periodically samples exact information, which comes from event counters and timers.

3.3 Profile Analysis

As discussed in Section 2.1.3, a profile analysis can be performed at various stages from post-code-generation to pre-visualization. Some profiling tools perform the profiling analysis in a simulated environment. *VPPB* [4], *PARAVER* [57] and *Chiron* [20] are examples of this approach.

Paradyn [50] performs the profile analysis during a program's run-time and uses the information about the current state of the profiled program to make decisions about where and when to add or remove the profiling instrumentation. On-the-fly analysis of profiling events has the advantage that the user can possibly be involved in the analysis, selection of instrumented code blocks, and filtering and metric selection while the program is running, which is especially useful for long-running applications. The major draw-back of dynamic profile analysis is the additional cost and perturbation caused by the profile analyzer. Therefore, most profiling tools, especially those based on event traces, perform a post-mortem profile analysis. Examples of tools that use post-mortem analysis are *IPS* [52], *IPS–2* [51], *ZM4/SIMPLE* [53], *TAU* [5], *PV* [31], *UTE* [79], *Pablo* [59], *Upshot* [24] and *AIMS* [80].

S-Check [63] performs its profile analysis via *synthetic-perturbation screening* [46] [64], i.e., by executing various differently instrumented versions of the profiled program. Instrumentation in *S-Check* means calculating a sensitivity measure for each code segment and subsequently building different versions of the instrumented program by inserting controllable delay statements at the locations of potential performance bottlenecks inside the program.

ExtraP and *Speedy* [54] are performance extrapolation tools for *pC++* [48] that use profile analysis in a certain environment to predict the performance metrics in a different environment. This technique can, for instance, be used to run the parallel program on a sequential machine and predict its performance in an environment with multiple parallel processors.

3.4 Visualization

Visualization is the stage in the profiling cycle that aims at identifying and understanding the profiled program. Different profiling tools use different profiling visualization techniques and there exists a wide range of presentation methods, from text-based tables to sophisticated three-dimensional graphics. Some selected approaches are discussed in the following:

gprof is an example of a profiling visualizer that solely uses text-based output in table format to present the gathered profiling information.

An example of a general visualization program for event traces is POET [41] [74]. It allows visualizing the generated event data on-the-fly or post-mortem, and supplies a flexible interface to define the graphical output.

ParaGraph [23] is another graphical display system for visualizing parallel program performance that can be used on a number of different message-passing multicomputer architectures. Using the profiler's trace data file, it can be used to pictorially replay the profiled events.

The *Chiron* [20] parallel program performance visualization system uses three-dimensional folded graphs to display larger amounts of profiling information relevant to memory and synchronization behaviour.

Through *SDDF* [2], *Pablo* [59] has become a tool with a wide range of visual and audio-visual presentation back-end modules: simple tables and bar-charts, kiviat graphs and three-dimensional graphical devices. Even virtual reality support (see [60]) and performance data "sonification" modules (see [47]), i.e., modules that represent profiling events by different audio patterns, are available.

Two other examples of tools that visualize parallel code are *PARAVER* [57] and *Tau* [5]. *Tau* is a language-specific program analysis tool for the *pC++* [48] environment that provides visualization modules to graphically present the gathered profiling information, including information about the *pC++* run-time kernel.

3.5 Target Programming Environment

Profiling tools can be designed to apply to general or language-specific programming paradigms. As discussed in Section 3.1, the necessary instrumentation insertion can be performed at different stages, which represents the profiling tool's programming language or system dependency. Profiling systems, where the instrumentation is inserted before or during compilation, normally apply to a specific programming language for certain computer environments. The most commonly supported programming languages are parallel dialects of Fortran, e.g., HPF (High Performance Fortran) [25], and variations of C/C++ [34] [66] with support for concurrent programming. Tools that perform the instrumentation by linking with modified libraries, binary re-writing or during the program's execution rather depend on the computer architecture than the underlying programming language.

JEWEL [42], *Quartz* [1], *TAU* [5], *Speedy* [54], *VPPB* [4], *Paradyn* [50] and *AIMS* [80] all support parallel versions of C or C++. *Paradyn* and *AIMS* additionally support HPF and allow profiling programs that make use of the PVM [73] message passing library. Commonly supported platforms are large-scale multi-processor systems like the IBM SP2 [67], the Intel Paragon [13], or the CRAY T3D [10].

Although the profiling tools presented in this chapter cover different aspects and programming environments, they do not incorporate facilities that cover performance analysis, algorithm analysis, coverage analysis, tuning and debugging (see Chapter 2) for user-level thread-based shared-memory multiprocessor environments given in the μC++ [8] concurrent programming system.

Most tools supporting multiple platforms are based on instrumented message-passing libraries, only support data parallel programming languages, or do not meet scalability or extendibility requirements. Dynamic instrumentation, for instance, is a promising approach to profiling large-scale long-running applications, but causes too much overhead to accurately profile smaller-scale short-running programs, as it requires running the profiled program under the entire control of the profiling monitor that performs the necessary instrumentation insertion. In addition, extendibility requirements are not met by many tools: the program analyst might want to add new metrics that reveal some information specific to the program under examination, aggregate the profiled data in a certain way, provide back-end interfaces to other visualization tools, etc.

Since retrieving monitoring information on a per-thread level requires the profiler to know about details intrinsic to the run-time system, a tight coupling between the profiler and the run-time system kernel becomes necessary. Although replacing the original run-time library by an instrumented one can handle this scenario, profiling at a very fine-grained level, i.e., a function or even statement level, can only be achieved when the profiler and the run-time kernel co-operate.

Chapter 4

The Target Environment

As discussed in Chapter 3, the design of a profiling tool for concurrent programs strongly depends on its target environment. Both the underlying concurrent programming language paradigms and the architecture of the programming environment play an important role. This work focuses on a concurrent object-oriented programming environment supporting user-level light-weight threads in shared-memory. Profiling user-level threads requires intrinsic knowledge about the run-time system; in this chapter, the most important aspects of the target system, μC++, as well as previous approaches to run-time kernel visualization for this system are discussed. Although this work mainly discusses issues dealing with the target environment described here, the ideas are applicable to numerous other environments.

The target programming environment of this work is a user-level thread library for C++, called μC++ [8]. Section 4.1 presents a brief discussion about the programming paradigms and the architectural and implementation issues of μC++ relevant to profiling. Some initial work for visualizing a μC++ program's activity has been done (see [6]) and is discussed in Section 4.2. Section 4.3 describes the built-in tracing functionality of μC++ and a visualization tool for event traces, called POET (see [41]).

4.1 μC++

μC++ [8] is a dialect of the object-oriented programming language C++ [66] introducing programming primitives to support concurrent entities, such as processors and tasks, and synchronization primitives, such as monitors and condition variables, semaphores, barriers and locks. In addition, it provides various communication facilities for precisely controlling the scheduling order at different levels of abstraction. At a task level, μC++ has real-time support and provides the possibility to implement user schedulers for tasks; at a mutual exclusion level, it supplies new mechanisms to control the order in which requests to enter a mutually exclusive object are served.

4.1.1 Implementation Issues

Mechanisms for supporting a concurrent programming environment in μC++ are provided at different levels: a modified pre-processor parses μC++ language primitives and performs a translation to C++. A standard C++ compiler generates the μC++ application's object code and links it to the μC++ run-time library, which implements the run-time system kernel and the concurrent programming functionality.

μC++ defines two different kinds of active entities operating at different levels: *processors* and *tasks*.

Processors in μC++ are virtual processors. μC++ operates in two modes: uni-processor and multiple-processor. In the uni-processor case, all processors are simulated by the run-time kernel inside of one UNIX process. In the multi-processor mode, each processor is implemented by a UNIX process, i.e., all scheduling for a processor is done by the underlying operating system. Only some parallel processor environments allow binding a UNIX process to a specific physical processor; therefore, the execution of a UNIX process cannot be predicted in any manner, which is an issue with respect to real-time programming. An example where binding a UNIX process to a specific processor is possible is the Sequent Symmetry [45]. Unless specified otherwise, in this thesis, the term *processor* refers to a μC++ processor and the term *process* refers to a

UNIX process.

Tasks in μC++ are user-level threads that are created and managed by the μC++ run-time kernel. Using user-level threads, as opposed to kernel-level threads, to implement a task has the advantage of greater portability, since still not all operating systems support multiple kernel-level threads. An example of an operating system supporting multi-threaded processes created and managed at the kernel level is the AIX 4.1 UNIX operating system. Tasks in μC++ are executed by μC++ processors and are preemptively time-sliced. Therefore, time-slicing is performed by the μC++ kernel among tasks, and time-slicing is performed by the underlying operating system among processors. In the following, only time-slicing and task scheduling at the μC++ kernel level is considered.

4.1.2 Clustering

In addition to creating and managing processors and tasks, μC++ introduces the notion of clusters, which binds tasks with processors. A cluster is a collection of tasks and processors, which can be defined by the programmer to administratively structure the concurrent program's active entities. Tasks on a cluster are only executed by the processors for that cluster. Thus, each cluster has to have at least one processor associated with it to execute its tasks. The order in which tasks are scheduled is defined by the policy of the cluster's scheduler, which can be replaced by a user. Migration of tasks or processors to other clusters during run-time is also supported.

4.1.3 Communication and Synchronization

Various synchronization facilities are available in μC++: locks, counting semaphores, barriers, and monitors with accept statements and condition variables. Monitors are objects in which all or some member functions are always executed in a mutually exclusive manner. For a further discussion on monitors see [27]. μC++ supplies very precise mechanisms to explicitly control which mutual exclusive member function executes next. Tasks in μC++

are implemented as monitors, i.e., a class with mutually exclusive member functions and a thread of control. This approach allows tasks to directly communicate with each other by making and accepting calls to their mutual exclusive member functions. Communication among all objects in μC++ is done through parameter passing, as opposed to message passing. Parameter passing has the advantage that it can be statically type-checked and it allows the implementation of a concurrent problem to be performed in a coding style that is also used in sequential programming.

4.1.4 Platforms and Memory Model

μC++ supports various uni- and multi-processor platforms with shared-memory and provides the same programming language primitives with identical interface for both the uni- and multi-processor case. As described in Section 4.1.1, in the uni-processor mode, all tasks and processors are managed by one UNIX process, whereas in the multi-processor case, each virtual processor is embedded into a UNIX process. In the latter case, it is assumed that the operating system distributes the different UNIX processes among the existing hardware processors. μC++ provides a shared-memory programming model and uses memory mapping between UNIX processes in the multi-processor case to make the multiple UNIX processes share a common memory.

Researchers in parallel computing generally agree that it is important to support a shared-memory programming model, since it does not burden the programmer with orchestrating all inter-task communication through explicit messages.

μC++ is currently implemented on various shared-memory multi-processor computer architectures. Recent research in developing large-scale parallel machines has shown that message-based processor communication is only more effective than shared-memory for distributed data-parallel applications and a few other specialized situations. For a more detailed discussion on architectural support for both shared-memory and message-passing, see [40].

4.1.5 Profiling μC++ Programs

There are many challenges when profiling a concurrent program in an environment similar to μC++. First, since the environment provides user-level tasks, the profiling tool must be able to profile at a task level. This requirement is non-trivial and has to be considered when designing both the run-time environment and the execution monitoring profiling tool. Knowing about the existence of user-level threads is not enough; the profiler has to know the exact locations where the information representing the separate tasks' state is stored and has to be able to correctly handle the run-time system's scheduling mechanisms.

Shared memory is the second challenge when designing a profiling tool for μC++, since the code segment is shared among different tasks. If the code segment is not shared, interesting profiling locations can be instrumented, and each hit in the instrumented code during execution can implicitly reveal which task is executing. In the case of shared code, it is impossible to identify the task that is executing the instrumented block of code by simply identifying where the instrumented segment is located. Thus, knowledge intrinsic to the run-time kernel has to be made available to the profiler.

A third challenge arises from the requirement that a profiling tool should be able to perform both exact and statistical profiling on different levels of abstraction, i.e., from a very fine-grained level to a coarse-grained level, e.g., function or statement level through to high-level language constructs, where the possible abstraction depends on the concurrent primitives supplied. In μC++, it should be possible to build profiling metrics that perform exact profiling on task creation, destruction and communication.

Further design considerations concerning a profiling tool for μC++ are discussed in Chapter 5.

4.2 MVD

Some initial work on monitoring and visualizing a μC++ program's run-time kernel activity has been done with the *Monitoring, Debugging and Visualization (MVD)* tool set [6]. The MVD tools and libraries allow building monitoring samplers for μC++ programs that visualize the current state of a running program.

4.2.1 Thread-Safe X/Motif Support

The visualization primitives are provided by a modified version of the X11/R6 Window System [17], and using the Xt [49] and Motif libraries. The modified version for μC++ allows thread-safe access to the available graphic routines. All X requests are handled by a server task that executes on its own μC++ cluster. Using the X/Motif visualization routines, a programmer can write application-specific visualization modules that display the program's behaviour and help in performing the algorithm analysis. Examples of such visualized concurrent programs are also described in [6].

4.2.2 Watchers and Samplers

The MVD toolset for μC++ allows monitoring a program's execution-time behaviour by supplying *watcher* and *sampler* objects. Samplers and watchers provide a mechanism to retrieve information about a program's run-time behaviour. A watcher is a task that periodically invokes routines of one or more sampler objects registered with the watcher in order to collect information or to display the collected information on the screen. Sampling can be done on any memory location or accessible variable. For instance, the watcher-sampler mechanism allows monitoring kernel-intrinsic information, such as the current task's stack sizes and its current state of execution. The sampled information is analyzed and visualized on-the-fly. Figure 4.1 shows an example graphical output of a collection of sampler and watcher objects visualizing the current system state at different levels of detail, i.e., at a system level, a cluster level and a task level.

The MVD toolset does not supply any mechanisms for exact profiling or statistical profiling on a fine-grained level. Also, it requires some effort from the programmer to choose and include the watcher and sampler modules required. Nevertheless, the watchers and samplers allow statistically monitoring of a program with respect to its kernel activity and provide a first mechanism to gain profiling information about a concurrent μC++ program.

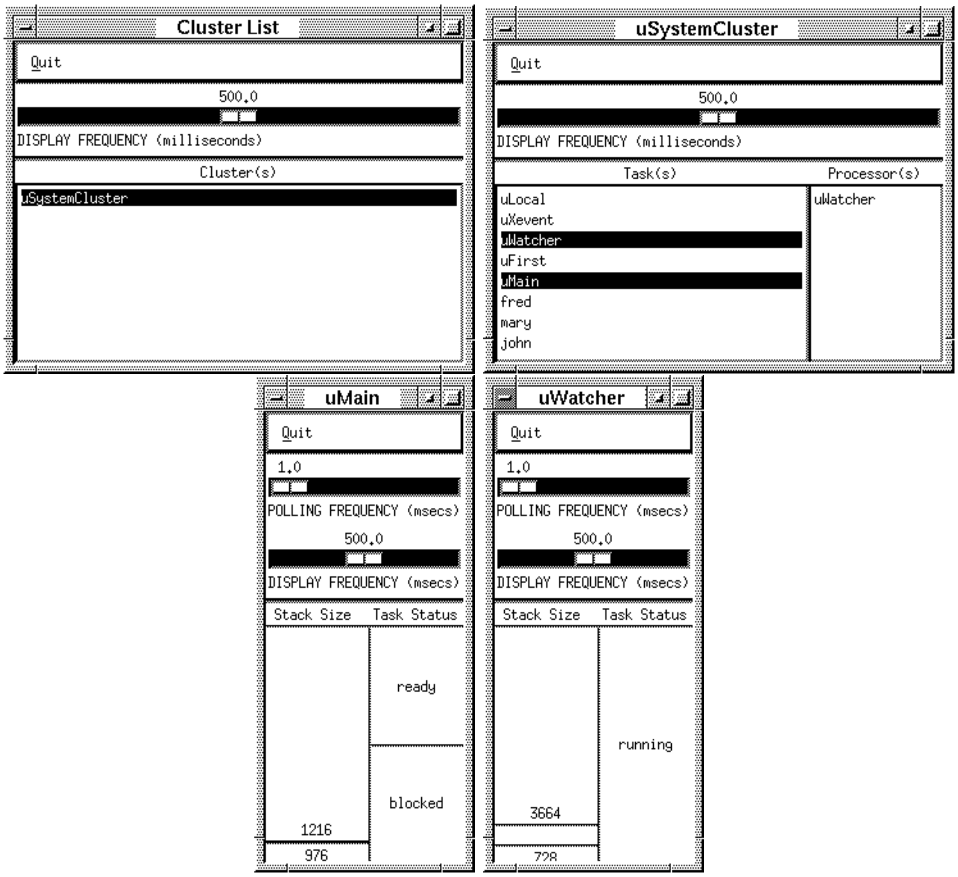

Figure 4.1: MVD Watchers and Samplers.

4.3 μC++ Built-In Tracing

μC++ has built-in tracing support that allows generating an event trace of communication and synchronization events for concurrent programs. This event stream can then be sent to an event trace visualizing tool. POET [41] is such a visualization tool providing a general configurable interface that accepts incoming events and graphically displays them. An interface to accept μC++ trace events has been developed and is described in detail in [74]. μC++ supports generating trace events for both active concurrent entities, such as tasks and processors, and passive concurrent entities like monitors and semaphores.

Figure 4.2 shows an example event trace of a simple concurrent consumer-producer program. The consumer (`cons`) and the producer (`prod`) synchronize insertions and removals

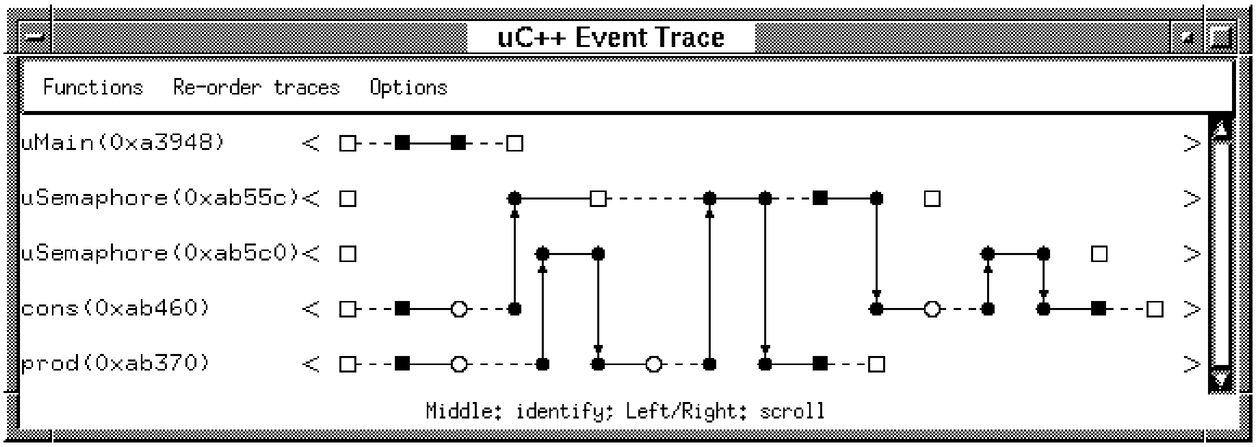

Figure 4.2: μC++ and POET: Event Trace.

from a buffer by making calls to semaphores as shown in the figure. In the current version of μC++, inserting the necessary code to produce the trace information is done at the translation level and provides information about concurrent language constructs. Only the relative order of the events is recorded, but no information is collected about the duration between two subsequent events. (However, POET is capable of supporting real-time events.) Nevertheless, a certain degree of exact profiling is achieved and the cooperation between μC++ and POET allows debugging and performing an algorithmic analysis of a concurrent program by visualizing the communication and synchronization events.

Chapter 5

The μProfiler Design

The discussions of Chapter 3 and Chapter 4 illustrated that previous work in profiling concurrent applications has been done and various profiling techniques and methods for different environments are available. Nonetheless, with the existing tools and techniques described, it is impossible to perform both exact and statistical, fine-grained profiling on a per-thread basis when using a shared-memory user-level thread library like μC++: the profiler must understand the specific architecture of the thread library and run in cooperation with the run-time kernel in order to provide detailed and thread library specific information.

As part of this thesis, I have designed and implemented an extendible prototype profiler for μC++ called *μProfiler*, which meets these requirements and allows exact and statistical per-thread profiling at a fine-grained level.

This chapter describes the design of μProfiler and Chapter 6 discusses the relevant implementation issues.

5.1 Design Objectives

The design objectives for the profiler prototype developed are derived directly from the requirements discussed earlier. The following sections summarize the design goals in more detail.

5.1.1 Profiling on a Thread Basis

When writing concurrent programs, the programmer creates different threads of control, which have to be independently monitored by the profiling tool. This requirement has the consequence that the profiler must know how the underlying thread-management system, i.e., the μC++ kernel, manages multiple threads of control. Also, since μC++ can operate in both a uni- and multi-processor environment, the profiler should work in both environments as well.

5.1.2 Profiling at Different Levels of Detail

Different metrics require profiling to be done at different levels of detail. For μC++, this requirement means that the profiler must be designed in such a way that it has the potential to profile at a cluster, at a processor, at a task and even at a function level. Techniques to collect information at a statement level should be provided as well, for the rare cases where profiling at a function level is insufficient. Nevertheless, since all instrumentation increases the probe effect, the instrumentation insertion should normally be done only at a function level; if finer-grained profiling is required, other mechanisms have to be invoked. Thus, another general and important design consideration is to ensure a reasonable level of accuracy and to minimize the probe effect introduced.

Both exact and statistical profiling have to be supported by the profiling tool. It should be possible to build metrics that support exact and statistical profiling at different levels of detail.

5.1.3 Selective Profiling

Selective profiling means that profiling is integrated into the system in such a way that the program analyst can specify which modules are profiled. Therefore, the design of the profiler has to ensure that instrumentation insertion and profiling is only done for the specified modules, and that instrumented program or library code and code that is not instrumented interact correctly.

Also, the profiler should become an optional tool in the μC++ tool set, i.e., a full installation of the profiler should not be mandatory to write and run μC++ programs.

5.1.4 Support Different Visualization Devices

Different profiling modes calculate different metrics and require different visualization techniques. μProfiler should provide various graphical and textual visual devices to present the collected profiling data in an informative manner. The basic types for visualization devices, such as tables, bar charts, or even more advanced devices, e.g., Kiviat graphs, should be supported in order to allow building various types of metrics.

5.1.5 Extendibility

One of the main design objectives for μProfiler is its extendibility. When profiling concurrent programs, situations can occur, where the program analyst may want to add functionality to the profiling tool. This new functionality can consist of new metrics, new analyzing routines, new visualization devices, etc. The design of the profiler should not preclude the program analyst from adding functionality to the profiler, but rather encourage it. μProfiler should provide a modular interface, which allows easy addition of new profiling facilities, or possibly modifying or replacing existing ones.

5.1.6 Portability, Interoperability and Maintainability

The μC++ system is supported on various computer architectures and operating systems, including SunOS, Solaris, Ultrix, IRIX, AIX and Linux on different processors. Therefore, the design of the profiler should not preclude a port to any of these or other computer architectures.

Since μC++ provides various tools to help develop concurrent programs, such as a debugger for μC++ programs (see [7]) and the MVD tool set, μProfiler should be designed so that it does not restrict existing functionality; for example, profiled programs should still be debuggable, and event trace visualization should still be possible.

Maintainability is another important design aspect in any software development. The design of μProfiler should allow adapting the underlying system to new versions with as little modifications to the profiler code as possible. The goal is best accomplished if μProfiler is implemented in a high-level object-oriented concurrent programming language.

5.2 Design Considerations

This section discusses what design considerations have to be taken into account in order to fulfill the design objectives stated in Section 5.1.

In order to incorporate profiling into μC++, work had to be done at two different levels: first, some new features had to be added to the existing μC++ environment to support profiling, and second, the application that performs the actual profiling had to be developed.

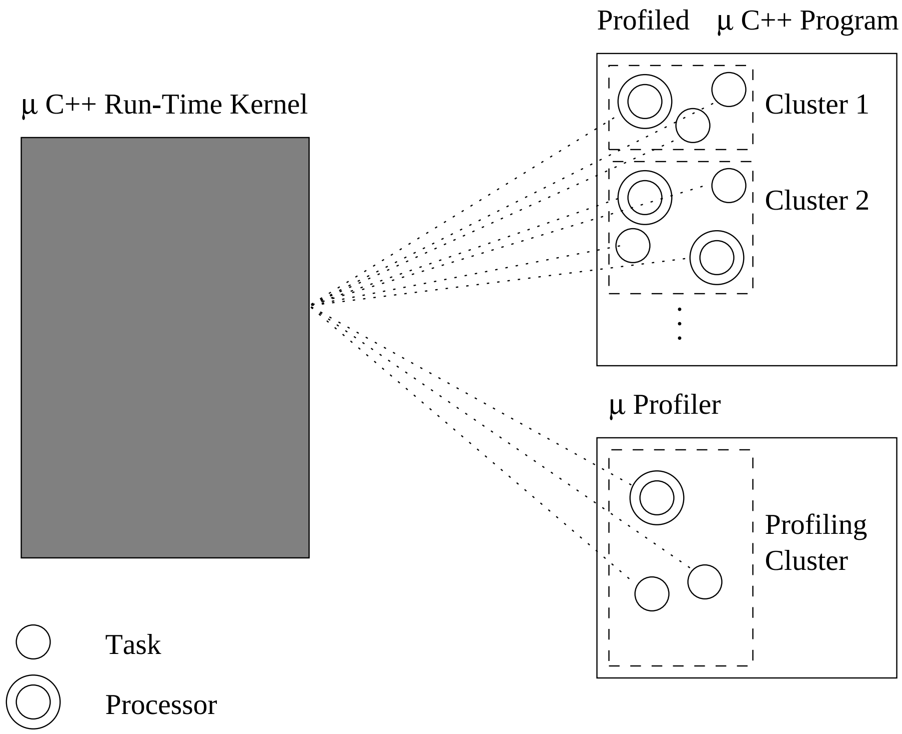

Figure 5.1: The μProfiler as a μC++ Application.

The profiling application is designed to be a concurrent program written in μC++, and executing in parallel with the profiled application on its own cluster and processor. Figure 5.1

illustrates this scenario. The μC++ run-time kernel co-operates with the profiler and is responsible for its execution in parallel with the profiled application. This tight coupling between the kernel and the profiler allows the profiler to access data structures intrinsic to the run-time kernel while maintaining a maximum degree of interoperability and flexibility.

Since the profiler application operates on its own cluster and is executed by its own processor, the influence on the profiled program's run-time behaviour is kept to a minimum; this is especially true in the multi-processor mode of μC++, since the profiler application executes inside its own separate UNIX process and does not share the profiled application's code image. Note that the data section of the program is shared, but the profiler performs read-only operations, which do not interfere with the profiled program. Nevertheless, the program analyst has to be aware of the fact that, as with any other possible profiling architecture designs, the profiler still introduces some effect on the execution behaviour due to increased bus traffic and memory usage.

5.2.1 Incorporating Profiling into μC++

When compiling a μC++ program, different flags are available to specify whether, for example, debugging or tracing code is generated or what running mode is used, i.e., uni-processor or multi-processor. Profiling must be incorporated in a similar way into the system; that is, when compiling a program, the programmer can specify whether the program is to be profiled by simply adding a command line parameter when invoking the compiler. In the case of profiling, the necessary instrumentation gets inserted and the program is linked with the profiling libraries that supply the profiler application code. Thus, in the simplest case, the instrumentation insertion and activation of the profiling modules is completely transparent to the programmer.

Nevertheless, to equip the programmer with precise control about what parts of the program are profiled, routines were added to the μC++ run-time system, which allow turning profiling on and off for a particular thread at any point during its execution. In addition, the programmer can compile parts of the program with the profile flag and other parts without the profile flag and then link them together, thereby creating an executable where only some parts are instrumented and profiled.

Allowing exact profiling on a per-thread basis requires certain *hooks* to be inserted into the run-time kernel, which are used to register important events with the profiler that would be missed with purely statistical monitoring. These hooks have to be designed so they become active only when a profiler application is present; i.e., the existence of the profiler must be checked for dynamically inside the run-time kernel. This approach has the advantage that it becomes possible to build user or system libraries that automatically and dynamically detect when profiling is active and activate the necessary hooks, which then register with the profiler.

Integrating the profiler into the profiled program clearly has the disadvantage that it increases the size of the instrumented executable, but it has the advantage that profiling becomes possible with minimal effort by the programmer, since it is only necessary to re-compile the program with an additional command line flag and then start the program to be profiled again in the same manner it is started without profiling activated. Also, the necessary tight coupling with the run-time kernel and the required accuracy of the collected profiling data are other reasons for having the profiler operating in the same address space as the application itself. The disadvantage of the larger executable file size resulting from the profiler may be avoided in future versions by dynamically linking only the necessary functions during run-time. Initial related work for supporting general link and unlink editing of single functions and modules during execution for a variety of computer platforms has been done and is described in [26].

The instrumentation insertion method used for μProfiler is static instrumentation insertion at compile-time using shared trampolines; the instrumentation points are function entry and function exit. Dynamic instrumentation is considered too expensive when profiling programs with short or intermediate execution times. Profiling with no instrumentation insertion would require developing a whole new run-time kernel for the μC++ system and would introduce too great a probe effect (see Section 2.1.1). Profiling on a thread basis using shared trampolines requires maintaining a separate data structure for each μC++ task. In these data structures, the current profiling information about the executing tasks is maintained, i.e., information about the current function's address, the address of the function from where the current function was called, etc. A pointer was added to each μC++ task object pointing to its corresponding profiling data structure in order to allow an executing task to correctly update the profiling data structures inside a shared trampoline. As profiling data structures for each task are independent, no mutual exclusion is required inside of the shared trampoline where the profiling data is updated. Since the per-thread data structures are created by the profiler, it knows their location and can therefore sample the data structures to retrieve the necessary information about the current location of the different tasks in the profiled program. This mechanism is used when statistically monitoring tasks.

5.2.2 The Profiler as a μC++ Program

As stated before, the profiler is a μC++ program, i.e., a concurrent object-oriented program, operating on its own cluster through its own processor. To ensure high flexibility and extendibility, it is split into different parts representing the underlying functionality. This section briefly discusses the design considerations of the different parts, i.e., the *μProfiler kernel*, *μProfiler execution monitors*, *μProfiler metric analyzers* and *μProfiler visualization devices*, and their purpose.

μProfiler Kernel

The core part of μProfiler is the *μProfiler kernel*. It has its own thread of control and acts as an administrator [16] handling incoming profiling events when exact profiling is selected, or performing the sampling for statistical profiling. It also provides an interface for other profiling modules. For instance, when *μProfiler execution monitors* or *μProfiler metric analyzers* are created, they register with the *μProfiler kernel*, which manages them from that point onwards. This design allows any combination of monitors and any combination of analyzers to be used during the same profiling session. In addition, it facilitates the implementation and integration of new profiling monitors or analyzers in the system. The *μProfiler kernel* has complete access to the μC++ run-time kernel's data structures and methods, and is created before the profiled application. Since it is possible to dynamically turn profiling on and off for each task, the profiler kernel must be aware of the current state in terms of profiling for each profilable task.

μProfiler Execution Monitors

Monitoring the profiled program's execution behaviour is performed by *μProfiler execution monitors*. All μProfiler monitors have the same base type and register with the profiler upon creation. The program analyst is able to interact in the selection of the profiling metrics after the μProfiler kernel is created, but before the profiled part of the application is started. Corresponding to the choice of the profiling metrics, only the necessary execution monitors are created. An extendible general base type for *μProfiler execution monitors* is provided, with an interface for both statistical and exact monitoring functionality at different levels of detail. The execution monitors are designed as passive objects, i.e., they are executed by the μProfiler kernel's thread. Each execution monitor that supports statistical sampling is assigned a sampling frequency, and the μProfiler kernel invokes the monitor's sampling member routines at that frequency. The sampling frequency should be changeable by the user before profiling starts and dynamically resettable by the execution monitors during execution.

μProfiler Metric Analyzers and Visualization Devices

In most situations, an execution monitor has a corresponding *μProfiler metric analyzer* object that performs the profile analysis for the profiling data collected. Normally, the μProfiler metric analyzers are created after the program's execution has completed; i.e., a post-mortem profile analysis is performed. Nonetheless, the modularization of the profiler does not preclude a dynamic profiling analysis.

The analyzers manage their corresponding graphical data visualization devices. Since μC++, in combination with the MVD tool set, provides facilities for graphical user interface programming using X Windows/Motif, the visualization devices were developed using this existing library. In order to supply a high degree of flexibility in creating and re-using visualization devices, a library of basic data visualization widgets is being developed rather than directly integrating the code for the visualization devices into the profiler application.

μProfiler must incorporate metrics based on statistical profiling and metrics based on exact profiling at a function-level. Analysis and visualization of the profiled program's operating system resource usage and metrics representing the load of the underlying system during execution are additionally provided to allow verifying the accuracy of a test run. On the platforms supported by μC++ , this important piece of information is available through the /proc file system [70] or by the getrusage system command [69].

Another important design aspect is to supply different hierarchies of abstraction in the profile analysis and visualization stages. Profiling information is accessible beginning at a high level of abstraction and information aggregation, for example, by exposing the list of the system's profiled clusters with summarized profiling data, and then allowing the analyst to enter levels that reveal more detailed information, such as information about the profiled tasks on the selected cluster.

5.3 Static Design

In this section, the static design aspects of μProfiler are discussed. Section 5.3.1 gives an overview of the underlying object-oriented analysis model for μProfiler, and further static design issues are delineated in Section 5.3.2.

5.3.1 Design Model Overview

The static design of μProfiler is presented using an object-oriented analysis model, whose main parts are illustrated in Figure 5.2 and Figure 5.3. The notation used conforms to the object-oriented analysis modeling techniques introduced in [9], with one exception: classes and objects, where the member functions and attributes are not shown, are drawn in a compressed format. (Appendix A summarizes the relevant notations used in the object-oriented analysis model presented here.) For clarity reasons, object messages are not shown and only the member functions and attributes of the objects and classes that are important to understand the conceptual design are displayed.

μProfiler Kernel and Execution Monitors

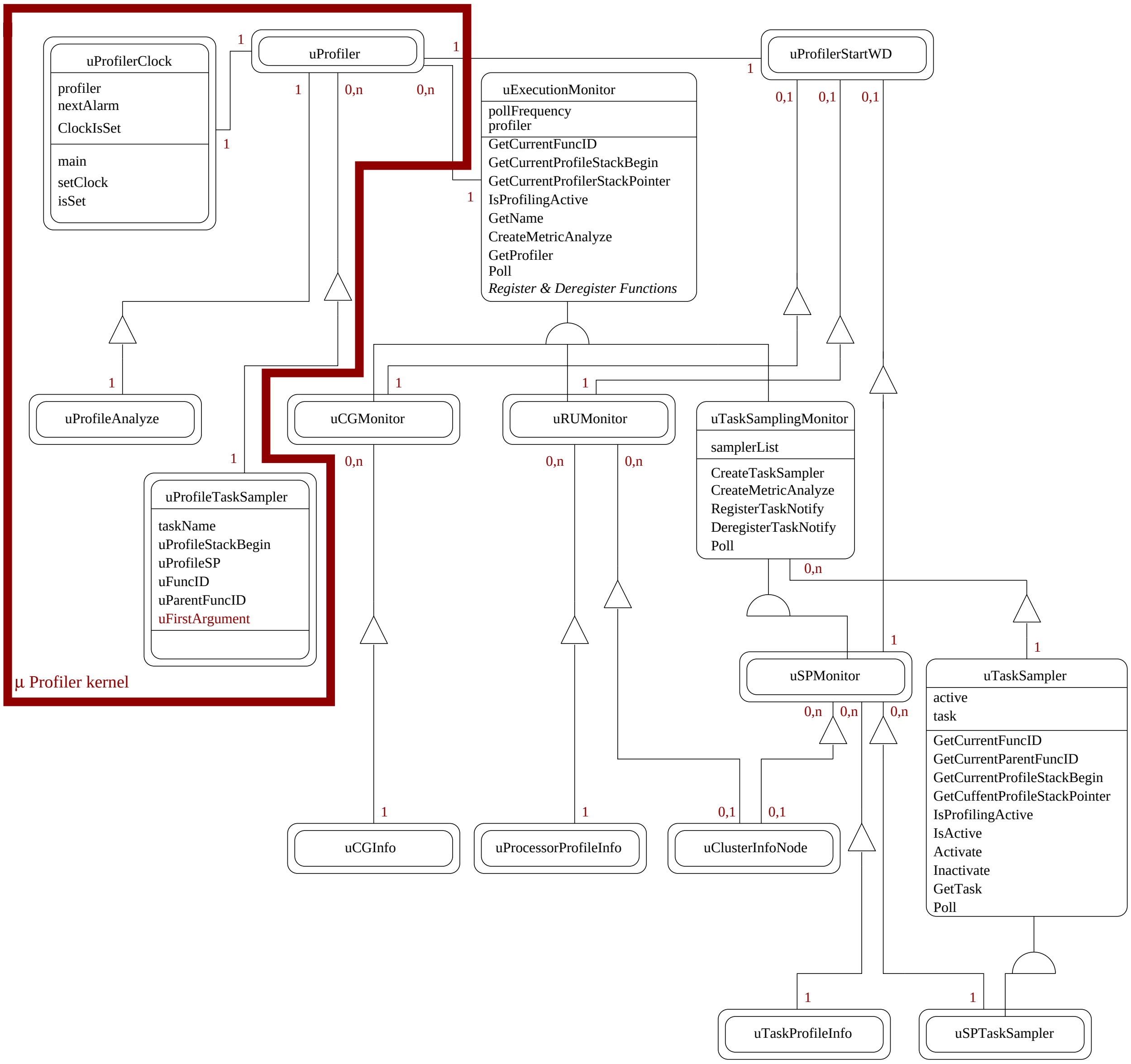

Figure 5.2: Object Model of the μProfiler Kernel and Execution Monitors.

Figure 5.2 shows the object relationships of the μProfiler kernel and the execution monitors. The μProfiler kernel consists of the objects uProfiler, uProfileAnalyze, uProfileTaskSampler and uProfilerClock. uProfiler and uProfilerClock are tasks communicating with each other to perform statistical profiling at a certain frequency (for more details, see Section 5.4).

uProfileTaskSampler is a passive object that contains the data structures needed to represent a task's state information and corresponds to *one* profiled task. The data structures needed for profiling encompass a profiling stack where the current call stack information of the task is stored, and information about the currently executed function (see Section 6.1.2); when in statistical profiling mode, these data structures are read periodically by the sampling execution monitors. uProfileAnalyze is the object executed once the profiling data is ready for analysis. uProfileAnalyze creates the necessary specific analyzers that perform the analysis. The object relationships of uProfileAnalyze are described in the next sub-section and are shown in Figure 5.3.

uExecutionMonitor is an abstract class providing the basic functionality for both statistical and exact profiling. All execution monitors inherit from it and specialize the virtual member functions needed for their specific purpose; examples of such member functions are Poll(), which is the member function that is invoked at the execution monitor's frequency when performing statistical profiling, or the register and deregister functions invoked when a processor or task is created or finishes. Since uProfiler invokes these specialized member functions, it must know of their presence for each active execution monitor. Therefore, upon its construction, each execution monitor has to register all necessary member functions with uProfiler. Using the virtual member function mechanism, it is possible to determine if a member function of a derived class specializes a certain member function of the base class, by comparing member function pointers of the base class and the derived class. This mechanism is used by uExecutionMonitor to facilitate the implementation of new execution monitors. The member function Initialize() provided by uExecutionMonitor can be used to dynamically check what virtual member functions are overwritten and registers them accordingly with uProfiler. Thus, adding a new execution monitor to the profiler simply requires inheriting from uExecutionMonitor, providing special purpose versions of the member routines that have to be called by the profiler kernel, and last, making an initialization call to Initialize().

uProfiler maintains a list of all execution monitors and their member functions and invokes them during execution as needed. uExecutionMonitor also provides access functions to get information about the current task's state, and therefore, allows type-safe read-only access to data structures internal to the run-time kernel or the profiler. A special purpose execution monitor developed for statistical monitoring on a task level is supplied by the uTaskSamplingMonitor class. A uTaskSamplingMonitor object maintains a list of uTaskSampler objects, one for each task on each cluster to be sampled.

uCGMonitor, uRUMonitor and uSPMonitor are three execution monitor prototypes. uCGMonitor supplies exact profiling at a function level. For each profilable task on each profilable cluster, it collects information about the number of calls to each function and each of its callers. uRUMonitor collects information about the run-time resource usage and load of the underlying operating system. uSPMonitor statistically samples a task measuring the time the task spends on a certain cluster in a certain function in a certain state. For instance, it collects data about how long the profiled task is in the blocked state inside of an I/O routine on the I/O system cluster. All information gathered by uSPMonitor is based on statistical sampling. Each execution monitor is responsible for operating and updating its own objects and possibly accumulating or summarizing the profiling data collected. These operations on the profiling data structures are encapsulated in the objects uCGInfo, uProcessorProfileInfo, uClusterInfoNode and uTaskProfileInfo, respectively.

μProfiler Metric Analyzers and Visualization Devices

The object model of the metric analyzers and visualization devices is shown in Figure 5.3.

Corresponding to the execution monitors described in the previous subsection, there exist different metric analyzers, i.e., uRUAnalyze, uCallGraphAnalyze, uSPAnalyze and uSPClusterAnalyze, which all inherit from the base type uMetricAnalyze. All metric analyzers are created and managed by uProfileAnalyze. uMetricAnalyze is the class providing the basic facilities to create and manage selection windows. Since displaying profiling information in a hierarchical manner is an important feature to allow efficient maneuvering through the presented data along a particular dimension, selection windows are important; they display a list of selectable objects, which represent the next level of detail. For instance, a selection window, see Figure 5.4, shows the list of profiled tasks in the system (left box) and reveals more

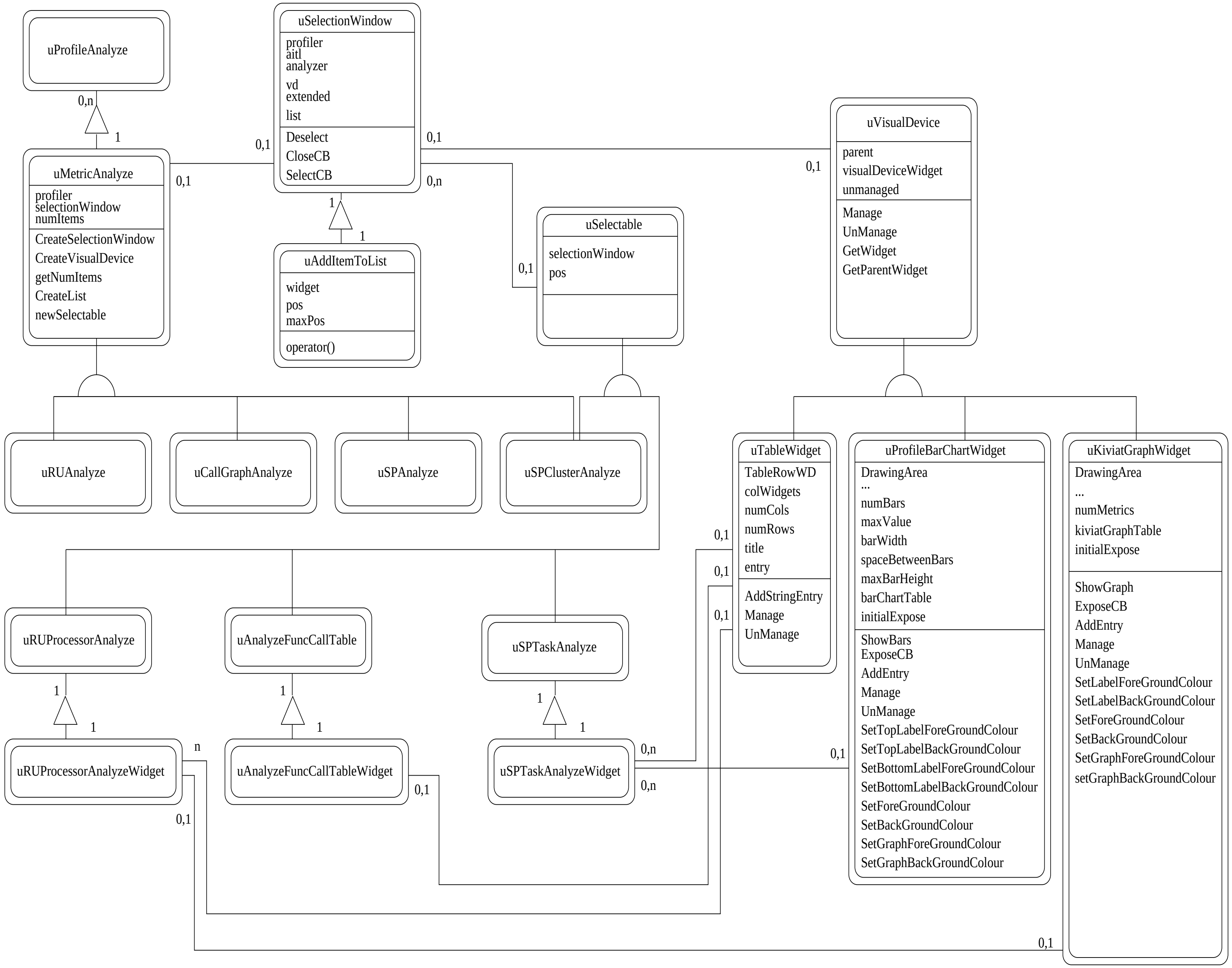

Figure 5.3: Object Model of the μProfiler Analyzers and Visualization Devices.

detailed information upon selection of a task. uSelectionWindow is the class that provides this functionality.

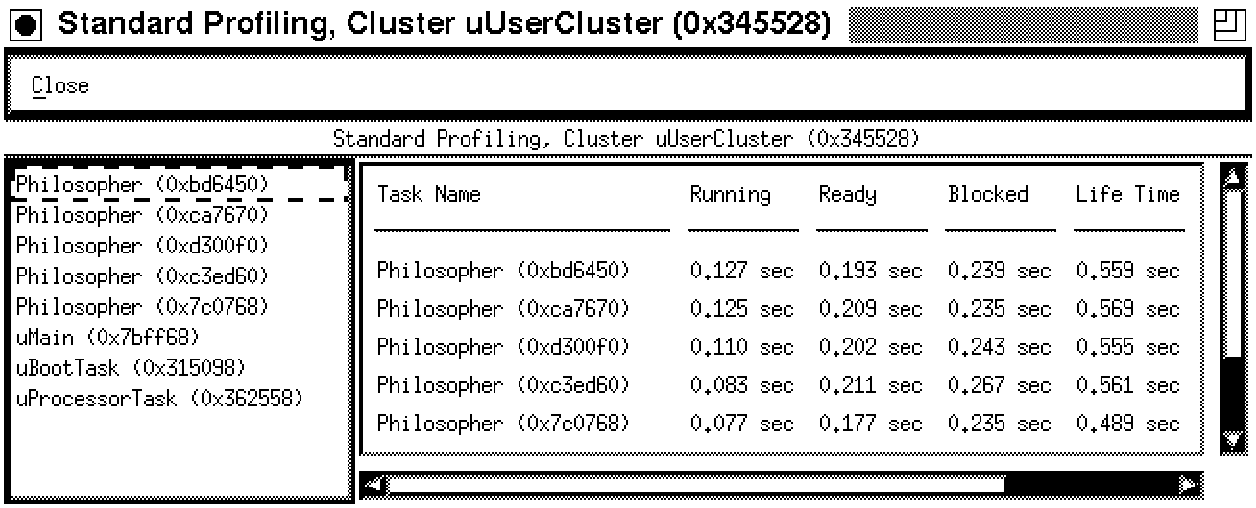

Figure 5.4: An Example Selection Window.

As illustrated in Figure 5.4, a selection window can also contain an additional visual device displaying information about the different list items. In the example selection window, an additional visual device shows information about the time each task spent in different execution states (right box). The corresponding objects to each selectable list item inherits from the class uSelectable to provide a type-safe precisely defined interface to uSelectionWindow. Visual devices are all derived from the uVisualDevice class and provide different mechanisms to expose information. uTableWidget shows the data in table format as illustrated in the right box in Figure 5.4. uProfileBarChartWidget and uKiviatGraphWidget visualize the information in a bar chart or a kiviat graph, respectively. uRUProcessorAnalyzeWidget, uAnalyzeFuncCallTableWidget and uSPTaskAnalyze use these visual devices to appropriately display the profiling information collected.

5.3.2 Further Static Design Aspects

Accessing the Symbol Table

Accessing the symbol table of the profiled program is important for many reasons: The profiler must know, for example, the address of all functions to determine in which function the currently profiled task is executing. Also, when visualizing the profiling information, object and function names are important. Since the symbol table information is dependent on the underlying computer architecture and file and process formats, a common interface is required to encapsulate the architecture dependent specifics. This interface is provided through the uSymbolTable class, which maintains a necessary subset of the symbol table entries and supplies member functions to retrieve the address and the name of a function. Also, uSymbolTable allows checking whether a certain function is a class member function, which is important when profiling on an object instance basis (see Section 5.4.4). The symbol table information is obtained by using the *Binary File Descriptor Library* [12], which implements the architecture dependent details and provides an interface to inspect and demangle symbol table entries.

5.4 Dynamic Design

As discussed in previous sections, the profiler is a μC++ program, which is integrated into the profiled application's code. Thus, starting the profiled application first starts the profiler and its start-up menu appears, where the program analyst selects the profiling metrics to be applied. After the program analyst has accomplished this selection, the profiled application is started and its execution behaviour is monitored by the profiler.

This section discusses the crucial dynamic design aspects of μProfiler. Section 5.4.1 describes the communication among the tasks involved in profiling, Section 5.4.2 discusses design issues concerning profiling information filtering, Section 5.4.3 describes thread-based profiling, Section 5.4.4 discusses issues concerning object-based profiling, and Section 5.4.5 briefly introduces some ideas concerning dynamic sampling frequency adaption for statistical profiling.

5.4.1 Task Communication

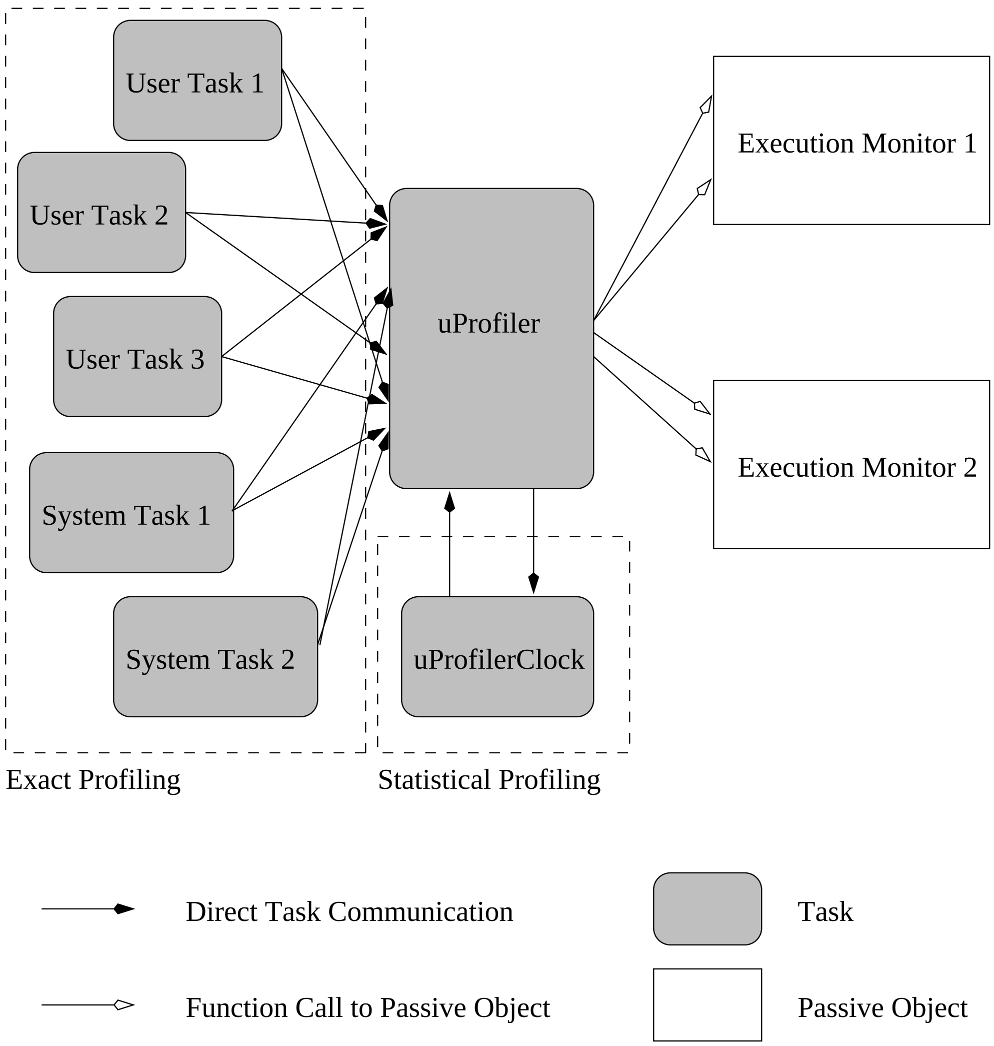

Figure 5.5: Dynamic Design: Task Communication.

Figure 5.5 illustrates how the uProfiler task communicates with the other tasks in the profiled program and how the execution monitors are invoked. The example scenario consists of three user-tasks, two system tasks and two execution monitors. User tasks are created in the profiled program's user code, whereas system tasks are tasks that are intrinsic to the μC++ run-time kernel. Note that all task communication is direct, i.e., communication is performed through calls to a task's mutual exclusive member functions. When exact profiling is applied, the user and system tasks call into the profiler upon reaching different profiling hooks implicitly inserted into the code (see Section 5.2). Since the uProfiler task works as an administrator, it accepts these calls in a mutual exclusive manner and executes the corresponding member routines of all execution monitors that have registered with the profiler for this specific profiling hook. To achieve greater concurrency, the work that is performed inside the mutually exclusive member routines, where the profiled tasks call in, is reduced to a minimum; i.e., the necessary information is copied into data structures internal to uProfiler and the caller exits the member function immediately, thereby returning control to the profiler. Then, the uProfiler task can safely read the contents of the communication data structures and pass the information to the corresponding execution monitors.

In the case of statistical profiling, another task, uProfilerClock, communicates with the uProfiler task. Whenever uProfiler has finished a statistical profiling cycle, it sets the alarm clock uProfilerClock, which is also part of the μProfiler kernel, by making a call to setClock(). A statistical profiling cycle in this context means the work that is done by the uProfiler task for all ready sampling events. If there are no other events to process, uProfiler puts itself to sleep by blocking for incoming events. uProfilerClock waits the specified time and wakes up the uProfiler task by calling its member function WakeUp(). Thus, this mechanism allows the uProfiler task to be ready to accept calls to all of its member functions at any time, unless it is in the process of performing work itself. Also, the design allows each metric to have a different and possibly variable sampling frequency.

Notice that an execution monitor does not have its own thread of control, i.e., execution monitors are passive objects as opposed to active tasks. All the work performed by the execution monitors is done by the uProfiler task. The main reason for this design model is to lessen the scheduling and synchronization overhead on the profiler cluster, because only one dedicated processor executes on the profiler cluster. This processor performs the work of the profiler, i.e., the uProfiler and uProfilerClock tasks, and it does not do any other work. Since only one processor is executing on the profiling cluster, changing the currently passive uExecutionMonitor objects to active tasks would unnecessarily increase the scheduling and synchronization overhead. Nevertheless, when running μC++ on a large-scale parallel machine, it might become desirable to assign more than one processor to the profiling cluster. In this case, it is necessary to make each execution monitor a distinct task, thereby providing greater concurrency. Since the current design already encapsulates the corresponding functionality into different objects and provides a well-defined interface to uProfiler, changing the currently passive objects to active tasks can easily be accomplished if required.

5.4.2 Filtering

Filtering is a very important mechanism that should be supported by any profiling tool. The purpose behind filtering profiling data is two-fold: first, filtering should be done to reduce the amount of profiling data collected to a relevant subset; second, filtering information should be done when presenting the analyzed profiling information to a user. In the latter case, the purpose of filtering is to provide a mechanism to focus on a certain section of the analyzed profiling information according to a certain, possibly selectable, perspective. Thus, two filtering mechanisms can be distinguished and are supported by μProfiler: *dynamic filtering* and *post-mortem filtering*.

Dynamic Filtering

The general design of μProfiler supports dynamic filtering facilities in several ways. Since collection and processing of profiling information is the responsibility of the execution monitors, they can immediately filter the incoming profiling information, or simply collect everything and leave the processing work for the corresponding metric analyzer. A certain degree of dynamic filtering is embedded into the design of the statistically profiling monitor uSPMonitor. uSPMonitor allows detecting on-the-fly whether a profiled function belongs to the run-time kernel, a μC++ specific library or the actually profiled user program; it filters all profiling information that is not part of the user program and charges any calls to kernel and library functions to the user function calling them. This mechanism is especially useful when profiling long-running applications, since the profiling information collected is reduced to a minimum. Therefore, similar to dynamic instrumentation, dynamic filtering allows profiling large-scale programs. Also, dynamically filtering kernel specific information protects the user from being overwhelmed with information that is part of the run-time kernel and not part of the user code. It is imaginable to integrate even more intelligent filtering facilities into execution monitors: they could, for example, not only filter the incoming profiling events, but additionally assign different weights to the collected information, or possibly discard already collected profiling information, which is considered to be unimportant, thereby freeing important memory space.

Post-Mortem Filtering

Post-mortem filtering is used to reduce the information collected, and possibly aggregate and categorize the profiling data in order to permit a human understandable presentation of the data. Post-mortem filtering is performed by the metric analyzers, which are invoked after the program's execution has finished. For example, the uCallGraphAnalyze analyzer uses post-mortem filtering to mark profiled run-time kernel and library functions differently from profiled functions in the user's code. Thus, the program analyst can easily identify the parts of the profiled program that belong to the actual user code. Other applications of post-mortem filtering are imaginable: for instance, when performance analyzing large-scale programs, irrelevant data values could be filtered out to help the program analyst focus on the actual performance bottlenecks. Also, depending on the metric used, wrong profiling information could be detected by the metric analyzer and be filtered out.

5.4.3 Thread-Based Profiling

Thread-based profiling requires retrieving the state information of each executing task in a safe, robust and correct way. When performing exact profiling, the profiled tasks register important events with the profiler to assure that no events are missed. Unfortunately, no other mechanism exists that satisfies the requirement of guaranteeing the detection of all events, but introduces less of a probe effect. In the case of statistical profiling, the situation is more difficult to judge. Theoretically, some statistical profiling can be done without any instrumentation insertion, but it is impossible to get all the state information of each task

Data Structure	Description
funcID	Address of the current function
parentFuncID	Address of the function from where the current function is called
firstArgument	First Argument to the Function (if existing)
uProfileStackBegin	Base of the Profiling Stack
uProfileSP	Current Profiling Stack Pointer

Table 5.1: A Task's Profiling State Information.

without instrumentation insertion. For instance, the currently traversed function cannot be identified without help from the executing task, since on most architectures it is impossible to read the current program counter of another task. Therefore, a different design approach is necessary: each profiled task updates its own profiling data structures and the uProfiler task periodically reads them from inside all activated statistically profiling execution monitors. The state information updated by each profiled task on a function entry consists of the table items in Table 5.1.

funcID and parentFuncID are used to identify the executing task's location, i.e., the current function and the current function's parent. If it exists, firstArgument holds the value of the first parameter passed as an argument to the function. This information is used for object-based profiling (see Section 5.4.4). uProfileStackBegin and uProfileSP are used to store the task's call stack information, which is needed for nested function calls and explained in more detail in Section 6.1. These data structures contain the information that cannot be directly retrieved from inspecting the run-time kernel, and therefore, have to be updated by the profiled task itself. I believe that the additional probe effect to update this state information cannot be eliminated when the profiling metric is based on profiling at a function level. In addition to the data structures given in Table 5.1, valuable information about a profiled task, for example its current execution state, are retrieved by examining the run-time kernel data-structures.

5.4.4 Object-Based Profiling

μProfiler is a profiling tool for μC++ programs, i.e., programs written in a concurrent *object-oriented* programming language. In contrast to developing a profiling tool for procedural programming languages, additional design issues have to be taken into account when profiling an object-oriented program at the function-level. It is imaginable to build metrics where the object instance as well as the object member functions are required. In C++ and μC++, different instances of an object share the same code for their member functions and only their data fields have distinct locations in memory. Therefore, simply monitoring the code address where the profiled tasks execute is not enough to determine which object instance the executed code belongs to. C++ provides a simple mechanism to identify the current object instance that belongs to a member function: a pointer to the current object instance's memory location (the "this-pointer") is passed to member functions as the first argument (see [66]). Thus, the first argument to an object member function is an important piece of information allowing object-based profiling. When loading the profiled program's symbol table (see Section 5.3.2), all object member functions are identified and the symbol table access class uSymbolTable provides a function to determine whether a code address is located in an object member function. Thus, the execution monitor and analyzer can determine if a traversed function belongs to an object instance and as a consequence store this information separately. An application of object-based profiling is, for instance, profiling task communication, synchronization and scheduling through mutually exclusive objects such as monitors or semaphores. This information allows a better understanding of the algorithmic behaviour of the profiled program, since it reveals not only when synchronization and communication among tasks occurs, but it also allows the identification of the communication partners.

5.4.5 Dynamically Controlled Statistical Profiling

When performing statistical profiling, the sampling frequency plays an important role. Sampling at a high frequency potentially produces a great amount of profiling data, which can become difficult to manage for long-running problems and increases the probe effect. On the other hand, sampling at a low frequency may not provide enough information about the profiled program. Also, different monitors might require different sampling frequencies. A good profiling tool should permit different execution monitors to use different sampling frequencies and must not preclude building execution monitors that dynamically control their sampling frequency. The μProfiler design provides the necessary mechanisms to dynamically control and adjust the execution monitors' sampling frequencies. Also, each execution monitor can be invoked at a different sampling frequency. Initial work is underway in developing algorithms that try to dynamically optimize the sampling frequency: one approach is to gradually reduce the frequency when the sampled task appears to have the same execution state, and reset it to its original sampling frequency when the state changes. This approach deals with profiling a great number of communicating tasks, because some tasks will remain in a blocked state until some synchronization or communication event occurs. Thus, the profiler can focus on the currently active and interesting points in the concurrent program's execution. Obviously, reducing the sampling frequency can only be done to a certain degree, because it also increases the probability of missing important profiling events when a profiled task becomes active again. An approach to possibly eliminate this problem is to include another profiling hook, i.e, another exact profiling facility, into the μC++ kernel, which sends a message to the profiler when a task becomes active so that sampling can restart immediately.

Another important aspect is that if the user-specified sampling frequency for a certain execution monitor is chosen too high, the profiler kernel cannot service all necessary sampling requests on time. Therefore, the user-specified sampling frequency should be regarded as a suggested maximum. The design of μProfiler allows some dynamic control at this level as well: the profiler can measure the computation time that it needs to perform one profiling cycle, i.e., to service all execution monitors for a certain sampling event. If the profiler discovers that it cannot service all requests, it re-adjusts the profiling frequency accordingly and notifies the program analyst about the problem. Some initial work on run-time measurement and re-adjustment of the frequency on behalf of the profiling kernel is integrated into the design of μProfiler and being developed.

5.5 Design Validation

This section presents a brief discussion about how the μProfiler design model meets the design objectives stated in Section 5.1.

5.5.1 Profiling on a Thread Basis

Profiling on a thread basis is achieved by maintaining independent data structures for each profiled task in the program. The design aspects introduced in Section 5.4.3 allow monitoring each profiled task's state information, where the state information consists of information available through the run-time kernel and additional information to be collected for different profiling metrics. Both exact and statistical profiling on a thread basis are possible. In exact profiling mode, no events are missed since the profiled tasks register and deregister with the profiler. In statistical profiling mode, distributing a small part of the work to the profiled task itself allows more precise and detailed information, and therefore, justifies the additional probe effect.

5.5.2 Selective Profiling

Selective profiling is integrated into the design of μProfiler by allowing parts of the program to be compiled with the profiling flag and parts without the profiling flag, and then linking them together, which results in profiling only parts of the program. In addition, new run-time system routines for dynamically turning profiling on and off on a task basis are introduced and allow a precise mechanism to control what parts of the program are to be profiled. Also, the design allows a flexible implementation of the execution monitors, which can decide on-the-fly what aspects of the program are to be profiled.

5.5.3 Different Visualization Devices

μProfiler provides a library of different visualization devices, such as tables, bar charts and kiviat graphs. As described in Section 5.3.1, it also supplies an extendible interface to add and customize new visualization devices. All analyzers can make use of the visualization devices provided and visualize the profiling information in a concise format.

5.5.4 Extendibility

The design of μProfiler supplies a flexible and easy-to-extend interface for both the execution monitors and the metric analyzers. Extendibility is provided through inheritance mechanisms. New statistical and exact profiling metrics can be built without having to change the underlying structure of μProfiler. Hooks inside the run-time kernel allow registering important events for exact profiling. If necessary, additional hooks can be inserted without much effort, since the design requires all hooks to conform to the same interface structure and act according to the mechanisms supplied by the profiler kernel. Nevertheless, all extensions to the existing design require some programming effort.

5.5.5 Portability, Maintainability, Interoperability

Portability, maintainability and interoperability are maximized by designing most parts of μProfiler as a μC++ program. These parts become directly available on all platforms where μC++ operates. In addition, they can interoperate with existing μC++ tools, such as, the concurrent debugger *KDB* (also written in μC++). Also, it is imaginable to profile parts of μProfiler with μProfiler itself. The usability of debugging tools and the μProfiler's design with its underlying object-oriented analysis and design model, advocate maintainability and make it possible to re-use code when extending or re-engineering μProfiler.

Chapter 6

The μProfiler Implementation

Based on the design model discussed in Chapter 5, I have built a prototype implementation of μProfiler. This chapter discusses the fundamental and relevant issues of the μProfiler implementation that reveal the additional challenges arising from creating a working prototype profiler that fulfills the design goals stated in Section 5.1. Section 6.1 describes how compiler support is used to perform the compile-time instrumentation insertion; Section 6.2 and Section 6.3 discuss challenges that have to be considered at the monitoring and analysis stage of the profiling cycle. Further implementation issues, especially concerning efficient data structures and additional profiling features are presented in Section 6.4. Finally, the last section in this chapter, Section 6.5, discusses the applicability of μProfiler and describes limitations of the current implementation. It also discusses how to possibly eliminate them in future implementations.

6.1 Instrumentation Insertion

6.1.1 Compiler Support

The prototype implementation for μProfiler uses static instrumentation insertion at compile-time. The C++ compiler underneath μC++ supplies rudimental profiling instrumentation insertion facilities for sequential programs, which are used to implement the necessary instrumentation insertion for μProfiler: a compiler flag, −pg, for the C++ compiler causes a call to a function called mcount() to be inserted at each function entry in that compilation unit. Note that the inserted call to mcount() has originally been designed to support sequential profiling for the standard UNIX profiling tools *prof* and *gprof*. For example, Table 6.1 shows the first assembler lines for a compiled C++ function on the SPARC architectures without and with compiler supplied instrumentation insertion. μProfiler uses this mechanism to

Without Instrumentation

```
...
          .global f__Fv
          .type   f__Fv,#function
          .proc   020
f__Fv:
          save %sp,-112,%sp
...
```

With Instrumentation

```
...
          .global f__Fv
          .type   f__Fv,#function
          .proc   020
f__Fv:
          save %sp,-112,%sp
.section          ".data"
          .align 4
.LLP0:
          .word   0
.section          ".text"
          sethi %hi(.LLP0),%o0
          call mcount
          add %lo(.LLP0),%o0,%o0
...
```

Table 6.1: Compiler Supported Profiling Instrumentation Insertion on the SPARC Architecture.

perform the necessary instrumentation, but supplies its own version of mcount(), thereby replacing the standard implementation. In order to replace the standard implementation, different versions of the standard libraries have to be passed as arguments to the link editor. μProfiler achieves this effect by supplying a filter program to possibly replace the arguments passed from the compilation stage to the linking stage.

Unfortunately, inserting profiling instrumentation only at function entries is insufficient for μProfiler. Figure 6.1 illustrates why it is necessary to additionally insert profiling instrumentation at function exists: in the example scenario a call to *Function B* is performed from inside *Function A*. Since both functions are instrumented, two calls to the profiling routine mcount() are performed: one from inside A and one from inside B. Inside of mcount(), the profiling state information is changed and accounting started for mcount()'s caller. The problem is that without re-setting this state information upon function exit, all profiling information collected after returning from the callee B to the caller A is erroneously associated with B instead of A. This example shows that a technique is needed to additionally invoke instrumentation code upon a profiled function's exit.

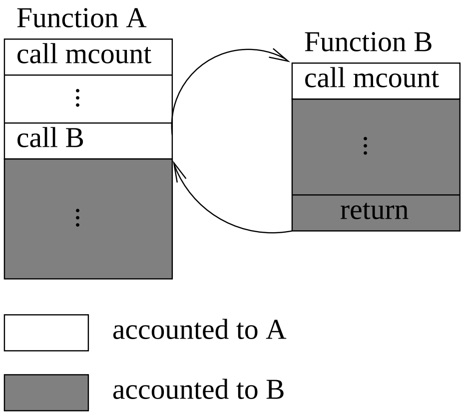

Figure 6.1: Profiling Nested Function Calls Without Function Exit Instrumentation.

This problem is solved in μProfiler by manipulating the return address of the instrumented function from inside mcount() as shown in Figure 6.2: In addition to mcount(), μProfiler supplies both a function prologue routine, uFunctionPrologue(), and a function epilogue routine, uFunctionEpilogue(), which compose the shared function entry and exit trampolines. Inside mcount(), a check is performed to determine if profiling is currently active. If this is not the case, control returns immediately to the instrumented function. Otherwise, uFunctionPrologue() is executed, and the return address of the instrumented function is changed, so that upon the instrumented function's exit, control returns to uFunctionEpilogue() instead of its original caller. Inside of uFunctionPrologue(), the old state information has to be saved on the profile stack for the currently executing task (see Section 6.1.2). This state information includes the original return address, which is later used inside of uFunctionEpilogue() to return to the instrumented function's caller. If exact profiling on function

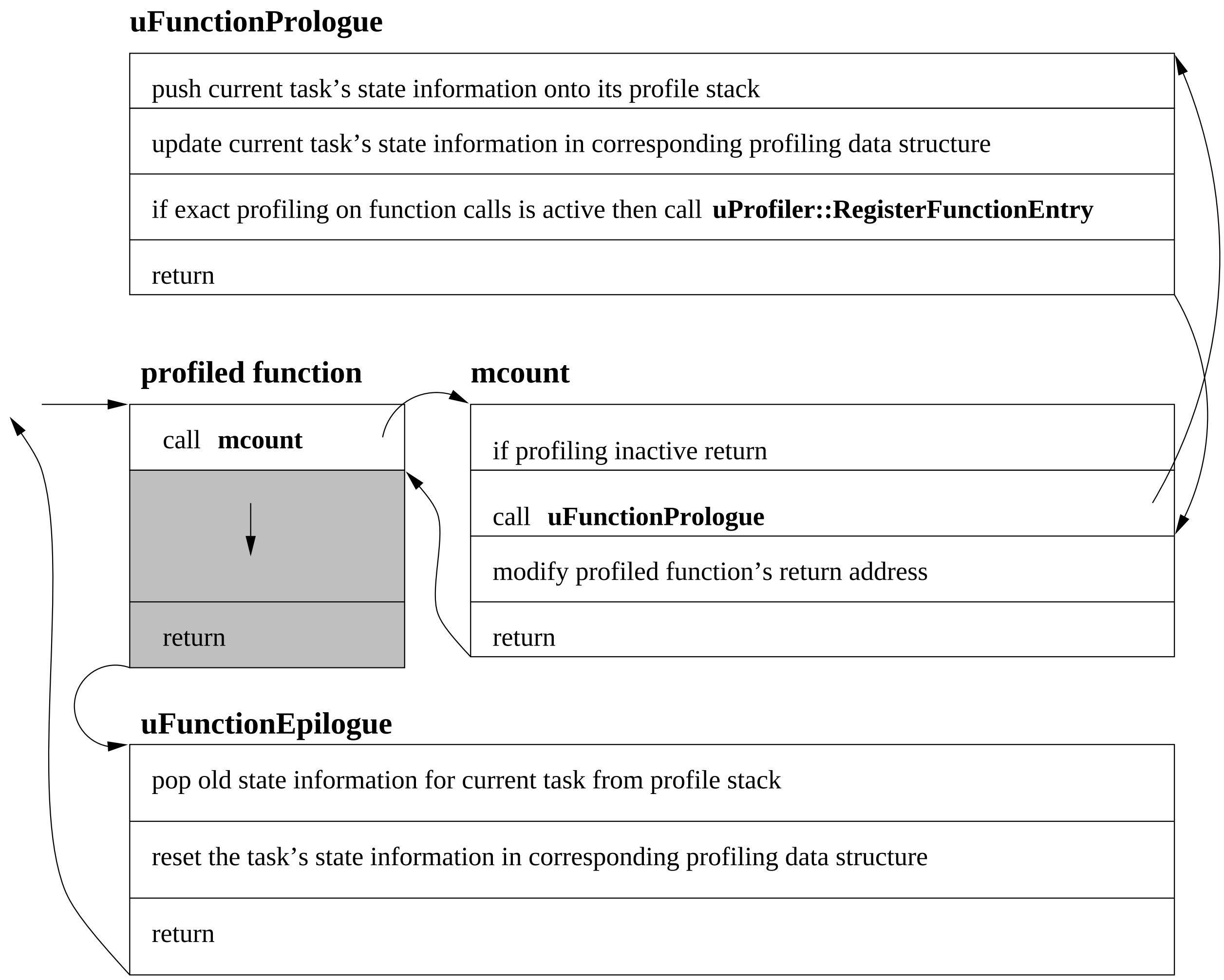

Figure 6.2: Compile-Time Instrumentation using Shared Trampolines.

calls is active, the task executing the function prologue trampoline calls into the profiler from inside of uFunctionPrologue() to register the current function entry event.

6.1.2 Profile Stack

As explained in the previous section, the state information of the current function's caller must be stored before the profiling data structures can be updated to hold information about the current function. This requirement is mandatory in order to allow retrieving the old state information before returning to the caller, especially since the state information includes the original return address to the caller, i.e., the value of the return address before it was re-adjusted to uFunctionEpilogue. The location where the profiling state information is stored defines the *profile stack*.

Since there has to be a profile stack for each profiled task at a memory location accessible to the task, μProfiler simply uses the memory that is already allocated for the task's stack. Figure 6.3 illustrates how the profile stack is located within the ordinary stack for a μC++ task. The profile stack is located at the opposite end of the allocated block of memory for the stack, and it grows in the opposite direction.

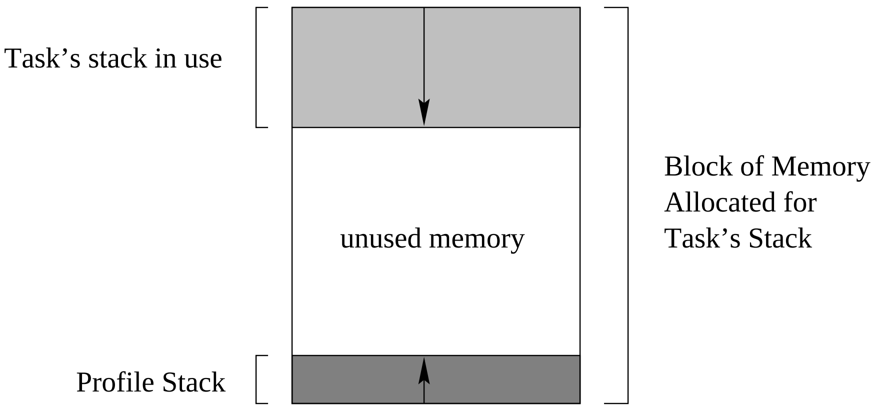

Figure 6.3: A Profiled μC++ Task's Stack.

Note that for each nesting level of function calls, only a few bytes of information have to be stored on the profile stack. For this reason and for efficiency reasons, this technique is preferable to dynamically allocating new storage for the profiling stack. Nevertheless, for highly recursive programs, there exists the danger of the two stacks interfering with each other. In order to ensure stack integrity, the profiler checks if the profile stack overwrites the task's ordinary stack. If this is the case, the profiler informs the user about the stack corruption and terminates profiling.

6.1.3 Platform Dependencies

As described in Chapter 5, the state information to be gathered encompasses the address of the current function, the address of the caller and the first argument to the called function.

Both the address information and the arguments passed into a function are handled differently on different architectures. Retrieving these pieces of information and manipulating the return addresses requires intrinsic knowledge about the computer architecture for which the code is generated. For this reason and in order to maximize the efficiency of the implementation, mcount() is implemented in assembler language. Prototype implementations of μProfiler for both a RISC and a CISC architecture have been developed. The target RISC architecture is Sun's SPARC Processor [65] and the target CISC architecture is Intel's x86 processor family [32]. On the SPARC, efficiency is increased by making use of the internal register ring structure. According to the SPARC coding specification (see [65]), the return address, the first six arguments to a function and the return value of a function call are all passed through registers. mcount() makes use of this fact and retrieves the required data directly from the registers in the register window. This technique is more efficient than flushing the register window to the stack and then walking the stack frame. The latter would be required if mcount() was not implemented in assembler, because of the compiler generated entry code for functions. For further details about the SPARC implementation and issues concerning the Intel implementation, refer to the documented source code of μProfiler.

6.2 Monitoring

6.2.1 Exact Monitoring

Exact monitoring using μProfiler can be done at various levels, depending on which exact monitoring hooks are used. Exact monitoring hooks are provided for different events: for example, hooks have been integrated into the run-time kernel for cluster creation and deletion, processor or task creation, migration and deletion, and inside of the trampolines, a hook exists to register function entry events (see Section 6.1). These hooks are only activated when there is at least one execution monitor, i.e., an object derived from uExecutionMonitor, which has registered for this particular hook with uProfiler. uProfiler maintains a list of execution monitors for each hook and invokes the member function of the execution monitor that corresponds to the hook, each time an event for this hook occurs. Figure 6.4 shows the implementation of a hook for task creation. uProfiler::uProfiler_RegisterTask is a function pointer that only points to the RegisterTask() member function of uProfiler if at least one execution monitor has registered for exact monitoring on task creation events. It points to NULL otherwise and can therefore be used to check whether the hook is active. All other hooks are implemented in a similar manner and allow a flexible mechanism to activate and deactivate exact profiling for a certain type of event without causing much overhead.

```
// profiling hook activated & task registered for profiling ?

if ( uProfiler::uProfiler_RegisterTask && uProfileActive ) {
    // ...

    // call into profiler through function pointer
    (*uProfiler::uProfiler_RegisterTask)( uProfiler::uProfilerInstance, *this );

    // ...
} // if
```

Figure 6.4: Exact Profiling Hook for Task Creation.

Using this mechanism to precisely control which hooks are activated makes it possible to selectively enable and disable the hooks according to the metrics used. For instance, a statistically sampling execution monitor can use exact monitoring of task creation events to ensure that all tasks in the profiled program are detected, and then perform statistical sampling for these tasks at a function level. This approach is, for example, implemented by the uSPMonitor execution monitor.

6.2.2 Statistical Monitoring

Robust Sampling

Performing accurate statistical monitoring for concurrently executing tasks requires a careful and robust implementation of the sampling routines. The problem is that while the statistical execution monitor is sampling a task's state information, the task could potentially be in the process of changing exactly this state information, and the execution monitor might gather erroneous data. Therefore, all sampling information is verified in order to assure accurate profiling data. Figure 6.5 shows how robust and accurate sampling is performed: all state information to be sampled is verified and only accepted if the profiled task has reached a consistent state. The robust sampling loop mechanism demonstrated in Figure 6.5 ensures that the information concerning the task's current cluster, the task's current state, the currently executed function, the function's parent and the profiling state of the task all correspond to one single state of the profiled task. While in theory there is no bound on the number of iterations of this loop, it is not a problem in practice. Note that although the information corresponds to one single state, the state information itself can be incorrect, e.g., when the sampled task is on the ready queue after being time-sliced in an inconsistent state.

```
// note: task holds a reference to the sampled task

static const uCluster          *CurrentCluster;
static uBaseTask::uTaskState CurrentTaskState;
static unsigned int           funcID;
static unsigned int           parentFuncID;
static bool                   CurrentTaskuProfileActive;

// ...

while ( CurrentCluster != &task.uGetCluster()
        || CurrentTaskState != task.uGetState()
        || funcID != GetCurrentFuncID( task )
        || parentFuncID != GetCurrentParentFuncID( task )
        || CurrentTaskuProfileActive != IsProfilingActive( task ) ) {
    CurrentCluster            = &task.uGetCluster();
    CurrentTaskState          = task.uGetState();
    funcID                    = GetCurrentFuncID( task );
    parentFuncID              = GetCurrentParentFuncID( task );
    CurrentTaskuProfileActive = IsProfilingActive( task );
} // while
```

Figure 6.5: Robust Sampling Loop.

Dynamic Filtering

Run-time filtering of profiling data is supported by μProfiler through the class uNameFilter, which supplies facilities to check whether the currently executed function is part of the user's code. If filtering is active and the current function is not part of the user's code, but rather part of the kernel or a μC++ internal library, the state information is charged to the user function that called the function either directly or indirectly. Thus, the amount of profiling data is reduced by a significant factor and the user is protected from being overwhelmed by information about non-user code supplied by the underlying system.

Dynamic Sampling Frequency Calibration

Some initial work is done to dynamically calibrate the sampling frequency in the case when the duration to perform a sampling cycle is greater than the user-specified sampling frequency for a statistically sampling execution monitor. A *sampling cycle* means the work performed on behalf of a particular execution monitor when the profiler invokes the monitor's Poll() routine. In order to perform dynamic sampling frequency calibration, the profiler measures the time it takes to execute one sampling cycle, and if user specified frequency requirements cannot be met, it re-adjusts the sampling frequency to the time needed to execute this sampling cycle. Since the time spent in a sampling cycle may change over time due to the user program's run-time behaviour, a validation of the sampling frequency and a possible re-adjustment are performed on every tenth sampling cycle.

In order to allow a validation of the accuracy of a profiling test run, the program analyst is informed about the frequency range in which the sampling has actually been performed.

Note that in the case of multiple monitors with different sampling frequencies this approach of dynamical calibration might not solve the problem, since calibrating the frequency of one execution monitor interferes with the calibration of other monitors with different frequencies. To properly handle this situation, a different scheduling method is needed, which should be further investigated in future work.

6.3 Profile Analysis and Visualization

6.3.1 Profile Analysis

Analyzing profiling information consists of three steps: first, the profiling data is extracted from the data structures into which the corresponding execution monitor has stored the information; second, irrelevant information is filtered out, and third, calculations according to the underlying metric are performed on the data. Extraction of the profiling data requires the analyzer to have full access to the execution monitor's data structures. Filtering at the profile analysis stage is unnecessary for the currently implemented metrics, since the data is already filtered at the monitoring stage. Nevertheless, the design of μProfiler does not preclude an analyzer from filtering if required by its corresponding profiling metric.

An example of an algorithmic calculation done at the profile analysis stage is the detection of call cycles in the profiling data of a particular task. Call cycle detection is performed when using the exact profiling option of μProfiler: a cycle detection algorithm is applied in order to find call cycles that occurred during the execution of the profiled tasks. Call cycle detection provides valuable information about user program interdependencies: for example, consider

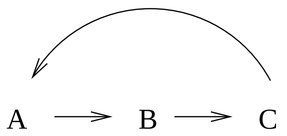

Figure 6.6: An Example Call Cycle.

the scenario of Figure 6.6. Function A calls function B, which calls function C, which calls back to function A. In this example scenario, the time spent in function A directly contributes to the time spent in function C, which also contributes to the time of A through B. By exposing this situation, μProfiler enables the program analyst to better understand and validate the presented profiling information.

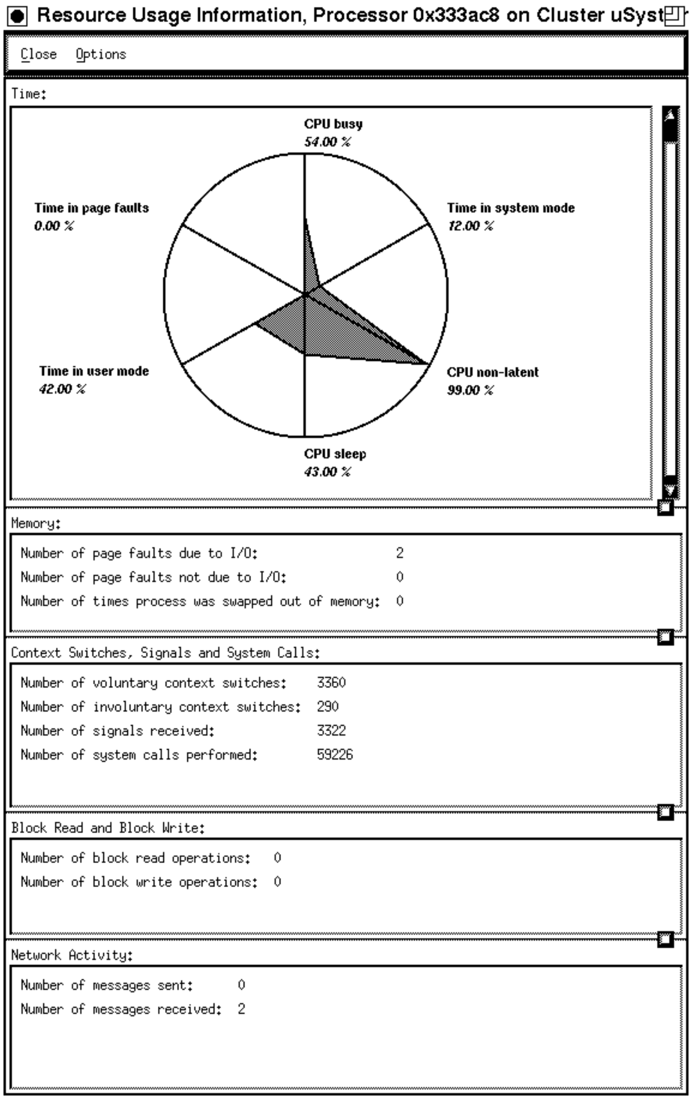

Figure 6.7: Visualization of Operating System Resource Information.

6.3.2 Visualization

Visualizing the profiling information is done using the visualization devices described in Section 5.3. For example, Figure 6.7 shows a kiviat chart and additional information, which is presented by μProfiler when information about the operating system's resource usage is selected. The screen-shot shows how a kiviat graph is used to visualize profiling data of a μC++ processor at the operating system level. Six metrics selected from the operating system's run-time information are presented and correlated in the presented kiviat graph: *CPU busy*, *time in system mode*, *CPU non-latent*, *CPU sleep*, *time in user mode* and *time in page faults*. In the example output of Figure 6.7, the shape of the kiviat graph shows that the inspected processor spent a relatively long time in the sleeping state, which could, for example, be caused by blocking I/O operations. In addition to the data represented in the kiviat graph, tables give detailed information about the memory behaviour, i.e., about page faults and the number of times memory had to be swapped out to disk, about UNIX process context switches, signals and system calls, about filesystem block reads and writes, and about the network activity during a test-run. As discussed earlier, information provided by the operating system is important to validate the test-run, because it represents the system's load during the profiled program's execution.

Figure 6.8 demonstrates an example visualization of profiling information for a task, which has been statistically monitored at a function level by a uSPMonitor execution monitor. For each execution state of the task, i.e., *running*, *ready* and *blocked*, the time spent in functions detected by the execution monitor is represented by sorted bar charts. The bar charts give information about absolute time values while, in addition, revealing the relative "importance" of the different functions. During the profiling test-run that corresponds to the example output of Figure 6.8, kernel profiling (see Section 6.4.3) was activated, which results in both user functions and run-time kernel function calls being monitored.

6.4 Further Implementation Aspects

In this section, further relevant implementation aspects are discussed, including access of the profiling data structures via *hashing*, the *profiling scope*, and issues concerning *kernel profiling*.

6.4.1 Hashing

Performing monitoring at a function level generates an enormous amount of data to be managed by the execution monitor. After possibly filtering or aggregating the data, profiling

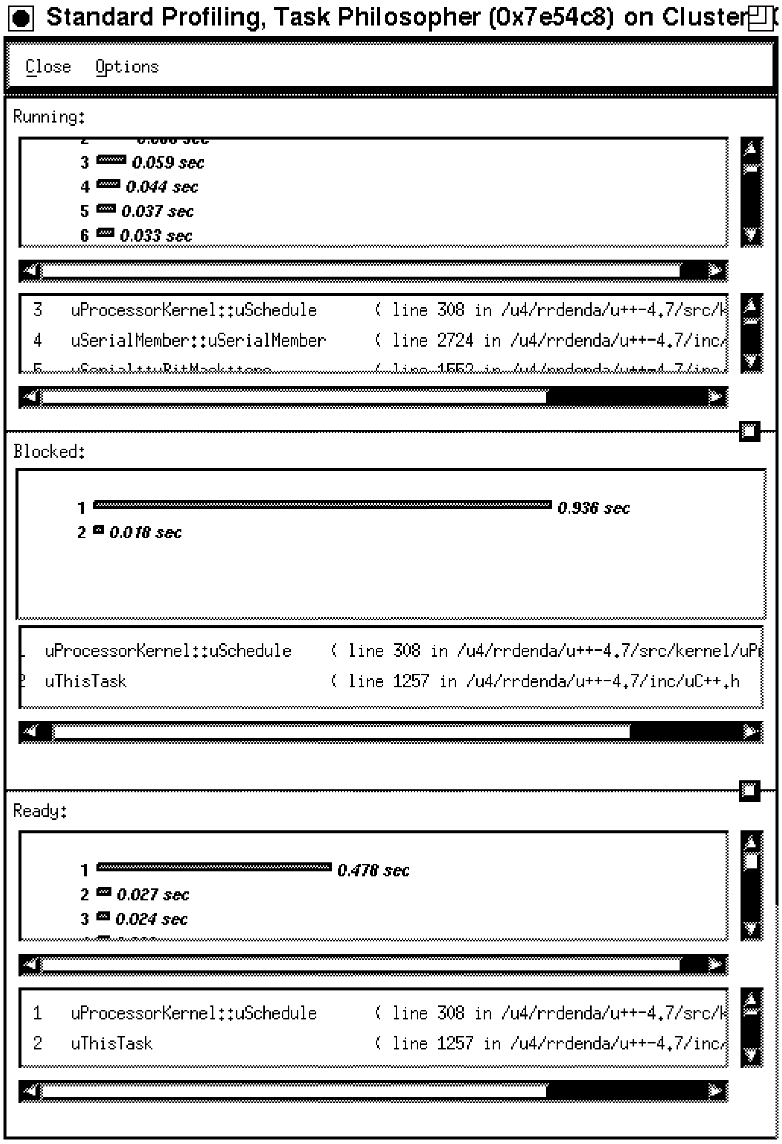

Figure 6.8: Visualization of Statistically Profiled Task Information at Function Level.

information has to be stored in adequate data structures. The main aspect in choosing appropriate data structures for this purpose is the speed with which these data structures can be accessed and queried. Using hashing algorithms results in, on average, constant query time and is therefore the selected technique for inserting and looking up profiling data during execution. The hashing method used by μProfiler is a simple but efficient variation of hashing using division: for a given hashing key K, the hash function $h(K) = K \ mod \ P$ is applied,

collisions	number of buckets
0	3353
1	329
2	31
3	1
4 and more	0

Table 6.2: Collisions in Profiling Hash Table Buckets for a large μC++ Program.

where P is the size of the hash table. It has been shown that the best results are obtained when choosing P a prime number with the additional requirement that $R \not\equiv 1 \ (\ mod\ P)$, i.e., R is *not congruent* to 1 modulo P, where R is the radix of the keys to be inserted. In the case of μProfiler, address values are used as keys into the hash table, which means that the key values are divisible by the required address alignment of the underlying computer architecture. For example, on the SPARC, properly aligned addresses are divisible by four, which results in radix $R = 4$. Therefore, the implemented hash table for μProfiler adjusts its size to a prime number that is not congruent to possible address alignment values. Note that many other hashing methods have been developed (see, e.g., [37] or [38]), but hashing using division has been shown to provide a comparatively good behaviour in terms of collision reduction. For a detailed discussion on the behaviour of hashing using the division method, see [18].

Another important aspect to be considered when implementing a hash table is its size. If the size of a hash table is chosen too small, an unacceptable number of collisions may occur; if the size of a hash table is chosen too great, the hash table has an unnecessarily sparse distribution of entries. The hashing tables used by μProfiler are approximately four times the size of the number of function name entries in the profiled program's symbol table. Empirical results have shown that the dispersion resulting from the hash function implemented by μProfiler is acceptable.

μProfiler performs collision resolution by separate chaining for each hash table bucket, i.e., each hash table entry is an entry point to a dynamically linked list. In order to distinguish the entries, the entry's key is also stored in the linked list.

Table 6.2 shows an empirical collision distribution for a large μC++ program. In the example, 4108 function addresses of an example μC++ program are inserted into a hash table used by μProfiler. 3353 of these insertions did not collide with another insertion into the corresponding hash table bucket; for 329 buckets, one collision occurred, i.e., two insertions were made; for 31 buckets, the hashing function caused two collisions, and in one case, three collisions were detected. For the examined program, the entry distribution can be considered the worst-case scenario, since all function addresses of the example program have been inserted into the hash table.

6.4.2 Profiling Scope

The integration of new start-up routines into the run-time kernel ensures that μProfiler is started before any of the user's functions are executed. Note that by applying this method, it is possible to profile the complete execution of the user code, including all the work done in global constructors and destructors. Since the global constructors and destructors of a μC++ program are executed by a μC++ system task (uBootTask), it is necessary to profile this task in addition to the tasks created by the user. By doing so, the profilable scope of a μC++ program includes all of the user's code.

6.4.3 Kernel Profiling

With μProfiler, some initial work has been done to partially profile the run-time kernel of μC++. Two variants of kernel profiling can be distinguished: first, since most of the run-time kernel is written in μC++ it can also be compiled with the profiling command line option, which causes profiling instrumentation to be inserted into the run-time library. With the second variant, only the parts of the run-time kernel that are compiled together with the user program (see [8]) are instrumented.

In the first case, profiling information about the μC++ kernel can be retrieved with only a few limitations: the part of the run-time kernel that manages the preemptive task scheduling, and some other parts such as the parts of the run-time library that are executed before μProfiler is created cannot be profiled, because the profiler itself strongly depends on their functionality. Profiling the μC++ run-time kernel is useful for debugging, analyzing and performance tuning of μC++ itself, i.e., it mainly serves the μC++ system developer.

The second variant provides a "light" version of kernel profiling: in addition to the regular profiling command line option, a second command line parameter called −kernelprofile has been integrated into μC++. When compiling a μC++ program with the −kernelprofile flag, profiling information about the part of the run-time kernel that is compiled together with the user program is collected and analyzed in an identical manner to the profileable part of the user code.

6.5 Limitations

Although μProfiler has been designed to avoid restrictions on profiling functionality and scope, some limitations still apply and are described in this section.

6.5.1 Hard-Coded Filename Access

As discussed in Section 5.2, integrating μProfiler into the profiled program has many advantages, but also entails some implementation problems: μProfiler needs access to the symbol table of the profiled program, i.e., the symbol table of the executable into which μProfiler is integrated. The problem is that there exists no simple mechanism to obtain the symbol table of the executable file that corresponds to the executing UNIX processes, into which the μC++ program is embedded, from inside the executing μC++ program itself. Note that the necessary information becomes available only when the main() function in the user part of the profiled application is started. Unfortunately, the symbol table information is already needed during the boot process of the uProfiler object; so, this approach cannot be taken. The solution used by μProfiler is to insert the filename into the profiled application during compile-time. This technique has the disadvantage that the filename of the profiled μC++ program cannot be modified after the compilation stage, i.e., changing the executable's name is not allowed without re-compilation. The simplest solution to this problem is for UNIX to provide global access to the shell arguments as it does for shell environment variables (see [68]).

6.5.2 Applicability and Availability

Since μProfiler has been developed and optimized for usage with μC++, it cannot be used with other thread packages in its current implementation. Nevertheless, the underlying design and ideas are applicable to other concurrent programming languages and provide a general and fundamental approach to monitoring a concurrent program's execution-time behaviour.

Although μProfiler provides profiling facilities and techniques to perform profiling at a very fine-grained level, it is questionable whether using μProfiler at a level lower than a function level generates information accurate enough for a reasonable profiling analysis. Nevertheless, with some effort, the program analyst can abstain from performing the profiling instrumentation insertion at a function level during compile-time and manually insert the instrumentation, i.e., the calls to function `mcount()`, into the profiled code wherever appropriate. This technique makes it possible to perform profiling at a very fine-grained level of detail.

Currently, μProfiler has been ported to architectures based on Sun's SPARC processors running SunOS and Solaris, and to architectures based on Intel's x86 processors running Linux. Since the shared trampolines for function entry and exit have to be implemented in machine-dependent assembler language, a simple re-compilation for a new target machine is insufficient. Nevertheless, the parts of μProfiler that are written in μC++ become directly available on all platforms for which a port of μC++ exists, and the μProfiler's design requires only few machine-dependent parts of the code, so that porting μProfiler to other machine architectures and operating systems is feasible with minimal effort.

Chapter 7

Conclusions and Future Work

The intent of this thesis was to identify problems and challenges that occur when designing and implementing novel methods to perform fine-grained profiling of a concurrent object-oriented program on a thread base. The resulting solutions lead to an extendible prototype implementation of a profiler for the μC++ language that incorporates exact and statistical profiling techniques and provides useful information for the analysis and improvement of a concurrent object-oriented program.

7.1 Conclusions

In this thesis, the essential profiling methods and methodologies are defined and the problems arising when profiling a concurrent program have been discussed.

This work focused on profiling user-level threads for concurrent object-oriented programs running in a shared-memory multi-processor environment. The main reasons for profiling have been identified as *performance analysis*, *algorithm analysis*, *coverage analysis*, *tuning* and *debugging*, and an abstract illustration of the profiling cycle has been presented.

Existing approaches to profiling concurrent programs have been discussed and it has been discovered that none of the previous work fulfilled the design requirements to perform extendible user-level profiling of concurrent object-oriented programs at a fine-grained level in a shared memory environment as it is provided by μC++.

The implemented prototype profiler, μProfiler, allows performing both exact and statistical per-thread profiling at a function level for μC++ programs. Designing and implementing μProfiler in the high-level concurrent object-oriented language μC++ makes it easy to extend, which is an important aspect of any good profiling tool. Also, interoperability with existing tools for μC++ becomes possible: for example, the profiler could be debugged using the debugger for μC++, the debugger can be profiled, and it is even imaginable to profile μProfiler with μProfiler. The tight coupling with the μC++ run-time kernel allows precisely controllable profiling at a fine-grained level, and the incorporation of the profiler into the program to be profiled provides user-friendly, flexible profiling with minimal effort. Finally, μProfiler's portability has been proven through implementations for both RISC and CISC computer architectures.

The novel techniques introduced by this work include *user-level thread profiling at a function level* for μC++, *dynamic filtering* of profiling information, *dynamic frequency calibration* for statistical profiling, and the ability to partially profile a thread library's run-time kernel.

Experiments using μProfiler have validated the statement that profiling user-level threads is useful for performance analysis, algorithm analysis, coverage analysis, tuning and debugging.

The extensions to the current implementation of μC++, which were necessary to incorporate profiling into the target environment, will be integrated into the future release of μC++, and μProfiler itself will be made publicly available as a part of the MVD package.

7.2 Future Work

The framework presented in this thesis provides a useful profiling environment for concurrent programs. Nevertheless, the extendible design of μProfiler should be utilized to pursue further research in the development and applicability of new metrics and corresponding visualization techniques.

Although this work presents initial research in incorporating object-based profiling into the system, metrics using this functionality still have to be developed. In addition, future work should encompass: research on an automation of the profile analysis, further investigation of dynamically controlled profiling, and the development of methodologies to ensure a better cooperation among profiling, debugging and event-tracing tools.

Another field for future work is to investigate the applicability of the presented ideas to other thread packages. Also, it is imaginable to develop an interface to visualization tools like *POET* [41], or extend the functionality of μProfiler to allow exporting the profiling data in a commonly supported file format, which is readable by other performance evaluation tools. An example of such a file format is the *Pablo Self Defining Data Format (SDDF)* [2].

Finally, porting μProfiler to all of the architectures that are supported by the underlying target environment (i.e., μC++) should be considered.

Appendix A

Object-Oriented Analysis Model Notations

The following gives a brief description of the notation format for classes, objects, object relations, inheritance and aggregation in the object oriented analysis model. The notation format used is based on the notation proposed by Coad and Yourdan [9]. Classes and objects

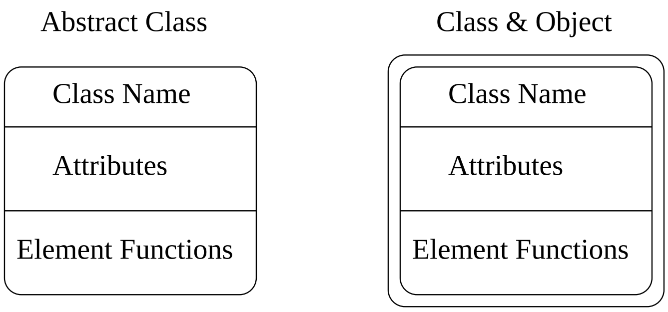

Figure A.1: Classes and Objects.

are represented as rectangles with round edges, which are divided into three sections. The top section contains the name, the middle section contains the attributes and the bottom section contains the element functions of the illustrated class. Classes that can be instantiated are embedded into a second frame with round edges; only purely abstract classes are represented by a single frame rectangular box. Figure A.1 shows the notation format of classes and objects. For clarity reasons, a simplified notation format for classes and objects is used in this thesis when considered appropriate. The simplified class and object notation is shown in Figure A.2.

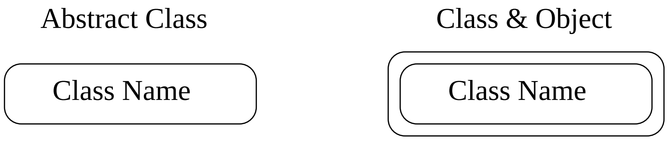

Figure A.2: Simplified Class and Object Notation.

Figure A.3 shows how object relations, i.e., instance connections, are represented. The cardinality notation shows how many objects of each type are related. In the example shown in Figure A.3, An instance of *Class A* is related to zero or any number of objects of type *Class B*, whereas each instance of *Class B* is related to exactly one object of type *Class A*, i.e., a many-to-one relationship.

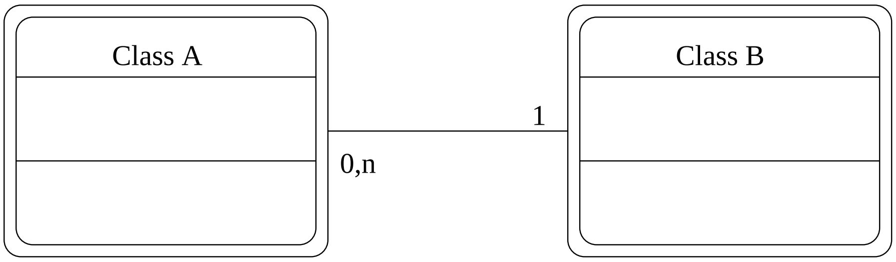

Figure A.3: Object Relationship.

Inheritance is represented as shown in Figure A.4. Note that, since inheritance is performed on a class basis rather than on an object basis, the connecting lines begin at the inner rectangular frame. The derived classes represent a specialization of the more general base class. Note also, that only type inheritance, as oposed to implementation inheritance, is represented: for example, the notation does not differentiate between attributes or element functions that are inherited by all derived classes, and attributes or element functions that are private to the base class. Figure A.5 illustrates how object aggregation is represented in the notation. Aggregation describes a situation where an object contains other objects as its attributes; i.e., objects are layered inside of other objects. In the example, a whole-part aggregation structure is shown: each "part" corresponds to exactly one "whole" entity,

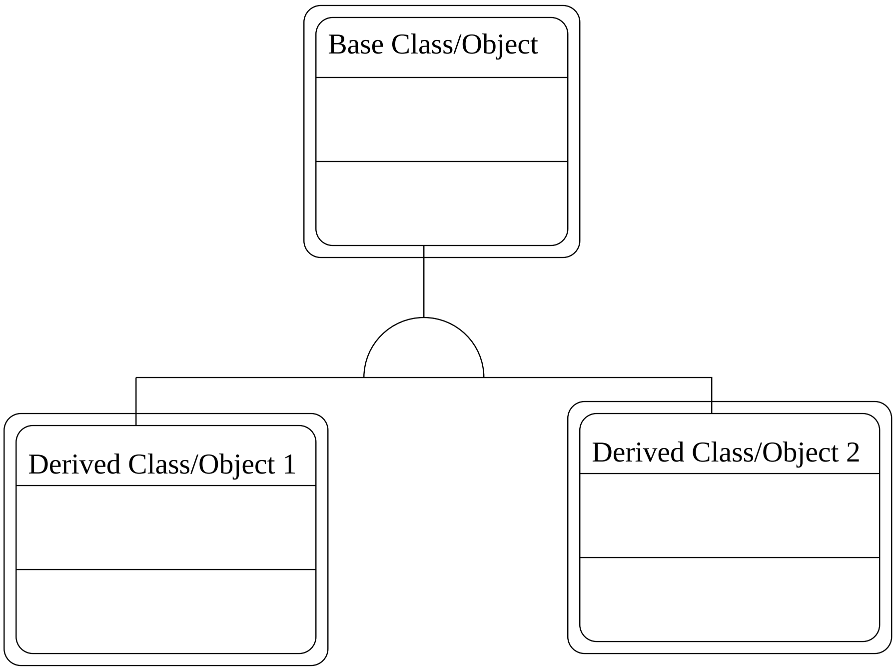

Figure A.4: Inheritance.

whereas the "whole" object consists of 1 to n "parts".

For a more detailed description of the object-oriented analysis notation format used in this thesis, refer to [9].

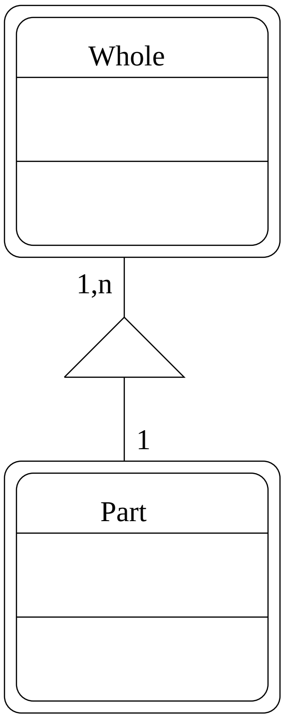

Figure A.5: Aggregation.

Bibliography

[1] ANDERSON, T., AND LAZOWSKA, E. Quartz: A Tool for Tuning Parallel Program Performance. In *Proceedings of the 1990 SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Boston, May 1990), pp. 115–125.

[2] AYDT, R. A. The Pablo Self-Defining Data Format. Tech. rep., Pablo Research Group, Department of Computer Science, University of Illinois, Urbana, Illinois, March 1992.

[3] BALL, T., AND LARUS, J. Optimally Profiling and Tracing Programs. In *19th ACM Symposium on Principles of Programming Languages* (Albuquerque, New Mexico, January 1992), pp. 59–70.

[4] BROBERG, M. Visualization of Parallel Program Behaviour. Master's thesis, Dept. of Computer Science and Business Administration, University of Karlskrona/Ronneby, 1996.

[5] BROWN, D., HACKSTADT, S., MALONY, A., AND MORH, B. Program Analysis Environments for Parallel Language Systems: The τ Environment.

[6] BUHR, P. A. μC++ monitoring, visualization and debugging annotated reference manual, version 1.0. Tech. rep., Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, Sept. 1995. Available via ftp from plg.uwaterloo.ca in pub/MVD/Visualization.ps.gz.

[7] BUHR, P. A., KARSTEN, M., AND SHIH, J. KDB reference manual, version 1.1. Tech. rep., Department of Computer Science, University of Waterloo, Water-

loo, Ontario, Canada, N2L 3G1, Dec. 1996. Available via ftp from plg.uwaterloo.ca in pub/MVD/KDB.ps.gz.

[8] Buhr, P. A., and Stroobosscher, R. A. μC++ annotated reference manual, version 4.6. Tech. rep., Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, July 1996. Available via ftp from plg.uwaterloo.ca in pub/uSystem/uC++.ps.gz.

[9] Coad, P., and Yourdon, E. *Object-Oriented Analysis*, 2 ed. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[10] Cray Research Inc. *Cray T3D System Architecture Overview.* HR-04033.

[11] Crovella, M. E., and LeBlanc, T. J. Performance Debugging Using Parallel Performance Predicates. 140–150. ACM/ONR Workshop on Parallel and Distributed Debugging.

[12] Cygnus Support. *LIBBFD, the Binary File Descriptor Library.*

[13] Esser, R., and Knecht, R. Intel Paragon XP/S — architecture and software environment. Tech. Rep. KFA-ZAM-IB-9305, Central Institute for Applied Mathematics, Research Center Jülich, Germany, r.esser@kfa-juelich.de, Apr. 26 1993.

[14] Fineman, C. E., and Hontales, P. J. Selective Monitoring Using Performance Metric Predicates. 162–165. Scalable High Performance Computing Conference, Williamsburg, Virginia, USA.

[15] Gait, J. A Probe Effect in Concurrent Programs. *Software - Practice and Experience 16*, 3 (March 1986), 225–233.

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

[17] Gettys, J., and Scheifler, R. W. Xlib - C Language Interface. electronic document.

[18] Ghosh, S. P., and Lum, V. Y. Analysis of collisions when hashing by division. *Information Systems 1*, 1 (1975), 15–22.

[19] Goldberg, A. J., and Hennessy, J. L. Mtool: An Integrated System for Performance Debugging Shared Memory Multiprocessor Applications. *IEEE Trans. on Parallel and Distributed Systems* (Jan. 1993), 28–40.

[20] Goosen, H. A., Hinz, P., and Polzin, D. W. Experience Using the Chiron Parallel Program Performance Visualization System. Computer Science Department, University of Cape Town, South Africa, 1995.

[21] Gosling, J., Joy, B., and Steele, G. *The Java Language Specification.* Sun Microsystems, Inc, August 1996.

[22] Graham, S. L., Kessler, P. B., and McKusick, M. K. gprof: a Call Graph Execution Profiler. *SIGPLAN Notices 17*, 6 (June 1982), 120–126. Proceedings of the SIGPLAN'82 Symposium on Compiler Construction, June 23–25, 1982, Boston, Massachusetts, U.S.A.

[23] Heath, M. T., and Finger, J. E. *ParaGraph: A Tool for Visualizing Performance of Parallel Programs.* University of Illinois and Oak Ridge National Laboratory, September 1994.

[24] Herrarte, V., and Lusk, E. *Studying Parallel Program Behavior with Upshot.* User Manual.

[25] High Performance Fortran Forum. *High Performance Fortran Language Specification, Version 0.4*, 1992.

[26] Ho, W. W., and Olsson, R. A. An approach to genuine dynamic linking. *Software - Practice And Experience 21*, 4 (Apr. 1991), 375–390.

[27] Hoare, C. A. R. Monitors: An Operating System Structuring Concept. *Communications of the ACM 17*, 10 (Oct. 1974), 549–557.

[28] Hollingsworth, J., and Miller, B. P. Slack: A New Performance Metric for Parallel Programs. Tech. Rep. CS-TR-95-1260, Computer Sciences Department, University of Wisconsin-Madison, Madison, Wisconsin, December 1994.

[29] Hollingsworth, J. K. *Finding Bottlenecks in Large Scale Parallel Programs.* PhD thesis, University of Wisconsin, Madison, 1994.

[30] Hollingsworth, J. K., and Miller, B. P. Parallel Program Performance Metrics: A Comparison and Validation. In *Supercomputing 1992, Minneapolis* (Computer Science Department, University of Wisconsin-Madison, 1992).

[31] IBM Corporation. *IBM Program Visualizer (PV) Tutorial and Reference Manual, Release 0.8.*

[32] Intel Corporation. *Intel Architecture Software Developer's Manual*, 1997.

[33] Jain, R. *The Art of Computer Systems Performance Analysis*, first ed. John Wiley & Sons, Inc, 605 Third Avenue, New York, N.Y., 1991. ISBN 0471503363.

[34] Kernighan, B. W., and Ritchie, D. M. *The C Programming Language*, second ed. Prentice Hall Software Series. Prentice Hall, 1988.

[35] Kerola, T., and Schwetman, H. Monit: A Performance Monitoring Tool for Parallel and Pseudo-Parallel Programs. In *Proceedings of the 1987 ACM SIGMETRICS Conference* (May 1987).

[36] Kessler, P. B. Fast Breakpoints: Design and Implementation. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation, published in ACM SIGPLAN Notices* (June 1990), vol. 25, pp. 78–84.

[37] Knott, G. D. Hashing functions. *The Computer Journal 18*, 3 (Aug. 1975), 265–278.

[38] Knuth, D. E. *The Art of Computer Programming, Sorting and Searching*, vol. 3. Addison-Wesley, Reading, MA, USA, 1973.

[39] Kolence, K. W., and Kiviat, P. J. Software Unit Profiles and Kiviat Figures. 2–12. Performance Evaluation Review.

[40] Kranz, D., Johnson, K., Agarwal, A., Kubiatowicz, J., and Lim, B.-H. Integrating Message-Passing and Shared-Memory : Early Experience. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, published in ACM SIGPLAN Notices* (July 1993), vol. 28, pp. 54–63.

[41] Kunz, T., Taylor, D. J., and Black, J. P. POET: Target-system-independent visualizations of complex ditributed executions. 452–461. Proceedings of the 30th Hawaii International Conference on System and Science, January 1997, Maui, Hawaii, U.S.A.

[42] Lange, F., Kroger, R., and Gergeleit, M. JEWEL: Design and Implementation of a Distributed Measurement System. *IEEE Transactions on Parallel and Distributed Systems 3*, 6 (November 1996), 657–671.

[43] Larus, J. R. Abstract Execution: A Technique for Efficiently Tracing Programs. *Software– Practice and Experience* (Dec. 1990), 1241–1258. Also Univ. of Wisconsin Computer Science Tech. Report 912.

[44] Lehr, T., Black, D., Segall, Z., and Vrsalovic, D. MKM: Mach Kernel Monitor Description, Examples and Measurements. Tech. rep., Carnegie-Mellon University, Pittsburgh, March 1989. PA-CS-89-131.

[45] Lovett, T., and Thakkar, S. The Symmetry multiprocessor system. In *Proceedings of the 1988 Conference on Parallel Processing* (1988), pp. 303–310.

[46] Lyon, G., Snelick, R., and Kacker, R. Synthetic-perturbation of MIMD programs. *Journal of Supercomputing 8*, 1 (1994), 5–8.

[47] Madhyastha, T. M. A Portable System for Data Sonification. Master's thesis, Rutgers State University, 1990.

[48] Malony, A., Mohr, B., Beckman, P., Gannon, D., Yang, S., and Bodin, F. Performance Analysis of pC++: A Portable Data-Parallel Programming System for Scalable Parallel Computers. In *Proceedings of the 8th International Parallel Processing Symbosium (IPPS), Cancun, Mexico* (April 1994), pp. 75–85.

[49] McCormack, J., Asente, P., and Swick, R. R. X Toolkit Intrinsics - C Language X Interface. electronic document.

[50] Miller, B. P., Callaghan, M. D., Cargille, J. M., Hollingsworth, J. K., Irvin, R. B., Karavanic, K. L., Kunchithapadam, K., and Newhall, T. The Paradyn Parallel Performance Measurement Tools. Tech. rep., Computer Science Department, University of Wisconsin-Madison, 1210 W. Dayton Street, Madison, WI 53706, USA.

[51] Miller, B. P., Clark, M., Hollingsworth, J. K., Kierstead, S., Lim, S., and Torzewski, T. IPS-2: The Second Generation of a Parallel Program Measurement System. In *IEEE Transactions on Parallel and Distributed Systems* (April 1990), vol. 1, pp. 206–217.

[52] Miller, B. P., and Yang, C.-Q. IPS: An Interactive and Automatic Performance Measurement Tool for Parallel and Distributed Programs. In *Proceedings of the 7th International Conference on Distributed Computing Systems* (September 1987).

[53] Mohr, B. Standardization of Event Traces Considered Harmful or Is an Implementation of Object-Independent Event Trace Monitoring and Analysis Systems Possible ? In *Advances in Parallel Computing*, J. Dongarra and B. Tourancheau, Eds., vol. 6. 1993, pp. 103–124.

[54] Mohr, B. W., Malony, A. D., and Shanmugam, K. Speedy: An Integrated Performance Extrapolation Tool for pC++ Programs. Department fo Computer and Information Science, University of Oregon, Eugene, Oregon.

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

[56] Perl, S. E., and Weihl, W. E. Performance Assertion Checking. 134–145. 14th ACM Symposium on Operating Systems Principles.

[57] Pillet, V., Labarta, J., Cortes, T., and Girona, S. PARAVER: A tool to visualise and analyze parallel code. In *Proceedings of WoTUG-18: Transputer and occam Developments* (Amsterdam, April 1995), vol. 44 of *Transputer and Occam Engineering*, IOS Press, pp. 17–31. ISBN 90 5199 222 x.

[58] Ponder, C., and Fateman, R. Inaccuracies in Program Profilers. *Software - Practice and Experience 18*, 5 (May 1988), 134–145.

[59] Reed, D., Aydt, R., Noe, R., Roth, P., Shields, K., Schwartz, B., and Tavera, L. Scalable Performance Analysis: The Pablo Performance Analysis Environment. *Scalable Parallel Libraries Conference* (1993). A. Skjellum, IEEE Computer Society.

[60] Reed, D. A., Shields, K. A., Scullin, W. H., Tavera, L. F., and Elford, C. L. Virtual Reality and Parallel Systems Performance Analysis. Department of Computer Science, University of Illinois, Urbana, Illinois, 1995.

[61] Ries, B., Anderson, R., Auld, W., Breazeal, D., Callaghan, K., Richards, E., and Smith, W. The Paragon Performance Monitoring Environment. In *Supercomputing 1993* (Portland, OR, November 1993), pp. 850–859.

[62] Segall, Z., and Rudolph, L. PIE: A Programming and Instrumentation Environment for Parallel Processing. *IEEE Software 2*, 6 (November 1985), 22–37.

[63] Snelick, R. S-Check: a Tool for Tuning Parallel Programs. National Institute of Standards and Technology.

[64] Snelick, R., Ja'Ja', J., Kacker, R., and Lyon, G. Synthetic-perturbation techniques for screening shared memory programs. *Software - Practice and Experience 24*, 8 (1994), 679–701.

[65] SPARC International. *The SPARC Architecture Manual.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, 1992.

[66] Stroustrup, B. *The C++ Programming Language*, second ed. Addison Wesley, 1991.

[67] Stunkel, C. B., Atkins, M., Hochschild, P. H., Swetz, R. A., Shea, D. G., Bender, C. A., Joseph, D. J., Stucke, R. F., Varker, P. R., Abali, B., Grice, D. G., Nathanson, B. J., and Tsao, M. The SP2 Communication Subsystem. Tech. rep., IBM Thomas J. Watson Research Center, August 1994.

[68] Sun. *getenv(3C).* C Library Functions.

[69] Sun. *getrusage(3C).* C Library Functions.

[70] Sun. *proc(4).*

[71] Sun. *ptrace(2).*

[72] Sun, X.-H., and Zhu, J. Performance Considerations of Shared Virtual Memory Machines. Department of Computer Sicence, Louisiana State University, Baton Rouge, Louisianna.

[73] Sunderam, V. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience 2*, 4 (Dec. 1990), 315–339.

[74] Taylor, D., and Buhr, P. A. POET with μC++. Tech. rep., Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, Dec. 1996. Available via ftp from plg.uwaterloo.ca in pub/MVD/Poet.ps.gz.

[75] Tufte, E. R. *The Visual Display of Quantitative Information.* Graphics Press, Box 430, Cheshire, CT 06410, USA, 1983.

[76] United States Department of Defense. *The Programming Language Ada: Reference Manual*, ANSI/MIL-STD-1815A-1983 ed., Feb. 1983. Published by Springer-Verlag.

[77] VEENSTRA, J. Mint tutorial and user manual. Tech. Rep. 452, Computer Science Department, University of Rochester, June 1993.

[78] WILLIAMS, W., HOEL, T., AND PASE, D. The MPP Apprentice Performance Tool: Delivering the Performance of the Cray T3D. *Programming Environments for Massively Parallel Distributed Systems, North Holland* (1994).

[79] WU, C. E., FRANKE, H., AND LIU, Y.-H. UTE: A Unified Trace Environment for IBM SP Systems. IBM T.J. Watson Research Center.

[80] YAN, J. Performance Tuning with AIMS - An automated Instrumentation and Monitoring System for Multicomputers. In *Proceedings of the 27th Hawaii International Conference on System Sciences, Wailea, Hawaii* (January 1994), vol. 2, pp. 625–633.

[81] YAN, J., SARUKKAI, S., AND MEHRA, P. Performance Measurements, Visualization and Modelling of Parallel and Distributed Programs using the AIMS Toolkit. *Software-Practice and Experience 25*, 4 (April 1995), 429–461.