μProfiler: Profiling User-Level Threads in a Shared-Memory Programming Environment

Peter A. Buhr[1] and Robert Denda[2]

[1] University of Waterloo, Waterloo, Ont., Canada
[2] Universität Mannheim, Mannheim, Germany

Abstract. A *profiler* is an important tool for understanding the dynamic behaviour of concurrent programs to locate problems and optimize performance. The best way to improve profiling capabilities and reduce the time to analyze a concurrent program is to use a target-specific profiler that understands the underlying concurrent runtime environment. A profiler for understanding execution of user and kernel level threads is presented, which is target specific for the μC++ concurrency system. This allows the insertion of hooks into the μC++ data structures and runtime kernel to ensure crucial operations are monitored exactly. Because the profiler is written in μC++ and has an extendible design, it is easy for users to write new metrics and incorporate them into the profiler.

1 Introduction

As programs grow more complex, a greater need arises for understand their dynamic behaviours, to locate problems and optimize performance. Concurrency increases the complexity of behaviour and introducing additional problems not present in sequential programs. An important tool for locating problems and performance bottlenecks is a *profiler*. However, sequential profiling techniques cannot be trivially extended into the concurrent domain. A concurrent profiler must deal with multiple threads of control, all potentially introducing errors and performance problems. Profiling concurrent programs has been done for performance analysis, algorithm analysis, coverage analysis, tuning, and debugging.

We believe the best way to improve concurrent profiling capabilities and reduce the time to analyze a concurrent program is to use a target-specific profiler that understands the underlying concurrent runtime environment. Our experience in designing several target-specific concurrency tools (high-level concurrent extensions for C++, called μC++ [1], a debugger [2], a profiler [3], and other concurrent toolkits) leads us to conclude that construction of a universal profiler for all languages and concurrency paradigms is doomed to failure.

2 Motivation

The basis of this work is μC++, a shared-memory user-level thread library running on symmetric multiprocessor architectures (e.g., SUN, DEC, SGI, MP-PC); kernel threads associated with shared memory provide parallelism on multiprocessors, and user threads refine that parallelism. The μC++ environment provides

©*Proceedings of the Second International Symposium (ISCOPE'98) on Computing in Object-Oriented Parallel Environments*, volume 1505, pages 159–166, Santa Fe, New Mexico, U.S.A., December 1998. Springer-Verlag.

a target-specific debugger for break-point debugging on a user-level thread basis, and experience has shown it aids in the development of robust concurrent programs. Nevertheless, debugging is normally based on a hypothesis concerning the reason for the erroneous behaviour of a program. To reason about general runtime behaviour, including performance analysis, coverage analysis and tuning, a profiling tool is needed to monitor execution and reveal information at different levels of detail.

Several profilers for concurrent programs exist, but most are general purpose tools with little understanding of the concurrency paradigm. Each concurrent environment provides a different paradigm, which a profiling tool must be aware of in order to provide effective monitoring, analyzing and visualizing of the program's behaviour. The analysis of a concurrent program's performance and algorithmic behaviour becomes more effective and efficient through target-specific profiling, where the profiler has internal knowledge about the runtime system intrinsics and the underlying programming paradigm.

Extendibility is also crucial in designing and implementing an effective profiler, since it is impossible to predict suitable metrics for all imaginable situations. Therefore, a profiling tool should provide a set of general purpose metrics *and* a mechanism enabling a program analyst to quickly develop new problem specific metrics. Hence, an analyst must use knowledge about the profiler, which is easier when the profiler operates as part of the target system and when the metric extensions can be written in a familiar language.. Ideally, the same language is used for the profiled program and the profiler extensions.

Finally, a profiler must operate at different levels of detail on concurrent programs to provide the functionality for both exact and statistical profiling. To profile large-scale concurrent programs, selective profiling must be supported: It must be possible to turn profiling on and off dynamically to target specific parts of a large program.

3 Related Work

Most profiling tools have been developed for analyzing the performance of scientific, mostly data-parallel programs, written in a message-based programming environment. For this arena, successful and powerful tools with a wide range of analysis and visualization modules exist. For example, Pablo [4] is a tool with many visual [5] and audio [6] performance data presentation modules. Pablo also introduced a standard trace log format, which is adopted by other profile analysis and visualization tools. Another example is Paradyn [7], a tool for profiling large-scale long-running applications. These program characteristics require some novel instrumentation and analysis methods: dynamic instrumentation insertion and removal based on execution-time profiling information, or user interaction. The results of dynamic instrumentation are promising, but the overhead introduced may reduce effectiveness when profiling code-parallel (in contrast to data-parallel) programs with shorter execution times.

Concurrent profiling tools may be available only as part of the operating system, which allows monitoring of programs, and information about calls for kernel thread creation, synchronization and communication primitives. For example, the Mach Kernel Monitor [8] instruments kernel thread context switching. This approach assumes that the concurrent program's runtime system uses only operating system features, instead of providing portable, user-level thread creation, synchronization and communication primitives.

Among the first profiling tools for a user-level thread-library was Quartz [9]. A target-specific profiling environment for concurrent object-oriented programs is pC++ [10]. pC++ is one of the few cases where the integrated performance analysis environment TAU [11] was implemented in concert with the language and runtime system. However, the design of pC++/TAU incorporates most of its profiling functionality into the preprocessor and runtime system, so extending the profiling metrics by the program analyst takes significant effort. The tight coupling between the language/runtime system and the profiling tool makes integration into other existing thread-libraries infeasible.

4 μProfiler

μProfiler [3] is a concurrent profiler, running on UNIX based symmetric shared-memory multiprocessors, that achieves our goal of target-specific, extendible, fine-grained profiling on a user-level thread basis. μProfiler supports the μC++ shared-memory programming model, which shares all data and has multiple kernel and user-level threads. Profiling μC++ programs requires incorporating both concurrent and object-oriented aspects, i.e., profiling different threads of control at the per-object level.

Profiling sequential programs is non-trivial but well-understood. Additional challenges arise when profiling concurrent programs in a shared-memory environment similar to μC++. Since the environment provides user-level tasks, the profiler must monitor the program's activity at that level. The profiler also needs internal knowledge about the runtime system to identify and monitor each executing task independently and exactly. The μProfiler design deals with these challenges and presents a mechanism to effectively integrate extendible profiling into the concurrency system.

μProfiler is a concurrent program written in μC++, executing concurrently with the profiled μC++ application (see Figure 1). A *cluster*, which groups user and kernel threads and restricts the execution of those user threads by the kernel threads in the cluster, is the μC++ capability which enables concurrent application and profiler execution. The user threads in the profiler cluster monitor execution of the runtime kernel and other clusters using direct memory reads via the shared memory. On multiprocessor computers, the kernel thread in the profiler cluster executes the profiler user threads in parallel with the application. If the amount of the monitoring is large, more kernel threads can be added to the profiler cluster to increase parallelism. So application performance is degraded only by the contention created by profiler operations. This cost can be 100 to

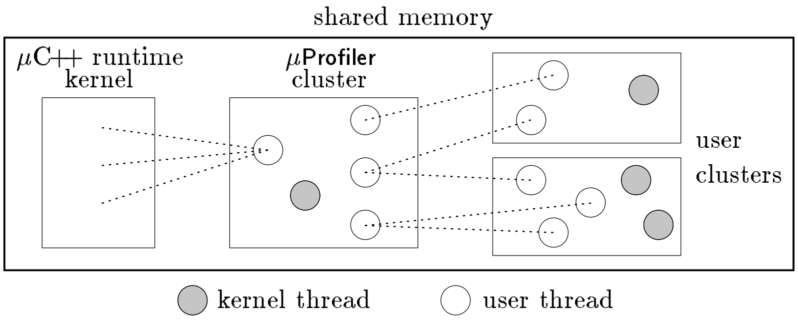


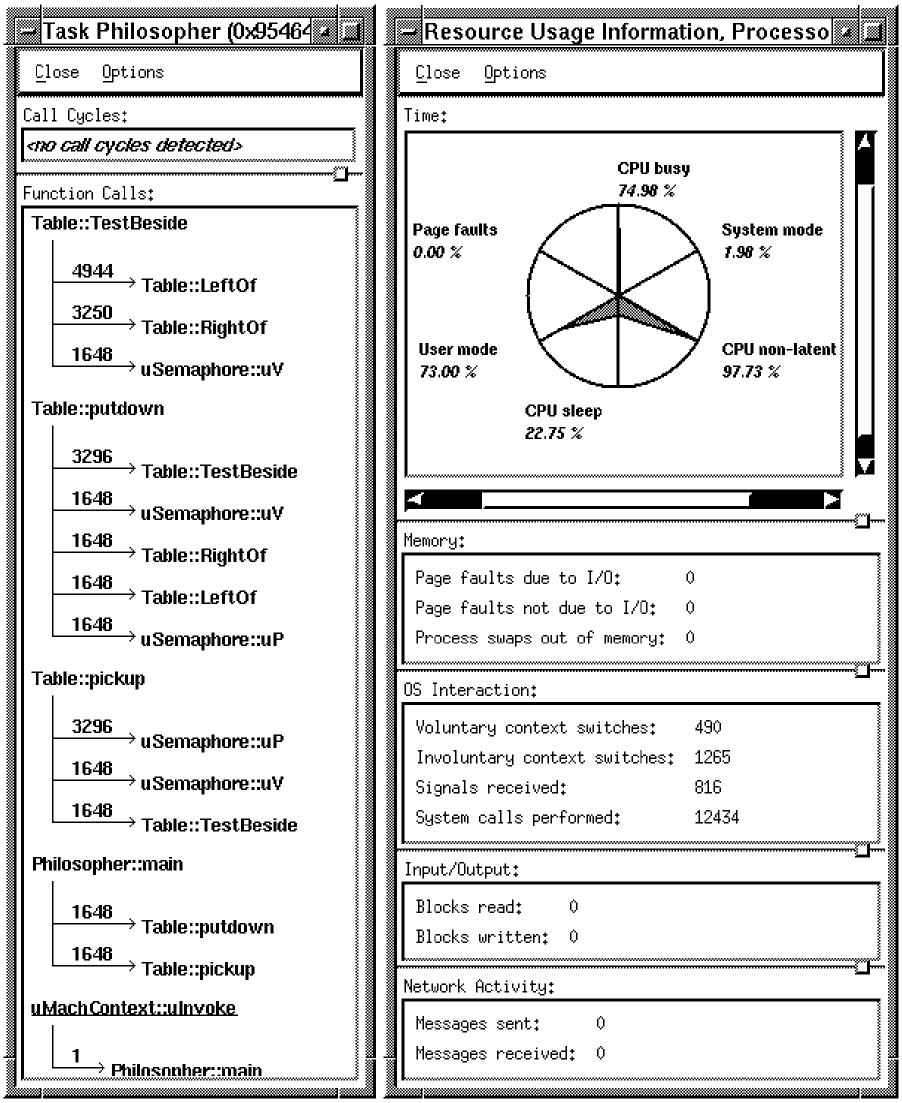
Fig. 1. Integration of μProfiler into the profiled program.

1000 times less than monitoring from a separate UNIX process, which requires cross-address-space reads. More complex monitoring is thus possible, while still having only a small effect on the application.

To access μProfiler, compilation flags -profile and -kernelprofile cause the necessary instrumentation insertion and linking with the profiling libraries. The first flag profiles only the user program; the second flag profiles the μC++ kernel calls made by the program. (The latter information is often inappropriate and confusing to users.) When the program starts, a menu appears, from which a user selects several builtin metrics, after which the program is run, and the metric output appears. Thus, in the simplest case, instrumentation insertion and activation of the profiling modules is completely transparent to the programmer. Additionally, parts of a program may be compiled with or without the profile flag(s), and then linked together, creating an executable where selected parts are instrumented and profiled. Finally, even more precise control is available through routines in the μC++ runtime system to turn profiling on and off for a particular thread at any point during execution.

Because μProfiler is truly integrated with μC++, it was possible to insert hooks into the μC++ runtime kernel to ensure crucial operations are monitored exactly, such as user and kernel thread creation/destruction, and migration of user and kernel threads among clusters. Purely statistical monitoring and dynamic instrumentation could miss some of these events. Also, dynamic instrumentation is considered too expensive when profiling programs with short or intermediate execution times. Exact routine counts are obtained via static instrumentation insertion at compile-time using shared trampolines. The C -pg option is used to generate routine-entry instrumentation; the routine-entry instrumentation is augmented to generate routine-exit instrumentation. In addition, arbitrary hooks can be inserted into user code by the program analyst.

All hooks can be dynamically activated and deactivated on a per thread basis, but only when the profiler is present in the application (i.e., the existence of the profiler is checked for dynamically inside the runtime kernel). Each activated hook results in the profiled thread sending event(s) to the profiler, which passes the information to the active profiling monitors. Figure 2(a) shows a (simple)



(a) Exact User Thread Metric (b) Exact Kernel Thread Metric

Fig. 2. μProfiler Exact Metrics

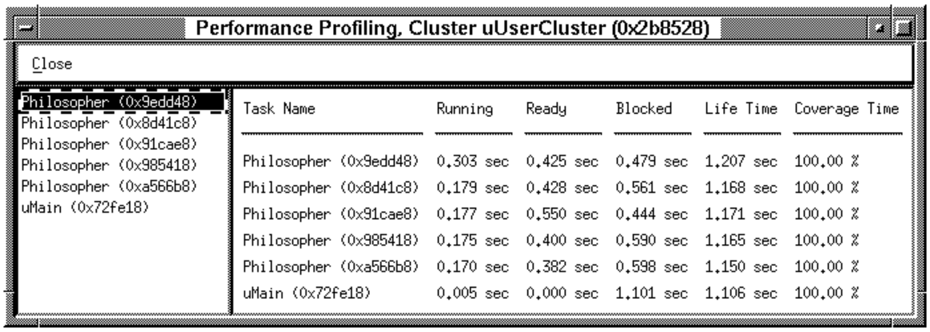
exact metric operating at the user thread level. For each user thread, *gprof*-like [12] routine call information is available, including call cycles. Function calls from within each executed routine are presented with the corresponding routine call count information. Figure 2(b) shows a (simple) exact metric operating at the kernel thread level. For each kernel thread, *UNIX* level information is available, including a Kiviat graph to quickly relate different time metrics.

For statistical monitoring, μ**Profiler** can monitor selected threads by periodically sampling at a dynamically adjustable frequency to collect profiling data with minimum interference. The current implementation of μ**Profiler** monitors the profiled program at the per-cluster, per-thread, routine level. Figure 3(a) shows a statistical metric operating at the cluster level. For this cluster, performance statistics are displayed for each task executing on it, broken down by task states (running, ready, blocked). "Coverage Time" is the percentage of time the task is sampled. By clicking on any of the tasks listed for a cluster, detailed information is available for that task. Figure 3(b) shows a statistical metric operating at the task level. For this task on the cluster, performance statistics are displayed for each routine call executed by the task, broken down by task states.

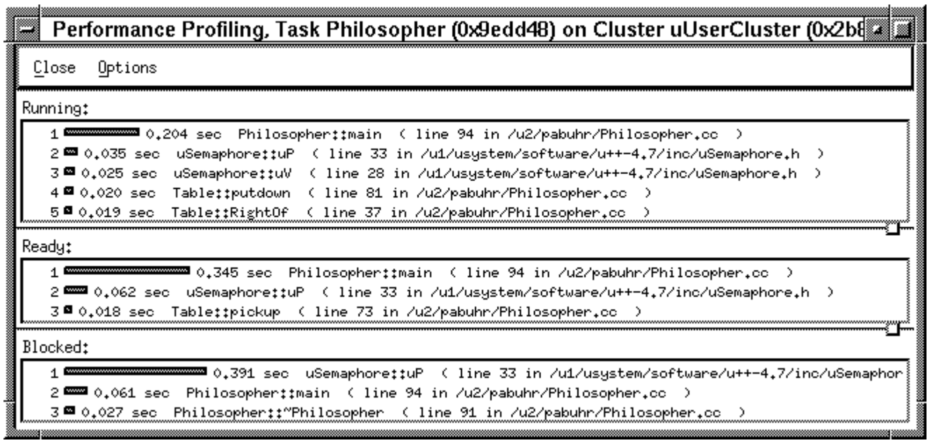
To ensure a high degree of flexibility and extendibility, μ**Profiler** is subdivided internally into parts representing the underlying functionality, including a profiling kernel, execution monitors, metric analyzers, and visualization devices. Each of these parts is split into submodules, which are ordered in a class hierarchy. To build a new metric requires building at least two components: an execution monitor component and a metric analysis component. An execution monitor is built as follows. Determine the functionality of the metric: e.g., exact, statistical or both kinds of profiling. Then create a C++ class that inherits from the abstract class **uExecMonitor**, and specialize a subset of **uExecMonitor**'s virtual routines to provide the necessary functionality. Class **uExecMonitor** provides virtual members for different purposes such as routine entry/exit notification, periodical polling, etc. Finally, add an initialization call to the routine **Initialize** in the constructor of the new class. Each execution monitor is responsible for operating and updating its own objects and possibly accumulating, filtering or summarizing the profiling data collected when the profiler task calls the registered members. Creating new metric analysis components is done in a similar manner by inheriting from a class called **uMetricAnalyze**. Specialized members of **uExecMonitor** are automatically registered with the profiler during the call to **Initialize** along with the new execution monitor. μ**Profiler** maintains a list of all execution monitors and their member routines, and invokes them during execution as needed. Since the registration process of new metrics is done dynamically, they can simply be linked with the application, restart it, and μ**Profiler** calls into the metric class's member routines when the requested events occur.

Additional reuse is provided by inheriting from existing metrics that come with the μ**Profiler** library or previously built by the program analyst. All μ**Profiler** metrics conform to the above mechanism. **uSPMonitor**, for instance, is an execution monitor that statistically samples a task and measures the time the task spends on a certain cluster in a certain routine in a particular state. Because **uSPMonitor** is based on statistical sampling, it inherits from **uExecMonitor** and specializes the poll routine, in which the data collection is performed.

Through this mechanism, an analyst can efficiently extend the functionality of μ**Profiler** by metrics that fit the analyzed problem much better than general



(a) Cluster Performance Metric



(b) Task Performance Metric

Fig. 3. μProfiler Performance Metric

purpose metrics created by the developers, resulting in profiling results that directly correspond to the problem under investigation. This approach enables an analyst to extend μProfiler's functionality by any metric, analysis or visualization device utilizing using exact or statistical monitoring. It is more important to integrate the basic functionality for different execution monitoring, analyzing and visualizing methodologies on which both general and problem-specific modules can operate, than to build a fixed set of highly sophisticated metrics.

Concurrency is also part of some object-oriented language, e.g., μC++. μProfiler can identify the corresponding objects (both caller and callee side) when a monitored task invokes an object's member routines. While there are no metrics using this feature, we anticipate them soon.

5 Conclusion

Concurrent systems have increased dynamic behaviour with significant implicit information embedded in the runtime environment. Our claim is that target-specific profilers can do a better job extracting and displaying information from this environment. We show tight integration is possible with a target-specific profiler, i.e., between μProfiler and μC++, resulting in better information gathering at lower cost, and the ability to easily add new metrics through a single programming language. The μProfiler displays are simple but informative, requiring the program the analyst to manually locate performance issues, e.g., hot spots, by examining the data. We have found manual determination to be straightforward, and have discovered several performance problems using μProfiler while examining both μC++ and μC++ applications to understand their dynamic behaviour.

References

1. Peter A. Buhr and Richard A. Stroobosscher. μC++ annotated reference manual, version 4.6. Technical report, Univ. of Waterloo, July 1996. Available via ftp from plg.uwaterloo.ca in pub/uSystem/uC++.ps.gz.
2. Peter A. Buhr, Martin Karsten, and Jun Shih. A multi-threaded debugger for multi-threaded applications. In *Proc. of SPDT'96: SIGMETRICS Symp. on Parallel and Distributed Tools*, pages 80–87, Penn., U. S. A., May 1996. ACM Press.
3. Robert R. Denda. Profiling Concurrent Programs. Diplomarbeit, Universität Mannheim, Mannheim, Deutschland, September 1997. Available via ftp from plg.uwaterloo.ca in pub/MVD/DendaThesis.ps.gz.
4. D.A. Reed et al. Scalable Performance Analysis: The Pablo Performance Analysis Environment. *Scalable Parallel Libraries Conference*, 1993. Computer Society.
5. Daniel A. Reed et al. Virtual reality and parallel systems performance analysis. *IEEE Computer*, 28(11), November 1995.
6. Tara Maja Madhyastha. A portable system for data sonification. Master's thesis, Rutgers State Univ., 1990.
7. Barton P. Miller et al. The paradyn parallel performance measurement tools. *IEEE Computer*, 28(11), November 1995.
8. T. Lehr et al. MKM: Mach Kernel Monitor Description, Examples and Measurements. Technical report, Carnegie-Mellon Univ., March 1989. PA-CS-89-131.
9. T.E. Anderson and E.D. Lazowska. Quartz: A Tool for Tuning Parallel Program Performance. In *Proc. of the 1990 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 115–125, Boston, May 1990.
10. A. Malony et al. Performance analysis of pC++: A portable data-parallel programming system for scalable parallel computers. In *Proc. of the 8th International Parallel Processing Symp.*, pages 75–85, Cancun, Mexico, April 1994.
11. Darryl Brown et al. Program analysis environments for parallel language systems: The τ environment. In *Proc. of the 2nd Workshop on Environments and Tools For Parallel Scientific Computing*, pages 162–171, Townsend, Tennessee, May 1994.
12. S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: a call graph execution profiler. *SIGPLAN Notices*, 17(6):120–126, June 1982. Proc. of the SIGPLAN'82 Symp. on Compiler Construction, June 23–25, 1982, Boston, U.S.A.