A Tool Suite for Simulation Based Analysis of Memory Access Behavior

Josef Weidendorfer[†], Markus Kowarschik[††], Carsten Trinitis[†]

[†]Technische Universität München, Germany
{weidendo,trinitic}@cs.tum.edu

[††]Universität Erlangen-Nürnberg, Germany
kowarschik@cs.fau.de

Abstract. In this paper, two tools are presented: an execution driven cache simulator which relates event metrics to a dynamically built-up call-graph, and a graphical front end able to visualize the generated data in various ways. To get a general purpose, easy-to-use tool suite, the simulation approach allows us to take advantage of runtime instrumentation, i.e. no preparation of application code is needed, and enables for sophisticated preprocessing of the data already in the simulation phase. In an ongoing project, research on advanced cache analysis is based on these tools. Taking a multigrid solver as an example, we present the results obtained from the cache simulation together with real data measured by hardware performance counters.

Keywords Cache Simulation, Runtime Instrumentation, Visualization.

1 Introduction

One of the limiting factors for employing the computational resources of modern processors is excessive memory access demands, i.e. not taking advantage of cache hierarchies by high temporal and spatial locality of sequential memory accesses. Manual optimization activities trying to overcome this problem typically suffer from difficult-to-use and -to-setup tools for detailed bottleneck analysis. We believe that profiling tools based on simulation of simple cache models can significantly help in this area. Without any need for hardware access to observation facilities, the simulator produces data easy to understand and does not influence simulated results. Therefore, runtime instrumentation and sophisticated preprocessing is possible directly in the simulation phase. Optimizations based on cache simulation usually will improve locality of an application in a general way, therefore enabling better cache usage not only on a specific hardware platform, but on cache architectures in general. This gives the user more flexibility in terms of target architectures. Furthermore, it is essential to present the obtained measurement data in a way easy to understand.

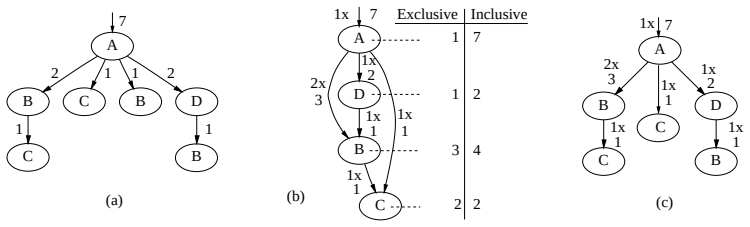

Fig. 1. A dynamic call tree, its call graph, and its context call tree

In this paper, we discuss the tool suite Calltree/KCachegrind[1]. The profiling tool uses the instrumentation framework Valgrind [18], running on Linux/IA-32. The visualization front end, which is based on QT/KDE libraries, runs on most UNIX platforms.

The paper is organized as follows: in the next section, we give a short overview on profiling methods and problems. In section 3, simulation based profiling and the implementation we use is covered in detail. Visualization concepts and implementation are presented in section 4. In our ongoing project DiME[2], memory intensive multigrid algorithms are optimized with regard to their memory access behavior. Section 5 discusses this application and presents measurements from both simulation and hardware counters. We close by presenting related work and future research directions.

2 Profiling

It is important to concentrate on code regions where most time is spent in typical executions. For this, runtime distribution is detected by *profiling*. Also, it can approve assumptions regarding runtime behavior, or show the right choice of multiple possible algorithms for a problem. A dynamic call-tree (DCT), see fig. 1 (a), represents a full trace from left to right, and has a size linear to the number of calls occurring. For profiling, the much smaller dynamic call graph (DCG), see fig. 1 (b), typically is enough. However, a DCG contains less information: in fig. 1 (b), one cannot see that D → B → C actually never happened. Therefore, [1] proposed the context call tree (CCT), see fig. 1 (c), where each node represents the occurrence of a function in a call chain, a context, starting at the root node[3]. Our tool is able to produce a "reduced" $\mathrm{CCT}_{n_{max}}$: as a CCT can still get huge, we collapse two contexts if the trailing n_{max} contexts of the corresponding call chains are identical. Table 1 gives some numbers on the size of these reduced CCTs for various UNIX applications.

During the profiling processing, event attribution of nodes and arcs is done. Events can be memory accesses, cache misses, or elapsed clock ticks, for example.

[1] `http://kcachegrind.sf.net`

[2] `http://www10.informatik.uni-erlangen.de/Research/Projects/DiME/`

[3] In the CCT, recursive calls or mutually recursive functions are handled specially to avoid a size linear to the number of calls occurring at runtime.

Table 1. Number of nodes and arcs in reduced CCTs

Command		Call chain length limit n_{max} 0	1	2	5	10	20
bzip2 libm.so.6	Nodes	408	850	1 004	1 329	1 332	1 132
(Compressor)	Arcs	538	688	861	1 113	1 113	1 113
cc1 ct_main-i.c	Nodes	1 519	5 157	8 352	22 060	41 164	44 899
(C compiler)	Arcs	6 741	10 881	15 905	34 282	52 191	54 722
konqueror	Nodes	21 500	55 829	91 713	251 449	420 871	507 507
(KDE Browser)	Arcs	51 052	90 629	147 958	315 838	470 032	544 487

Attributes of interest for nodes are the number of events occurring inside the function (*exclusive* or *self* cost), and additionally in functions which are called from the given function (*inclusive* cost). For arcs, events occurring while the program is running in code called via this arc are interesting, as well as the number of times the call is executed. In fig. 1, event attribution is shown, assuming one event during the execution of each function.

Profiling should not perturb the performance characteristics of the original program, i.e. its influence on runtime behavior should be minimal. For this reason, modern CPU architectures include performance counters for a large range of event types, enabling *statistical sampling*: After a hardware counter is set to an initial value, the counter is decremented whenever a given event takes place, and when reaching zero, an interrupt is raised. The interrupt handler has access to the program counter of the interrupted instruction, updates statistical counters related to this address, resets the hardware counter and resumes the original program[4]. The result of statistical sampling is self cost of code ranges.

To overcome the limits of pure statistical sampling, one has to instrument the program code to be profiled. Several instrumentation methods exist: source modification, compiler injected instrumentation, binary rewriting to get an instrumented version of an executable, and binary translation at runtime. Instrumentation adds code to increment counters at function entry/exit, reading hardware performance counters, or even simulate hardware to get synthetic event counts. To minimize measurement overhead, only small action is performed[5]. When synthetic event counters are enough for the profiling result quality to be achieved, hardware simulation allows for runtime instrumentation with its ease of use.

3 Simulation Based Profiling

Runtime instrumentation can dramatically increase execution time such that time measurements become useless. However, it is adequate for driving hard-

[4] For the results of statistical sampling to be meaningful, the distribution of every n-th event occurring over the code range of a program should be the same as the distribution of *every* event of this type.

[5] GProf [8] instrumentation still can have an overhead of 100%. Ephemeral Instrumentation [21], can keep instrumentation overhead smaller.

ware simulations. In our profiling tool, we use the CPU emulator *Valgrind* [18] as a runtime instrumentation framework. The instrumentation drives the cache simulation engine, which is largely based on the cache simulator found in Valgrind itself: calls to the simulator are inserted on every instruction fetch, data read, and data write. Separate counters for each original instruction are incremented when an event occurs in the simulation of the simple, two-level cache hierarchy. The cache model enables the user to understand the numbers easily, but it has drawbacks compared to reality: it looks at memory accesses from user-level only, simulates a single cache, and assumes a processor without an out-of-order engine and without speculation. Despite of this, the synthetic event counters often come close to numbers of actual hardware performance counters [17]. We note that the simulation is not able to predict consumed wall clock time, as this would need a detailed simulation of the microarchitecture.

Our addition to the simulation is two-folded: First, multiple counters are maintained even for the same instruction, depending on the current thread ID or the call chain leading to this instruction. Thus, we support profiles per threads in multi-threaded code, and more fine-granular analysis by event attribution to CCTs. Second, we introduce the ability for construction of the dynamic call graph of the executed program. For every call site in the program, the list of target addresses called from that site is noted, and for each of these call arcs, the cost spent inside the called function. Optionally, this is even done for every context of a call site, where the context includes the current thread ID and call chain leading to this call site. To be able to provide the recent portion of a call chain with a given length, we maintain our own call stack for every thread and signal handler. Fortunately, thread switches can be trapped with Valgrind (using its own implementation of the POSIX Thread API). To be able to reliably trace a call stack, we always have to watch the stack pointer not only on detection of a CALL or RET instruction: most implementations of exception support in different languages (such as C++), and the C runtime functions setjmp/longjmp write the stack pointer directly. Function calls to shared libraries usually are resolved by the dynamic linker the first time they take place. After resolution, the linker directly jumps to the real function. As the notion of a jump between functions is not allowed in a call graph, we pretend two independent calls from the original call site.

4 Visualization of Profiling Data

Most profiling tools provide a post-processing command to output a text file with a list of functions sorted by cost and annotated source. While this is useful, it makes it difficult to get an overview of the performance characteristics of an application without being overwhelmed by the data measured. A GUI should enable convenient browsing in the code. Requirements are

– Zooming from coarse to fine granular presentation of profiling data, starting from event relation to all functions in a shared library, a source file or a

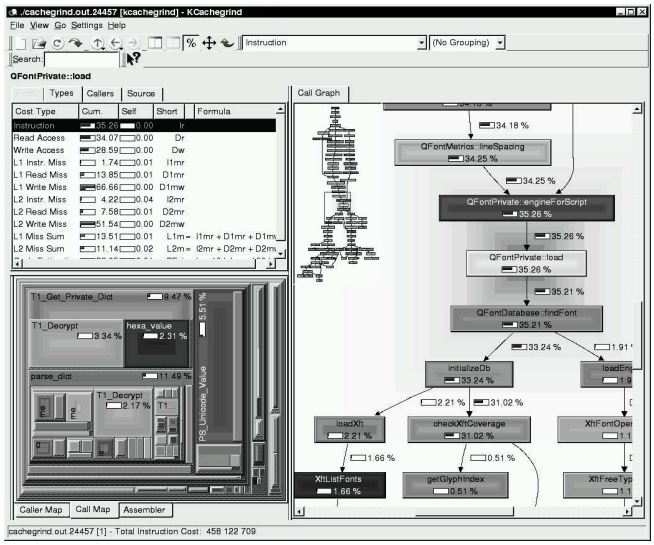

Fig. 2. Cost Types (Top), Call Graph (Right) and Call TreeMap (Bottom)

C++ class, and going down to relation to loop structure, source code lines and machine instructions.
- Support for arbitrary and user specified, derived event types, including support for profile data produced by different profiling tools. Additionally, it is important to be able to specify and browse according to derived event types, which have to be calculated on the fly from the given ones (e.g. MFLOPS).
- Visualization of metrics that makes it easy to spot performance problems.
- Possibility to combine and compare multiple profile measurements produced by different tools.

Even for the simplest program, the runtime linker, the setup of the environment for the C runtime library, and implementation of high-level print functions via low-level buffering are involved. Thus, only a fraction of a call graph should be presented at once. Still, it is desirable to show multiple levels of a call chain. As a function can be part of multiple call chains, we eventually have to estimate the part of cost which happens to be spent in the given chain. This is only needed when context information, i.e. an attributed CCT, is not available. Our call graph view starts from a selected function, and adds callers and callees until costs of these functions are very small compared to the cost of the selected function. Because the functions in the call graph view can be selected by a mouse click, the user can quickly browse up or down a call chain, only concentrating on the part which exposes most of the event costs. Another view uses a TreeMap visualization [20]. Functions are represented by rectangles whose area is proportional to the inclusive cost. This allows for functions to be fully drawn inside their caller. Therefore, only functions which are interesting for performance analysis

Table 2. Simulation and profiling results for the 3D multigrid code

	Simulation			Real Measurement		
	Instr. exec.	L2 Misses	Runtime	Instr. retired	L2 Lines In	Runtime
Standard	11 879 M	751 421 K	1 865 s	11 879 M	777 131 K	40.4 s
Optimized	11 666 M	361 336 K	1 798 s	11 666 M	383 609 K	27.1 s

are visible. In a TreeMap, one function can appear multiple times. Again, costs of functions have to be split up based on estimation. Figure 2 shows a screenshot with both the call graph and TreeMap visualization. Additionally, on the top left the list of available cost types is shown, produced by the cache simulator. As "Ir" (instructions executed) is selected as cost type, these costs are the base for the graphical drawings.

To be able to show both annotated source and machine code[6], we rely on debug information generated by the compiler. Our tool parses this information and includes it in the profile data file. Thus, the tool suite is independent from the programming language. Additionally, our profiling tool can collect statistics regarding (conditional) jumps, enabling jumps visualized as arrows in the annotation view.

5 Application Example and Results

In our DiME project, research on cache optimization strategies for memory-intensive, iterative numerical code is carried out. The following experiment refers to a standard and to a cache-optimized implementation of multigrid V(2,2) cycles, involving variable 7-point stencils on a regular 3D grid with 129^3 nodes. The cache-efficient version is based on optimizations such as array padding and loop blocking [11]. Table 2 shows simulated and real events obtained on a Pentium-M with 1.4 GHz with corresponding wall clock runtimes. In both cases, the advantage of cache optimization is obvious by effectively reducing the number of cache misses/cache line loads by 50%. Reduction of runtime gives similar figures.

Simulation runtimes show that the slowdown of runtime instrumentation and cache simulation is quite acceptable.

6 Related Work

The most popular profiling tool on UNIX are `prof`/`gprof` [8]: a profile run needs the executable to be instrumented by the GCC compiler, which inserts code to get call arcs and call counts, and enables time based statistical sampling. Measurements inside of shared library code is not supported. GProf has to approximate the inclusive time by assuming that time spent inside a function grows linearly with the number of function calls. Tools taking advantage of hardware

[6] For machine code annotation, the standard GNU disassembler utility 'objdump' is used.

performance counters for event based sampling are available by most processor vendors (e.g. DCPI for Alpha processors [2] or VTune for Intel CPUs). Most of them relate event counters to code, but Sun's developer tools also allow data structure related measurements since recently [10]. OProfile is a free alternative for Linux [12]. PAPI [4] is a library for reading hardware performance counters. TAU [16] is a general performance analysis tool framework for parallel code, using automated source instrumentation.

For instrumentation purpose, binary rewriting of executables is possible with ATOM [7]. DynInst [15] allows for insertion of custom code snippets even into running processes. Regarding runtime instrumentation, the Shade tool [6] for SPARC inspired development of Valgrind. Sophisticated hardware simulators are MICA [9] and RSIM [19]. Using simulation, MemSpy [13] analyses memory access behavior related to data structures, and SIP [3] gives special metrics for spatial locality. A profile visualization with a web front end is HPCView [14].

7 Conclusions and Future Work

In this paper, we presented a tool suite for profiling of sequential code based on cache simulation and runtime instrumentation, and a graphical visualization front-end allowing fast browsing and recognition of bottlenecks. These tools have already been used successfully, e.g. in projects at our lab as well as in other labs in Germany, and in the open-source community.

Improvement of the simulation includes data structure based relation, even for dynamic allocated memory and stack accesses. This is feasible with the runtime instrumentation approach. In order to avoid the measured data to become huge, relation to data types seems necessary. For large arrays, relation to address differences of accesses promises to be interesting. Suited visualizations are needed. Other directions for research are more sophisticated metrics to easily detect problems regarding spatial or temporal locality. LRU stack distances of memory accesses [5] are especially useful as they are independent on cache sizes. For the visualization tool, we want to support combination of simulated and real measurement: e.g. flat profiles from event based sampling with hardware performance counters can be enriched with call graphs got from simulation, and thus, inclusive cost can be estimated. In addition, we recognize the need to further look into differences between simulation and reality.

Acknowledgement We would like to thank Julian Seward for his excellent runtime instrumentation framework, and Nick Nethercote for the cache simulator we based our profiling tool on.

References

1. G. Ammons, T. Ball, and J. R. Larus. Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. In *Proceedings of PLDI '97*, June 1997.

2. J. M. Anderson, L. M. Berc, J. Dean, et al. Continuous Profiling: Where Have All the Cycles Gone? *ACM Transactions on Computer Systems*, 15(4):357–390, November 1997.
3. E. Berg and E. Hagersten. SIP: Performance Tuning through Source Code Interdependence. In *Proceedings of the 8th International Euro-Par Conference (Euro-Par 2002)*, pages 177–186, Paderborn, Germany, August 2002.
4. S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *The International Journal of High Performance Computing Applications*, 14(3):189–204, Fall 2000.
5. G. C. Cascaval. *Compile-time Performance Prediction of Scientific Programs*. PhD thesis, University of Illinois at Urbana-Champaign, August 2000.
6. B. Cmelik and D. Keppel. Shade: A Fast Instruction Set Simulator for Execution Profiling. In *SIGMETRICS*, Nashville, TN, US, 1994.
7. A. Eustace and A. Srivastava. ATOM: A Flexible Interface for Building High Performance Program Analysis Tools, 1994.
8. S. Graham, P. Kessler, and M. McKusick. GProf: A Call Graph Execution Profiler. In *SIGPLAN: Symposium on Compiler Construction*, pages 120–126, 1982.
9. H. C. Hsiao and C. T. King. MICA: A Memory and Interconnect Simulation Environment for Cache-based Architectures. In *Proceedings of the 33rd IEEE Annual Simulation Symposium (SS 2000)*, pages 317–325, April 2000.
10. M. Itzkowitz, B. J. N. Wylie, Ch. Aoki, and N. Kosche. Memory profiling using hardware counters. In *Proceedings of Supercomputing 2003*, November 2003.
11. M. Kowarschik, U. Rüde, N. Thürey, and C. Weiß. Performance Optimization of 3D Multigrid on Hierarchical Memory Architectures. In *Proc. of the 6th Int. Conf. on Applied Parallel Computing (PARA 2002)*, volume 2367 of *Lecture Notes in Computer Science*, pages 307–316, Espoo, Finland, June 2002. Springer.
12. J. Levon. OProfile, a system-wide profiler for Linux systems.
13. M. Martonosi, A. Gupta, and T. E. Anderson. Memspy: Analyzing memory system bottlenecks in programs. In *Measurement and Modeling of Computer Systems*, pages 1–12, 1992.
14. J. Mellor-Crummey, R. Fowler, and D. Whalley. Tools for Application-Oriented Performance Tuning. In *Proceedings of 15th ACM International Conference on Supercomputing*, Italy, June 2001.
15. B. P. Miller, M. D. Callaghan, J. M. Cargille, et al. The Paradyn Parallel Performance Measurement Tool. *IEEE Computer*, 28(11):37–46, November 1995.
16. B. Mohr, A. Malony, and J. Cuny. *Parallel Programming using C++*, chapter TAU. G. Wilson, editor, M.I.T. Press, 1996.
17. N. Nethercote and A. Mycroft. The Cache Behaviour of Large Lazy Functional Programs on Stock Hardware. In *Proceedings of the ACM SIGPLAN Workshop on Memory System Performance (MSP 2002)*, Berlin, Germany, July 2002.
18. N. Nethercote and J. Seward. Valgrind: A Program Supervision Framework. In *Proceedings of the Third Workshop on Runtime Verification (RV'03)*, Boulder, Colorado, USA, July 2003. Available at `http://valgrind.kde.org/`.
19. V. S. Pai, P. Ranganathan, S. V. Adve, and T. Harton. An Evaluation of Memory Consistency Models for Shared-Memory Systems with ILP Processors. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 12–23, October 1996.
20. B. Shneiderman. Treemaps for space-constrained visualization of hierarchies. `http://www.cs.umd.edu/hcil/treemap-history/index.shtml`.
21. O. Traub, S. Schechter, and M. D. Smith. Ephemeral instrumentation for lightweight program profiling. In *Proceedings of PLDI '00*, 2000.