Initial Experiments in Visualizing Fine-Grained Execution of Parallel Software Through Cycle-Level Simulation

Rick Strong, Jayaram Mudigonda, Jeffrey Mogul, Nathan Binkert

HP Laboratories
HPL-2008-210

Keyword(s):
Visualization, Parallel programs, Simulation

Abstract:

Programmers wishing to obtain the best possible performance from multi-threaded software on parallel hardware must often understand detailed architecture-level interactions. Visualizations based on execution traces are often useful in gaining such understanding. Previous visualization tools have used traces from actual executions. We have experimented instead with visualizations based on traces from cycle-level simulations; while this approach imposes significant performance penalties, it exposes thread behavior, architectural events, and inter-thread interactions in minute detail unavailable via other means. We describe experiments with a prototype of our tool on several simple applications and on kernel code, and sketch the design of a usable tool.

External Posting Date: December 6, 2008 [Fulltext] Approved for External Publication
Internal Posting Date: December 6, 2008 [Fulltext]

Submitted to First USENIX Workshop on Hot Topics in Parallelism (HotPar), Berkeley, CA, March 30-31, 2009

Initial Experiments in Visualizing Fine-Grained Execution of Parallel Software Through Cycle-Level Simulation

Rick Strong* rstrong@cs.ucsd.edu
Jayaram Mudigonda Jayaram.Mudigonda@hp.com
Jeffrey C. Mogul Jeff.Mogul@hp.com
Nathan Binkert binkert@hp.com
*HP Labs, Palo Alto, CA 94304 and *UC San Diego*

Abstract

Programmers wishing to obtain the best possible performance from multi-threaded software on parallel hardware must often understand detailed architecture-level interactions. Visualizations based on execution traces are often useful in gaining such understanding. Previous visualization tools have used traces from actual executions. We have experimented instead with visualizations based on traces from cycle-level *simulations*; while this approach imposes significant performance penalties, it exposes thread behavior, architectural events, and inter-thread interactions in minute detail unavailable via other means. We describe experiments with a prototype of our tool on several simple applications and on kernel code, and sketch the design of a usable tool.

1 Introduction

Multicore systems create the need to improve the performance of multi-threaded, shared-memory parallel programs. Optimizing the performance of these applications can be quite difficult, and low-level architectural interactions can create unexpected performance problems. Most programmers have a hard time understanding the nature and causes of these interactions.

Parallel programmers can gain insight into the behavior of their software through visualization tools, many of which have been described in prior work [4, 5, 6, 7, 8, 9, 12]. However, these tools have not been able to directly display the fine-grain interactions, such as cache conflicts and lock contention, that can define the performance of a parallel program. While some tools (e.g., DCPI [2]) have been able to infer the likelihood of cache misses afflicting a specific static instruction or line of code, inferential techniques have their limits: inferences can be inaccurate, and can still require significant guesswork on the part of the user.

We have developed an approach that allows us to directly visualize the detailed architecture-level interactions between threads in a parallel program. We can display the execution of a single iteration of a function or loop body (although, of course, real performance problems come from frequently repeated iterations). We can show when cache misses, memory barriers, or other stalls afflict the program's performance. We can also show how operating system execution interacts with user-mode applications, and our approach allows us to visualize operating system behavior just as easily as user-mode behavior – a significant benefit, given that the OS on multicore hardware may be the most critical parallel program that many users encounter.

Unlike previous approaches, which monitor the execution of programs on real hardware, our technique uses detailed traces generated by a cycle-accurate simulation. Accurate simulation has its costs, most obviously in the enormous slowdown it imposes, but it provides arbitrarily detailed visibility, and avoids having to solve the problem of synchronizing multiple traces.

We did not develop this technique for understanding the overall performance of a complex application; it is for zooming in on a "kernel" of an application, once it has been identified by other analyses.

In this position paper, we describe our approach

and show examples of how it can visualize behavior on several toy programs. We also describe how we have used it to understand and improve the performance of complex Linux scheduler code in a multicore system.

Note that we have not actually built a fully-usable tool based on this approach. Instead, we describe the features that such a tool might offer, and we discuss some of the challenges and drawbacks of our approach.

2 Related work

Numerous papers have addressed visualizing the structure and performance of parallel and distributed systems (e.g., [6]). Most of these papers focus on the overall behavior of a system, rather than fine-grained interactions between threads. Often, the goal has been to guide programmers in allocating work to processors. These visualizations have typically shown overall execution statistics, or time series sampled at relatively coarse intervals (for example, [7]).

Some prior work has visualized the timelines of threads interacting at the level of messages [5, 9] or pthreads calls [12]. In these systems, the goal is often to understand the communication patterns, with the aim of either changing work placement or reducing communication in other ways. These approaches typically work by instrumenting significant library calls (e.g., for message operations), or sometimes by instrumenting the application code directly.

Our approach is distinct from previous work in that we focus on the finest-grain interactions between threads in a shared-memory system, and between threads and the underlying hardware architecture. However, we recognize that much of the prior work on parallel-program visualization techniques in general (e.g., [4, 8]) is applicable to fine-grained interactions.

3 A simulation-based approach

The basic outline of our approach is simple: we run the application of interest (along with a real operating system) on a cycle-level simulator, using detailed models of the target hardware. The simulator generates a detailed execution trace, including instruction addresses, data addresses, cache and bus events, etc., with fine-grained timing information. We then distill the events of interest to the programmer, and present them as a timeline graph.

Architecture researchers have developed a number of cycle-level simulators. We use M5 [3], which supports the execution of the entire system, including operating system code (a slightly modified Linux 2.6.18) and models of network and disk devices. Since M5 support for x86 CPUs is not yet fully debugged, we have used its Alpha CPU models, but our approach should apply to any architecture.

M5 supports a variety of core and cache models of various complexities, and allows the user to specify various speeds, cache sizes, and core counts. While this flexibility imposes some extra work on the user, it also allows what-if analyses – for example, will my parallel application run well on a wide variety of systems, including some that might not be available for a few years?

3.1 Trace generation

Once the simulated hardware has been defined, the user can compile the application of interest and install it on the simulated system. In our setup, we simply copy the application binary to a disk image that is read by the operating system running on the simulated system, as well as a script to invoke the application once the simulated Linux system has booted.

Booting an operating system using a cycle-level simulator can take a long time, so M5 supports lower-fidelity (but faster) simulation modes. M5 allows us to boot the system and start the application in a fast-simulation mode, then checkpoint the system state; we can then start a new, cycle-level simulation from the checkpoint, so that only the interesting part of the application suffers the worst slowdowns. Checkpointing is easy; the programmer inserts an **m5_checkpoint** library call, which issues a special machine-code instruction telling M5 to emit a checkpoint.

Events traced: Since a cycle-level simulator can have arbitrarily accurate models of the hardware, it can trace essentially any interesting archi-

tectural event. Our raw traces are text files with one line per event, timestamped with picosecond resolution. Events include instructions executed, showing the PC and memory addresses referenced (if any), memory barriers, and all cache misses. We do not explicitly trace TLB misses and interrupts, but these are implied by the kernel functions they invoke.

We then convert these raw traces to XML, to make post-processing easier. Each sequence of instructions is represented as an XML element, which includes a timestamp, the function name, execution duration, and timestamped sub-elements for each interesting event such as cache misses. We also note "long-latency instructions" (those that take over N cycles; we use $N = 100$).

Converting addresses to names: Users need symbolic names to make sense of the visualizations. We convert a PC value to a *function_name+offset* by searching the symbol table for the highest-valued text-segment symbol less than or equal to the PC. M5 already has access to the kernel's symbol table. Currently, we add application symbols using a script that runs the **nm** command to get symbols, then translates all traced PCs within the range of these symbols. This hack works only for one application per trace. It should be simple to modify M5 to load all application symbol tables (assuming these are not "stripped") and to translate a process's address using the right symbol table, since M5 already knows which Linux process is running on each CPU.

Inferring calls and returns: Currently we detect calls and returns when the *function_name* changes. We keep a stack of these names, and if we start executing in a function whose name is already on the stack, we treat this as a return and pop the name; otherwise, it must be a call, and we push the name. This hack would not work for recursive functions, so we intend to re-implement this feature using M5's existing mechanism for parsing the stack into frames, which would directly provide the stack level. (Correctly determining the stack level in the presence of non-local gotos, etc., can be challenging.)

Tracing more events: It would not be difficult to add other interesting events to the trace, such as access to named global variables (including offsets for structure fields or array elements), and other architectural events, such as lock attempts, lock grants, unlocks, and power-state transitions.

It might also be possible to trace access to dynamically-allocated variables of a given type, if the language runtime provides appropriate cues.

3.2 Trace visualization

We visualize traces as plots of thread timelines, with time on the x axis and call-stack depth on the y axis. For each thread, we represent its timeline with a unique color, and we label function call points with the name of the functions. Interesting events, such as cache misses, are marked on the timelines. Fig. 2 shows an example; we will describe the details of this example in sec. 4.

Currently, we process the XML trace into gnuplot commands, which gives us the ability to create PostScript (PS). files. Unfortunately, these output files are huge, and the PS visualization tools do not let us search for a particular function name. We end up searching the XML files to find events of interest, then we re-generate the image file with timestamps limited to a window around the interesting events. This is tedious and error-prone.

We would like, instead, to have an XML-driven browser that allows us to zoom and pan on the trace visualization; to search for specific events (such as function names, variable accesses, or architectural events such as L2 cache misses); and to selectively suppress details such as the names of boring functions. The browser should also provide "calipers" to allow fine-grained measurements of time intervals, and could allow "diffing" of the fine-grained behavior between two traces. But this is definitely future work.

3.3 Automated analyses

Our current visualization approach depends greatly on the user's understanding of the software and hardware, and ability to form mental images. For example, the causal relationship between two threads competing for a lock might jump out at one user from the visualization, while another user might puzzle over the same image trying to figure it out.

A trace browser could provide relatively simple automated analyses to help the user. For example, it

should be possible to match up all events (in various threads) corresponding to a given lock address, and to display a simplified timeline showing how the threads contend for the lock. It should also be possible to match up references from multiple CPUs that are causing cache-coherency misses. (The DCPI tool [2] made similar kinds of inferences for events local to a single thread.)

4 Experiments with simple programs

We created two toy programs to illustrate our visualizations:

- **ping-pong**: two threads repeatedly access the same cache line. The (**senderThread**) sets a word to 1 (in a function called **setSharedVar**) and polls it until it becomes 0; the (**listenerThread**) poll-waits for it to become 1, then sets it to 0 (in a function called **clearSharedVar**). We put a function call in each inner loop to make the timelines more readable.
- **spin**: N threads set up as pairs of producer-consumer bucket brigades, using pthreads spinlocks to protect the queue between each pair of threads. (For simplicity, we show results for $N = 2$.) After a thread passes along an item (in a function called **move**) it does a modest amount of arithmetic (in a function called called **compute**) before iterating.

Each thread was pinned to a dedicated core in a 2-core system; we modelled "EV6-like" cores with per-core L1 caches and a shared L2 cache.

Figs. 1 and 2 show snippets of the timelines for **ping-pong** and **spin**, respectively. The stack-level values are arbitrary, and are offset somewhat to separate the threads vertically. The boxes at the bottom of each figure show the key; a square on the timelines means the thread had an L1 dcache miss, and stars show long-latency instructions ("lli"). We also mark memory barrier (MB) instructions ("#mb") explicitly. (No L2 or L1 icache misses happened during these snippets.)

Fig. 1 (**ping-pong**) shows simple behavior; each thread experiences a series of L1 dcache misses while it polls the variable waiting for it to change.

Fig. 2 (**spin**) shows more complex behavior. The bottom timeline (core=0) shows a complete iteration of the inner loop. It is easy to see the lock and unlock operations (note that the **move** function takes and releases two locks, but only one of these is shared between the two threads pictured; the other lock is not necessary in the two-thread case). One flaw in our current visualization is that function names are placed exactly at the time that a call is made, which causes labels to overlap; a better label placement algorithm would dither these.

Fig. 2 also shows that each **pthreadSpinlock** and **pthreadSpinUnlock** issues an MB, some of which cause long-latency instructions in all threads, and an MB in **pthreadSpinUnlock** appears to cause multiple L1 dcache misses in **pthreadSpinLock**. This kind of effect would be impossible to see with other tools.

5 Experiments with kernel code

The experiments described above show timelines for artificially simple programs. We initially developed our approach to solve real problems we faced, during our work on Linux modifications supporting fast switching of threads between cores [11]. Because those modifications involved the kernel scheduler, they were especially difficult to debug.

Fine-grained visualization (the pictures are too complex to show in the available space) allowed us to solve a number of performance problems, including:

- **Increasing inter-core parallelism**: Our code on one core sometimes needed to wake up another core from a powered-down state. We reduced overall latency significantly by "prefetching" this wakeup; our visualization allowed us to see that this actually led to the desired parallelism.
- **Removing unnecessary code from a fast path**: Rather than removing all of the code that we *thought* might be unnecessary, risking the introduction of subtle bugs, we found a few functions that consumed a lot of time, and removed them first.
- **Replacing inefficient code**: Linux was using a very inefficient mechanism, which was obscured by deep layering of simple functions, to decide which core to send an interprocessor interrupt to. One layer created a bitmap with just one bit set; another layer then spent much time figuring out

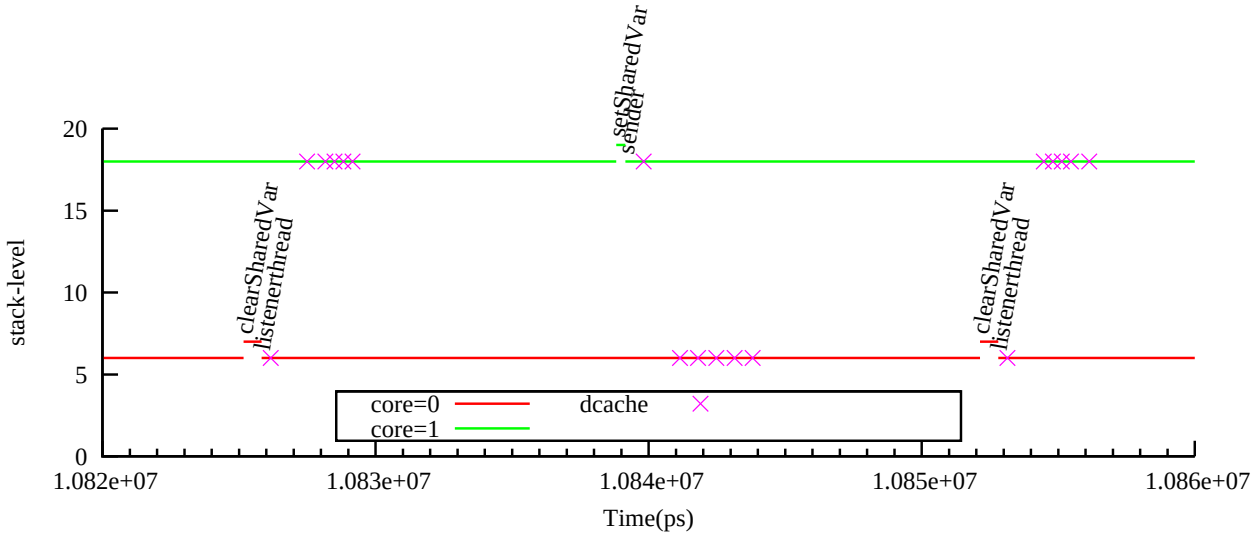

Figure 1: Partial timeline for cache-line ping-pong program

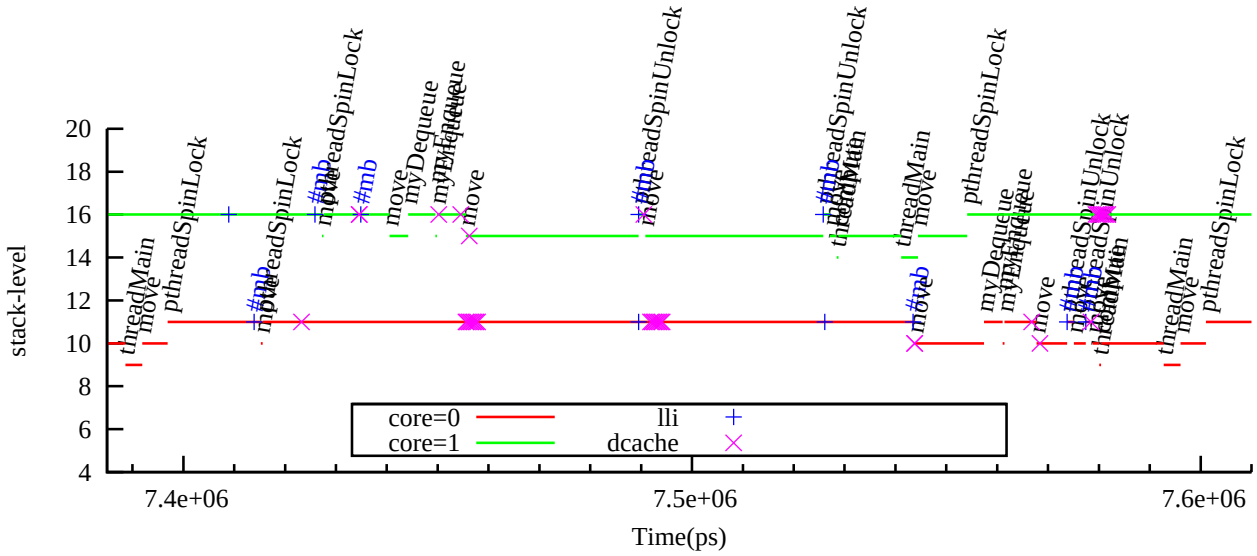

Figure 2: Partial timeline for spin lock program (Sorry: our automatic label placement is not very smart!)

which bits were set. Our visualization made this obvious.

In this work, we wondered whether some of our latency could be caused by excessive cache misses due to sharing of scheduler data structures between cores. Our visualizations revealed that this was not a problem, saving us the trouble of trying to experiment with different algorithms and data layouts. (As we make this code more "sophisticated," we may have to revisit this issue.)

6 Challenges and Drawbacks

While we have already found simulation-based visualization to be a powerful tool in performance de-

bugging for parallel software, we realize that the approach has its challenges and drawbacks.

Simulation overhead: Simulators are slow. Cycle-level simulators are very slow. For example, an M5 simulation of a two-core system running the Apache server, on a quad-core 2.8 GHz Xeon took 24660 elapsed seconds for 0.466 simulated seconds, a slowdown of about 53000:1. If one can restrict the fine-grained simulation to a specific inner loop, as we did for **ping-pong** and **spin**, simulations can finish much faster.

M5 simulates parallel systems, but currently is singled-threaded with one event queue. An upcoming version of M5 will exploit parallel hardware, with one event queue and thread per real core, using known techniques [10] to maintain causality. This should allow simulation speed to scale up on modern CPUs.

Our approach also generates huge trace files. For example, our fine-grain simulation of **spin** covering 157 simulated microseconds took 35.4 real seconds and generated a 8.2 MByte raw trace file and a 6.3 MByte XML file. (Also, our unoptimized Python script takes lots of CPU time to parse the raw trace files when generating XML.) Eliminating excess trace detail and using a better intermediate representation, such as DataSeries [1], could reduce trace size and processing time.

Simulation fidelity: Ultimately, programmers want to know how their software runs on real hardware. Our approach is useful only if the simulation is reasonably faithful. Simulators are complex beasts, and models for modern CPUs are also quite complex. While simulator developers take pains to validate their models against reality, there will always be gaps. Simulator developers don't model details they don't expect will be used, and so applications that encounter these corner cases will suffer inaccurate simulation.

These simulation gaps also mean that simulators are fragile, since someone is always trying to improve the simulator or its models. Our experience is that "simulation rot" sets in within a few months; experiments that used to work now require debugging. When simulators break, the average programmer can't debug them.

Simulator usability: Running applications on a simulator adds complexity. One has to work in a different execution environment, possibly requiring minor changes to program sources or makefiles. Simulators have lots of configuration options. And, of course, time dilation means that the user has to be very careful about what parts of the program run during cycle-level (as opposed to behavioral) simulation.

Simulators *could* be more usable (for example, VMware Player resembles a simulator in many ways, and it is fairly usable), but the market for simulators does not support the large programming effort to make them really usable.

Information overload: The biggest benefit of our approach, extremely fine-grained information about the interaction between threads and the hardware, is also a curse. Programmers trying to understand the performance of their code have to wade through an overwhelming amount of data. It helps to have both a good search strategy and some inspired guesses about where to look for problems; it also helps to have a good eye for red herrings in the data.

A trace browser clearly needs to provide powerful, efficient tools for searching through the trace, probably including some way to look for patterns of the form "L2 cache miss during function X" or "spin-lock acquisition delay of more than 1000 cycles during function Y."

Finally, we have tried out our approach only on systems with just a few cores (parallel threads). We expect it will be difficult to extend this kind of visualization to applications with dozens of threads or more.

7 Summary

We have found simulation-based visualization to provide unique and valuable views of the fine-grained behavior of parallel software, even in our current crude implementation. We expect it can be vastly improved.

References

[1] E. Anderson, M. Arlitt, B. Morrey, and A. Veitch. Dataseries: an efficient, flexible data format for structured serial data. *Operating Systems Review*, 43(2), 2009. To appear.

[2] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: where have all the cycles gone? *ACM Trans. Comput. Syst.*, 15(4):357–390, 1997.

[3] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 Simulator: Modeling Networked Systems. *IEEE Micro*, 26(4):52–60, 2006.

[4] R. Bosch, C. Stolte, D. Tang, J. Gerth, M. Rosenblum, and P. Hanrahan. Rivet: A Flexible Environment for Computer Systems Visualization. *Computer Graphics*, 34:68–73, 2000.

[5] M. T. Heath and J. A. Etheridge. Visualizing the Performance of Parallel Programs. *IEEE Softw.*, 8(5):29–39, 1991.

[6] M. T. Heath, A. D. Malony, and D. T. Rover. Parallel Performance Visualization: From Practice to Theory. *IEEE Parallel Distrib. Technol.*, 3(4):44–60, 1995.

[7] K. L. Karavanic, J. Myllymaki, M. Livny, and B. P. Miller. Integrated visualization of parallel program performance data. *Parallel Comput.*, 23(1-2):181–198, 1997.

[8] B. P. Miller. What to draw? When to draw?: An Essay on Parallel Program Visualization. *J. Parallel Distrib. Comput.*, 18(2):265–269, 1993.

[9] W. E. Nagel and A. Arnold. Performance Visualization of Parallel Programs - The PARvis Environment, 1994.

[10] S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. In *Proc. SIGMETRICS*, pages 48–60, May 1993.

[11] R. Strong, J. Mudigonda, J. C. Mogul, N. Binkert, and D. Tullsen. Fast Switching of Threads Between Cores. Under review.

[12] Q. A. Zhao and J. T. Stasko. Visualizing the Execution of Threads-based Parallel Programs. Tech. Rep. GIT-GVU-95-01, Georgia Inst. of Tech. Graphics, Visualization and Usability Center, Jan. 1995.