Examples from Integrating Systems Research into Undergraduate Curriculum

John H. Kelm and Steven S. Lumetta
University of Illinois at Urbana-Champaign
{jkelm2, lumetta}@illinois.edu

Abstract

In this paper we motivate and discuss the use of examples drawn from computer systems research for use in the classroom. We describe three case studies used in an advanced undergraduate course covering large software system design. The case studies document situations we have encountered while designing and implementing performance modeling infrastructure and benchmark applications for use in our research on parallel processor design. Two of the case studies cover debugging techniques and are available publicly. The third case study covers performance analysis using freely available tools. The goals of this work are to illustrate how classroom concepts are realized in computer systems, to provide examples of how performance analysis and tuning can be applied to complex real-world applications such as our C++ architectural simulator, to motivate the use of profiling tools in instruction, and to expose students to research topics and methodology.

1 Introduction

This paper contains the summary and discussion of three case studies that are intended for use in advanced undergraduate instruction. The case studies describe debugging and optimization experiences from RigelSim, a C++ simulator for the Rigel architecture [11], and its corresponding runtime system. Two of the case studies involved removing correctness bugs from RigelSim. The third case study discusses the application of performance analysis and optimization techniques to our simulator infrastructure. These case studies were used in a senior undergraduate-level software systems class and serve as models that other instructors could adopt. The goal of using case studies is to highlight the difficulty and subtlety involved in addressing correctness and performance bugs in large systems that are otherwise difficult for students to see firsthand in class projects.

The first case study documents the experience of removing a correctness bug in RigelSim that was hard to expose and required long running simulations to activate. The case study highlights the need for innovative and methodical approaches to debugging large computer systems. The case study discusses the need for regression testing and self-checking mechanisms when working with evolving software systems that have multiple contributors. We discuss the utility of determinism and robustness in the debugging process for large-scale applications. The goal of the study is to introduce students to one component of a large software system, describe a software error, and show them how one would go about removing that error. Using an example from our research allows us to provide a more in-depth perspective and exposes students to the tools that researchers in the area of computer architecture frequently use. An extended version of the case study is available online [9].

The second case study describes the process of isolating and removing a livelock from the runtime system used within RigelSim. The case study highlights three topics relevant to students. It describes the nature of a common, but difficult to diagnose condition in parallel systems. Secondly, we motivate methodical and structured approaches to software system performance analysis. Lastly, the case study illustrates how latent performance bugs can remain undetected for long periods of time while sapping performance unbeknownst to the developers. A greater emphasis is being placed on developing parallel software. However, there is a lack of widespread experience with such systems making case studies such as this a valuable resource for computer engineering students. An extended version of the case study is available online [10].

The third case study discusses a number of experiences using sample-based profiling of our simulator tools to diagnose performance pathologies in our code. We also discuss how students can benefit from these experiences and how other instructors could develop similar examples. We apply widely deployed and freely available tools to our simulator infrastructure, and in doing so, bring computer architecture resources into the classroom while providing students with tools they can apply more broadly.

The rest of the paper is as follows: Section 2 provides an overview of the course where these materials were first used. Section 3 summarizes the debugging case studies. Section 4 discusses the performance analysis experiences. Section 5 provides discussion of the use of research experiences in the classroom. Section 6 concludes the paper.

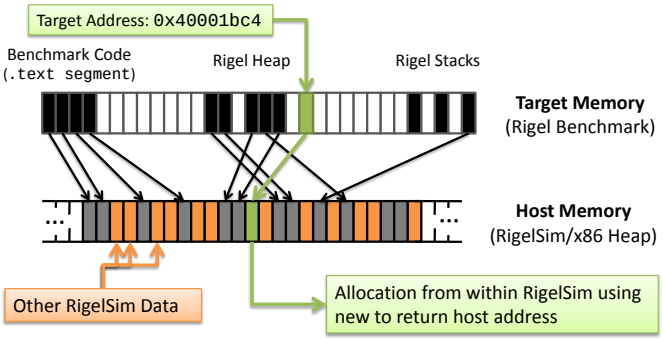

Figure 1. Target-to-host memory mapping for RigelSim.

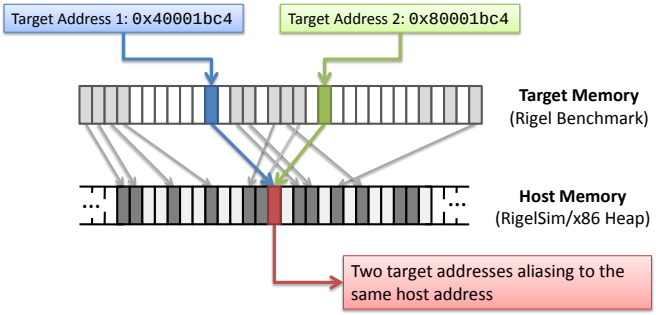

Figure 3. Target-to-host aliasing that caused the observed target memory corruption.

2 Course and Research Overview

The materials presented in this paper were used in an elective course offered in Spring 2009 at the University of Illinois. The course focused on large software system design. The course targeted advanced undergraduates and graduate students. Professor Steven S. Lumetta designed and instructed the course.

The goal of the course was to provide students with an understanding of the relationship between application software, compilers, runtimes, and computer architecture. The first half of the course focused on abstraction used in modern programming languages. The course used C++ as an example language and used Stroustrup [17] as a text. The second half of the course covered parallel runtimes, common parallel idioms, and the interplay between parallel software development and parallel architectures. Interwoven with the two major thrusts of the class were perspectives on debugging and performance analysis—the two topics considered in this paper.

Additional materials, including lecture notes, laboratory assignments, and supplemental material, are available on the course website [14].

3 Correctness Case Studies

In this section we provide an overview of two case studies used in the class. The first study discusses the isolation and removal of a software error from an architectural simulator used in our research. The second study concerns a livelock condition found in the simulated parallel runtime for our design. Both case studies are available online [9, 10].

3.1 Memory Model Bug

Motivation The motivation for this case study is to give students a perspective on debugging large-scale software systems. Developing tools and techniques to remove software errors from large systems requires ingenuity and experience. Many students learn the technique of debugging from class projects. Bottom-up approaches to introductory computer systems instruction [4, 15] exposes students to the design and implementation of computer systems. However, class projects rarely exceed a semester in length, thus limiting the size of the system with which students interact. Furthermore, when a large software system is used, such as the Linux kernel, debugging tools and vetted infrastructure already exists to aid in the isolation of software errors. The existence of debugging infrastructure and methodologies lessens the need for holistic and innovative approaches to debugging, thus leaving students ill prepared for debugging large computer systems that lack widely accepted tools and practices.

To help bridge the gap between the classroom and real world large system design, we use case studies based on our own experience developing the simulation infrastructure for the Rigel architecture. Using software errors found in the development of our research infrastructure as examples, we demonstrate how bugs can cross abstraction level boundaries from the application down to the microarchitecture and how to isolate bugs in such an environment.

Description and Debugging Process Throughout this paper, *host* refers to the x86 workstations that execute instances of RigelSim, while *target* refers to the simulated Rigel system. The case study concerns a bug in RigelSim that caused two addresses in the target address space to map to the same host address causing intermittent pointer corruption. The component responsible for the error was the memory model for the simulator.

The bug discussed in the case study was found during a nightly batch run of simulations. Of the 200+ jobs that were run, only four failed. Furthermore, the four failures occurred only after many hours of simulation. Due to the long time to activation, we were constrained by the rate at which we could make a change to the system and observe whether the bug was corrected. The case study discusses the use of information gathering techniques and testing philosophy that were employed to keep the test process tractable.

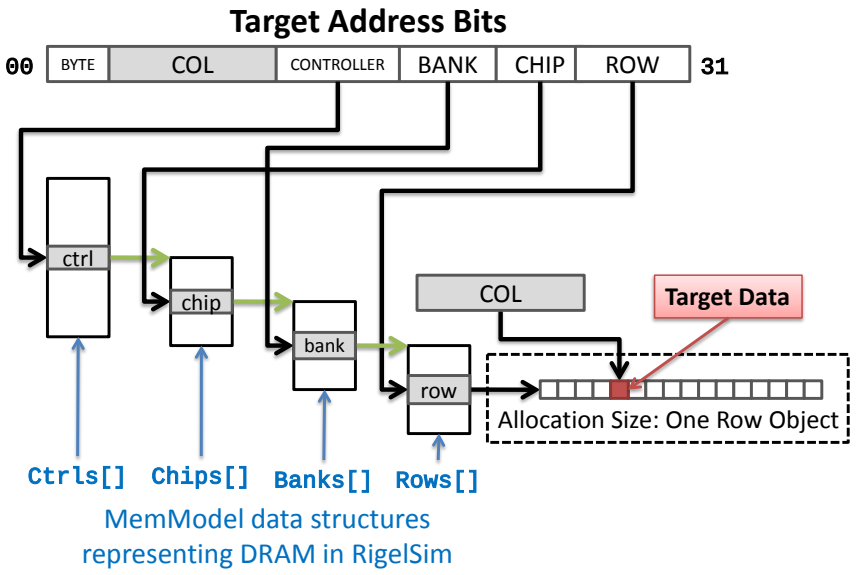

Figure 2. Structure of memory model in the Rigel Simulator.

Figure 1 depicts the memory map of RigelSim. The simulator is designed to efficiently use host memory by only allocating blocks of memory as needed. Each block in the figure represents a 2 KB allocation from within the simulator. Figure 2 shows how the RigelSim memory model hashes target addresses to the 2 KB host allocations show in Figure 1. The memory model mirrors the multi-level tree structure of the DRAM used in the simulator.

There are multiple hashes performed to generate the $address \rightarrow \{ctrl, chip, bank, row, col\}$ mapping shown in the figure. The mapping is intended to uniformly distribute random accesses to target memory while exploiting row-level locality for bursts of contiguous accesses. The number of bits of address needed for each hash is dependent on the five parameters and the number of cores, all of which are either command line options or statically defined constants. The multi-dimensional mapping and variability of the relevant parameters makes developing a robust hash function difficult. Our initial mapping failed for a particular configuration and was only exposed for a single benchmark. Even then, the latent bug did not become programatically visible for several hours. Figure 3 illustrates the bug.

Learning Objectives The aspects of the bug that make it relevant as a case study include the time needed to expose the bug and the number of possible components the bug could have touched. The case study focuses on isolation, removal, and regression testing to provide a perspective on the full debugging cycle. The location of the bug being the simulated memory system also allows the case study to review challenges in an important area of computer architecture research.

The rest of the case study [9] describes the techniques used to isolate and remove the bug. Other aspects of the debugging process covered include the use of in-line checks and assertions to aid in debugging. The case study discusses the need for intelligent debug output to make debugging long-running executions tractable. Self-checking techniques and their use in regression testing are discussed. Lastly, we discuss the importance of robustness and determinism in avoiding software errors when possible and isolating those errors when they occur.

3.2 Runtime System Livelock

Motivation The second case study involves a livelock condition in the runtime system simulated within RigelSim. The nature and causes of livelocks and methods to debug and correct them are discussed. The case study discusses the use of fair and unfair locks in the implementation of a piece of parallel software. The benefits of the case study for students are the description of a common, yet hard to debug class of parallel software errors and an introduction to parallel constructs that enforce fairness. The link between parallel software debugging and architecture is motivated by the increased use of multicore processors. Our research infrastructure serves as the environment for the case study, providing another example of how computer architecture research can be brought into the classroom even in non-architecture contexts.

In parallel systems, fairness is a concern when two or more components access the same resource and a total ordering on those accesses is enforced. In some situations fairness is imposed explicitly, such as in the case of a FIFO queue to order requests to a memory. In other cases, no fair-

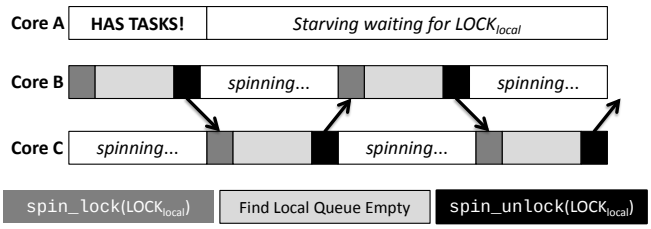

Figure 4. Timeline of cores participating in the livelock. Note that core **A** can never enqueue tasks because it is continually starved for $lock_{local}$ by **B** and **C**.

ness guarantees are given, such as the use of a simple spin lock for protecting a critical section in a parallel application. In the initial implementation of the runtime presented in the case study, the lack of fairness led to two threads starving a third, resulting in livelock. The case study examines how such a situation can occur and how to correct it by enforcing fairness explicitly.

Description and Debugging Process The runtime used in the case study is a multi-level hierarchical work queue structure. Tasks are inserted in the top level queue and are removed from the low level queues. Here we will assume two levels of queue, a local and a global level. When a local queue runs out of tasks, the core attempting to dequeue requests more tasks from the global queue. All queue management is performed in software by the runtime using atomic load-linked and store-conditional primitives provided by the architecture.

Figure 4 shows the timeline of events that leads to livelock when unfair locks are used. There are two locks involved. One lock must be held to access the local queue. The other lock must be held to access the global queue when the local queue runs out of work. To allow simultaneous enqueue and dequeue operations the local lock, $lock_{local}$, is dropped while attempting to access the global queue. The livelock occurs when two cores, **B** and **C** in the diagram, continually attempt to obtain tasks from an empty local queue and thus starve **A** that is trying to obtain the $lock_{local}$ so that it can insert more tasks.

Learning Objectives The value of the case study for students is that they can see how a transient parallel software error can be removed from a large system under simulation. The case study also discusses locking mechanisms and the tradeoffs inherent to fair versus unfair mechanisms. Another valuable insight is that not all software errors result in crashes or deadlocks that hang. Some software errors, such as livelocks, can reduce the system's performance unbeknownst to the developer resulting in disappointing performance and misplaced optimization efforts.

4 Performance Analysis Case Study

In large-scale hardware and software systems correctness is often the primary concern for the developer. However, for commercial applications and high-performance computing systems performance is a critical concern for competitive, economic, and tractability reasons. Introductory programming and software engineering classes stress *correctness* while more advanced computer science courses focus on algorithmic *complexity*. However, *optimization* is introduced to students late in their careers or not at all. Therefore, a disparity exists between the set of skills students have and the requirements of potential employers.

As multicore processors have become prevalent, parallel programming has been cited as a way to achieve higher performance for parallelizable applications [7]. While teaching parallel programming is one approach to training student to develop faster code, there is still substantial performance to be gained from sequential optimizations, which apply to both sequential and parallel applications alike. As an example, one case study shows orders of magnitude speedup for dense matrix multiply by applying algorithmic and sequential optimizations [2]. Moreover, Moore's Law alone is unlikely to reduce single-threaded runtime, thus placing more emphasis on sequential performance tuning [12] as a means to increase performance for sequential applications. In light of this, we present experiences optimizing a large sequential application, a C++ based simulator we use in our research, and discuss the use of these examples in undergraduate instruction.

4.1 Sample-based Profiling

A naïve approach to achieving higher performance is to develop many variants of an application and benchmark each to select the optimal design. Optimization by benchmarking alone is a time consuming process and achieves suboptimal results. Moreover, the N-version programming approach lacks directed feedback mechanisms to isolate where performance is being lost and masks performance degradation located in code and libraries common across benchmark versions.

A more methodical approach is to use sample-based profiling tools to target pathologies in large software systems. Sample-based profiling allows the developer to isolate performance problems and perform targeted optimizations. This section describes how we used performance regressions from our architectural simulator. We use sample-based profiling to diagnose such regressions and to perform targeted optimizations.

Many profiling, instrumentation, and analysis tools are freely available. Examples include the GNU profiler [8] (`gprof`), Oprofile [1], and Intel's PIN [13]. Each of these

tools was used at some point in the class; however this section focuses on our use of Oprofile. Oprofile is a suite of tools available on Linux systems that utilize the hardware performance counters on x86 hardware to perform low-overhead sample-based profiling.

4.2 Strength Reduction

Motivation Optimizing compilers have evolved to a point where developers now rely almost exclusively on automation to perform code generation. In most cases, hand-optimized assembly provides marginal performance gains and large productivity and portability losses compared to a state-of-the-art compiler. Many computer science programs espouse this viewpoint and teach students high-level languages. However, in doing so students may be unaware of the connection between statements in high-level languages and the instructions that they generate and memory allocation patterns [5, 6].

One approach to demonstrating the performance characteristics of high-level constructs in low-level or high-level languages is through performance analysis. As an example, a compiler for a high-level language, such as the GNU C++ compiler, can fail to optimize obvious cases. However, these cases may contribute little to overall runtime compared to caching effects and algorithm choices and thus not result in observable performance degradation. However, if such a case falls into a common code path, performance can suffer greatly. The performance analysis applied to our simulator provides a concrete illustration.

Description and Debugging Process Oprofile provides a utility to annotate source code with relative frequency of execution for each line. While analyzing the annotated source for RigelSim, we found that in many places 1-2% of runtime was spent doing integer divides. The sum of these overheads resulted in 5-8% slowdown across our benchmarks. Note that integer divide latencies on modern microprocessors can be as as high as 79 cycles [3].

In RigelSim, many common operations such as address hash functions involve integer multiplication, division, and modulus operations with operands known at compile-time to be powers of two. This enables a well-known optimization called strength reduction whereby expensive operations can be converted statically to logical shifts and bitwise masks, thus saving dozens of cycles of latency. The compiler was not performing this optimization. However, we were able to remove most of the overhead by performing the strength reduction at the C++ source level.

Learning Objectives The compiler example illustrates four points that are valuable for students. The first is that while compilers are quite good at generating high-quality code, they are not infallible and choices at the source level can impact code generation in measurable ways. The second is that benchmarking alone cannot easily detect all performance pathologies. In this case we did not even realize that there was a performance issue until we looked at the annotated source code. The example also illustrates that methodical approaches to performance debugging can lead the developer, working at the source level, to the underlying cause of a performance pathology, which happen to be at the instruction level in this example. Lastly, a naïve approach may have been to remove all modulus and divide operations. Doing so would have reduced code readability, possibly introduced bugs, and would have been unnecessary in almost all cases since most static divide instructions are executed few times dynamically in RigelSim.

4.3 Cache Blowout

Motivation The previous example used the number of committed instructions and halted clock cycles to determine when to take samples. While this works well in most cases, some performance pathologies are not localized and are not easily detected using instruction frequency-based sampling. One example from our simulator was the use of structures on the host side that track each miss status handling register (MSHR) used in the unified L2 cache inside the target.

Description and Debugging Process There are 128 target L2 caches in a full RigelSim simulation and 8–32 MSHRs associated with each L2. In the initial implementation, in every target cycle, each MSHR had its valid bit checked. The MSHRs are tracked as an array of objects at each L2 and thus use an array-of-structures (AoS) data layout. While AoS achieves good locality when many fields within a single record are accessed in succession, AoS provides little locality across a single field in multiple records. In this example, the valid flag, represented as a single bit in memory, requires that a full 64-byte cache line be pulled into the host data cache for each access. The other 511 bits of the line are of no use if the MSHR is invalid, which is the common case. So in each simulated target cycle $128 \times 32 \times 64b = 256$ *KB* of data are brought into the data cache to find a ready MSHR, blowing out both the host L1 and L2 data caches.

Sample-based profiling of host data cache misses showed there to be an abundance of misses whenever valid bits in MSHRs would be accessed. As a solution, we added facilities to track all valid bits for a cache in a single bit-vector structure. All of the valid bits could then be accessed without bringing large amounts of unnecessary data into the host's cache.

Learning Objectives This example illustrates that performance pathologies can be systemic and simple performance models, such as those based only on instruction count, fail to capture the behavior of large systems with caches. The example also illustrates a use of sample-based profiling beyond just committed instructions. A similar approach could be used for branch mispredictions and instruction cache misses to better isolate performance issues across module boundaries. The example shows that caching effects are a real problem for large software systems, but that with proper analysis and simple code changes, such as the SoA/AoS conversion performed here, some pathological cache behavior can be avoided.

4.4 STL Pitfalls

Motivation Software systems developers face a tradeoff between performance and programmer productivity, code readability, and maintainability. The C++ standard template library [16] can provide productivity gains by not forcing developers to re-implement common data structures and algorithms repeatedly. However, naïve use of STL, and libraries with opaque interfaces in general, can result in degraded performance. In this section we show how a misuse of the STL `map` container in RigelSim led to a performance degradation of over 60%. We then discuss how we were able to use sample-based profiling to isolate and remove the performance regression.

Description and Debugging Process During the development of RigelSim, the target statistics collection code for RigelSim was replaced. The old model relied upon a struct of counters that were incremented directly, making it difficult to easily print and gate statistics generation at runtime. The new model would use text-based strings to identify counters by name and could be instantiated automatically in the simulator. The implementation relied upon an STL map that used strings as keys and kept 64-bit integers as values. We found that not long after we added the new profiling facilities, simulator runtime more than doubled.

STL is used extensively for some of the more complex analysis we perform and had never been a performance concern. We analyzed the annotated output produced by Oprofile and found that the majority of execution time was attributable to internal methods of the STL map implementation and string constructors. To achieve better resolution, we used Oprofile to obtain a call graph of the execution showing cumulative runtime at each method invocation. Here it became clear that RigelSim was spending half its execution doing string compares to traverse the red-black tree data structure used by the STL map implementation.

Learning Objectives While STL can save programmers a good deal of effort, this example points out the importance of understanding the overhead of using a library and, if it is costly, how often it will be used. The solution in our case involved using an array of structs with statically constant identifiers, implemented using an enumerated type mapping counter names to array indices. The new implementation avoided the need for text-based compares and thus removed the overhead. The trade off was additional programmer effort in developing the statistics collection system and added time to add new counters. However, the $2\times$ slowdown of the initial implementation makes the slightly more inconvenient mechanism a better trade off in RigelSim.

The lesson demonstrated here is that while libraries and container classes such as STL can provide gain in productivity and a reduction in bugs, their use does not come without cost. The proper use of performance analysis tools, such as the performance counter annotated call graph and source code tools provided by Oprofile, are invaluable in isolating performance regressions.

4.5 Summary

We have shown how sample-based profiling can be introduced to senior undergraduates. We use a case study approach, relying upon examples from our own research and experiences applying freely-available analysis tools to our own simulator infrastructure. The examples can help students to better understand software performance, while also building a better understanding of the link between software performance and the underlying architecture.

5 Discussion

In the paper we have shown how computer systems infrastructure can be used in the classroom through examples. In this section we discuss the value of using case studies to bring computer systems research into an instructional setting. We also discuss two of the high-level points we illustrate in the paper. One point is the tension between different solutions and the second is the proper use of abstraction. We conclude by motivating the use of real world examples from computer systems research to connect theoretical concepts with practical systems.

Tools such as compilers, operating systems, and simulators represent large scale applications that the instructor, teaching assistants, and research assistants working research projects are intimately familiar with. However, while a graduate student or professor focusing on computer architecture may be familiar with a wide variety of large software systems that have code freely available, such as operating systems and compilers, seldom do they spend as much time

developing code for those systems as they do for simulators and related tools.

Using a simulator as an example application increases students' exposure to computer architecture research topics and methodology. Increased exposure can motivate students to investigate advanced courses or careers in the area of computer architecture. The students in the class where these case studies were used were undergraduates pursuing degrees in computer engineering. A variety of areas of interest within computer engineering were represented. Seeing the tools computer architects use for their research and the methods used to debug and optimize those tools may entice students to consider computer architecture in their choice of graduate school and in their job search.

When bugs manifest themselves in a large system, there are trade offs between reimplementation and quick fixes. The trade offs involve performance, programmer effort, and the probability of inserting or exposing new bugs with a proposed solution. In our case studies we show that in some cases targeted fixes were the proper solution. Examples include the strength reduction performance regression example and the memory aliasing bug. In other cases, we showed that structural changes were necessary, such as in the livelock example where we had to reimplement locking mechanisms to ensure forward progress.

Another trade off is the frequency versus cost in verification and validation techniques. It is important to understand the cost of verification and at what level to apply verification techniques to achieve high performance while having high confidence in the results and minimal occurrence of bugs. As an example, future memory aliasing bugs can be regression tested with a simple checker, but the simple solution also requires too much time to be run with every simulation. Instead, longer tests such as these are run at predefined intervals such as when code is committed to our source repository or in nightly regression tests.

We demonstrate that while abstraction can provide tangible benefits, it can also mask performance and correctness problems; One example being the use of STL for performance counters in RigelSim. While the abstraction provided by the STL map led to an easy solution, it created a performance regression. The use of proper analysis tools, such as Oprofile, can greatly reduce the difficulty in diagnosing such performance regressions. It also has the pedagogical benefit of making otherwise opaque abstractions transparent. Transparency during instruction increases the students' understanding of the underlying implementation of an interface, such as the STL container classes used in this example.

Lastly, we find the use of real world examples of performance and correctness issues valuable for students. Computer science courses often teach the theoretical underpinnings of pathological conditions such as livelock. However, it may be difficult for students to make the connection between dining philosophers and threads of computation racing for a lock, thus failing to make forward progress. Furthermore, a theoretical understanding of computer systems, such as the asymptotic complexity of our STL container classes, may not always be sufficient for understanding performance implications in real systems. Case studies have the advantage of making fundamental issues in computer science tangible for students thus strengthening the connection between theory and practice.

6 Conclusion

Although few courses in a typical curriculum focus on computer architecture, the observations made while developing large scale software and hardware systems while pursuing research in computer architecture can still be adopted for a wide range of classes. We believe that computer architecture research and the process it entails can provide useful and relevant material for use in the classroom. As we show, one way architecture research can be brought into the classroom is by example using case studies.

This paper explores the use of case studies in debugging and performance analysis. The examples in this work are derived from our experience developing, debugging, and tuning our research infrastructure. We find that having intimate knowledge of the system used in classroom discussion can greatly aid in instruction. The use of computer systems infrastructure exposes students to a large software system and illuminates problems that are unlikely to be found in class projects due to constraints on time and scope.

The use of case studies can provide students with a portal into the world of computer architecture research and large-scale computer system design in general. Using real world examples gives credibility to the presentation of the case studies. Lastly, we find that the impact of computer architecture on the classroom need not stop at designing microprocessors. Instead, we can use the process of computer system design as a vehicle for educating a wider audience of students.

Acknowledgment

The authors acknowledge the support of the Focus Center for Circuit & System Solutions (C2S2 and GSRC), two of the five research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation Program. The authors thank the Trusted ILLIAC Center at the Information Trust Institute for their contribution of use of their computing cluster. The authors also wish to thankMatt R. Johnson and the anonymous referees for their input and feedback. John Kelm was partially supported by a fellowship from ATI/AMD.

References

[1] Oprofile. http://oprofile.sourceforge.net.

[2] S. P. Amarasinghe. Performance engineering of software systems, lecture 1, 2008. Available Online: http://stellar.mit.edu/S/course/6/fa08/6.197/.

[3] AMD Staff. Software optimization guide for AMD family 10h processors, May 2009. Revision 3.11.

[4] R. E. Bryant and D. R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Prentice Hall, 2003. Website: http://csapp.cs.cmu.edu/.

[5] R. Dewar and O. Astrachan. Point/counterpoint cs education in the u.s.: heading in the wrong direction? *Commun. ACM*, 52(7):41–45, 2009.

[6] R. B. K. Dewar and E. Schonberg. Computer science education: Where are the software engineers of tomorrow? *CrossTalk: Journal of Defense Software Engineering*, January 2009.

[7] A. Ghuloum. Viewpoint face the inevitable, embrace parallelism. *Commun. ACM*, 52(9):36–38, 2009.

[8] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. In *SIGPLAN '82: Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pages 120–126, New York, NY, USA, 1982. ACM.

[9] J. H. Kelm. Anatomy of a bug, May 2009. https://netfiles.uiuc.edu/jkelm2/www/kelm-rigelsim-bug.pdf.

[10] J. H. Kelm. Case study: Rigel task model livelock, May 2009. Available at: https://netfiles.uiuc.edu/jkelm2/www/kelm-rtm-livelock.pdf.

[11] J. H. Kelm, D. R. Johnson, M. R. Johnson, N. C. Crago, W. Tuohy, A. Mahesri, S. S. Lumetta, M. I. Frank, and S. J. Patel. Rigel: An architecture and scalable programming interface for a 1000-core accelerator. In *Proceedings of the International Symposium on Computer Architecture*, June 2009.

[12] J. Larus. Spending Moore's dividend. *Commun. ACM*, 52(5):62–69, 2009.

[13] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM.

[14] S. S. Lumetta. ECE498SL Spring 2009 homepage, May 2009. http://courses.ece.illinois.edu/ECE498/SL/.

[15] Y. N. Patt and S. J. Patel. *Introduction to Computing Systems: From Bits and Gates to C and Beyond*. McGraw-Hill, 2003. Class Website at the University of Illinois: http://courses.ece.illinois.edu/ECE190/.

[16] A. Stepanov and M. Lee. The standard template library. Technical Report X3J16/94-0095, HP Laboratories, November 1995.

[17] B. Stroustrup. *The design and evolution of C++*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1994.