PIN: A Binary Instrumentation Tool for Computer Architecture Research and Education

Vijay Janapa Reddi, Alex Settle, and Daniel A. Connors
University of Colorado, Boulder.
{vijay.janapareddi, settle, dconnors}@colorado.edu

Robert S. Cohn
Intel Corporation
robert.s.cohn@intel.com

Abstract

Computer architecture embraces a tremendous number of ever-changing inter-connected concepts and information, yet computer architecture education is very often static, seemingly motionless. Computer architecture is commonly taught using simple piecewise methods of explaining how the hardware performs a given task, rather than characterizing the interaction of software and hardware. Visualization tools allow students to interactively explore basic concepts in computer architecture but are limited in their ability to engage students in research and design concepts. Likewise as the development of simulation models such as caches, branch predictors, and pipelines aid student understanding of architecture components, such models have limitations in the workloads that can be examined because of issues with execution time and environment. Overall, to effectively understand modern architectures, it is simply essential to experiment the characteristics of real application workloads. Likewise, understanding program behavior is necessary to effective programming, comprehension of architecture bottlenecks, and hardware design. Computer architecture education must include experience in analyzing program behavior and workload characteristics using effective tools. To explore workload characteristic analysis in computer architecture design, we propose using *PIN*, a binary instrumentation tool for computer architecture research and education projects.

1 Introduction

New applications and programming models are constantly emerging to complement new and improving hardware technology and paradigms. It is becoming essential to understand the workload characteristics of applications in order to design effective architectures. Often, in order to understand program behavior on a specific processor, students must have a significant amount of knowledge of the underlying hardware and the control and data flow of the application. For instance, modern performance is a confluence of many components working together - branch predictors, caches, pipelines, etc. Even with a deeply rooted understanding of the architecture, it is often extremely difficult to comprehend the flow and resource usage of the program because of the immense amount of data that needs to be collected and analyzed in order to study such behavior.

There are various tools available for computer architecture education. These tools can be divided into several categories, architecture visualization systems, simulation environments, and hardware event monitoring programs. Each of these categories play a role in bridging the divide between pedantic methods of illustrating computer architecture and real-world dynamic examination of architecture concepts. There are several areas of knowledge and skills that these tools address. For instance, architecture simulators provide insight into microarchitecture design and the behavior of the individual hardware components. These simulators are usually designed so that students can integrate additional emulated hardware components into the overall simulation system and analyze the impact of design parameters on the simulated processor. Likewise, performance monitoring support allows students to realize the performance impact of the different architecture components and compiler optimizations have on the overall system. Most importantly, monitoring systems provide accurate feedback on real workload applications.

Simulators and performance monitoring systems may not be sufficient for computer architecture education because they do not allow large amounts of information to be collected on a per-instance level at the instruction granularity. Likewise, profiling techniques, such as *gprof* and *pixie*, only provide coarse-grain profiling information and are not suitable for detailed computer architecture concept exploration. Rather it is necessary for students to attribute profile information to the instruction level of the program. Thus, tools that provide infrastructure for performing data analysis of both software and hardware events can be extremely valuable to the computer architecture fundamentals of performance analysis, design, and architecture validation. Overall, such integrated computer architecture examination results in a deeper and more detailed comprehension of the collected data.

In this paper, we present PIN, a binary instrumentation tool that can be used as a teaching aid by in-

structors in computer organization education to facilitate the study of real-world workloads and their impact on the design of architectures. In addition to discussing the benefits of using PIN in the area of education, this paper includes a complete description of PIN's extended profiling features and methods of operation. Overall, we demonstrate that PIN can be used to replace the time and complexity of using trace files and software simulators to explore computer architecture concepts and design.

In the following section, we present an overview of tools that have been used in computer architecture courses, followed by the motivation of using PIN in computer architecture education in Section 3. In Section 4 we present some projects to illustrate how PIN can be used to teach. Following that we present our conclusion.

2 Background in Computer Architecture Education Tools and Methods

Although various tools are integrated into computer architecture curriculum, we believe that the best systems for computer architecture education should primarily involve:

- Simulated Design Environments
- Performance Analysis
- Workload Characterization

We elaborate on each of the above points and present arguments for and against them as we percieve fit in the classroom environment.

Simulated Design Environments: Instructors often place much emphasis on designing architectural models in the classroom. Simulators are used as a means of getting the students to explore design issues or are even as models to deepen the students understanding of architecture concepts.

Design simulators such as the Liberty Simulation Environment [7] allow students to design architectural components that interact within simulation modeling framework. Such simulators allow them to gain a deep understanding of how various components interact with one another and could potentially create a bottleneck. Others such as MipsIt[2], DLXide[12] etc. are used in the classroom because they help students better understand the theoretical concepts taught and open up research ideas. There exist numerous other simulators that could be used in the classroom, [5] has a lengthy listing of such tools which could very effectively be integrated into the coursework. Building a simulation environment allows students to comprehend the details of real hardware and provides greater understanding of the subtlties which give rise to complex designs. However, when designing simulators, significant time is spent developing software rather than doing the actual performance analysis on their simulator. Simulation environments often abstract the concept of real workloads on real hardware too much. Some students have difficulty relating the effect of the simulation environment to the actual program and hardware.

The simulators students develop are often trace driven which are severly limiting as we elaborate here. The trace files used for simulation can be immensely large, several megabytes per benchmark, and are therefore difficult to distribute as inputs for programs. On average, for ASCII trace file formats with simple execution information (program counter, opcode, 1 field of behavior), for every 2 million instructions there is overhead of about 80MBytes of trace file. Burtscher[11] reports that even with specifically tailored trace-compression algorithms applied to the execution trace, real workloads will exceed several gigabytes of space.

Trace files already reflect control flow of the program which was used to generate the traces. Thus, it voids the students an opportunity to investigate changes to both the application (compilation, rewriting algorithms) and the underlying architecture. Another issue with trace-based simulators is the problem of simulation time, generally such simulators take days to model the entire run of a real program. Our own evaluation of trace-based efficiency has determined that trace-reading (parsing and I/O) consumes nearly 40% of a simple cache simulator program.

Performance Analysis: Hardware event monitors are currently being extensively invested into by chip developers. The Itanium Processor Family (IPF), IA-32, POWER, and Alpha systems provide various event monitors for use. However only a few combinations may be active at any given time; the combinations are often limiting in the events that can be simultaneously monitored. Nevertheless, providing hardware counters has facilitated the development of interesting tools. Tools such as Perfmon[6] and PAPI[10] access the underlying event counters to develop applicaton profiles and other interesting reports that are valuable in performing program analysis.

In the classroom, awareness of hardware event monitoring tools is essential because they are actively being used in the industry to study the performance of programs on the underlying hardware. As mentioned previously, current architectures currently come heavily armed with an arsenal of hardware counters.

Interesting works such as vertical profiling[14], dynamic optimizations and code caches [8] take advantage of event monitors to perform performance critical analysis on programs to help boost the applications performance.

The primary limitation of simply showing absolute counter values to students would not prove effective as a method of teaching because the students lack an

insight on how the various performance counters could be co-related. A very basic example is to realize that using a counter to realize the total instruction count for a program divided by the total execution time of the program in cycles results in getting the average cycles per instruction.

Hardware event monitors are not flexible in that they come as part of the hardware and are not easily customizable to suit the users needs. The event monitors usually perform sampling, they do not guarantee sequence of execution as outputs. Their inflexbility limits their use in the classroom.

Workload Characterization: The arguments presented thus far are to bring forth the realization that minimal emphasis if any has been placed on presenting students with real workloads in the classroom, an essential component to the coursework. Simulators and hardware event monitors though valuable are limited in their contributions to the class environment. The solution to their limitations is using tools that facilitate a means of observing real program behavior on real hardware. Tools that facilitate binary instrumentation - PIN[13], Dyninst[3], Atom[1], etc.

The use of such tools requires no compiliation of the source, any binary application can be directly used as input into the program. This gives the flexibility of being able to study the nature of numerous programs since the only requirement is the binary itself. It voids the requirement of traces for simulators or source codes for studying program behavior.

Since these programs along with their binary input sets run directly on the machine, their execution times are short which allows the study of programs for their entire run. It opens up new venues of concepts that may be presented in the classroom, concepts such as phase behavior. Phase behavior requires that programs be run for a very long period of time, a requirement that cannot be met by simulators, but one that binary instrumentation tools can.

Furthermore, instrumentation tools allow quick implementation of ideas and do not require complex infrastructure. This would be suitable in the classroom because students may quickly implement new concepts and test the concept's effectiveness. This fosters a research oriented environment in the class which motivates students to investigate deeper into the subject. The instrumentation tools would allow students to explore real workloads of varying characteristics, from scientific and engineering programs and even to commercial products.

There exist varied tools that could be incorportated into the architecture curriculum; the essential note is that instructors should realize that it is extremely beneficial for students to have a broad idea of the tools available for teaching and performing real workload studies. The idea is much similar to making a decision in selecting the appropriate programming language when designing a softare applicaton. Thus it is essential for instructors to design their course material such the students are exposed to multiple tools through the run of their computer architecture coursework.

3 PIN - An Approach to using Binary Instrumentation Tools in Education

3.1 About PIN

A tool that we believe fills in the missing pieces of the previously described tools is PIN[13]. PIN is provided free of charge from Intel. It currently runs on the Itanium systems, but work is under way to support ARM and the IA32 architectures. It provides a functionality similar to the ATOM[1] tool for Compaq Tru64 Unix.

The user writes instrumentation and analysis routines. Instrumentation routines insert calls to analysis routines into an application. They determine how an application is instrumented. The analysis routines are called while the program executes and can record information like the effective address of a memory instruction or the direction of a branch instruction. The instrumentation is customizable; the user decides where analysis calls are inserted, the arguments to the analysis routines, and what the analysis routines do.

PIN inserts instrumentation into an application at run time. It sees every instruction in the user process that is executed, including the dynamic loader and all shared libraries. The instrumentation and analysis execute in the same address space as the application, and can see all the application's data.

PIN passes instructions or a sequence of instructions (trace) to an instrumentation routine. The instrumentation routine can inspect the instructions, looking at the opcode class and its register and literal arguments. The instrumentation routine may insert a call to an analysis routine before or after an instruction. PIN tries to make the instrumentation and its own execution transparent to the application. It does not use the same memory stack or heap area (brk) as the application, and maps addresses in a special area. Addresses of local variables (stack) and addresses returned by calls to brk, malloc and mmap will not be changed when PIN is active.

3.2 Using PIN

Presented in Table 1 is a simple example that gives the instruction count of a program including all the shared library calls made by the application program. The sample program is run by executing: *$ pintool – /bin/ls* at the shell prompt.

Lines 13 and 14 register callback functions with PIN. The function "Instrument Instruction" is the *Instrumentation* function that is called on every instruction and "Finish" is the function called upon termi-

```
      /* Analysis Function */
1     void AnalyzeInstruction() {
2          icount++;
3     }
4
      /* Instrumentation Function */
5     void InstrumentInstruction(INS ins, void *v) {
6          PIN_InsertCall(
               /* Call analysis func. before instr. is executed */
               IPOINT_BEFORE,
               /* Current instruction */
               ins,
               /* Call analysis func. before instr. is executed */
               (AFUNPTR) AnalyzeInstruction,
               /* End of PIN_InsertCall's argument list */
               IARG_END);
7     }
8
      /* Executed at end of program */
9     VOID Finish(int n, void *v) {
10         cout << "ICount : " << icount;
11    }

      /* Register callback functions */
12    int main(int argc, char *argv[]) {
13         PIN_AddInstrumentInstructionFunction(Instruction, 0);
14         PIN_AddFiniFunction(Finish, 0);
15         PIN_StartProgram();
16    }
```

Table 1: PIN tool to count the total number of instructions in a program

```
1     void InstrumentInstruction(INS ins, VOID *v) {
2        /* Query the opcode */
3        switch(INS_Category(ins)) {
4           case TYPE_CAT_BRANCH:
5               PIN_InsertCall(IPOINT_BEFORE,
                    ins,
                    /* Call the branch prediction program */
                    (AFUNPTR) Branch_Predictor,
                    /* The instruction address */
                    IARG_IP_SLOT,
                    /* Non-zero if branch will be taken; otherwise 0 */
                    IARG_BRANCH_TAKEN,
                    IARG_END);
6               break;

7           case TYPE_CAT_STORE:
8           case TYPE_CAT_LOAD:
9               PIN_InsertCall(IPOINT_BEFORE,
                    ins,
                    /* Call the data cache program */
                    (AFUNPTR) Data_Cache,
                    /* The memory address */
                    IARG_EA,
                    IARG_END);
10              break;

11          default:
12              break;
13    }
```

Table 2: PIN tool that interfaces with Data cache and Branch prediction simulators

nation of execution of an application. The *Analysis* function for every instruction is specified through PIN_InsertCall on line 6. The instrumentation function is called only the first time an instruction is executed. The analysis function, AnalyzeInstruction() is called every time the instruction is executed.

Data Cache & Branch Predictor Simulation Interface: The pin tool in Table 1 can be changed to support simulations easily by changing the instrumentation function. The instrumentation function in Table 2 easily integrates a data cache and a branch prediction simulator into one tool while still providing the previous instruction count analysis. Detailed opcode analysis is avoided here for simplicity. Precise opcode detail is available in the actual source at [13].

The simulator codes for cache and branch prediction can be written in entirely separate modules, compiled and linked with the pin tool. Also, due to PIN's inherent interface with the hardware, simulator code sizes

IPOINT	
IPOINT_BEFORE	Call before the instruction/procedure is executed.
IPOINT_AFTER	Call after the instruction/procedure is executed.
IPOINT_TAKEN_BRANCH	Call after the instruction executes and before the target is executed. Only supported for IP relative branches.

Table 3: Instrumentation Points (IPOINTs) for PIN_InsertCall(*IPOINT*, INS, AFUNPTR, iarg1, iarg2, ..., iargN, IARG_END)

are small in relation to real simulators. The data cache and branch predictor simulator code size are approximately only 20 lines. The development time is dramatically shortened because the need to build surrounding infrastructure to understand the instruction set architecture (ISA) is no longer required.

The PIN_InsertCall(IPOINT, INS, AFUNPTR, iarg1, iarg2, ..., iargN, IARG_END) function is the key to instrumenting any binary in the PIN environment. Details of the various instrumentation points (IPOINT's) that can be placed for every instruction or every procedure call are provided in Table 3. The function can take up to a maximum of eight arguments to facilitate various types of instrumentation and is capable of instrumenting both procedures and instructions. Table 4 describes the various Instrumentation Arguments (IARG's) that may be passed into the analysis function. The type AFUNPTR defines the analysis function to be called during the run of the program. Only a few of the IARG's and IPOINT's are listed for conciseness.

4 In the Classroom

4.1 Students and Projects

Understanding concepts allows the principles of many different disjoint areas to be leveraged in solving problems and developing skillful intuition. In turn, teaching is about exposing the underlying principles of ideas in ways that are both clear and logical. A good approach to teaching computer architecture is to be able to teach a concept and immediately illustrate a working system to students. The PIN infrastructure can be used to illustrate such ideas and allow students to cultivate and exercise their creativity and intuitions in projects. Often, course projects are limited in scope because of time, however, by integrating PIN with existing tools for use in a project, more structured ideas can be realized. Furthermore, it is evident that the act of learning an existing tool for a project is similar to real engineering situations in becoming assimilated to a particular design team.

One of the most common projects in computer architecture is to build concept simulators to enhance understanding. These projects include instruction cache, data cache and branch prediction simulators. Such assignments are very costly in the amount of time the students spend building the interface to reading the trace

IARG	
IARG_IP_SLOT	Memory address of an instruction, where the low 4 bits encode the slot number (e.g. 0, 1, 2).
IARG_IP	Memory address of the bundle containing this instruction.
IARG_EA	For a load or store, the effective address of the memory location accessed by an instruction. Only valid for IPOINT_BEFORE.
IARG_QP_VALUE	The value of the qualifying predicate for this instruction. Only valid for IPOINT_BEFORE.
IARG_REG_VALUE	The value of a register, register name follows.
IARG_BRANCH_TAKEN	Non-zero if the branch will be taken, otherwise 0. Only valid for IPOINT_BEFORE.
IARG_BRANCH_TARGET_ADDR	The target address of a branch.
IARG_FALLTHROUGH_ADDR	The IP and slot of the next instruction to be executed. If this instruction is a branch, it is assumed that the branch is not taken.
IARG_THREAD_ID	Thread id, first thread is 0, successive threads are 1, 2, ...

Table 4: Instrumentation Arguments (IARGS) for PIN_InsertCall(IPOINT, INS, AFUNPTR, *iarg1, iarg2, ..., iargN*, IARG_END)

format. This often results in students being limited to building only one or two simulators per semester due to time constraints, furthermore the time is spent on details of programming/software engineering and not on analyzing the results of the architecture simulators.

However, if the students had an opportunity to first interact with the various simulators as the class progresses and are assigned simple assignments of optimizing a pre-built simulator by changing the parameters etc. they would be able to get a good feel of how design parameters affect performance. Further more, it would allow the instructor to design the course such that at the end of the semester a project could be assigned where the students could pick a simulator that intrigued them and build it from bottom up. The advantage of that is that students often reach deep into their work when they are keenly interested in it. Exploiting that in them would guarantee that they extract the most from the class.

4.2 Applying PIN

In Section 2 we gave a brief introduction to PIN as a tool and hereby wish to reflect upon why PIN would prove effective in the classroom environment. PIN has certain characteristics which we believe makes it a unique experience for projects in the classroom. They are as follows:

Unique simulation environment: PIN's simulation environment is perceived uniquely by students because they realize the input program are binary tools that they use regularly such as ls, sort, grep etc. instead of presenting their simulators with traces. The students run the programs on real hardware rather than on an abstract software layers which limit some students from understanding how a simulator is working.

Reduced development time: The development time for simulators is dramatically cut short and thus allows students to focus more on actual data analysis; students often loose precious time in just building the simulator infrastructure.

Flexibility: PIN is a valuable teaching tool because of its flexibility in being able to support simulation environments as well as being to monitor compiled binaries both statically and dynamically. Often students are expected to cope with multiple tools because no one tool provides enough flexibility to be able to last through the run of the course. The students could use PIN all through their computer architecture without having to change tracks to using a new environment.

To give a generic view of how PIN could be used in the classroom; we present through Figure 1 the various analysis that students can do as part of their class projects in computer architecture coursework. The example illustrates cache and value profile modules being used on the entire program. This can be achieved by instrumenting every single instruction in the program using the PIN callback function PIN_AddInstrumentInstructionFuction(...). In procedural level instrumentation for call_A() and call_C(), instrumentation is injected again by using the PIN_AddInstrumentInstructionFuction(...) function; however the instrumentation range is dedicated only to the range of instructions that fall in the scope of those procedure calls. Detailed application programming interface (API) is available online at [13]. We see that call_A() and call_C(); data capture varies from collecting fine grained opcode statistics to generic profiling.

The currently released PIN kit contains various tools for use, a few of which are the data cache, branch predictor simulators, a tool to measure instruction counts and to analyze the latency of load instructions. Also contained are tools that perform profiling of the program; time spent in procedural calls etc.. A tool to collect detailed program traces is also available. Details of all the tools mentioned are available at the pin website [13].

4.3 PIN's accessory tools/libraries

PIN provides fine-grained analysis with excellent flexibility however a limitation that often tends to exist is with students being unable to analyze the data being collected. Thus we provide complementary tools to help the students.

Statistical analysis package. The students are provided with *data analysis* packages. These play a significant part in using PIN as a teaching tool because it voids the students from having to analyze raw data by hand. Collected data that has been processed as in Table 5 and Figure 2 make it easy for students to comprehend their program behavior better.

CUT - Colorado Utility Tool: Data generated in Table 5 shows detailed analysis of a load instruction at memory address 0x200000000000db20 that was pro-

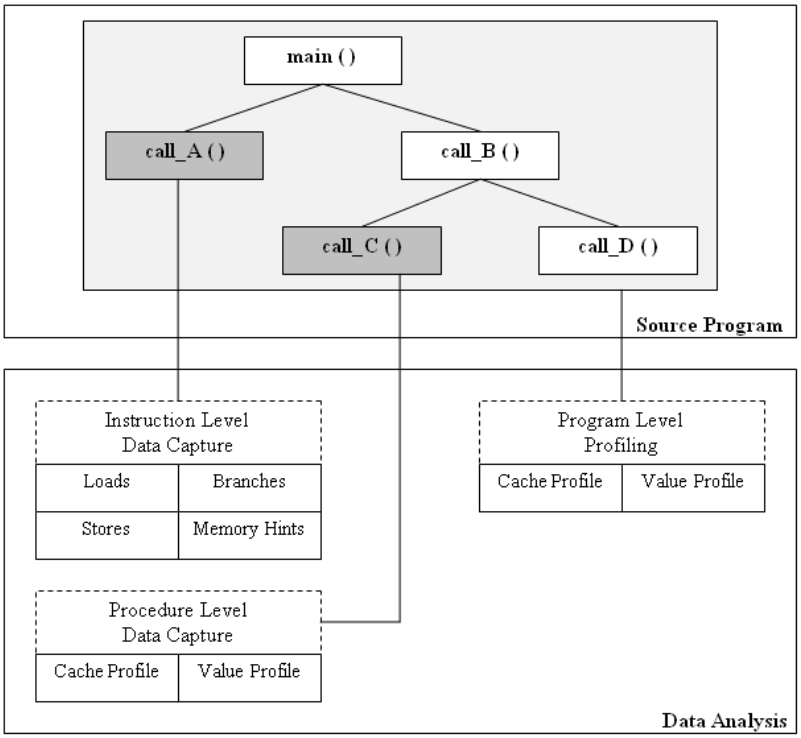

Figure 1: Instruction and Program level data capture

Colorado Utility Package (CUT): Data Analysis	
0x2000000000000db20-samples	7
0x2000000000000db20-mean	119
0x2000000000000db20-stddev	220.596
0x2000000000000db20-conf90	175.095
0x2000000000000db20-conf95	220.724
0x2000000000000db20-conf99	335.541
0x2000000000000db20-quantile-samples	7
0x2000000000000db20-Samples-6	3
0x2000000000000db20-Percent-6	42.8571
0x2000000000000db20-CumPercent-6	42.8571
0x2000000000000db20-Samples-12	1
0x2000000000000db20-Percent-12	14.2857
0x2000000000000db20-CumPercent-12	57.1429
0x2000000000000db20-Samples-45	1
0x2000000000000db20-Percent-45	14.2857
0x2000000000000db20-CumPercent-45	71.4286
0x2000000000000db20-Samples-158	1
0x2000000000000db20-Percent-158	14.2857
0x2000000000000db20-CumPercent-158	85.7143
0x2000000000000db20-Samples-604	1
0x2000000000000db20-Percent-604	14.2857
0x2000000000000db20-CumPercent-604	100

Table 5: Program level statistical analysis of a load instruction at address 0x2000000000000db20

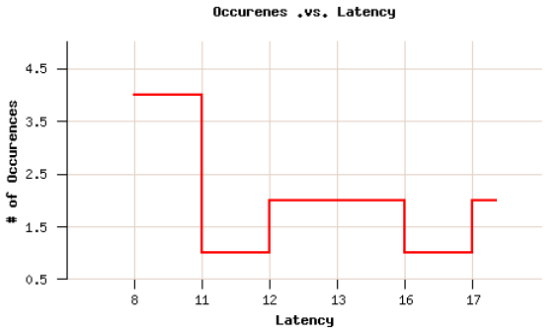

Figure 2: Dynamically generated load latency histogram

filed through the entire run of the program. The data is interpreted as follows: The first five lines reflect the samples, mean, standard deviation followed by the confidence intervals respectively. Thereafter the data reflects each of the individual samples; the load latencies every time the instruction was executed. *quantile-samples* is the total number of occurrences of this load instruction. Samples-*x* and its corresponding entry represent the frequency of occurrence of the load for latency *x*. Respectively followed by the percentage and cumulative percentages of occurrences of that latency for the given load instruction. The graph in Figure 2 reflects another data set; the graph was generated automatically through our CUT package.

Our analysis's package allows students to easily create digestible reports and graphs for post-run analysis.

Profiling structure library: Aside from PIN students are provided with a number of code modules for increasing the flexibility of the system as well as reducing development time. First, a set of data structure modules are provided that include generic caches, hash tables, time-line event record books, and symbol tables. In addition, a library module for value and memory address profiling is available for seamless integration with PIN instrumentation calls. The value profiler can be directed to keep a top*N* value (TNV) table for register operand values. The address profiler can track constant, stride, and finite-context matched patterns of addresses for load and store instructions.

Sampling interface: PIN provides a sampling interface that directs the binary instrumentation process. There are several management controls (known as PIN-pointing) which support triggering of the user-inserted instrumentation calls after an initialization period of instruction execution events or for periodic sampling. More detailed controls allow the instrumented code events to be called for a set interval of instruction executions after each periodic point has been reached.

4.4 PIN Projects

Numerous projects could be given out to students to select. The following are a few of those that could be used as a guide line:

Architectural models: It is helpful for students to see how the various architecture units of the system are performing while a program is running. A student could select/write the modules of interest such as: a Register Stack Engine(RSE), branch predictor, cache simulator etc.

Profiling: Profiling is a very common occurrence when studying program behavior. It serves as the fundamental step prior to doing in depth analysis. Thus profiling in conjunction with the data analysis packages would facilitate the study and generation of reports that reflect how the system performed through the run of the program based on the analysis the user has asked for. Some of the profiling tools could extend from simply collecting the opcode frequencies to observing the load referencing patterns where students may study how far ahead loads were fetched and record the actual use of the load. Yet another profiling tool could be one that looked for redundant loads and stores to the same location. PIN can facilitate this by looking at the source and destination registers and comparing them to see if the values are identical.

Trace collection: PIN is able of collecting traces of the programs as they execute. While there exist many tools that facilitate such a feature; the uniqueness of PIN is that the trace could actually consist of register values that are present at the time the instruction is being executed.

4.5 Future Development

Run-time program analysis is vital to understanding the essence of computer programming and not limited to comprehending the effectiveness of modern architecture designs. It applies to program writers at all levels, from students to software developers and especially to those involved in that area of research. It is vital to understand how a high level language such as C/C++ gets transformed into low-level code that runs on the underlying modern architecture since it affects the performance of the machine.

Lately interest has been growing in optimizing programming dynamically during their execution time - dynamic optimizations. With binary instrumentation tools, interesting research topics such as code caches, feed-back directed optimizations etc. may be simplified and presented in the classroom as projects to encourage research interests in students. Students are not aware of such concepts and presenting them with such ideas could give way to newborn interests in pursing the field further.

5 Conclusion

Instrumentation tools can be a vital teaching tool in the classrooms. We propose PIN as such a tool because it presents the students with live runs real compiled binaries on real hardware on a custom simulator if desired while also facilitating fine/coarse grained analysis and instrumentation functions. We believe that the ability to be able to merge all those into one program to be used as a teaching tool is of tremendous significance.

The tool would be extremely vital in helping students understand how programs are to be analyzed and how how their behavior can be monitored while still being able to teach them and making them understand the architecture upon which their computer programs run.

Acknowledgments

We would like to thank the Intel and Hewlett-Packard Corporations for the donation of Itanium Processor Family (IPF) systems. The systems allowed us to gain valuable insight and experience with the Itanium version of the PIN tool. Grant assistance in support of this work was provided by the Intel Corporation. We also extend our gratitude to Professor Dirk Grunwald at University of Colorado at Boulder for sharing with us his data analysis package - the Colorado Utility Tool (CUT).

References

[1] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In Proceedings of the ACM Symposium on Programming Languages Design and Implementation (PLDI'94), pages 196–205, 1994.

[2] Brorsson, Mats, "MipsIt: A simulation and development environment using animation for computer architecture education, Proceedings WCAE 2002", Workshop on Computer Architecture Education, Anchorage, AK, May 26, 2002, pp. 65-72. Tool available at http://www.embe.nu/mipsit

[3] Bryan Buck and Jeffrey K.Hollingsworth. An API for runtime code patching. The International Journal of High Performance Computing Applications, 14(4):317-329, Winter 2000.

[4] Doug Burger and Todd M. Austin and Steve Bennett "Evaluating Future Microprocessors: The SimpleScalar Tool Set" Technical Report 1996-1308, 1996.

[5] Greg Wolffe, William Yurcik, Hugh Osborne, and Mark Holliday, published in the Proceedings of the 33rd Technical Symposium of Computer Science Education (SIGCSE 2002), ACM Press, Northern Kentucky USA, Feb/March 2002.

[6] http://www.hpl.hp.com/research/linux/perfmon/index.php4

[7] Jason Blome, Manish Vachhajarani, Neil Vachhajarani, and David I. August., "The Liberty simulation environment as a pedagogical tool,", Workshop on Computer Architecture Education (WCAE), June 2003.

[8] Kim Hazelwood and Michael D. Smith. "Generational Cache Management of Code Traces in Dynamic Optimization Systems," 36th Annual International Symposium on Microarchitecture (MICRO-36). San Diego, December 2003, pp. 169-179.

[9] L.DeRose, Y. Zhang, and D. Reed. Svpablo: A multilanguage performance analysis system. In Proc. 10th International Conference on Computer Performance Evaluation - Modeling Techniques and Tools- Performance Tools '98, pages 352–355, 1998.

[10] London, K., Moore, S., Mucci, P., Seymour, K., Luczak, R. "The PAPI Cross-Platform Interface to Hardware Performance Counters," Department of Defense Users' Group Conference Proceedings, June 18-21, 2001.

[11] M. Burtscher. VPC3: A Fast and Effective Trace-Compression Algorithm. Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS04). June 2004.

[12] P.Lopez. http://www.gap.upv.es/people/plopez/english.html

[13] Robert S. Cohn, Intel Corporation. http://systems.cs.colorado.edu/Pin

[14] Vertical Profiling: Understanding the Behavior of Object-Oriented Applications Matthias Hauswirth, Peter F. Sweeney, Amer Diwan, Michael Hind, 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications.