Automatic Logging of Operating System Effects to Guide Application-Level Architecture Simulation

Satish Narayanasamy[†], Cristiano Pereira[†], Harish Patil[‡], Robert Cohn[‡], and Brad Calder[†]

[†]Computer Science and Engineering, University of California, San Diego
[‡]Intel Corporation

Abstract

Modern architecture research relies heavily on application-level detailed pipeline simulation. A time consuming part of building a simulator is correctly emulating the operating system effects, which is required even if the goal is to simulate just the application code, in order to achieve functional correctness of the application's execution. Existing application-level simulators require manually hand coding the emulation of each and every possible system effect (e.g., system call, interrupt, DMA transfer) that can impact the application's execution. Developing such an emulator for a given operating system is a tedious exercise, and it can also be costly to maintain it to support newer versions of that operating system. Furthermore, porting the emulator to a completely different operating system might involve building it all together from scratch.

In this paper, we describe a tool that can automatically log operating system effects to guide architecture simulation of application code. The benefits of our approach are: (a) we do not have to build or maintain any infrastructure for emulating the operating system effects, (b) we can support simulation of more complex applications on our application-level simulator, including those applications that use asynchronous interrupts, DMA transfers, etc., and (c) using the system effects logs collected by our tool, we can deterministically re-execute the application to guide architecture simulation that has reproducible results.

Categories and Subject Descriptors: I.6.7 [Simulation and Modeling]: Simulation Support Systems

General Terms: Experimentation, Measurement and Performance

Keywords: Architecture Simulation, Emulating System Calls, and Checkpoints

1. INTRODUCTION

Modern computer architecture research relies heavily on cycle-accurate simulation to help evaluate new architectural features. Our focus is on building and maintaining application-level simulators. These are simulators that perform cycle level simulation of the application code and system libraries, but do not simulate what happens while handling an operating system call or interrupt. A time consuming part of building such a simulator is correctly emulating the system effects executed as part of the workload under study. For example, the traditional solution [4, 19, 17] to emulate system calls for these simulators is by gathering the required input values from simulated registers and memory state and using them to invoke the call natively. In addition, most of these simulators do not support system effects such as DMA transfers or asynchronous interrupts because of their emulation complexity.

Emulating operating system effects, even just the system calls, can be a tedious exercise. For system calls, the programmer has to be aware of the input and output semantics of every call that needs to be emulated. Apart from having to handle the complexity of an emulator, porting the simulator to run on a different operating system is labor intensive. Even maintaining a simulator with system emulation can be quite expensive, since the emulator can break when the simulator is run on newer versions of the same operating system. Problems arise when there are changes to the operating system interface used by the application being simulated, since this can also require changes to the emulation system. In addition to all these problems, a good number of system effects are non-deterministic in nature, and as a result emulating them using native system calls during simulation can cause small variations across different simulations of the same program with the same input. Hence, simulation results may not be completely reproducible.

In this paper, we present a tool that can automatically capture the side effects of all the operating system interactions to support application-level simulation. We call our tool *pinSEL* (Pin-System Effect Logger), which is built using the Pin [9] instrumentation tool. We capture system effects by executing an instrumented version of the binary natively on the operating system for which the workload binary was compiled for. The instrumented code creates the *System Effect Log* (SEL) when executed. For each system call executed, the log contains the changes to the register state effected by the system call. The log also contains the values of memory locations accessed by load instructions executed after the system call, if those memory locations were modified by the system call. Our algorithm to identify the registers and memory locations modified by a system call is independent of the semantics of the system call and hence it is easy to implement and is portable across operating systems. The SEL also contains memory values modified by other system interactions such as asynchronous interrupts or DMA, if those modified memory values are accessed by the program being simulated. Thus the SEL enables deterministic simulation of a program-input pair across system calls, interrupts and DMA transfers. Deterministic simulation is important to accurately compare different alternatives during design space exploration. In addition, pinSEL can also support simulation of multi-threaded applications on uniprocessor systems, which we discuss in Section 5.6.

Using pinSEL, an application-level simulator can avoid the emulation of system effects and the associated complexity. As a result, we can easily simulate real applications from standard operating systems. For example, SimpleScalar [4], which has been widely used for over a decade, emulates just enough number of systems calls to support the simulation of SPEC and similar applications, and cannot support simulation of many real world programs. Using our approach, we can now simulate real world Linux applications on our x86 version of SimpleScalar [4] without having to emulate any system calls or complex interactions with asynchronous interrupts or DMA transfers. At Intel, engineers were successful in using pinSEL to quickly and easily support application-level architecture simulation of MAC OS and Windows applications. Without pinSEL it would not have been practical to port pinLIT to these operating systems for architecture simulator. PinLIT (Pin-Long Instruction Trace) is a tool used at Intel to gather checkpoints to support architecture simulation.

Using our approach, we can simulate only the execution of application code and the user level libraries. This is useful for studying applications like desktop and scientific programs, which spend a significant amount of execution time in the user level code. Even interactive applications like `acroread` and `powerpoint`, spend 80% and 76% of execution time respectively in application code and user level libraries [1], which will be captured by our approach. However, our approach has limitations in that it cannot be used to study applications that are heavily dependent on system interaction (e.g., I/O bound applications like TPC-C [6], web servers like DSS that spend significant amount of execution time in the kernel code).

This paper makes the following contributions:

- We examine how current popular simulators handle system effects through emulation. The discussion includes Intel's pinLIT and our x86 version of SimpleScalar [4].
- We present a technique to automatically log system effects for user-level architecture simulation. The benefits of our automated logging are (a) we do not have to build or support any infrastructure for emulating system effects, (b) our approach allows us to log and simulate more complex programs, including those that use asynchronous interrupts and DMA transfers, and (c) the system effect logs provide deterministic simulation across all kinds of system effects.
- We describe our tool called pinSEL (System Effect Logger) that is built using the Pin [9] instrumentation infrastructure. A user level architectural simulator built with pinSEL support can simulate programs that have complex interactions with the operation system and is also portable across different operating systems.

The rest of this paper is organized as follows. Section 2 discusses prior work. Sections 3 explains the application-level simulation technique used in a couple of widely used simulators. The complexity of manually emulating system calls in those simulators is discussed in Section 4. Section 5 presents our solution to automatically log system effects. Section 6 analyzes the time and space logging overhead. Section 7 summarizes.

2. PRIOR WORK

This section discusses existing solutions to handle system effects.

2.1 Handling system effects for Application Level Simulation

Many popularly used cycle accurate simulators [4, 19, 17] simulate just the user code and this is sufficient for studying many micro-architecture level optimizations and design choices using workloads like SPEC. However, even though their goal is to simulate only the user code, they still have to emulate the system calls to obtain correct execution of the program. The conventional solution to emulate system calls is to decode the system call and obtain the arguments. Then using those arguments the simulator invokes an equivalent system call that can be executed natively on the host machine on which the simulator is executing. The result values obtained from this native execution are then used to modify appropriate simulated registers and memory locations. The output of the system call can be stored in a trace (e.g., EIO trace in SimpleScalar) so that future simulations can use those traces instead of emulating the system call again. Using system call traces like EIO traces ensures deterministic simulation, and we describe this approach in more detail in Section 4.

The above approach is not desirable for a number of reasons. First, the programmer writing the emulator needs to explicitly handle each system call to find the registers and memory locations that contain the input/output operands. This code is then only valid for a given operating system. To use the simulator on multiple operating systems would require the emulation of the simulated system calls for each of these systems. Even maintaining the simulator to run on the same operating system requires changes over time to support newer versions of the operating system. Similarly, if the user desires to run a workload compiled for different versions of an operating system, the emulation may need to change if the operating system interface has changed. To top it all, complex system interactions due to asynchronous interrupts and DMA transfers cannot be handled easily with this form of emulation, which is required to correctly execute real world desktop applications like `acroread` and `powerpoint`.

In this paper we describe a simple binary instrumentation solution to capture the effects of all types of system interactions without having to explicitly emulate each system call or interrupt. Since our solution is independent of the operating system, it is very easy to provide simulator support to execute binaries compiled for various operating systems, as well as to allow the simulator to be compiled and executed in any operating system.

2.2 Full system simulation

There exist full system functional simulators like Simics [10], SimOS [15] and SoftSDV [20] that can emulate the full system including the operating system and all interaction with the external devices. Therefore, one option for building performance simulators would be to execute the binary inside a functional full system simulator and use that as a front end to feed traces of instructions executed to the cycle accurate performance simulator [7, 11, 5].

However, building and maintaining full system simulators is very expensive. It requires multiple person-years of effort to develop them. Also, they need to be modified constantly to support newer systems. In addition, the execution environment required for running real applications on full system simulators can be hard to reproduce, because of dependencies on specific kernel or device drivers versions, run-time license checking, elaborate installation procedures and high storage requirements. Therefore having a full system simulator in the front-end incurs higher runtime overhead during simulation. Moreover, if the goal is to analyze the performance of just the user code then it is an unwarranted complexity to have a full system simulator as a front end.

It is highly desirable to have a way of handling all forms of system effects to correctly execute the application during simulation, but still preserve the simplicity of application-level simulators. Our solution in the paper is targeted toward achieving this goal.

2.3 Checkpoint Mechanisms

Detailed cycle accurate simulation of the full program execution is very time consuming. Sampling techniques like SimPoint [16] and SMARTS [21] are used to find representative samples of program execution. Simulating only these samples have been shown to provide accurate simulation results. The *Sample Starting Image* (SSI) is the state needed to accurately emulate and simulate the sample's execution to achieve the correct output for that sample. Various checkpoint mechanisms have been proposed to capture the SSI [14, 2, 18] with minimal checkpoint size. In this section we describe those checkpoint mechanisms as they are related to the technique we use to collect our logs to capture system effects.

Szwed *et.al.* [18] proposed SimSnap, which instruments the application's binary, with the SSI corresponding to a sample and necessary code to restore it. Thus, during simulation, the simulated application's binary can itself restore the SSI for the sample to be simulated. To create such a binary, they first obtain the SSI of the application's state at the beginning of a sample by natively executing the instrumented binary of the application.

Ringenberg *et al.* [14] proposed an *Intrinsic Checkpointing* mechanism which also embeds SSI into the binaries and lets the application restore itself during simulation. Their focus is to create one binary, that restores the SSI for all of the simulation points needed to simulate the execution of that binary, for a specific input. In doing this, they make an observation that to create the SSI for a simulation point they can take the ending memory image of the last simulation point, and just update it with all of the stores that occurred between the end of the last simulation point and the start of the current simulation point. In addition, they optimize the restoration process by choosing to restore only those locations that are read at least once inside the simulation interval. The intrinsic checkpointing approach achieves the purpose of checkpointing the SSI at the beginning of a simulation interval by having a list of memory stores that need to be executed to get the memory image up to date for the start of the new simulation interval. This saves a significant amount of space over storing the full memory image state for each simulation point. Note that the simulator using this binary with intrinsic checkpoints still needs to have support for emulating the system call and other system interactions. This is because the only thing that the intrinsic checkpoint scheme ensures is that the simulation point has the correct SSI. Thus, intrinsic checkpointing does not address the problems of handling system effects, which is the focus of our paper.

Van Biesbrouck et.al. [2] also proposed an algorithm to reduce the size of SSI. Their technique assumes the EIO trace generation mechanism used in SimpleScalar to handle system calls. In the EIO traces generated by the default SimpleScalar, the SSI is the full memory image of the application at the beginning of the simulation interval along with a trace of result values of all the system calls executed (EIO trace) within the simulation interval. Instead of having the full memory image for SSI, they log initial memory values only for the locations that are accessed within the simulation interval. They also consider representing the same information in a different format in the form of Load Value Sequence (LVS) which is essentially a trace of all the load instructions. Their approach focuses on reducing the size of the SSI, and not upon providing system call logging. They still rely upon the EIO traces and system call emulation in SimpleScalar for that. Our focus is to not have to provide any system emulation for SimpleScalar, while at the same time enabling the simulation of real (non SPEC) programs on SimpleScalar.

3. BASELINE APPLICATION-LEVEL SIMULATION APPROACHES

In this section, we describe two system call logging infrastructures – pinLIT, which is used at Intel, and SimpleScalar, which is commonly used in academia.

3.1 pinLIT

An approach used at Intel for simulation is to first use SimPoint [16] to determine representative samples in a program's execution. Then a tool called pinLIT is used to create a checkpoint for each sample. A sample's checkpoint contains everything needed by their simulator to simulate the sample. In this section, we summarize this baseline technique used to create a sample's checkpoint.

3.1.1 SimPoint

The first step is to choose for a program-input pair where the execution interval for detailed simulation. SimPoint is used to choose the samples to be simulated. Note that other methods can be used to choose the simulation samples; the selection algorithm is not the focus of this study.

The SimPoint [16] sampling approach picks a small number of samples, that accurately creates a representation of the complete execution of the program. It breaks a program's execution into intervals, and for each interval creates a code signature. It then performs clustering on the code signatures, grouping intervals with similar code signatures into phases. The notion is that intervals of execution with similar code signatures have similar architectural behavior, and this has been shown to be the case in [16, 8, 13, 22]. Therefore, only one interval from each phase needs to be simulated in order to recreate a complete picture of the program's execution. SimPoint then chooses a representative from each phase and performs detailed simulation on that interval. Taken together, these samples can represent the complete execution of a program. The set of chosen samples are called *simulation points*, and each simulation point is an interval on the order of millions of instructions.

3.1.2 Creating Checkpoint Image

Once the simulation points are chosen, the next step is to create checkpoints for each simulation point using pinLIT (Pin-Long Instruction Trace) tool that is built using the

Pin [9] dynamic binary instrumentation tool. The checkpoint and system call tracing mechanism used in pinLIT provides the logs used to guide simulation as described in the Intel's UserLIT [17] simulation infrastructure.

A checkpoint image for a simulation interval contains all the necessary code and data information that is required for simulating the interval that it represents. This includes a trace of all the input and output values for the system calls executed within the simulation interval.

A checkpoint image for a simulation point is created as follows. The instrumented binary is executed natively and once the execution reaches the simulation point, the processor's architectural register state is copied to the checkpoint. In addition, pinLIT copies all the pages that contain application code and shared libraries to the checkpoint.

For the code and data pages, pinLIT tries to avoid checkpointing the entire data image of the process that exists at the beginning of the simulation point. Instead, pinLIT copies the pages lazily to the checkpoint when they are first used during the simulation interval. This approach avoids logging those data and code pages that are never accessed inside the simulation point and thus reduces the size of the checkpoint. The address locations inside the checkpoint image where the code and data pages are copied to are stored in a table at a particular location in the checkpoint image. This table, which we call as CheckpointPageTable, is required during simulation to restore the code and data pages.

In addition to copying pages accessed by the program to the checkpoint, pinLIT also logs enough information about the execution of system calls so that they can be handled during simulation. pinLIT has code specific to each system call that determines the inputs and outputs for every one of them. Before executing a system call, the analysis code in pinLIT logs information about the input values to the system call along with their address location (for memory operands) or the register name. After the return from the system call, the return value and any memory location and values modified by the system call are logged. When the system calls are encountered during simulation, the control is transferred to a special system call handler that verifies the arguments and writes the output in the proper memory and register locations. If the input arguments are different, then simulation is halted, since the simulation environment requires and only supports deterministic simulation.

3.1.3 *Simulation Using pinLIT's Checkpoint Image*

We now describe how the simulator uses the checkpoints. The simulator first loads the checkpoint image into its address space and starts the program's execution from address 0, which contains a specially inserted (by pinLIT) minimal operating system code or mini-OS. The mini-OS initializes the real page table using CheckpointPageTable to map the virtual addresses of the application to the physical addresses where the code and data pages from the checkpoint image are loaded. The mini-OS also registers a system call handler which is invoked whenever a system call is encountered during the program's execution inside the simulator. Finally, the architectural register's contents are read from the checkpoint image and written to the registers. Note that this sets the PC to the first instruction executed at the beginning of the simulation interval.

When a system call is encountered the system call handler verifies if the system call input values match the checkpoint image values and writes the outputs to the simulated registers and memory. The system call itself is ignored.

3.2 SimpleScalar

SimpleScalar supports a system call checkpoint environment called EIO (External I/O) logging, which is a trace of the output values of system calls. Playing back the system calls effects from the log ensures deterministic behavior, even if the system call has non-reproducible behavior (e.g. `gettimeofday`).

An EIO file contains a checkpoint of the initial program state that includes memory and architectural state that represents the state of the system at the beginning of the simulation interval. The rest of the EIO file contains information about every system call, including all input and output values and the name of the registers and memory address locations where those values should reside.

When the simulator encounters a system call, it restores the necessary register and memory values by reading them from the EIO trace. This method enables deterministic program execution across all the simulation runs.

4. COMPLEXITY AND EXAMPLE OF LOGGING SYSTEM EFFECTS

User level simulators need to emulate system calls for correct execution of applications. In this section, we discuss in more detail the solutions for emulating system calls and provide some concrete examples to illustrate the complexity involved in emulating them.

4.1 Emulating System Calls

We describe in more detail how system calls are emulated in SimpleScalar [4]. SimpleScalar's instruction decoder can interpret Alpha, ARM and PISA instruction set architectures. Recently, support for x86 ISA have been provided. For clarity, here we assume Alpha OSF binary emulated on a Linux x86 architecture.

4.1.1 *Approach*

When a system call is invoked by the simulated application, a special system call handler in the simulator is called to emulate it. The system call handler's operation can be summarized in three parts.

First, the system call handler has to decode the system call invoked by the application and obtain the necessary input arguments from the simulated register and memory locations. Decoding a system call is dependent on the system call numbers, which are specific to an operating system. For Linux, these numbers are specified in the header file *unistd.h*. This decoding part of emulation should support the operating system for which the application has been compiled for.

Second, the system call arguments are used to invoke an equivalent system call, that can be executed natively on the host machine. This part of the emulation should support the operating system on which we want to execute the simulator, because the arguments to the system call are specific to the system. For example, SimpleScalar can support execution of Alpha binaries compiled for DEC Alpha Unix systems (determined by the decoding part of the emulator) on x86 Linux systems (on which the emulator natively executes the system calls).

Third, result values obtained from the native execution of the system call are used to modify appropriate registers and memory locations in the simulator.

4.1.2 *Examples*

Let us consider how `open` system call is emulated. For the `open` system call, register ECX contains the flag input and EBX contains the address to the location containing the filename. The flag input format can change between operating systems and we personally have experienced problems while trying to run SimpleScalar on some newer versions of RedHat Linux, which required changes to the emulation system.

For `open`, the filename is copied into a temporary buffer. The temporary buffer and an integer containing the flag value are used as arguments to invoke the `open` system call natively. The file handle returned from the native system call is then copied into the EAX register. Note, the emulation of the `open` system call would be affected if either the binary is compiled for a different operating system or if the host on which the simulator is executed change.

Let us consider another example. The `read` system call is used to read a specified number of bytes from a file and copy the values read to a buffer. To emulate this system call, SimpleScalar invokes the `read` system call natively using the contents of register EBX and EDX as arguments, where EBX contains the file handle and EDX contains the size of the buffer. The `read` system call also requires a pointer to the location where the read contents need to be stored. To accomplish this, SimpleScalar allocates a buffer of a size specified by EDX register and passes the pointer to the `read` system call. Once the system call returns, the contents of the buffer are copied to the location whose address can be found in the ECX register. Finally, the EAX register is written with the error code returned from the native execution of the `read` system call. Note that, the `read` call can modify the memory location pointed to by ECX and the number of locations modified is dependent on the size specified in the register EDX. Thus, it is necessary to capture the system effects on the memory locations.

For this example, EAX, ECX and EDX determine the memory locations modified by the system call. Other system calls have different interfaces (e.g. pointers to structures, etc), and each case must be handled individually. These memory inputs and outputs are system call specific and this is why creating these emulation systems is tedious, error prone, and hard to maintain.

4.1.3 Handling Asynchronous Interrupts and DMA

Emulating more complex interactions with the system through asynchronous interrupts and DMA are even tougher to handle in an execution driven simulator. It would require modeling the full system including the external peripheral devices, like in Simics [10]. Hence, applications affected by interrupts and DMA are not supported in the user-level architectural simulators [4, 19, 17], but our logging approach captures the memory effects seen during application level execution.

4.2 Providing Automated System Effects Logging

The above implementation for logging system effects is not desirable for a number of reasons. Note that handling system calls involves identifying the input and output values of each system call. This requires decoding and writing code to handle each system call. This method is not portable to simulate applications compiled for a different operating system or even for a different version of the same operating system. In addition, pinLIT and SimpleScalar do not support applications that use asynchronous interrupts and DMA transfers. We solve these issues with our automated system effect logging to capture all forms of system effects which we describe next.

5. AUTOMATIC LOGGING OF SYSTEM EFFECTS

In the previous sections, we described how popular cycle accurate simulators [4, 19, 17] need to emulate system calls to achieve correct program execution. For example, SimpleScalar emulates 81 unique system calls to support simulation of SPEC and similar programs. In comparison, the pinLIT simulation tool used at Intel emulates 258 system calls to support a much more wider range of applications compiled for the most popular Linux kernels. Emulating these system effects is tedious to implement, hard to maintain, and error prone.

In this section, we discuss an instrumentation tool that can automatically capture system effects in a log, which can then be used to guide architecture simulation. The tool that we describe here can also support simulation of multi-threaded programs on a time-shared uniprocessor system, which is discussed in detail in Section 5.6. It can also be extended to support deterministic simulation of multi-threaded programs on multi-processor systems, but we leave that for future work.

5.1 Overview

Our goal is to automatically capture all the system effects to a program's execution in a *System Effect Log* (SEL) which can be used to replay the program's execution and simulate it without having to emulate any system effects. The SEL replaces the system effect logging approach used for pinLIT and the SimpleScalar EIO checkpoint trace described in Section 3. Our logging approach is much easier to implement and maintain, and it provides support for asynchronous interrupts and DMA transfers, which are supported neither in the pinLIT nor the SimpleScalar EIO tracing mechanism.

We built our system effect logger called *pinSEL* using the Pin [9] dynamic instrumentation tool. We briefly describe the key concept that allows us to automatically capture system effects. Our algorithm is inspired by the checkpoint scheme used in BugNet [12]. A straight-forward way to capture the system effects to a program execution is to log the value of every single load instruction executed by the program, and to log the register states and the PC value after handling a system call or an interrupt. However, this method is clearly too expensive in terms of runtime and log size overhead. Instead, we need to log a load value, only if (a) the load is the first memory operation to access the memory location or (b) the memory location accessed by the load has been modified due to a system effect. We determine the second condition by keeping track of a *user-level* copy of the memory space that is read and written by the application during execution. The redundant copy is called the user-level copy, because it is maintained in the pinSEL's address space, and is updated by pinSEL for load and store operations executed by the application. The user-level copy is *not* updated when the system modifies the corresponding application's memory state while it is handling system calls, interrupts or DMA transfers. Hence, if an application's memory location is modified due to a system effect, and later if a load accesses the same location, pinSEL detects a mismatch between its user-level copy and the corresponding value in the application's address space. When pinSEL detects such a mismatch for a load, it can determine that the program's memory value has been changed by some system event external to the program being profiled, and hence it knows that the load value needs to be logged. We use a similar mechanism to capture the system effects to the register states before and after a system call or interrupt. Thus, for a program's execution, we are able to automatically log external system effects to its execution state, without having to explicitly model and emulate the system interactions.

The SELs can then be used to deterministically replay a program's execution to guide architecture simulation and avoid the need to emulate the system interactions. The simulation of application level execution is accurate as SELs enables deterministic replay of program execution. However, there can be slight inaccuracies (less than 1% error) while simulating mis-speculated paths if those execution paths access memory locations that have not been logged. We discuss this limitation in more detail in Section 5.8.2.

5.2 Introducing pinSEL

Our goal is to collect a *System Effect Log* (SEL) to guide reproducible architecture simulation. The SEL contains the initial register, program counter and memory (code and data) values accessed by the program execution and all the system effects to those memory and register states. In addition, for multi-threaded programs executing on time shared uniprocessor systems, it also contains information about thread interleaving which is discussed in Section 5.6. The SEL can be for the complete execution of a program or just for a sample of program execution. The sample could be hand picked, or chosen using tools like SimPoint [16].

The SELs are collected by dynamically profiling the program execution using Pin. PinSEL is similar to pinLIT in that it is used to collect checkpoint traces for simulation. The difference is that in pinSEL, the system effects to both memory and register states are captured using a generic algorithm that is completely independent of the operating system. As a result, unlike pinLIT, it can also easily capture system effects due to interrupts and DMA transfers.

5.3 Dynamic Instrumentation

To profile a program using pinSEL, we execute the program natively on the system that it was compiled for. PinSEL then dynamically instruments the program binary, and the SELs are gathered as the program executes. Pin [9] provides interfaces that allows us to instrument classes of instructions, specific functions, system calls and interrupt events, allowing us to register call-backs to our analysis routines at those instrumentation points. When a pinSEL's analysis routine is invoked for an instrumentation event, pinSEL can examine the program's architectural register and memory states, update its internal data structures, and log information to the SEL files if necessary. Then after we are done with the analysis for an instrumentation event, the program's execution continues until the next instrumentation point before invoking an analysis routine again.

We use Pin's interface in our pinSEL tool to instrument every load and store instruction, so that the analysis routines can keep track of *user-level* memory state of application's data sections and capture its initial state and subsequent system effects to them. We also instrument every basic block to log the initial state and system effects to code regions in an application's memory.

Finally, we instrument every system call and the interrupt handlers. Pin allows us to register call-backs to our analysis routines that are invoked before and after the execution of system call and interrupt handlers, allowing pinSEL to capture system effects to the register state.

5.4 Timestamps

Every log entry in SEL contains a timestamp that tells us when that entry has to be used during simulation.

We use two types of timestamps in our logs - memory operation count and instruction count. The *current memory operation count* of a program execution is the number of dynamic load and store instructions executed since the start of the logging, whereas *instruction count* is the total number of instructions executed since the start of the logging. Tracking instruction count at the granularity of every instruction incurs high instrumentation overhead. Instead, we update the instruction count only after executing a basic block, where a basic block is a sequence of instructions with a single entry and a single exit point.

The above two counts are tracked only for the application's execution (user code and user level libraries) and are not updated during the execution of the system kernel code. Hence, while simulating the application's execution, we can accurately keep track of these timestamps and determine when to use a log entry. To reduce the size of the timestamp being logged, instead of logging the full memory and instruction counts as the timestamp, we optimize the size by just logging the difference between the prior count and the new count for the current log entry.

5.5 System Effects Log Files

A SEL for a program's execution is composed of the following three log files, at a minimum.

5.5.1 Code Update Log

The purpose of this log is to record the initial memory values of the code regions and the system effects to them. This ensures that we can handle programs using self-modifying code and dynamically loaded libraries. Each entry contains (a) an instruction count and (b) the code contents of a basic block and its size. During simulation, when the number of instructions simulated is equal to the instruction count of the next log entry to be used, we restore the logged code for the basic block to the simulated memory before executing the next instruction. The effective address for restoring the code log entry is the simulated program counter (PC) value. We therefore do not need to log the starting address for the code block.

5.5.2 Data Update Log

The purpose of this log is to record the initial memory values of the data regions and the system effects to them. Each entry contains (a) a memory operation count, and (b) the value of a load operation. During simulation, before executing a load operation, if the simulated memory operation count is equal to the memory count of the next entry in the log, the logged value is restored to the simulated memory. The effective address which we should use to restore the log value is the effective address of the simulated load, which can be determined during simulation and hence need not be logged.

5.5.3 Register Update Log

The purpose of this log is to record the initial states of the architectural register values and the program counter values, and capture subsequent updates to them due to the execution of both synchronous interrupts (system calls) and asynchronous interrupts. At the beginning of execution, we log the initial values of all the registers and the program counter in this log. Then an entry in the log is created whenever we encounter an interrupt with the following information: (a) the instruction count, (b) the value in the program counter before the execution of the interrupt, (c) the sequence of register values modified by the interrupt handler along with the name of the modified registers, and (d) the program counter value after the execution of the interrupt, if it had been modified. The instruction count along with the PC value, before the execution of the interrupt, together accurately capture the time at which the interrupt was executed during the program's execution. During simulation, when it is time to use an entry from this log, we restore the logged register values in the corresponding simulated registers. Also, we restore the logged PC value to the simulated program counter if it was also logged. Note, any memory value updated by the interrupt is logged in the Code and Data Update Log.

In addition to the above logs, SEL also records necessary information to simulate multi-threaded programs on a uniprocessor system which is described in Section 5.6. We describe now how each of the above logs is created in more detail.

5.5.4 *Code and Data Update Log*

To capture changes in memory due to system interaction, we maintain a data structure called the *UserMemState* in pinSEL. The UserMemState keeps track of the values for every memory location accessed by the application. The values in UserMemState are updated only for the load and store instructions executed by the application and not by the system code. Thus, it keeps track of what we call *user-level* memory state. It is essentially a hash map table, indexed by the address. Each entry in the table mirrors 4KB of application's address space. The initial value for each address location in the table is set to zero.

We instrument each load and store to keep track of data values in UserMemState. To keep track of code regions in the application's address space, we instrument each basic block.

Analysis for Store - Whenever the application executes a store to an address in its address space, we update the value in UserMemState for that address with the store's output value.

Analysis for Load - When executing a load, we check if the value in the application's memory for the load's effective address differs from the corresponding value in the UserMemState. If they differ, then it implies that, (a) it is first time we are accessing that memory location or (b) the accessed memory location has been modified by the system while handling a system event. Note, if it is the first access to the data address and the value loaded is zero, we do not need to log it, because we initialize all of memory to zero in our simulator. In addition, if the application's store instruction modifies a memory location, then we would have correctly updated the UserMemState when the store was executed. Therefore, whenever the load value is different from the corresponding value in the UserMemState, we log the value in the Data Update Log along with the current memory count. This ensures that we capture the initial memory data values as well as system effects to them in the Data Update Log.

If we have to log the value for the load because it differs from the value in UserMemState, we also update the UserMemState's value for the load's address with the new value observed in the application's memory. This is required to make UserMemState consistent with the new value we found in the application's memory state, so that future loads to the same location will not result in additional logs, unless it gets modified due to a system effect.

During simulation, we initialize all the memory locations to zero. Thereafter, before simulating a load instruction, if the simulated application's memory count is equal to the memory count value of the next log entry, then we restore the logged memory value to the simulated memory. To restore the value, we need to know the effective address of the load instruction, which we would not have logged. However, during simulation, we know the input operands for all the instructions, including the load instruction, and hence we are be able to compute its effective address and use it to restore the value from the log.

Analysis for Basic Block - The Data Update Log created by analyzing load and store instructions captures the initial memory values and system effects to only the data values accessed by the application. It does not contain the instructions fetched for execution, unless they are loaded by some load instruction.

For our analysis, an instruction fetch can essentially be treated as a load from the address specified by the program counter value. To capture the code, we instrument every basic block in the program to register a call-back routine that is invoked before the execution of each basic block. A basic block is a sequence of instructions with a single entry and a single exit point. This means, if the program control reaches the beginning of a basic block, we can be assured that all the instructions in the basic block are going to be executed.

When our analysis routine is invoked just before the execution of a basic block, we compare the N bytes of application's memory values at the location specified by the program counter value with the corresponding values in the UserMemState. If the comparison fails for any of the bytes, then we log the value in the Code Update Log along with the instruction count. Also, we update the value in UserMemState with the up-to-date value in application's memory in order to make them consistent.

During simulation, when the simulated instruction count equals the instruction count in the next code log entry to be consumed, we restore the code from the log to the simulated memory using the address in the simulated program counter.

Handling Self-Modifying Code and DLLs - Our mechanism ensures that we are able to handle applications using self-modifying code. The Code Update Log captures the initial values in the code regions during execution. An application using self-modifying code modifies itself by executing store instructions, which will be deterministically replayed during simulation. That is, we know the exact input and output values for each store instruction and hence handling self-modifying code is not an issue.

We can also handle applications using dynamically loaded libraries (DLLs). A dynamic library can be loaded during a program sample's execution through the invocation of a system call (eg: `mmap` system call in Linux). Since the Code Update Log captures any changes to the code regions, it will also capture the contents of the dynamically linked libraries when they are fetched from memory for execution.

Note, a more light weight approach (in terms of run-time overhead) for logging code is to integrate the code logging with the run-time system used to execute the program, instead of instrumenting every basic block. Using this type of run-time system, no code can execute until it has first been pre-processed. The first time it is executed, it will be analyzed once and the code to be executed will be logged. Then the code will not have to be analyzed for logging again, unless it is modified. If there is self-modifying code, then the code would be invalidated by the run-time system. It will then be re-analyzed before it can execute again, and when it is, it will be re-logged. The run-time system to support this type of approach could be a virtual machine or a dynamic binary instrumentation system like Pin. Since the code is analyzed for logging only once before it is first executed, the run-time overhead will be minimal compared to instrumenting every basic block.

5.5.5 *Register Update Log*

The Code and Data Update logs described in the previous section can capture initial memory values and the system effects to them. In addition, we also need to log the initial register and program counter (PC) values, and system effects to them due to the execution of a system call or an interrupt handler.

At the beginning of the execution of a sample (chosen using sampling techniques like SimPoint [16]), we log the initial values of the architectural registers and the program counter in the Register Update Log. Thereafter, we create an entry in the log whenever the program execution encounters a system call or an interrupt.

Pin [9] provides us APIs, *SIGNAL_AFTER_CALLBACK* and *SIGNAL_BEFORE_CALLBACK*, to register call-backs to our analysis routines before and after the execution of system call (synchronous interrupt) and signal (asynchronous interrupt) handlers.

Before the execution of a system call or an interrupt, we record the state of all the application's architectural register values and the program counter value (which are accessible through Pin's interface) in pinSEL's internal data structure. Then right after the execution of the system call or the interrupt, we compare the current register and PC states with the recorded values. The values of the registers and PC for which the comparison fail are logged in the log entry.

Each log entry also contains the instruction count and the program counter value before the execution of the system call or the interrupt. These two values, together constitute a timestamp that tells us when the log entry should be used during simulation. During simulation, we use the next log entry from this log, if the simulated instruction count is greater than or equal to the logged instruction count, *and* if the simulated program counter value matches with that of the logged PC value.

5.6 Simulating Multi-threaded Programs on Uniprocessor Systems

Our approach also allows simulation of multi-threaded programs on uni-processor systems. For each thread, we create a SEL consisting of Code, Data and Register Update log and all the data structures in the pinSEL used to create these logs are kept private to each thread. Whenever a new thread is created within a sample's execution, we create a Register Update Log for the thread and log the thread's initial register and program counter values. Thereafter, for each system call or interrupt executed as part of the thread, a new log entry in the thread's Register Update Log is created.

During simulation, we simulate the thread inter-leavings just as they would occur on a uni-processor. To achieve this we need to capture context switches and log sufficient information about them in a *Context Switch Log.* This log file is shared among all the threads in the program's execution and is created as follows. Whenever there is a context switch from one thread to another, we create an entry in this log. We detect context switches between the threads of the profiled application as follows. Pin internally keeps track of a unique thread ID for each thread and these IDs are accessible from the analysis routines. Inside each analysis routine, we compare the current thread ID with the thread ID seen by the last executed analysis routine. If they differ, then it means that there was a context switch.

On detecting a context switch, we create an entry in the Context Switch Log. The entry contains the thread IDs of the thread that is context switched out and the thread that is context switched in. Also, the log entry contains the memory count corresponding to the last memory operation executed by the thread that is context switched out. While simulating a thread, we keep track of its memory count, and if it equals to the memory count for an entry in this log, then we know that the thread needs to be context switched out. Also, we know which thread we should start simulating next.

The above mechanism is useful for simulating multi-threaded programs on uniprocessor systems by reproducing the thread inter-leavings. Evaluating the utility of this simulation methodology and extending this to model multi-processor systems is left for future work.

5.7 Atomic Analysis

In Section 5.5.4, we described our analysis functions that can automatically capture the system effects to memory. To record the Data Update Log, pinSEL's analysis routine compares the load value with the value in UserMemState when executing every load. However, between the execution of the analysis routine and the application's load, there can be an interrupt that modifies the memory value accessed by the load. If so, the value seen by the analysis routine can be different from the value that is actually loaded. Essentially, the execution of the application's load and the analysis routine is not guaranteed to be atomic. In addition to interrupts, atomicity can also be compromised if a new thread that gets context switched in, modifies the load's memory location.

This is not a problem for single threaded SPEC programs, but it needs to be handled for programs with asynchronous interrupts, and multi-threaded programs with shared memory interactions. For those programs, we solve this problem by keeping track of a bit for each thread, called the *interference bit.* The interference bit is set for a thread when it is context switched out or when it encounters an asynchronous interrupt.

The interference bit is cleared before the execution of the load for the thread. Then, after the execution of the load, we check to see if the interference bit is still clear. If so, we determine if we need to log the load's value as described in Section 5.5.4. However, if the interference bit is set, then this means that either a context switch or asynchronous interrupt occurred between executing the load and our logging analysis code. If this is the case, we log the value of the load's output register in the Register Update Log along with the current timestamp. During simulation, all that we need to do is restore the logged register value in the simulated registers, *after* executing the load instruction corresponding to the logged timestamp.

It should be noted that the above solution works even if the "interfered" load operation is a complex x86 CISC instruction that updates multiple memory locations with newly computed values (eg: a read-modify-write instruction). This is because the UserMemState will contain the value seen only by the analysis routine executed before the interfered load. When a load accesses the modified memory location in future, we will detect a mismatch between that load's value and the value in the UserMemState and log it correctly in the Data Update Log.

Our basic block analysis can also be affected by interrupts and context switches, which can be solved as follows. After handling an asynchronous interrupt or when a thread gets context switched back in, we compare the values of the current basic block in the application's memory with the values in the UserMemState if the interference bit is set. If they differ, we log it in the Code Update Log entry along with the current timestamp. The timestamp we use for this log entry comprises of both the current instruction count and the PC value before the context switch or the interrupt execution. Using this timestamp, we can precisely restore the logged basic block contents during simulation. Note that, register and PC values modified by the interrupt are logged in the Register Update Log, as described in Section 5.5.5.

5.8 Architecture Simulation

PinSEL's logging approach replaces the pinLIT logging approach for system effects and SimpleScalar EIO traces to deterministically guide the program's execution through simulation. The above sections describe how and when to use each of the logs to guide simulation. We have implemented a version of SimpleScalar that runs x86 binaries, and have modified it to consume our logs to guide simulation. At UCSD, we currently use pinSEL to collect SELs for Linux applications which can then be used for simulating them in x86 SimpleScalar. At Intel, we use the pinSEL approach of logging system effects inside of pinLIT for Linux, Mac OS, and Windows applications to guide architecture simulation.

5.8.1 Advantages of PinSEL

The main advantage of using SELs we have described thus far is our ability to automatically log system effects to avoid emulation of system calls in simulators. Another advantage of using our pinSEL approach is that the simulator can easily support the simulation of applications compiled for any operating system as long as Pin [9] can support it.

The other advantage of using SELs is that it provides deterministic re-execution of the program to guide simulation. Since the same SEL is used across all simulation runs, the load instructions read exactly the same values and hence the execution of the program follows the same path in all the simulation runs. This is an important property that helps us to simulate user interactive applications.

5.8.2 Limitation

Simulations based on checkpoints and traces can be affected when it comes to simulating the wrong path (misspeculated path) in the program's execution. When we simulate using pinSEL logs in SimpleScalar, it is still possible to model the wrong path execution similar to execution driven simulation. However, there could be slight inaccuracies in the simulation, if the wrong path of execution tries to execute code or data that was neither logged nor is regenerated during simulation. In such an event, the wrong path execution can either stop executing down the mis-speculated path or it can proceed by consuming a null value.

Van Biesbrouck et.al. [2] examined this issue for their technique to reduce the checkpoint sizes which we described in Section 2.3. Their simulation uses optimized checkpoints that contain only the code and data addresses used during the sampled simulation interval and as a result experience the same problem as ours. However, they found that the error in performance metrics due to the above inaccuracy is less than a 1% on average. Moreover, this inaccuracy is consistently biased in one direction across different simulation runs for an application while exploring the architecture design space, enabling a fair comparison across different design alternatives.

PinSEL's utility is limited when we want to use it to study applications whose performance is heavily dependent on system interaction. For example, we may only be able to capture less than half of the execution of I/O bound applications like TPC-C [6] and other server applications like DSS (Darwin Streaming Server), since they spend so much time executing in system code. However, there are many interactive desktop applications like `acroread` and `powerpoint`, etc, which spend 76% to 80% of their execution time in application code and user level libraries [1] (non-kernel code), which we will capture with our approach. This makes pinSEL useful to evaluate these types of applications.

6. LOGGING RESULTS

In this section we examine the runtime overheads in collecting pinSEL logs along with the log size overheads for most of the SPEC programs and a handful of desktop interactive programs. We traced the execution of all these programs using pinSEL, and used the logs in our x86 SimpleScalar to simulate them.

6.1 Benchmarks

For this study we ran all the SPEC programs that we could compile and run in our environment. We also ran a handful of desktop interactive Linux programs. We first ran each one natively for about two minutes, and then replayed the same actions with the program instrumented with pinSEL to gather the logs.

The programs we examined were `xpdf`, `acroread`, `ggv`, `xv` and `rdesktop`. The first two are used to read PDF documents and `ggv` is used to read Postscript documents. `xv` is an image processing application. `rdesktop` is used to remotely access a Windows system. We ran the first three programs to open and read the files, browsed through them, enlarged them and exited. We ran `xv` by using it to open two JPEG images, zoomed in and out on an image and ran an image sharpening utility that comes along with the application over those images. For the `rdesktop` program, we opened a connection to a windows machine, and browsed a few web-sites using `firefox` and worked on a `power-point` file before logging out.

6.2 Avoiding Software Complexity of System Effects Emulation

One important result of this paper is that to collect logs and to simulate a wide variety of applications, including real interactive programs, and we do not have to provide any system emulation support. In comparison, Intel's pinLIT has a large body of switch-case statements to handle each of the 258 different system calls and it still can simulate only a limited variety of applications. x86 SimpleScalar has support for emulating only 81 different system calls which is the set of system calls that are sufficient to simulate the SPEC workload. However, this support is inadequate to simulate interesting desktop applications.

Using our automated logging approach, we can now simulate any type of application in our x86 SimpleScalar. Also, since our application level simulation approach is independent of the operating system, it ensures the portability of our simulator to any version of Linux operating system. The approach is easily portable to other operating systems, as long as there is a binary instrumentation tool that can allow us to collect SEL files, which is why at Intel, pinSEL has enabled simulation of Windows and MAC OS based applications.

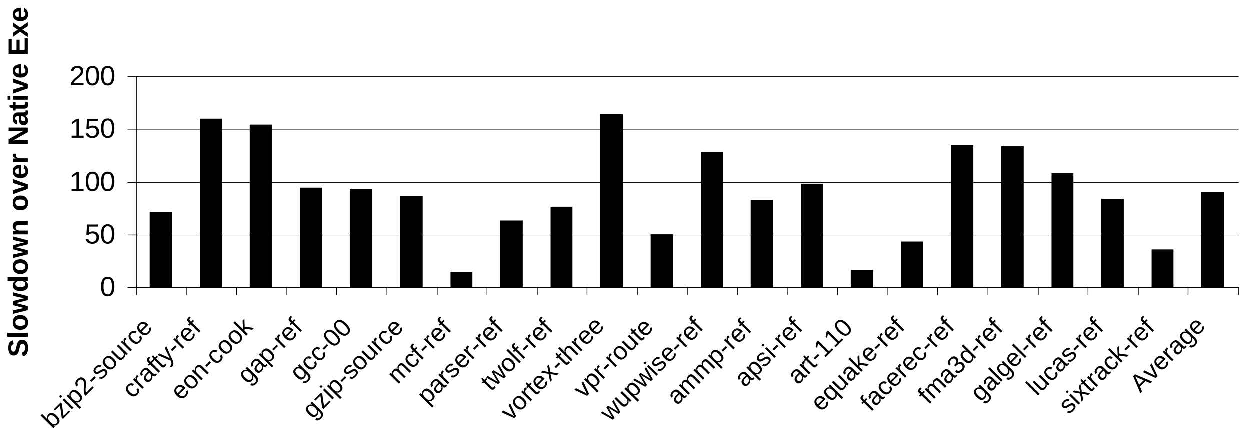

Figure 1: pinSEL logger runtime slowdown over native execution for the SPEC programs.

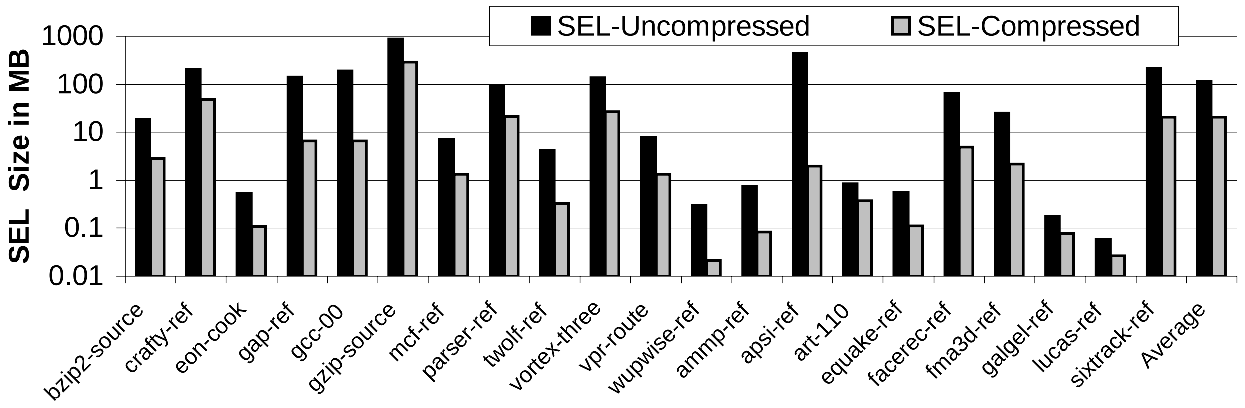

Figure 2: pinSEL log sizes to capture the full execution of the SPEC programs, with and without compression using bzip2.

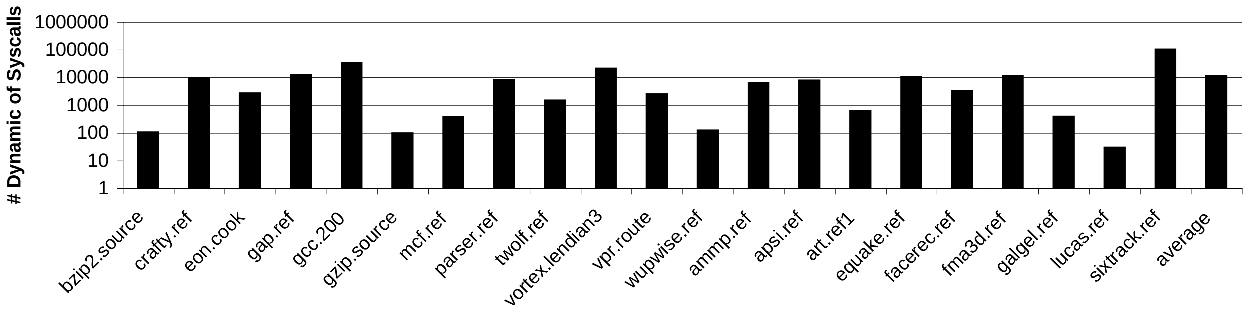

Figure 3: Number of system calls executed in SPEC

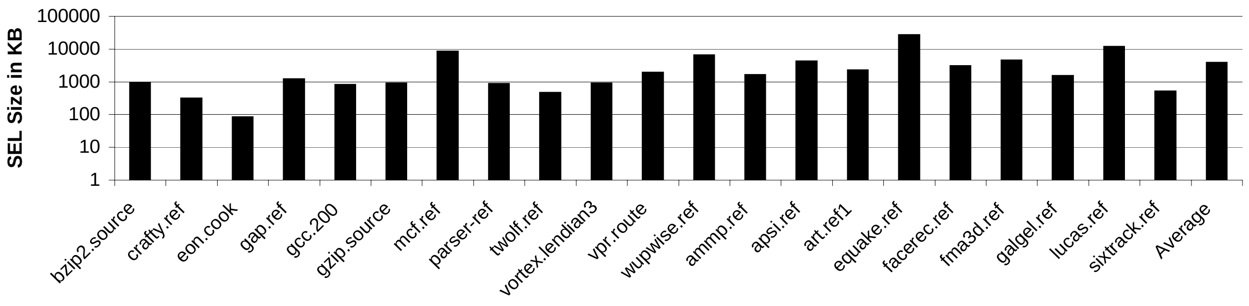

Figure 4: SEL size required to capture a simulation point of 100M instructions for each SPEC program on average, without compression.

Though our mechanism can enable simulation of any application in our x86 SimpleScalar, as we pointed out in Section 5.8.2, the evaluation will be meaningful only for those applications that spend a significant proportion of execution time in the application and user level shared libraries.

6.3 Log Sizes and Logging Overhead

We now examine the runtime and space overhead in generating SEL logs for the full execution of the SPEC programs. For this, we ran the SPEC programs over the reference inputs with the pinSEL tool to generate the logs.

We used a hash map table (UserMemState) indexed by the address. Each entry in the hash table keeps track of 4KB of data or code. Figure 1 shows the slowdown for running the program with our pinSEL tool. The runtime overhead is with respect to natively executing the workload. We can see that the worst case runtime overhead is about 163x for `vortex`. On average, we experience slowdown of about 89x. These overheads are for tracking the full execution of the program. Also, we can notice that the runtime overhead due to instrumentation for programs that usually have low IPC (eg: `mcf`) is only in the order of 10x to 20x. Whereas, programs with high IPC experience slowdowns in the order of 150x (`crafty`, `eon` and `vortex`).

Figure 2 shows the log sizes for capturing the SEL for the full execution of SPEC programs that we studied. The results show log sizes with and without compressing the logs using bzip2 using the default compression level. In the worst case, we require about 866MB of un-compressed SEL to capture the full execution of `gzip`, and we only require 291MB of SEL after compressing it using bzip2. On average, we require only about 115MB of uncompressed SEL, which when compressed requires only 22MB of disk space to capture the full execution of a SPEC program running the reference input. The sizes of SELs are dependent on the number of system calls executed and are also heavily dependent on the amount of data read from the system through those system calls. Since `gzip` reads a large amount of data from the files through system calls to perform compression and decompression, it incurs a large log size overhead.

Figure 3 shows the total number of system calls executed in the SPEC programs that we studied. On average, there are about 11,400 system calls executed during the full execution and in the worst case for `sixtrack` there are about 104,000 system calls being executed.

6.4 Log Sizes Per Simulation Point

It is a common practice in computer architecture to choose representative samples of program execution [16] and perform detailed simulation only for those samples to save simulation time. Hence, we would now like to quantify the SEL size overhead for capturing an arbitrary sample of program execution.

To quantify the average SEL size for an arbitrary sample, we broke each program's execution into 100 million consecutive intervals (samples). For a program, we collected a SEL from scratch for each interval. To create a SEL for an interval, we clear all the entries in the pinSEL's data structures (e.g., UserMemState, register values, etc) at the beginning of the interval of execution. This will ensure that the SEL captured for an interval of execution (sample) has sufficient information to replay the program's execution starting at the beginning of that sample. Thus, using a SEL for a given sample we can simulate the sample's 100 million instructions.

Figure 4 shows the average SEL size (without compression) for 100 million instructions of execution for each program. The SEL size shown for a program is an average of the SEL sizes of all 100 million intervals for the program. The results show that on average, we require 4 MB of SEL to capture a program's 100 million instruction sample. We can see that in the worst case, for `equake`, we require 26MB of SEL to capture a sample of program execution (100 million instructions). But as we show in Figure 2, the SEL size required to capture the full execution of `equake` is only 0.54MB. We observe similar results for `mcf`, `wupwise`, `ammp`, `art`, `galgel` and `lucas`.

The reason for this difference is that for an arbitrary sample, we have to log all the data live coming into the sample which are used (through a load) before getting redefined (through a store) during the sample's execution. In the worst case, this could be potentially be every single load executed in an arbitrary sample of execution. In comparison, when we start generating a SEL from the start of execution, the only data that is live to that SEL is the data read from the global data segment and input data files used during execution. All the other data generated by the program itself during its execution are not logged when we are generating one SEL starting from the beginning of execution. This is the reason for the average sample size for a SEL being larger than the size of a single SEL generated for the full execution. A similar observation was made by Bronevetsky et.al. [3] in their design of a checkpoint and recovery system. They choose to create a checkpoint of the application's state during program execution when the amount of live data is smaller, so that the resulting checkpoint size is also smaller.

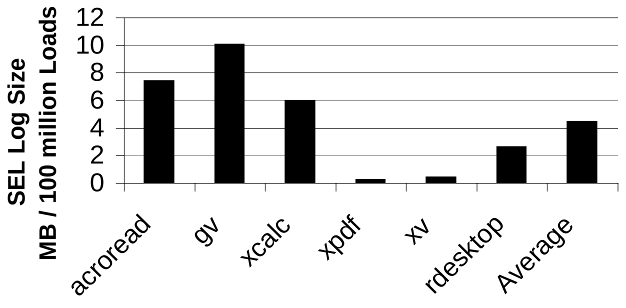

Figure 5: SEL size required to capture 100 million load instructions for interactive desktop applications with compression.

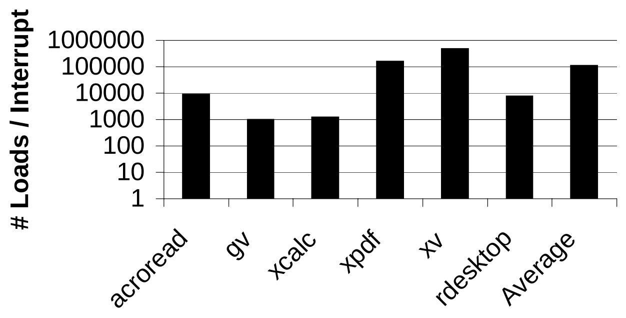

Figure 6: Average number of loads executed between two interrupts (including system calls and asynchronous interrupts).

6.5 Log Sizes for Non SPEC Programs

We also gathered logs for a few interactive desktop applications which we can now simulate easily in x86 SimpleScalar. We executed each of these interactive programs for a few minutes performing some common tasks. Figure 5 shows the average SEL size (with compression) required to capture 100 million load instructions. On average, we require about 4.4MB of compressed SEL to capture 100 million load instructions for these interactive applications. We also show the average number of load instructions executed between two system calls or interrupts in the Figure 6 for these applications. It varies from 1000 load instructions for `gv`, which incidentally also requires the largest SEL size, to about 460,000 load instructions for `xv`, which requires a smaller SEL size.

7. SUMMARY

One of the primary requirements for an architectural performance simulator is the ability to handle interactions with the system through system calls, asynchronous interrupts and DMA transfers. Conventional solutions such as SimpleScalar and pinLIT provide system support by emulating system calls, and they do not provide support to deal with asynchronous interrupts nor DMA transfers.

In this paper we presented an automated logging solution for capturing system effects. We capture the system effects without the knowledge of the semantics of any system interaction. This was accomplished using a binary instrumentation tool to gather system effect logs, which are then used to guide architecture simulation. This approach is very easy to implement and is easy to port to other operating systems and architectures. Previously, SimpleScalar was capable of emulating only 81 system calls, which essentially limited its use to SPEC workloads. But with the help of our pinSEL logging support, it is now capable of simulating any linux application. As a result, any application that spends significant amount of time in application code and user level shared libraries can be evaluated using SimpleScalar.

Our pinSEL tool and a version of x86 SimpleScalar that can consume SEL can be downloaded from:
`www.cse.ucsd.edu/users/calder/sims/`

Acknowledgments

We would like to thank the anonymous reviewers for providing helpful comments on this paper. We would also like to thank Geoff Lowney and the rest of the Pin team for their support. We thank Steven Wallace and C K Luk for help with the Windows and MacOS port of pinLIT, and Mahesh Madhav for his consultation for the userLIT approach. This work was funded in part by NSF grant No. CCR-0311710, NSF grant No. CCF-0342522, UC MICRO grant No. 05-115, and a grant from Intel.

8. REFERENCES

[1] R. Bhargava, J. Rubio, S. Kannan, L. K. John, D. Christie, and L. Klaes. Understanding the iimpact of x86/nt computing on microarchitecture. In *Chapter 10. Workload characterization of emerging computer applications. Kluwer Academic Publishers*, 2001.

[2] M. Van Biesbrouck, L. Eeckhout, and B. Calder. Efficient sampling startup for sampled processor simulation. In *International Conference on High Performance Embedded Architectures and Compilers*, November 2005.

[3] Greg Bronevetsky, Daniel Marques, Keshav Pingali, Peter K. Szwed, and Martin Schulz. Application-level checkpointing for shared memory programs. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 235–247, 2004.

[4] D. C. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[5] H. Cain, K. Lepak, B. Schwartz, and M. Lipasti. Precise and accurate processor simulation. In *In Proceedings of the Fifth Workshop on Computer Architecture Evaluation Using Commercial Workloads (CAECW)*, 2002.

[6] The Transaction Processing Performance Council. Tpc benchmark c: Standard specification. *http://www.tpc.org/tpcc/spec/tpcc current.pdf*, Dec 2003.

[7] J. Emer, P. Ahuja, E. Borch, A. Klauser, C.K. Luk, S. Manne, S. S. Mukherjee, H. Patil, S. Wallace, N. Binkert, R. Espasa, and T. Juan. Asim: A performance model framework. *Computer*, 35(2):68–76, 2002.

[8] J. Lau, J. Sampson, E. Perelman, G. Hamerly, and B. Calder. The strong correlation between code signatures and performance. In *ISPASS*, March 2005.

[9] 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 *Programming Language Design and Implementation*, Chicago, IL, June 2005.

[10] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hllberg, J. Hgberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002.

[11] C. J. Mauer, M. D. Hill, and D. A. Wood. Full-system timing-first simulation. *SIGMETRICS Perform. Eval. Rev.*, 30(1):108–116, 2002.

[12] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *ISCA*, June 2005.

[13] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation. In *MICRO-37*, December 2004.

[14] J. Ringenberg, C. Pelosi, D. Oehmke, and T. Mudge. Intrinsic checkpointing: A methodology for decreasing simulation time through binary modification. In *ISPASS'05*, March 2005.

[15] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod. Using the simos machine simulator to study complex computer systems. *Modeling and Computer Simulation*, 7(1):78–103, 1997.

[16] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *ASPLOS-X*, October 2002.

[17] R. Singhal, K.S. Venkatraman, E. Cohn, J.G. Holm, D. Koufaty, M.J. Lin, M. Madhav, M. Mattwandel, N. Nidhi, J. Pearce, and M. Seshadri. Performance analysis and validation of the intel pentium 4 processor on 90nm technology. In *Intel Technology Journal*, February 2004.

[18] P.K. Szwed, D. Marques, R.M. Buels, S.A. McKee, and M. Schulz. Simsnap: Fast-forwarding via native execution and application-level checkpointing. In *Proc. HPCA 2004 Interact-8: Workshop on the Interaction between Compilers and Computer Architectures*, February 2004.

[19] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *ISCA*, pages 191–202, 1996.

[20] R. Uhlig, R. Fishtein, O. Gershon, I. Hirsh, and H. Wang. Softsdv: A pre-silicon software development environment for the ia-64 architecture. In *Intel Technology Journal*, December 1999.

[21] Roland E. Wunderlich, Thomas F Wenisch, Babak Falsafi, and James C. Hoe. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *ISCA-30*, June 2003.

[22] J. J. Yi, S. V. Kodakara, R. Sendag, D. J. Lilja, and D. M. Hawkins. Characterizing and comparing prevailing simulation techniques. In *HPCA-11*, February 2005.