Simulation and Analysis Engine for Scale-Out Workloads

Nadav Chachmon[†] Daniel Richins[‡] Robert Cohn[†]
Magnus Christensson[†] Wenzhi Cui[‡] Vijay Janapa Reddi[‡]

[†]Intel Corporation [‡]The University of Texas at Austin

ABSTRACT

We introduce a system-level Simulation and Analysis Engine (SAE) framework based on dynamic binary instrumentation for fine-grained and customizable instruction-level introspection of everything that executes on the processor. SAE can instrument the BIOS, kernel, drivers, and user processes. It can also instrument multiple systems simultaneously using a single instrumentation interface, which is essential for studying scale-out applications. SAE is an x86 instruction set simulator designed specifically to enable rapid prototyping, evaluation, and validation of architectural extensions and program analysis tools using its flexible APIs. It is fast enough to execute full platform workloads—a modern operating system can boot in a few minutes—thus enabling research, evaluation, and validation of complex functionalities related to multicore configurations, virtualization, security, and more. To reach high speeds, SAE couples tightly with a virtual platform and employs both a just-in-time (JIT) compiler that helps simulate simple instructions efficiently and a fast interpreter for simulating new or complex instructions. We describe SAE's architecture and instrumentation engine design and show the framework's usefulness for single- and multi-system architectural and program analysis studies.

CCS Concepts

•**Computing methodologies** → **Simulation environments; Simulation tools;** *Interactive simulation;*

Keywords

Analysis, instrumentation, transparency, full-system, scale-out, big data, JIT, multicore, multisystem

SAE website: https://software.intel.com/en-us/intel-sae-sdk

This research was funded by the U.S. Government. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the U.S. Government.

ICS '16, May 29-June 02, 2016, Istanbul, Turkey
© 2016 ACM. ISBN 978-1-4503-4361-9/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2925426.2926293

1. INTRODUCTION

The landscape of computing continues to evolve rapidly as do the workloads. Computing workloads have evolved from being simply single- or multithreaded to running across distributed systems, mostly driven by large-scale programming frameworks, such as Spark and Hadoop, that support scale-out applications and analytics. With the emergence and proliferation of these workloads, there is a need for new and robust instrumentation tools that can facilitate deep program introspection without compromising transparency. Characterizing and analyzing scale-out workloads has been challenging, if not impossible, due to the lack of industry-strength tools that can enable fine-grained, transparent program introspection both within and across nodes.

We identify three fundamental requirements for tools to study scale-out workloads. (1) *Instrumentation*: Researchers must be able to build a centralized and comprehensive view of distributed execution. This requires that they be able to write arbitrary tools and control them from a centralized interface. (2) *Full-system*: Datacenter workloads rely on kernel services and inter-process interactions. A tool to study these workloads must capture everything that executes on a processor, both in kernel- and user-space and across processes and nodes. (3) *Transparency*: In a scale-out workload, the instrumentation tool must be transparent not only to the process under study, but also to its interactions with the rest of the system and the rest of the network. A single instrumented system running slower than the rest of the nodes, for example, could compromise transparency because the node interactions would be altered by the changed speed.

We introduce the Intel Simulation and Analysis Engine (SAE)—a system-level dynamic binary instrumentation engine that meets all three requirements. (1) SAE supports fine-grained distributed workload instrumentation with a simple API to create arbitrary analysis tools. It instruments all systems from a single interface, enabling a coherent view of distributed execution. (2) At its heart, SAE is a processor model that simulates instruction execution in the context of a full or distributed system. Hence, it is not limited to user-space exploration, instead capturing literally all activity on a CPU, including even kernel, driver, and BIOS operations. (3) SAE is unique in being built atop a mature virtual system platform: Wind River Simics. Consequently, SAE resides entirely in the host machine's space: it uses none of the virtual machines' memory space, nor does it change the progress of time within a virtual machine. And because all the systems are controlled from within SAE, there is no relative slowdown between any two machines.

Simulating unmodified scale-out workloads demands high speed. In order to achieve good performance, SAE combines a lightweight just-in-time (JIT) compiler—for simulating simple, frequent instructions efficiently—with a fast interpreter—used for infrequent and complex instructions. It incorporates numerous optimizations and manages sufficient speed to boot unmodified operating systems and execute large distributed workloads (e.g. Hadoop).

To enable powerful and flexible instrumentation, SAE's instrumentation interface allows for plug-in tools which can be loaded and unloaded dynamically while simulation is active. Tools are developed in C/C++ and they programmatically describe what to instrument and the analysis actions to perform during execution. The instrumentation API strongly decouples the tool implementation from SAE's internal implementation, allowing the developers to focus on their task.

The vast majority of scale-out workload studies rely on hardware performance counters [11, 16, 26, 20, 14]. While the insights gathered from performance counters can be informative, they are typically limited to existing platforms. Hardware performance counters do not facilitate architecture design space exploration (e.g. better cache designs, prefetch engines, interconnects) and software program analysis (e.g., information flow, memory alias analysis, program slicing), as both of these tasks require instruction-level program tracing coupled with the flexibility to develop customizable tools. While tools such as Pin and derivatives of DynamoRIO support fine-grained instrumentation, they fall short on transparency and full-system instrumentation.

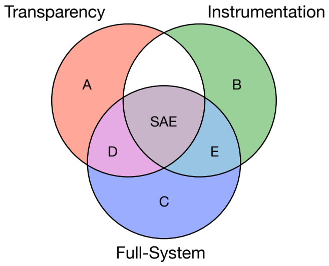
In summary, SAE offers capabilities that no other single tool can offer. It enables easy but powerful instrumentation (like Pin) for the entire software stack (like gem5 [4]) while maintaining perfect transparency (like QEMU [1]). SAE combines these attributes and not only allows for hardware and software exploration within a system but extends that capability to workloads distributed across systems.

The paper is organized as follows. Section 2 motivates the need for SAE. Section 3 presents an overview of SAE. Section 4 gives details about the instrumentation system and describes the CPU- and OS-related instrumentation interfaces that users can use to build custom program analysis tools. Section 5 describes the underlying hybrid interpreter and JIT compiler technology that enables fast and efficient instrumented simulation. Section 6 evaluates the performance of SAE from a multitude of dimensions and presents our best effort to compare SAE against existing technology. Section 7 presents prior work and summarizes SAE's distinguishing features. Section 8 concludes the paper.

2. REQUIREMENTS

Scale-out workloads present several unique challenges for researchers. The fact that they represent a single workload executing concurrently across multiple machines means that no single part of the execution can be studied in isolation. We posit that a research tool must provide instrumentation capabilities, support full-system analysis, and guarantee transparency to study these applications (see Figure 1).

Instrumentation Providing instrumentation APIs is not a new concept—Pin, Valgrind, and DynamoRIO all provide this capability—but in providing instrumentation to support a scale-out workload, we refine the definition of instrumentation: a framework must provide a single unified point of instrumentation control. This would allow for capturing a single coherent picture of the distributed execution. This definition means, for example, that simply launching multiple instances of Pin on each of the nodes in a datacenter does not qualify. Such an approach lacks the single coherent picture of distributed execution, instead forcing post-processing to precisely align distributed events.



Group	Examples
A	Performance counters
B	Pin, Valgrind, DynamoRIO
C	gem5 FS
D	QEMU, SimFlex, BOCHS, QTrace, DECAF
E	PinOS, DrK

Figure 1: SAE satisfies all three fundamental requirements (transparency, programmable instrumentation, and full-system view) for studying scale-out workloads.

Full-System A research platform must also support full-system analysis. Gone are the days when virtually all code of interest was restricted to user-space execution; modern programs, especially in datacenters, rely heavily on kernel-space services. Ignoring that aspect of program execution could miss as much as 60% of the committed instructions [11, 14]. Hence, these workloads must be studied in their entirety, using a tool that can capture the full execution stack.

Transparency We define transparency to be the property that a framework has *zero impact* on the system under study; this precludes, among other things, sharing a target application's memory space, slowing the application's execution relative to the operating system and other processes, and changing the timing of the target application relative to other systems on the network. Our definition of transparency expands upon that used by previous projects. Pin and Valgrind, for example, claim transparent operation because the application under study requires no modifications to operate under each tool. With our expanded definition, however, we see where transparency breaks down: these tools commandeer a portion of the application's memory space and, more importantly, they alter the progress of the application relative to the rest of the system (operating system interrupts, for example, are not slowed to match the slowdown in application progress) and relative to other systems on the network.

As Figure 1 and the table associated with it show, there are a number of "open source" research tools. But we believe that no single existing tool fully satisfies all three requirements for faithfully studying scale-out workloads. In Section 7, we explore these more fully and discuss the various shortcomings of prior work, particularly relating to transparently introspecting scale-out workloads.

3. ENTER SAE

SAE lies at the intersection of the three requirements for a system to analyze scale-out applications. It is an instrumentation engine. SAE allows users to easily create custom tools capable of everything from ISA extensions to security analyses. SAE is effectively a CPU simulator for Simics, enabling instrumentation of *everything* that executes on the processor, regardless of OS (or even lack of an OS). SAE

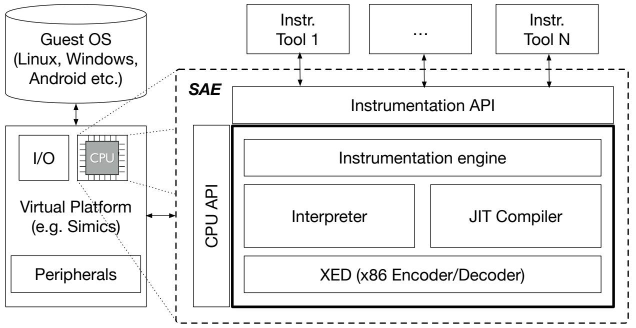


Figure 2: Overview of the system and how the SAE CPU model interfaces with a virtual platform to support full-system instrumentation via an instrumentation interface.

is perfectly transparent: because it resides fully in the host machine's memory space, SAE and its instrumentation tools cannot encroach on the system under study. Furthermore—and very importantly—because SAE is a system simulator, any slowdown it imposes is reflected only on the host and not in the guest. Because SAE is even a *multi*-system simulator, time progresses at the same rate across all the systems so that even if one system is slow (because of instrumentation), that slowdown is invisible to the simulated datacenter.

SAE is a functional x86 CPU, fitted with instrumentation capabilities that achieve high-performance using an intelligent combination of an interpreter and just-in-time compiler. Figure 2 shows how it provides a high-level instrumentation interface that is implemented using an interpreter and compiler, using the x86 XED [9] encoder/decoder. In addition to serving as the computation engine of the virtual platform (i.e., simulating instructions), SAE modifies the execution's behavior according to user-defined instrumentation tools. SAE has a powerful set of APIs that allow users to have full architecture and program state access at instruction granularity. It can also provide information at a coarser grained per-core and per-system level. Moreover, it supports multiple concurrent instrumentation tools, and it is also capable of instrumenting multiple systems.

The SAE instrumentation engine is designed specifically to address an important problem present in instrumenting scale-out workloads: how to observe and modify a program's behavior with an external instrumentation agent without using the observed system's memory or other resources [6]. The widely used Pin [19] and DynamoRIO [5] tools share vital resources with the instrumented target application and hence could compromise system robustness and transparency. For instance, Pin's compiler and code cache is injected into the instrumented application's virtual address space. Often, this can result in virtual memory conflicts between the application and the instrumentation system. In practice, we observe large applications, such as the Oracle database engine, using hard-coded virtual addresses for caching and shared global memory (across multiple threads and processes). Pin and DynamoRIO's solutions to this problem are OS-specific, requiring custom effort and solutions across different OSs.

The instrumentation engine solves the shared resource problem using two approaches. First, the system uses the virtual platform to enable accurate execution of the program. Second, the system enables instrumentation as an external agent. To enable full-system instrumentation, SAE is integrated into a virtual platform (e.g., Simics) and can boot an unmodified guest OS. The external agent communicates with SAE and introduces no modifications to the guest OS's behavior—there is no need to share resources between the guest program and the external instrumentation agent as they reside in completely isolated and different spaces.

As an example use case, Figure 3 shows the source code for a simple instruction counting tool that collects the distribution of instructions executed across different cores on multiple systems. We focus our discussion here on the basic instrumentation interface and delay the explanation of the multicore and multisystem instrumentation until later. Lines 46-56 register a runtime callback function. The registered function (`before_ins_exe`) is set to be called before each instruction's execution (line 48) and increments a running instruction count (line 20). It is tracking instruction execution at a per-system and core-level. Lines 58-67 register a finalization callback function to be called when simulation terminates. The function (`ztool_fini`) concludes the tool's work by printing the final instruction count (line 30).

For each callback, SAE exposes a handle as well as getter and setter functions. It does not expose its internal data structures, thus decoupling the internals from the tool interface. For example, the `ztool_state_handle_t` (line 15) can be used to observe and modify current register state and current memory state. In addition to memory and instruction notifications, SAE can be notified upon interrupts and exceptions. Users can analyze interrupt and exception triggers as well as service routines, which we discuss later.

4. SAE INSTRUMENTATION ENGINE

SAE provides a rich set of instrumentation capabilities at the CPU level for studying large and complex real-world scale-out workloads. We summarize the instrumentation events that the engine provides to tool writers, starting with event notification types (Section 4.1). Because SAE supports multiple active instrumentation tools, challenges arise that must be solved to ease the programming burden (Sections 4.2 and 4.3). The engine also provides OS-level instrumentation interfaces for tracking OS-level activity (Section 4.4). The engine supports multicore and multisystem instrumentation (Section 4.5), which is useful for studying multithreaded and distributed large-scale applications (e.g. MapReduce), and we explain how that is enabled in SAE.

4.1 Instrumentation Events

To study the workloads with varying levels of granularity and flexibility, the instrumentation engine can deliver different event notification types to the tools during execution. Broadly, these are categorized into (1) configuration events, (2) runtime events, and (3) finalization events.

Configuration events These events are triggered when there is a change in the simulated machine's state. When a new tool is loaded into the simulation the `ztool_init()` function is invoked in the tool (line 34 in Figure 3), providing notification that the tool has been loaded. Within the scope of this function, a tool can register to receive other configuration, runtime, or finalization event callbacks.

A key and novel feature of the configuration event type is the dynamic configuration mechanism, which can enable or disable callbacks dynamically using a special configuration key—for example, entering a new phase that requires switching from lightweight to heavyweight instrumentation.

```
1  #include <iostream>
2  #include "ztool-api.h"
3  using namespace std;
4
5  struct system_data_t {
6      ztool_handle_t zhandle;
7      unsigned int num_cores;
8      unsigned long *icount_before;
9  };
10
11 ztool_system_data_key_handle_t ztool_key =
12             ZTOOL_SYSTEM_DATA_KEY_HANDLE_STATIC_INITIALIZER;
13
14 // Analysis function. Called before instruction execution
15 void before_ins_exe(ztool_state_handle_t handle, void* data)
16 {
17   void* v2 = ztool_state_get_system_data(handle, ztool_key);
18   system_data_t* sd = reinterpret_cast<system_data_t*>(v2);
19   unsigned int core_num = ztool_state_get_core_num(handle);
20   sd->icount_before[core_num]++;
21 }
22
23 // Fini function. Call at program termination or shutdown
24 void ztool_fini(ztool_fini_handle_t handle, void* data)
25 {
26   void* v = ztool_fini_get_system_data(handle, ztool_key);
27   system_data_t* sd = reinterpret_cast<system_data_t*>(v);
28
29   for (unsigned num = 0; num < sd->num_cores; num++)
30     cout << "INST COUNT = " << sd->icount_before[num] << endl;
31 }
32
33 // Init function. Called by SAE when tool is first loaded
34 extern "C" void ztool_init(ztool_init_handle_t handle)
35 {
36   system_data_t* sd = new system_data_t();
37   ztool_handle_t zhandle = ztool_init_get_tool_handle(handle);
38   sd->zhandle = zhandle;
39   sd->num_cores = ztool_init_get_core_count(handle);
40   sd->icount_before = (unsigned long *)
41     calloc(sd->num_cores, sizeof(unsigned long));
42
43   ztool_system_alloc_data_key(zhandle, &ztool_key);
44   ztool_system_set_data(zhandle, ztool_key, sd);
45
46   { // Instruction execution callback registration
47     ztool_instruction_exe_desc_t desc;
48     desc.when = ZTOOL_INSTRUCTION_WHEN_BEFORE;
49     desc.fn = before_ins_exe;
50     desc.data = NULL;
51     desc.order = ZTOOL_CB_ORDER_DEFAULT;
52     desc.config_key = ztool_init_get_default_config_key(handle);
53     desc.zhandle = ztool_init_get_tool_handle(handle);
54
55     ztool_instruction_exe_register_cb(&desc);
56   }
57
58   { // Simulation fini callback registration
59     ztool_fini_desc_t desc;
60     desc.fn = ztool_fini;
61     desc.data = NULL;
62     desc.order = ZTOOL_CB_ORDER_DEFAULT;
63     desc.config_key = ztool_init_get_default_config_key(handle);
64     desc.zhandle = ztool_init_get_tool_handle(handle);
65
66     ztool_fini_register_cb(&desc);
67   }
68 }
```

Figure 3: Instrumentation tool code for counting the total number of instructions executed. The tool tracks the total instructions executed by each core and system.

A tool can control its callbacks dynamically by allocating a new configuration key and setting its initial state—enabled or disabled—during callback registration. Whenever the tool determines it has to toggle the callbacks' state, it issues a dynamic configuration event that enables or disables the configuration keys accordingly. Dynamic reconfiguration is controlled using APIs, such as `ztool_config_enable_key()` and `ztool_config_disable_key()`. All instrumentation callbacks require a key (lines 52 and 63 in Figure 3). The system provides a default key for convenience.

Runtime events These events are delivered to the tool when execution encounters a specific architectural event such as instruction execution, memory access, interrupts, and exceptions. Upon event delivery, the tool can query per-event data. For example, on a memory access, the tool can query the virtual, linear, and physical addresses of the memory access. It can also query the root cause of the memory access, i.e. whether the access was caused by a program instruction or on behalf of an instruction—such as a page walk while accessing the instruction. The tool can also query architectural register state during the event, including general purpose registers, vector registers, control registers, etc.

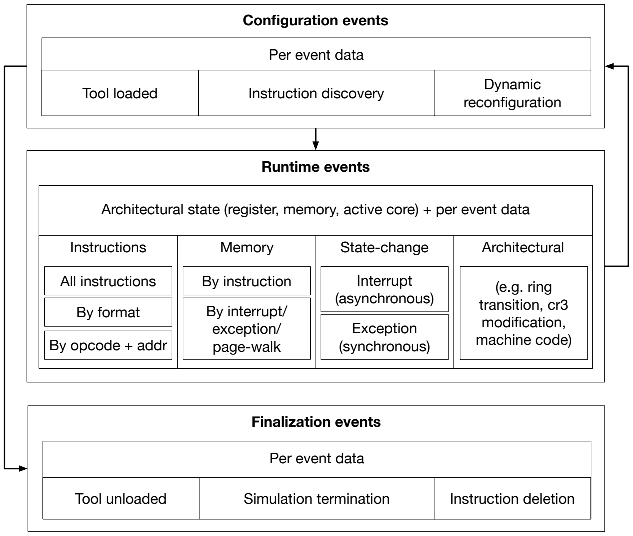


Figure 4: Tool developers can register for three event notification types. Configuration events allow developers to request notifications when SAE detects new state (e.g., instruction discovery). Runtime events allow developers to request notifications about the simulation's runtime activity (e.g., memory accesses). Finalization events allow developers to request notifications when anything is invalidated from the simulation (e.g., termination).

Finalization events These are the simplest of events and they are delivered to the instrumentation tool to enable the tool to deallocate memory or summarize its analysis. Finalization events are delivered when the tool is unloaded from the simulation or when the simulation terminates.

4.2 Precise Event Callback Delivery

SAE can be used for microarchitectural analysis and architectural exploration of complex workloads that exercise the full system stack, including the user space and kernel space code, and as such it is important that SAE faithfully mimic and inform the users about the underlying processor activity at the instrumentation interface level. Existing tools, such as Pin, fail to faithfully trace full system activity. Memory accesses, for instance, caused by page walks are not observed by Pin because it does not instrument the kernel, and also because it does not simulate the CPU behavior.

Many different events might be associated with a single instruction. For example, a user might request callback notifications for before and after an instruction's execution, for memory accesses performed by the instruction, and for exceptions triggered by the instruction. Table 1 shows examples of event delivery order. We use five commonly executed x86 instruction types to illustrate the ordering of the event notifications. Within a SAE tool, it is possible to uniquely attribute the root cause of events. We can trace when a hardware interrupt event triggers a memory access (e.g.,

Instruction #1 (*e.g.* `mov r11, rdx`)	**Instruction #2** (*e.g.* `mov rax, [r8+r9*8]`)	**Instruction #3** (*e.g.* `div ecx`)
○ Before instruction ○ After instruction	○ Before instruction □ Before memory access □ After memory access ○ After instruction	○ Before instruction ☒ Interrupt start □ Before memory access □ After memory access ☒ Interrupt ready

Instruction #4 (*e.g.* `inc r9`)	**Instruction #5** (*e.g.* `sub [rcx+r9*8], r11`)	**Hardware Interrupt**
○ Before instruction ○ After instruction	○ Before instruction □ Before memory access □ After memory access ○ After instruction	☒ Interrupt start □ Before memory access □ After memory access ☒ Interrupt ready

Table 1: Example order in which event notifications are delivered. Nesting of events within one another shows that other event types may be triggered due to an ongoing event. Depending on the tool's instrumentation objective, faithful event-order delivery can be essential, especially if the tool is extending the ISA functionality.

page walks). The tool developer has the power to decide whether these nested event notifications should be delivered or masked away (i.e., ignored). The order in which the notifications are delivered is enforced for all the instructions to faithfully represent program execution on a real CPU.

4.3 Multiple Event Callbacks and Tools

In a production environment, when doing complex analysis on scale-out programs, it is useful to break down the instrumentation analysis into separate event callbacks and tools. Developing in such a way allows for the instrumentation to be composable and software to be manageable. However, it introduces correctness issues that SAE addresses.

When a tool is loaded, SAE flushes its instruction code cache to allow the tool to register its own runtime callback events. Otherwise, the old instrumented code will be executed. SAE flushes the caches again when the tool is unloaded to remove all previously registered callbacks notifications. Tool loading and unloading is expected to be rare, so the performance penalty for flushing the caches is minimal.

Support for multiple event callbacks and tools becomes a challenge when one or more callback(s) modify the simulated CPU state—for example, a callback that implements a new ISA extension as part of architectural exploration. There are two important mechanisms that SAE uses to provide robust support for multiple event callbacks and tools: (1) control of callback notification order and (2) propagation of architectural memory accesses generated by tool callbacks.

Callback notification order When multiple instrumentation tools are loaded, the tools must be able to notify one another. For example, a tool may modify a register value, and this change must be made visible to the other tools to ensure all tools see consistent machine and program state. Though it is not usually the case, the notifications may need to be delivered to the tool(s) based on the order in which they were registered at configuration time.

SAE supports callback ordering both across tools as well as in the same tool (where multiple callback notifications are registered for same event) by exposing a new data structure. Using the `ztool_cb_order_enum_t` enumerated type (provided in the callback descriptor of runtime events), a tool developer can control the relative order of different callbacks. A tool that does not modify any CPU state values—and most do not—would be provided with a default value (`ZTOOL_CB_ORDER_DEFAULT` on lines 51 and 62 in Figure 3).

Propagation of memory accesses When a callback implements new CPU functionality, such as one that performs architectural memory accesses (recall that SAE allows the developer to extend the capabilities of the CPU for architectural studies), other memory access callbacks should be notified. For instance, one tool could implement a new instructions set architecture (ISA) extension that affects memory accesses, while another tool implements a cache simulator. Any of the memory accesses performed by the ISA extension tool must be closely reflected in the cache tool.

To support such scenarios, SAE provides special instrumentation interfaces for architectural read, write, and read-modify-write memory accesses. The memory access interfaces first verify that the memory address is accessible (and issue an exception if not) and provide a memory access buffer to the user. For a read operation, the buffer is filled with the required data and provided to the caller. For a write operation, the buffer is filled by the tool and committed to memory. For a read-modify-write operation, the tool gets the required data, modifies the data, and commits it. In this way, the specialized interfaces for architectural memory accesses *automatically* notify other registered memory access callbacks about the memory accesses performed by the tool.

4.4 Operating System Events

Scale-out workloads tend to exercise the operating system heavily, as much as 20% to 50% of their execution time in the kernel [14, 16]. Context switches involving threads and processes, thread creation and destruction etc. occur frequently and put pressure on the OS that can have significant impact on the architecture [26]. Therefore, it is important to study the OS behavior of scale-out workloads. A key challenge lies in understanding the idiosyncrasies of each OS to know how to track process and thread interactions with the OS.

To ease the burden on programmers, SAE's instrumentation engine provides high-level abstraction APIs for OS-level notifications. For example, a tool can register for context-switch notifications. These OS-level notifications are provided by the "OS awareness" instrumentation system that is built on top of the CPU notifications described thus far.

OS awareness obviates the need for a user to hold intimate platform- and OS-specific knowledge. For example, using only CPU-level notifications to track context switches would require understanding the `cr3` page table register. A change in the `cr3` register indicates a context switch in the hardware. However, tracking such low-level changes requires intimate domain knowledge. OS awareness hides these low-level idiosyncrasies and automatically calculates the context-switch address based on symbolic information and registers an instruction execution callback notification. When the instruction execution notification is delivered to the OS awareness tool, it triggers the context-switch event in the tool.

The OS awareness module exposes a set of core OS-level events that hook directly into the underlying kernel functions. Direct hooking enables accurate interceptions. OS-level events include tracking process creation, thread switch-

Process	OS-level Event
Start	Process creation Thread creation
Active	After task switch (from another process to this process) Program load (i.e., `exec()` like behavior) Image load (e.g., `libc`) Function entry (e.g., `malloc()`) Function exit (e.g., `malloc()`) Before task switch (from this process to another process) ... another process is running After task switch (from another process to this process) Thread creation (i.e., multi-threaded process) Thread destruction (i.e., multi-threaded process) Before task switch (from this process to another process)
End	Thread destruction Process destruction

Table 2: The OS awareness module supports various OS-level event notifications to ease programmer burden, and the table shows the runtime event delivery order for the various OS-level events. Some events, such as thread destruction and task switch, may trigger more than once during execution depending upon program behavior.

ing, and more. The module also facilitates querying OS-level data structures, providing access to, for example, process ID and thread ID in the scope of a context switch.

The OS awareness module supports three different task-related event types and they are as follows: (1) process-start-related events, (2) process-active-related events, and (3) process-end-related events. Table 2 summarizes the various OS event types and captures the event delivery order. Process-start events can track, for example, process creation in the OS. Process-active events monitor changes in a process's behavior, such as library loading, context switching, and thread creation and destruction. Process-end events can track process termination. Beyond these three main event types, there are additional symbolically oriented events that allow users to query symbols in an image and register function execution events at those symbols' addresses.

Since the OS awareness subsystem is OS-specific, we currently only support the Linux kernel. OS and instrumentation *tool* developers can extend support to include additional operating systems, including Windows, using SAE APIs.

4.5 Multiple Cores and Multiple Systems

Scale-out applications almost always rely on some form of parallelism, be it multi-threaded or multi-process execution, and moreover they leverage multiple systems to do their processing [18]. Studying such workloads requires an instrumentation engine that can capture all of that behavior in a unified fashion. SAE is designed specifically for studying such workloads. It can capture fine-grained instruction level instrumentation for a distributed workload, running across multiple systems and on multiple cores (within each of those systems), using a single unified instrumentation interface.

Virtual platforms enable support for multiple cores that are part of the same system. SAE leverages this feature to support multicore-aware instrumentation. During initialization, the instrumentation engine informs the user of the active core count to enable allocating per-core data structures (line 39 in Figure 3). During runtime, for each runtime event, the tool can query the initiating core and perform the appropriate instrumentation analyses (lines 19-20).

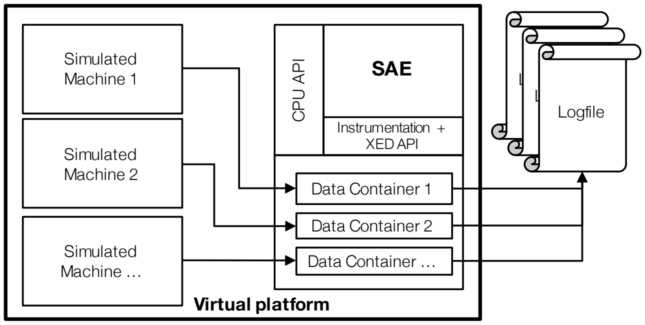


Figure 5: Multisystem instrumentation capability using SAE, as shown previously in Figure 3. Users can use one (or more) instrumentation tool(s) to monitor and collect instrumentation data from all the simulated machines.

SAE also supports multisystem instrumentation, which suits scale-out applications like Hadoop and MapReduce. Figure 5 shows how this works. The feature rests on the virtual platform's ability to simulate multiple systems concurrently. The instrumentation engine is designed to support the concurrent systems, but it is up to the tool developer to ensure thread safety; that is, since each system is simulated on a separate host thread, the tool developer must ensure, for example, that data structures are private to each system or, if shared, properly protected during access. This is no different than the thread safety guarantees in current user-level production systems such as Pin. During tool initialization, the instrumentation engine allows the developer to allocate system-specific data (labeled as "Data Containers" in Figure 5) that can be filled with any data structures that are needed (e.g., counters to hold system-specific data).

Leveraging the Wind River Simics platform, SAE simulates multiple systems using concurrent threads on the host machine. Hence, multisystem simulation can easily scale up to the number of cores on the host. It can continue to scale even beyond the resources of the host, though it may be subject to performance penalties (Section 6.2). Even when exceeding its core count, simulation on a single host works quite well because Simics can collapse idle time, only spending simulation cycles on active work. As the load factor for distributed systems is often rather low, the overhead for simulating such large systems is surprisingly low. In addition, the Simics/SAE combination allows for different simulation modes at core granularity so only the actual system under study will incur more overhead compared to the fastest execution mode. Multicore simulation within a single SAE simulated system is limited to one host thread. The simulated cores are time-multiplexed on the host thread corresponding to that system. Similar to the multisystem case, Simics can collapse idle time even within a system for idle cores.

Referring back to Figure 3, we show how the instruction counting example supports multicore and multisystem simulation. Lines 11-12 and 36-44 configure the tool for multisystem instrumentation. Lines 17-18 provide access to the system-specific data containers, and, similarly, lines 26-27 extract the statistics gathered from each system at the end of the program. Within each system, (line 19), the tool can query the originating core for each instruction. In summary, *the tool requires only 13 lines of new code to support and capture fine-grained multicore and multisystem information.*

5. DESIGN AND IMPLEMENTATION

SAE's instrumentation interfaces are enabled by an interpreter and just-in-time (JIT) compiler. We describe and motivate the design of and interaction between the interpreter and the compiler (Section 5.1). We present our interpreter design and optimizations (Section 5.2). We also present the compiler's design and implementation (Section 5.3).

5.1 Overview

The interpreter is a complete functional simulator. The SAE interpreter provides high performance and can simulate on the order of tens of million of instructions per second (MIPS). When the JIT compiler is active, performance improves to hundreds of MIPS. The JIT compiler accelerates execution by generating x86 code at runtime to simulate instructions. The generated code improves performance by reducing the overhead of simulated instruction dispatch, maintaining a one-to-one relationship between simulated instructions and host instructions, and keeping simulated registers in host registers.

The compiler only compiles the frequently executed instructions, and falls back to the interpreter for complex, less-frequent events, such as interrupts and exception handling. Moreover, in SAE we introduce a *hybrid mode* to deal with the case that frequently executed code cannot be JITed. A hot piece of code might, for example, rely on instructions that are not supported by the physical hardware. In this case, the JIT compiler drops into the hybrid mode, producing code that jumps directly into the appropriate interpreter routines without the overhead of leaving the code cache and fully invoking the full interpreter. In this unique manner, the JIT can support instructions that are not supported physically on the hardware. For example, the JIT compiler can support AVX2 instructions on a machine that does not physically have AVX2.

The hybrid mode provides significant improvements for some workloads since it obviates the need for costly switches back and forth between the interpreter and JIT compiler and confines execution to the code cache for as long as possible.

5.2 Interpreter Design and Implementation

The interpreter's structure closely mirrors the implementation of a non-pipelined, single-issue CPU. It is broken down into two parts: the front-end and back-end. The front-end fetches and decodes instructions then advances the instruction pointer for the following instruction. The decoded output is called SIIS (static instruction information structure) and is analogous to microcode; it is an orthogonal representation of the instruction that is straightforward to execute. An x86 instruction operand size depends on mode and prefixes. The SIIS contains the effective operand size. The size and signed-ness of immediates can vary. The SIIS contains the immediates sign/zero extended to 64 bits. The addressing modes of x86 are expressed in a canonical form:

```
displacement + base + index * scale.
```

Operands can come from a register, memory, or immediate. A simulation function specific to the instruction operation and operands is selected. The back-end dispatches execution to the simulation function. It reads operands from registers or memory and writes back the updated register or memory values according to instruction semantics. Most simulation functions are simple because the irregularity of x86 is factored out by the SIIS. After execution, control passes back to the front-end to fetch the next instruction.

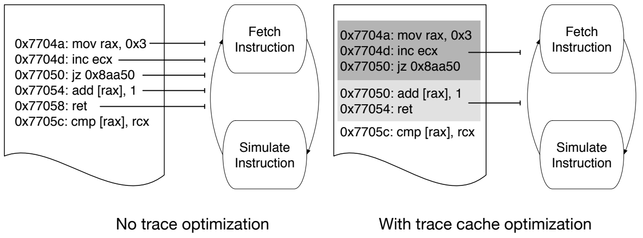


Figure 6: Trace-cache optimization reduces fetch count by about 40%. The instruction fetch count in the above example is reduced from five to two fetch sequences.

In addition to implementing conventional optimizations, such as lazy flags [23], we implement new optimizations that are unique to accelerating full-system simulation. The principal rule is to make the common path fast. The interpreter optimizations are categorized into *front-end optimizations*, which focus on fetching and decoding instructions, and *back-end optimizations*, which focus on acquiring values from registers or memory and executing the instructions.

5.2.1 Front-end optimizations

The front-end of the interpreter is responsible for fetching and decoding the x86 instruction byte stream. As we described earlier, its input is an instruction pointer and its output is a handle to the SIIS, which is passed to the back-end of the interpreter. A naive and straightforward implementation takes approximately 1000 cycles, with most of the overhead coming from decoding an instruction and the rest coming from address translation and copying the instructions from memory. Such a naive interpreter implementation would limit our peak simulation speed to a 1000× slowdown over native execution, which is in the order of 3 MIPS.

To reduce its overhead, SAE's interpreter exploits locality in the executed code stream and caches address translations. It exploits both *temporal* and *spatial locality.* Temporal locality comes from executing the same instruction multiple times. Spatial locality comes from executing the same sequence of instructions repeatedly. Both are exploited by our *trace caching* optimization, as shown in Figure 6, which reduces the overhead of the frequently performed task.

Temporal locality Trace caching improves the interpreter's performance by storing the result of previous mappings from the physical address to the SIIS in a hash table. The optimization works especially well if there is strong temporal execution locality in the code because a miss in the physical cache requires an expensive decode. We determined that it is worthwhile to use a complex lookup mechanism in order to reduce any hash table misses caused by conflicts.

We implemented the hash table with physical indexing. The major benefit of using a physically indexed trace cache is that it does not need to be invalidated when there is an address space change, something that occurs frequently in a full-system simulation environment. Moreover, the trace cache can be shared across cores. We heuristically determined that a 4-way set associative cache with 64K entries combined with the least recently used (LRU) replacement policy performs best and implemented that configuration.

Spatial locality The trace caching optimization also exploits spatial locality by retrieving a sequence of SIIS with a single lookup. The physical cache entries contain a trace instead of a single SIIS. Fetching an entire instruction trace amortizes the overhead of the cache lookup over the execution of several instructions. For instance, in Figure 6 the number of interpreter fetch, decode, and dispatch lookups is reduced by 40%. Lookups drop from five (one per instruction) to two (one per trace). Traces are terminated whenever an instruction modifies the `RIP` (the x86 register holding the instruction pointer): when it is modified, we cannot safely determine the next instruction. Traces are also terminated when they become long (>10 instructions).

5.2.2 Back-end optimizations

After an instruction is fetched and decoded, the interpreter's back-end performs the heavy-lifting tasks of fetching the operands from registers or memory, performing the actual operation, updating condition code flags, and storing the result to memory or registers. Because each instruction has different logic, it is hard to optimize each instruction manually. Moreover, optimizing for every instruction will usually yield little benefit. We instead focus on operations that are common to all instructions—the interface to memory, address translation, and condition code evaluation—where performance is key.

Direct memory interface When a simulated instruction references memory, SAE may need to invoke the virtual platform to complete the operation—a costly operation. The address may be for system memory or to a peripheral device for memory-mapped I/O. To improve performance, SAE asks the platform for a pointer to the simulated memory (called a host pointer or direct memory interface). If the address corresponds to system memory, the platform returns a pointer to the simulated memory and SAE uses it to directly access the memory. As a further optimization, we cache the host pointers in SAE for future references to the same memory page. For a request to memory-mapped I/O, the platform returns an error, forcing SAE to call out to the platform for every memory operation, passing the address and data pointer—a highly costly operation.

Translation lookaside buffer (TLB) Address translation is the most common operation performed by the CPU. Besides translating the linear instruction pointer to a physical address before fetching each instruction, many instructions operate on data from memory, requiring another address translation. The trace cache described in Section 5.2.1 eliminates most translations for instruction fetch, leaving data access references to be translated by the back-end.

Although a hardware TLB translates linear addresses to physical addresses, SAE's interpreter TLB must translate to both physical addresses and host pointers. We implement a software TLB to efficiently translate addresses from linear to both physical and host addresses. While the x86 ISA supports multiple page sizes, a software TLB implementation of variable page sizes would require an expensive lookup function. Instead, SAE TLB entries only map 4K pages in the address space. The number of SAE TLB entries used for a large page depends on how much of the page is referenced.

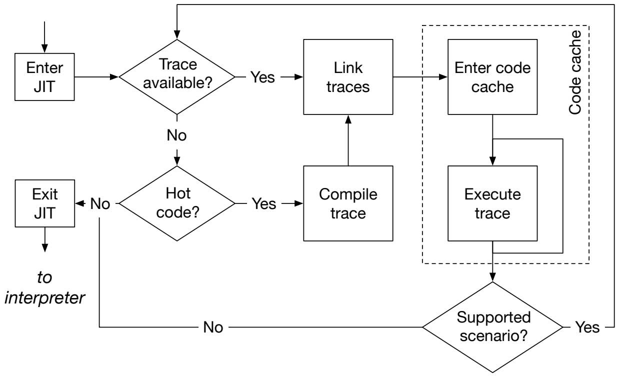


Figure 7: Overview of the SAE's JIT compiler. It is an optimizing compiler designed to improve the performance of frequently excuted code under steady state.

Although the address translation algorithm itself is typically straightforward, caching the results efficiently to allow fast translations is important. Several checks must be performed for every access: (1) check if a cached translation exists, (2) validate permissions, and (3) detect page crossings that require cross-page memory accesses. SAE performs all three checks in a single operation to minimize overhead.

The physical instruction code cache must be invalidated whenever any x86 code is overwritten in system memory. We use SAE's TLB to detect when that occurs. When an entry is filled in the code cache, the write permission for the page containing that code is removed from the TLB. Any subsequent write access that fails the above check causes SAE to invalidate entries in the code cache that correspond to that particular page, thus guaranteeing that the instruction code cache is alway consistent with the system's memory.

5.3 JIT Compiler Design and Implementation

The second part of SAE is the JIT compiler. The JIT compiler performs binary translation of guest x86 instructions to host x86 instructions. Its purpose is to enable efficient execution of the (instrumented) x86 guest instructions on the host with minimal overhead. The compiler assumes that the original guest code is already optimized and therefore does not perform heavyweight compiler optimizations.

The JIT compiler does not provide full instruction coverage. Since the JIT compiler is performance oriented, it does not cover complex and rarely executed scenarios, such as the handling of exceptions or the execution of privileged instructions. Whenever such a scenario occurs, the JIT compiler aborts and lets the interpreter complete the operation.

Code generation Figure 7 shows the internal workings of the JIT compiler. It is a typical industry-strength hot-code optimizing JIT compiler. When invoked on not-yet compiled "hot" (frequently executed) code, the compiler starts by fetching an instruction stream until it finds an instruction that triggers it to stop. Trigger instructions could be branch or unsupported instructions. While fetching the stream, the compiler decodes the instructions and keeps their Intermediate Representations (IR) in an internal trace data structure. To support execution on the host, some additional instructions are inserted. For example, each region embeds a call to a function that checks if the simu-

	Win 7 boot	Fedora 5 boot	bzip2	mcf
JIT+Interpreter	1.0×	1.0×	1.0×	1.0×
JIT only	1.4×	1.2×	1.0×	1.1×
Interpreter only	2.1×	2.6×	6.6×	6.7×

Table 3: Slowdown of isolated SAE components, as compared to running SAE with the interpreter and the JIT.

lated instruction count has exceeded the platform quota. Another example is a call to a TLB function that translates guest address to host address. These frequent functions are tuned and optimized. Once the compiler has embedded the additional instructions into the trace, it performs linear scan register allocation [22]. The compiler generates code to fill guest registers into the physical host registers before guest execution starts; before transitioning from guest back to host instructions, the updated guest registers are spilled into an in-memory spill area. The register management code is added directly into the (unbounded) code cache [8] to improve performance, rather than performing costly world switches in and out of the code cache. The compiler performs direct linking/chaining [23] in the code cache to avoid unnecessary transitions from the code cache to the compiler and vice versa. Indirect branches are handled using a typical fast and efficient lookup table [13], which is updated based on previous target resolution. The lookup table maps from the guest linear address to the host address; it is flushed on any kind of guest address space modification.

Optimizations To strike the right balance between interpretation and compilation, SAE employs a hot-spot detector that predicts when a guest code trace will have frequent subsequent execution. If a hot spot is not detected, the compiler is not invoked. We use heuristics to determine a predefined threshold to decide when we invoke the compiler.

The compiler implements optimizations to reduce instrumentation overheads. For instance, advanced instrumentation capabilities, such as dynamic reconfiguration (described earlier in Section 4.1), is based on *trace coloring* in the code cache. In trace coloring, each trace in the code cache is marked as the combination of all active keys (remember that multiple configuration keys can be active for the same callback, and that any combination between active keys is legal). Whenever a trace is being looked up in the code cache, it will be compared against the currently active key combination. If the key combination matches, the trace will be used. Otherwise, a new trace will be generated, allowing the right callbacks to be registered. Whenever a key is disabled, the generated traces remain in the code cache, and whenever the key is enabled again, they can be reused.

The alternative to trace coloring for supporting dynamic reconfiguration is to flush the entire code cache whenever the instrumentation needs to change. This can impose significant performance degradation, especially for workloads with large working footprints. The compiler would have to re-interpret the code to detect hot spots, and then re-instrument and re-compile the code.

6. EVALUATION

We start by evaluating SAE's JIT compiler and interpreter performance (Section 6.1). Next, we study SAE's overheads to conduct detailed full- and multi-system instrumentation (Section 6.2). We present three use cases on how SAE enables studies that could not be easily performed by existing tools—one from an architecture perspective, another from a program analysis standpoint, and the final one from a scale-out perspective (Section 6.3).

The use cases and evaluation are largely centered on showing what SAE enables, rather than on developing and showcasing new insights—we defer that to our future SAE users. We show that SAE can enable new hardware and software analyses. Although it is difficult to predict precisely how the tool will be used by the community, we expect that some new applications might include multi-system program analysis, fine-grained visibility into operating system services (I/O, disk, etc.) with perfect transparency, and ISA exploration before silicon tapeout. Already it has been used within Intel to develop fast performance simulators, multi-system workload analysis, and simulation of new ISA features.

6.1 Component Effectiveness

Though not all the performance enhancements of SAE can be easily isolated for study, we are able to evaluate the individual contributions of the JIT compiler and interpreter. Table 3 shows the relative (to default SAE operation) execution times of booting Windows 7 (32-bit) and Fedora 5 (64-bit) and of executing *bzip2* and *mcf* from SPEC CPU 2006 when either the JIT compiler or interpreter operates in isolation. Here, we pick *bzip2* and *mcf* because they are representative stress test cases for SAE's two core functionalities: code generation and address translation. *bzip2* stresses the generated code because it is a compute-bound workload; *mcf* stresses address translation because it is a memory-bound workload. Later, we discuss all of CPU 2006.

Disabling the interpreter forces the compiler to compile everything it can (some instructions must still be interpreted) without waiting for hotspot detection. Both operating systems boot quickly in both limited modes. Importantly, boot sequences execute a lot of varied (non-repetitive) code; as such, the interpreter performs well, reflecting its importance. The JIT is faster than the interpreter, and it shows tolerable slowdown compared to the native code. On the other hand, *bzip2* and *mcf*, which heavily reuse code, show virtually no difference between the JIT and baseline operation, while the interpreter incurs a major blow to performance. Together, the JIT and interpreter provide the best performance.

6.2 Overheads

For all its instrumentation capabilities, SAE must operate at reasonable speeds if it is to prove useful. Though SAE is slower than native execution and its instrumentation introduces further overheads, we demonstrate that, without instrumentation, it is less than 1.3× slower than QEMU+BT (using binary translation) and, with instrumentation, between 3.6× and 6.9× slower than QEMU+BT. This is certainly fast enough to enable useful evaluation.

Uninstrumented SAE boots full, unmodified operating systems at noteworthy speeds. Table 4a shows boot times for Linux, Android, and Windows. We compare Simics and SAE (without any instrumentation) against QEMU with direct execution ('QEMU+KVM') and with binary translation ('QEMU+BT'). Both QEMU+KVM and Simics (by default) operate using hardware virtualization. Though all other performance data on Simics and SAE in this paper are measured only on Simics 4.8, Table 4a also lists data for Simics 5, the latest major release. Simics 5 compares favorably

	Native	QEMU	Simics	Simics	QEMU	SAE
		+KVM	4.8	5	+BT	
Ubuntu 14.04.3	54s	7.9s	14.4s	8.5s	99.8s	117.7s
Android 4.4	Failed	10.0s	11.8s	9.0s	100.3s	126.8s
Windows 7	30s	8.6s	14.7s	9.7s	Failed	95.5s

(a) OS boot times (measured on the host machine). Simulation can boot faster than native hardware because the disk images are often cached in the host memory.

	QEMU	Simics	QEMU	SAE	Pin
	+KVM		+BT		
Geomean Slowdown	1.02×	1.79×	31.15×	38.51×	1.28×

(b) SPEC CPU 2006 (`ref` inputs) slowdown relative to native execution. See Figure 8a for the slowdown results that correspond to the individual CPU 2006 benchmarks.

	Pin+	SAE+	Pin+	SAE+
	ihist	ihist	memtrace	memtrace
Geomean Slowdown	7.13×	2.93×	2.89×	5.55×

(c) Instrumentation tool slowdown relative to uninstrumented Pin and SAE execution.

Table 4: SAE performance. We show the geometric mean slowdown of SAE and other tools in executing OS boot, SPEC CPU 2006, and instrumented code using instruction mix tools and memory tracing tools.

with QEMU+KVM, achieving nearly the same speed.

Worth extra attention is the fact that boot times on virtual platforms are often significantly faster than on real hardware. This is largely because virtual platform read virtual disk image data from the host operating system where it is often already cached, while booting on real hardware requires bringing in the data from actual storage. SAE and QEMU+BT both use binary translation as their base simulation technology, and performance is similar with SAE being closely behind QEMU+BT in the boot scenarios.

In program execution, SAE introduces a 1-2 order of magnitude slowdown (38.51×, geometric mean) compared to native execution. Table 4b summarizes the results. The table also includes data for (uninstrumented) Pin, QEMU+KVM, QEMU+BT, and Simics. Simics (using direct execution) and QEMU+KVM show near-native speeds. SAE is only 21.50× slower than Simics. Figure 8a shows the detailed running time breakdown of SAE across all the benchmarks from the SPEC CPU 2006 benchmark suite (using the `ref` input set). The results are relative to native execution. QEMU+BT relative times are also included for comparison.

Instrumentation The addition of instrumentation imposes additional speed penalty in SAE. These results are summarized in Table 4c for two different tools: instruction mix (ihist) and memory tracer (memtrace). These tools exercise frequently used APIs. For reference, we also use comparable tools in Pin. SAE's tools have different slowdowns than Pin (2.93× and 5.55× versus 7.13× and 2.89×).

The slowdown of the ihist SAE tool is the product of SAE's base slowdown (38.51×) and the additional slowdown imposed by the tool (2.93×), or 112.83×. While this seems high, we emphasize that the majority of the slowdown comes from the overheads of binary translation and interpretation, which even QEMU is subject to because it uses similar technology. *Compared to QEMU+BT, the* ihist *and* memtrace *slowdowns are only 3.62× and 6.87×, respectively.* Figure 8b shows the execution times of SAE (relative to uninstrumented execution) with the two tools across the entire SPEC CPU 2006 benchmark suite (using the `ref` input set).

	1 System	6 Systems	12 Systems	24 Systems	96 Systems
Time	108.6s	198.9s	252.4s	446.9s	1697.1s
Slowdown	1×	1.83×	2.32×	4.11×	15.62×

Table 5: Multisystem simulation performance. Until the host's resources are exhausted, the penalty for simulating multiple systems is small. Simulating multiple systems, however, is not limited to the host's resources we use for our experiments. Each SAE system simulated a single-core CPU and 2 GB of RAM. The host had 12 hyper-threaded cores and 160 GB of RAM.

SAE provides more powerful instrumentation insights than Pin (e.g. full- and multi-system visibility). Additionally, SAE is orders of magnitude faster than a full microarchitectural simulator for collecting similar statistics and it can run the programs to completion within a reasonable timeframe.

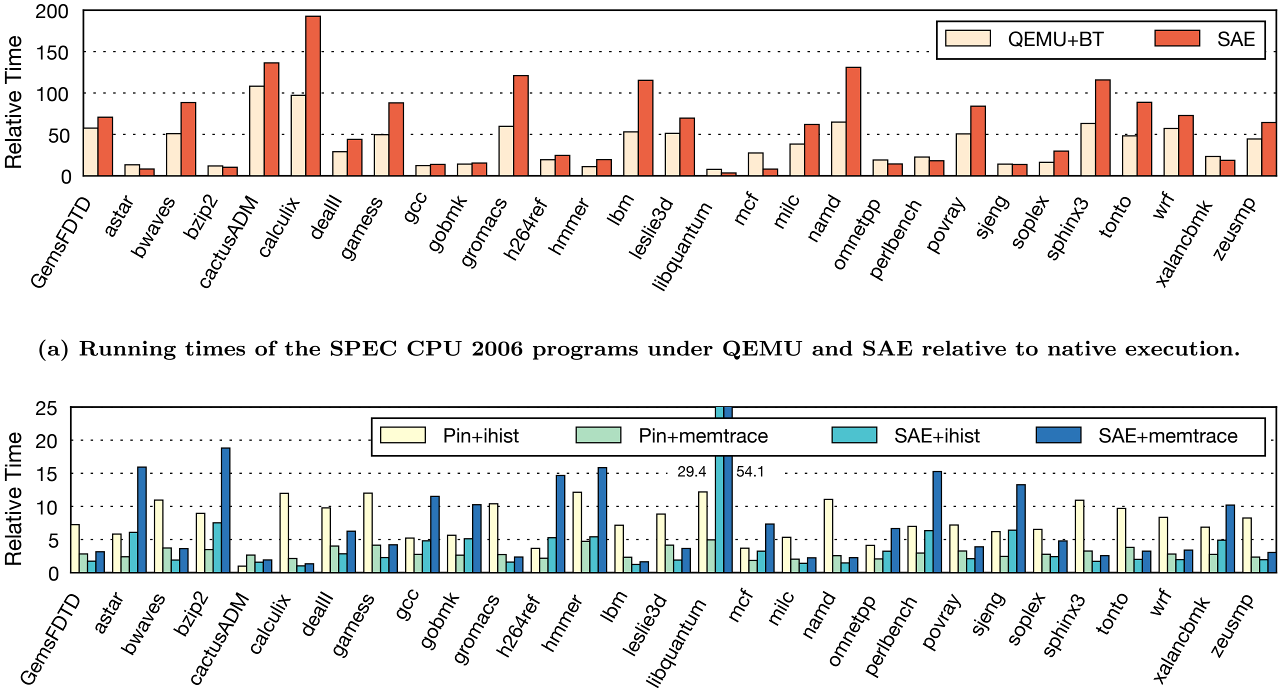
Multicore When simulating a multicore CPU, SAE only uses one host thread, serializing all simulated threads. SAE slowdown scales almost perfectly linearly with the number of *active* cores; i.e., inactive simulated cores have a negligible contribution to simulation time. We verified this using the *swaptions* benchmark from the PARSEC 2.1 benchmark suite [2], which scales linearly on multicore hardware [3].

Multisystem: To simulate multiple systems, SAE relies on multiple threads—one host thread per simulated system. Though Simics suffers some penalty for synchronizing the systems, the overhead is small. Table 5 shows the speeds at which SAE on a single host is able to simultaneously boot one or more simulated systems. Each simulated system boots CentOS 6.7 with a single-core CPU and 2 GB of RAM; the host system boasts 160 GB of RAM and uses a single 12-core Intel Xeon E5-2697 v2, with hyper-threading enabled. Booting 6 or 12 systems offers each simulated system its own host core and imposes only a 1.83× or 2.32× slowdown, respectively. Jumping to 24 systems exhausts the logical cores of the host (12 cores, each with hyper-threading) but only increases the slowdown to 4.11×. To demonstrate that the number of simulated systems is not limited to the resources of the host (though performance may suffer), we also boot 96 systems, thoroughly overwhelming the host's resources and this is accomplished with only a 15.62× slowdown.

6.3 Use Cases

To show how SAE can be useful for program analysis and architecture research, we provide three high-level examples.

Kernel Vulnerability Detection: The power of SAE is showcased in a kernel vulnerability detection tool we implemented called Kernel Double-Fetch (KDF). It is based on prior work on security vulnerabilities in the Windows NT kernel [15]. When an application makes a system call, the kernel should copy function arguments into kernel space in a single fetch; if an argument is fetched more than once, an attacker could change the argument between fetches and compromise system security [15]. Though the research targeted Windows NT, we reimplemented KDF for Linux with-



(a) Running times of the SPEC CPU 2006 programs under QEMU and SAE relative to native execution.

(b) Running times of the SPEC CPU 2006 benchmarks with instrumentation enabled. SAE tools' times are relative to uninstrumented SAE execution. Similarly, the Pin tools' times are relative to uninstrumented Pin execution.

Figure 8: Relative times (a) without and (b) with instrumentation enabled for the SPEC CPU 2006 benchmarks.

out any OS-specific changes and identified a potential bug in the Linux kernel that has been confirmed by the kernel team (though, fortunately, it turned out to be unexploitable).

The KDF tool showcases much of SAE's power. It incurs only a ~1.2× slowdown during OS boot and utilizes many of SAE's unique features, including memory access notification, instrumentation of both kernel- and user-space code, kernel *ring* 0 instruction-level instrumentation, dynamic tool reconfiguration (to restrict its scope to system calls), and OS awareness (to provide names of functions involved in the double-fetch). At only ~600 lines of C++ code, the KDF tool demonstrates that powerful analysis tools can be written with very moderate coding investment.

ISA Exploration and Software Enablement: As clock frequency scaling has diminished, CPU innovation has come to increasingly rely on ISA and architectural changes to deliver improved performance and new features. SAE is an instruction set simulator, and so an existing ISA can be extended by simulating new features within SAE tools.

SAE can enable software development ahead of the hardware. Software to exercise new hardware features must be developed well before the working silicon is available. Compilation tools must be extended to generate new instructions; runtime libraries must be rewritten to use them; operating systems need to be updated; and new applications must be created to evaluate their benefit. For these types of software changes, SAE excels because modified software can be run on top of the simulated new ISA. Therefore, SAE facilitates prototyping, evaluation, and validation of software support for future ISA extensions and architectural designs.

Simulation Description	Total LOC	Task LOC
`hlt` instruction (*ring* 0)	153	58
`jmp` instruction	143	40
`inc` instruction	180	77
`mov` instruction	184	66
`lgdt` instruction	176	71
Memory redirection	377	266
Shadow memory	434	321
`kmalloc` interception	290	215
Average	242	139

Table 6: Lines of Code (LOC) for hardware simulation using SAE. Total LOC corresponds to the total lines of code for the entire tool. Task LOC corresponds to the implementing the actual functionality, minus environment setup and instrumentation configuration.

Table 6 shows example SAE tools for instruction and memory system simulation and OS call interception. Due to SAE's extensive and easy-to-use APIs, developers, on average, wrote only 242 total lines of tool code. At their core, however, these tools rely on an average of only 139 lines of new code, the remaining lines come from instrumentation setup and configuration that can be reused across tools.

Big Data: Due to its unique ability to simultaneously simulate and instrument multiple systems, SAE is ideally suited to studying big data workloads. To demonstrate its capabilities and potential, we installed Hadoop on a simulated cluster and instrumented three representative Hadoop workloads: *K-means*, *Wordcount*, and *Sort*. We used real

	Simics	SAE	Tool	Slowdown
K-Means	6840s	13860s	37320s	5.45×
Sort	5160s	22980s	132540s	25.7×

(a) Instruction mix instrumentation results.

	Simics	SAE	Tool	Slowdown
K-Means	6840s	13860s	26340s	3.85×
Sort	5160s	22980s	92280s	17.9×
Wordcount	4860s	9660s	58260s	12×

(b) Memory reuse distance instrumentation results.

	Simics	SAE	Tool	Slowdown
K-Means	6840s	13860s	62160	9.1×
Sort	5160s	22980s	133020	27.4×
Wordcount	4860s	9660s	96300	18.7×

(c) Memory footprint instrumentation results.

Table 7: Multisystem instrumentation. We use a 4-socket Xeon-EP E7-8890 v3 @ 2.5 GHz with 512 GB.

datasets and instrumented four worker nodes that perform computation. Recall from Section 6.2 that SAE can scale to a much larger system count. In the interest of time, we scaled down the size of datasets to accelerate workload completion. The size does not affect the accuracy of the tools.

We used three tools—instruction mix, memory reuse, and memory footprint—to instrument Hadoop and collect system-wide profiles of every process. Instruction mix reports statistics using the instruction discovery, inspection, and execution API. Memory reuse and memory footprint both exploit SAE's memory access API to collect a memory profile.

Table 7 shows the results for instrumenting Hadoop. On average, SAE instrumentation tools introduce a 15× slowdown compared to running on Simics. Despite the overhead, recall that a user need only write a *single* tool to aggregate all the instruction-level statistics across all the systems.

7. PRIOR WORK

We believe that no prior tool is fully adequate for studying scale-out workloads, and we contend that there are three key attributes that a tool must have to qualify: it must support flexible instrumentation, simulate the entire system, and maintain perfect transparency. SAE meets all three of these requirements. But much prior work has laid the foundation for developing SAE. We group the prior work according to five different categories, indicated using the group labels **A** through **E** in Figure 1. We encourage the reader to refer back to Figure 1 and its associated table before proceeding further. We omit discussion of a sixth category that arises from combining transparency with user space-only instrumentation because we are not aware of any system that is capable of this particular combination, given the stringent requirements we present for transparency.

Category A [Transparency]: Performance counters are the best example of transparent inspection. In their simplest form, performance counters report counts of a small set of events with zero impact on program execution. However, the transparency is lost when performance counters are used to trigger interrupts for program introspection or when used to record kernel behavior; though they are capable of introspection of the kernel, performance counters must be controlled via the kernel, which is a violation of transparency. Furthermore, they provide no extensibility (i.e. a researcher cannot write a tool providing support for a new counter).

Category B [Instrumentation]: Valgrind [21], DynamoRIO [5], and Pin [19] provide instrumentation capabilities, including the requisite flexibility. However, they are limited to user-space analysis. They also reside in their applications' memory space and slow execution relative to the kernel and the rest of the system, violating transparency.

Category C [Full-System]: The gem5 [4] full-system simulator is a powerful platform, supporting a complete system setup. Unfortunately, it relies on the simulated OS being modified specifically for gem5 execution (violating transparency) and lacks an instrumentation system. Only those with expert knowledge of its source code can introduce custom analysis by directly modifying its internal source code.

Category D [Transparent Full-System]: QEMU [1], SimFlex [25], and BOCHS [17] combine transparency with full-system capabilities. SimFlex is a microarchitectural simulator like gem5, but because it builds on Simics it can boot unmodified OSs. QEMU and BOCHS are functional simulators, capable of the same feat. However, none of these systems offers an instrumentation interface; researchers are constrained to whatever analyses are provided out-of-the-box. QTrace [24] and DECAF [12] have attempted to add instrumentation capabilities to QEMU, but their APIs represent only a small subset of SAE's extensive API. Additionally, these systems' transparency breaks down when extended to multiple systems, as one system's slowdown changes its interactions with the other systems. This would be a problem, for example, if instrumenting a single node in a Hadoop cluster, which would likely change the work assignments and therefore alter the behavior of the workload.

Category E [Full-System Instrumentation]: PinOS [7] and DrK [10] extend the instrumentation frameworks of Pin and DynamoRIO to support kernel-level instrumentation. However, these solutions are OS-specific and require substantial porting effort to support additional OSs. Because they rely on OS specifics, they also affect execution behavior, which violates the transparency requirement.

8. CONCLUSION

New instrumentation systems, such as SAE, are crucial for advancing research and development. We believe that SAE has the potential to help its tool writers unlock new research because of the following features: *(1) Transparency:* Since it operates within a simulator, SAE in no way alters the execution or perceived state of the target system. This is critical in commercial workloads where relaxing transparency or compromising execution is not an option. *(2) Performance:* Many scale-out workloads exercise OS functionalities and rely on complex software stacks. SAE is fast enough to boot an OS and execute a full Hadoop stack in real-time. SAE couples an interpreter and just-in-time compiler for flexibility and performance. Moreover, it leverages multi-threaded execution to efficiently support multisystem instrumentation. *(3) Extensibility:* SAE's program and architecture introspection capabilities are simple and easily understood. It enables early-stage pathfinding by abstracting the instruction set architecture and operating system idiosyncrasies. It can even extend ISA capabilities for early software development for future anticipated hardware. *(4) Interoperability:* SAE requires no OS-specific support. Because it is plugged into a virtual platform simulator, SAE can transparently run any operating system, including Windows, Linux, and Android, and all applications contained therein.

ACKNOWLEDGEMENTS

We thank the entire Intel SAE team for their extensive development work that has made this paper possible, including but certainly not limited to Yulik Feldman, Mohammad Mahajna, Yair Lifshitz, Yoav Weiss, Aviv Segall and Inbal Livni Navon. We also thank the tool developers who contributed to the experimental data published in this paper, including but not limited to Valentin Andrei and Tahrina Ahmed. Last but not least, we appreciate the constructive feedback on developing this paper from the UT Trinity Research lab.

9. REFERENCES

[1] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference, FREENIX Track*, 2005.

[2] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *Proc. of PACT*, 2008.

[3] C. Bienia and K. Li. *Benchmarking modern multiprocessors*. Princeton University USA, 2011.

[4] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, et al. The gem5 simulator. *SIGARCH Computer Architecture News*, 39, 2011.

[5] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, 2003.

[6] D. Bruening, Q. Zhao, and S. Amarasinghe. Transparent dynamic instrumentation. In *ACM SIGPLAN Notices*, volume 47. ACM, 2012.

[7] P. P. Bungale and C.-K. Luk. PinOS: a programmable framework for whole-system dynamic instrumentation. In *Proceedings of the 3rd international conference on Virtual execution environments*. ACM, 2007.

[8] K. H. Cettei. *Code cache management in dynamic optimization systems*. PhD thesis, Harvard University Cambridge, Massachusetts, 2004.

[9] M. Charney. In *https://software.intel.com/en-us/articles/xed-x86-encoder-decoder-software-library*. Last accessed: Sep. 1, 2015.

[10] P. Feiner, A. D. Brown, and A. Goel. Comprehensive kernel instrumentation via dynamic binary translation. In *ACM SIGARCH Computer Architecture News*, volume 40, 2012.

[11] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *SIGPLAN Notices*, volume 47, 2012.

[12] A. Henderson, A. Prakash, L. K. Yan, X. Hu, X. Wang, R. Zhou, and H. Yin. Make it work, make it right, make it fast: Building a platform-neutral whole-system dynamic binary analysis platform. In *Proceedings of the International Symposium on Software Testing and Analysis*. ACM, 2014.

[13] J. D. Hiser, D. Williams, W. Hu, J. W. Davidson, J. Mars, and B. R. Childers. Evaluating indirect branch handling mechanisms in software dynamic translation systems. In *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society, 2007.

[14] Z. Jia, L. Wang, J. Zhan, L. Zhang, and C. Luo. Characterizing data analysis workloads in data centers. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2013.

[15] M. Jurczyk and G. Coldwind. Identifying and exploiting windows kernel race conditions via memory access patterns. In *The Symposium on Security for Asia Network*, 2013.

[16] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, D. Brooks, S. Campanoni, K. Brownell, T. M. Jones, et al. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. ACM, 2015.

[17] K. P. Lawton. Bochs: A portable pc emulator for unix/x. *Linux Journal*, (29es), 1996.

[18] P. Lotfi-Kamran, B. Grot, M. Ferdman, S. Volos, O. Kocberber, J. Picorel, A. Adileh, D. Jevdjic, S. Idgunji, E. Ozer, et al. Scale-out processors. In *ACM SIGARCH Computer Architecture News*, volume 40, 2012.

[19] 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 *ACM Sigplan Notices*, volume 40, 2005.

[20] M. Michael, J. E. Moreira, D. Shiloach, and R. W. Wisniewski. Scale-up x scale-out: A case study using nutch/lucene. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*. IEEE, 2007.

[21] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, 2007.

[22] M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21, 1999.

[23] J. Smith and R. Nair. *Virtual machines: versatile platforms for systems and processes*. Elsevier, 2005.

[24] X. Tong, J. Luo, and A. Moshovos. QTrace: An interface for customizable full system instrumentation. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2013.

[25] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe. SimFlex: statistical sampling of computer system simulation. *IEEE MICRO Special Issue on Computer Architecture Simulation and Modeling*, 26, 2006.

[26] C. Zheng, J. Zhan, Z. Jia, and L. Zhang. Characterizing os behavior of scale-out data center workloads. In *Workshop on the Interaction amongst Virtualization, Operating Systems and Computer Architecture (WIVOSCA)*, 2013.