Fast Microcode Interpretation with Transactional Commit/Abort

Jens Tröger
jens.troeger@intel.com

Darek Mihočka
darekm@emulators.com

Pardo
david.keppel@intel.com

Abstract

This paper describes the design and implementation of a fast microcode interpreter for functional system simulation. While we primarily target architecture simulation, the design principles can easily be applied to high-level and byte-code interpreters as well.

A functional system simulator executes an entire guest operating system stack and applications, inside a software simulated environment. Such simulators do not simulate cycle accuracy, but run fast enough to run real-world workloads. System simulators depend on the architecture of both guest and host, so are traditionally hand-written and tailored to the respective architectures. Performance optimizations like dynamic compilation to host code or assembly coding of key routines are common, but further increase host and guest dependency. The interpreter described here is substantially independent of the host, is easily retargeted to new guests, yet achieves performance competitive with much more hand-crafted simulators.

Most functional simulators implement a traditional guest instruction set with simple and independent operations such as loads, adds, and so on. In contrast, simulating microcode is difficult because each instruction has many sub-operations – which may depend on each other, and which must atomically commit or abort as a single transaction. This paper describes techniques for high-performance transactional simulation.

The paper measures real-world workloads like Windows7[1] *and Linux boot, and application runs of Office, web browsers, and more. We also report very machine specific performance details using micro-benchmarks. We then use that data as a guide for the design and implementation of an interpreter that achieves high performance, host portability via coding in "plain" C++, and guest retargetability via a machine-generated instruction parser and abstracted operations that are reused in retargeting.*

Initial results show execution on modern processors as fast as a few tens of host cycles per simulated guest instruction, thus executing hundreds of millions operations per second, and tens to hundreds of millions of guest instructions per second, depending on their complexity.

[1] Trade names in this paper are the property of their owners.

Keywords: functional system simulation, microcode interpretation, binary translation, instrumentation

1. Introduction

System simulation allows for execution of system level software in a controlled software environment. The simulator implements a "guest" CPU architecture, physical memory, TLBs, optionally caches, and some essential devices like timers and serial I/O devices. The goal is to run unmodified firmware and an operating system stack with applications, ideally with the guest software unable to detect it is not running on real hardware. The system simulator itself is an application that runs on the "host" machine.

System simulation has many applications. One common use is migration of legacy software to new and different host architectures. Architecture research and experimentation uses simulation to explore novel features without building hardware, and to collect detailed information used to refine proposed new features. Simulators can sandbox a guest software stack in a secure and controlled execution environment for scalable cloud computing. Simulators are also used for debugging low-level code where using a debugger on real hardware is hard, and for developing and validating code where new hardware does not yet exist. Finally, a system simulator can observe, log, and manipulate a guest software stack, thus collecting arbitrary information about the dynamic behavior of the guest software and allowing performance analysis and tuning that is difficult or impossible on real hardware.

There are many approaches to simulation, depending on the desired accuracy. For example, cycle- and timing-accurate simulators mimic low-level implementation details of guests. Performance data is accurate, but may cost thousands or millions of host cycles to accurately simulate bus contention, full-buffer stalls, and so on. The slow speed usually limits what software can be run. Conversely, functional simulators hide many details of the guest, to improve simulation speed. As a result, functional simulators can boot and run modern operating system stacks, executing billions of guest instructions in tens of seconds.

Even among functional simulators, there are many ap-

proaches. Decode-and-dispatch interpreters are often simplest to build, but may take a thousand host cycles per simulated guest instruction. Instruction decode cost often dominates simulation speed, so threaded-code simulators that cache decoded results are often much faster. Dynamic cross compilation is faster yet because it can reduce inter-instruction simulation costs, but code generators are a significant effort to build and retarget [EG03]. Finally, user-level simulation often provides highest performance and simplest implementation, but is only able to run limited software and cannot observe system details.

Most simulators in the literature are for guests with relatively simple instructions. Even "complex" instructions, such as floating-point `sin`, are still conceptually orthogonal and independent of other instructions. Microcode simulation is more complex, because each instruction is composed of sub-operations that may have interdependencies, can give rise to several simultaneous exceptions, and which complete atomically. For example, a single microcode instruction may include memory, integer ALU, floating-point ALU, and branch sub-operations; a compare sub-operation may be forwarded to conditional branch logic in the same microinstruction; multiple units may give rise to exceptions, which need to be prioritized; and results from all sub-operations follow atomic commit/abort semantics. Existing simulator designs are poorly-suited to simulate such a machine.

This paper discusses challenges implementing a portable and high performance functional system-level interpreter. The simulator is written in C++ and is easily recompiled on different hosts. It avoids most direct host dependencies – for example, it avoids multiplexing guest condition codes on to host condition codes, and host/guest endianess affects performance but not correctness. The simulator is easily retargeted to different guest ISAs and gives good performance for both conventional macro instruction sets and complex microcode.

Section 2 discusses different types of workloads and benchmarks, their application, and how they influence the design of our interpreter. Section 3 analyzes how design decisions of existing interpreters and dynamic compilers for system simulators affect their performance. Section 4 describes our internal virtual microcode architecture, and introduces the design and implementation of our fast and portable microcode interpreter. Finally, section 5 presents initial performance results of our interpreter which demonstrate the feasibility and benefits of our design.

2. Workloads and Benchmarks

A fast simulator must run common operations quickly. To design one, we need to understand:

- what common and dominant guest workloads users will run;
- what critical interpreter paths dominate performance of these guest workloads; and
- how these paths are best executed on a given host machine.

For guest operations, it is important to measure large workloads that are representative of what will be used in simulation, since "inner loop" benchmarks often hide system-level behavior that in practice can dominate total simulation cost.

For host operations, it is important to measure processor performance in detail, to ensure the simulator makes good use of the hardware microarchitecture, since small differences in cost often have a big impact on running time.

2.1. Application Benchmarks

We use guest workload data to discover what guest operations are common or rare, and thus drive performance decisions in the simulator design.

A workload is the code executed and data consumed by an application to solve a given problem. A benchmark exercises one or several dedicated workloads. One widely-used benchmark is SPEC [Cor10]. SPEC CPU, for instance, is "designed to provide performance measurements ... used to compare compute-intensive workloads on different computer systems." Other SPEC benchmarks focus on power consumption, render speed and 3D workflows, parallel computation, server loads for virtual machines, and so forth.

Another common approach is to run real-world applications concurrently so they compete with each other for resources. One scenario, for example, is running a web browser that renders media content, while also checking email or playing a game. The data we used to design our fast interpreter is largely from such real world scenarios, and less from benchmarks.

2.2. Micro-Benchmarks

We use micro-benchmarks to understand performance bottlenecks of a host CPU, and use that data to design the data and code layout of our fast interpreter.

A micro-benchmark is a short sequence of instructions that exercises one particular feature of the CPU. Listing 1 shows a micro-benchmark we designed to measure latency of the x86 `lahf` instruction, an instruction often used by interpreters and dynamic instrumentation tools to retrieve some host arithmetic flags. Micro-benchmarks

```
1       mov  ecx,1000000000
2   loop:
3       lahf
4       xor      eax,eax
5       lahf
6       xor      eax,eax
7       lahf
8       xor      eax,eax
9       lahf
10      sub      ecx,1
11      jne      loop
```

Listing 1: Micro-benchmark to execute 4 billion LAHF instructions with EFLAGS update after each instruction to prevent CPU internal microcode cheats.

	Instruction Count
WindowsXP/32 boot	4,292,484,077
Windows7/64 boot	12,105,336,209
Windows7/32 boot, HD Video	32,549,751,890
Windows7/64 boot, Office 2010	581,278,454,698
LiveCD Suse Linux/64 boot	64,857,312,020
LiveCD Suse Linux/64 apps	3,313,655,494

Figure 1: Instructions executed for some real-world scenarios.

can measure latency of individual instructions, instruction sequences, load-hit-store or store-forwarding latencies, cache miss penalties, and so forth.

2.3. Performance Data, Interpreter Design

Today's most popular desktop architecture is Intel's x86 architecture, often 32-bit but increasingly 64-bit, running different operating systems like Windows, Linux, or OS X. We use Intel's 64-bit x86 architecture as our initial host, though we plan to port the interpreter to other architectures.

Windows and Linux are the most popular and accessible operating systems today, so we used them to gather real-world guest workload data. We instrumented the Bochs [Boc10, MS08] system simulator to collect data.

Figure 1 shows instructions executed for some real-world scenarios. Instruction counts are absolute and are "polluted" numbers, with every instruction the CPU sees, not just instructions from a single application. We break these numbers down further in later sections.

In contrast, figure 2 shows sample results from host micro-benchmarks. These micro-benchmarks are particularly useful for implementating a fast interpreter, as they dominate the implementation of control flow, function calls and returns, and access to arithmetic flags and other execution context. The results are for three implementations of

	P4	i7	SB
LAHF	8	2	2
PUSHF/POPF	97	26	22
SETC AL	1	1	1
FSTENV	316	98	97
FNSAVE	488	160	167
FXSAVE	135	92	64
CALL/RET	14	4	4
CALL mispredict	51	15	12
JECXZ predict	2	1	1
JECXZ mis-predict	22	13	6
L1 D$ miss latency	9	4	4
Load 32-byte cross	1	2	1
Load 64-byte cross	20	4	5
Load mapped page cross	73	24	28
W32, R8R8R8	39	3	3
W8W8W8, R32	35	16	18

Figure 2: Approximate cycle counts of instructions for the Intel Pentium4 Xeon, Intel Core i7, and Sandy Bridge Core i7 processors.

the Intel 64 architecture, and show how low-level implementation details may affect performance portability.

The first two sets of rows show approximate instruction latencies to save and restore arithmetic flags and execution context. Saving context is very expensive: even saving only arithmetic flags is more expensive than just a single `setcc` to store flags. The next two rows show approximate latencies for predicted and mis-predicted function calls and conditional control flow. These numbers demonstrate the penalty of control flow misprediction.

The last set of numbers show the latency of an L1 data cache miss with L2 hit, the costs of loads that cross alignment boundaries, and the penalties of store-forwarding through writing a 32-bit value and reading it back in three 8-bit quantities, and vice versa. The importance of these numbers is explained in following sections.

2.4. Further Data Analysis

Booting operating systems and running common user applications produces the instruction counts of various traces shown in figure 1. The distribution of x86 instructions shows 60% of dynamic instructions are made up of only about 20 actual instructions: mainly loads and stores, selected arithmetic instructions, and conditional and unconditional control flow.

Figure 3 shows total counts of select interesting instructions across four of our scenarios. Figure 4 shows the distribution of the 20 hottest dynamic instructions of the SuseLinux/64 Apps scenario. The breakdown varies little

	Win7/64 boot	SuseLinux/64 boot	Office2010/32	SuseLinux/64 Apps
Top 20	59.3%	59.1%	52.89%	59.00%
MOV/64	16.03%	13.28%	7.1%	15.90%
MOV/32	10.87%	5.56%	9.2%	8.40%
MOV/8, MOV/16	4.25%	5.87%	3.45%	4.53%
JZ, JNZ	9.57%	10.68%	6.64%	9.14%
Jcc	3.57%	4.07%	4.41%	3.25%
LAHF, SAHF, PUSHF, POPF	0.002%	0.22%	0.0007%	0.34%
ADC, SBB	0.35%	0.25%	0.05%	0.12%
CALL, RET	1.84%	3.78%	1.44%	5.14%
x87 FP	0.001%	0.000%	0.02%	0.001%

Figure 3: Break-up of the dynamic instructions from figure 1.

Operation	Count	Quota	Total
MOV $GqEqM$	232,397,067	7.013%	7.013%
MOV $GqEqR$	208,763,826	6.300%	13.313%
JZ	160,250,964	4.836%	18.149%
JNZ	142,558,476	4.302%	22.452%
POP RRX	116,318,547	3.510%	25.962%
PUSH RRX	109,188,708	3.295%	29.257%
MOV $EqGqM$	107,951,225	3.258%	32.515%
MOV $GdEdR$	101,166,140	3.053%	35.568%
ADD $EqIdR$	100,594,816	3.036%	38.604%
MOV64 $GdEdM$	86,416,769	2.608%	41.211%
RET	85,216,567	2.572%	43.783%
LEA GqM	82,643,279	2.494%	46.277%
CALL	77,833,440	2.349%	48.626%
TEST $EqGqR$	57,768,649	1.743%	50.369%
TEST $EdGdR$	57,299,987	1.729%	52.099%
MOVZX $GdEbM$	54,683,474	1.650%	53.749%
JMP	49,507,917	1.494%	55.243%
XOR $GdEdR$	43,189,064	1.303%	56.546%
NOP	40,838,707	1.232%	57.779%
CMP $GqEqR$	40,683,876	1.228%	59.006%

Figure 4: Top twenty instructions of the SuseLinux/64 Apps scenario. Most scenarios show similar numbers.

across scenarios, even though the compilers, operating systems, and applications are quite different and scenarios are built for either 32-bit and 64-bit host architectures.

Integer instructions are among the hottest instructions. The most common integer type is the 32-bit wide `int`; 16-bit `short` is used almost never and 8-bit `char` only sporadically. This aligns with the development of the Visual Studio, GNU, and Intel compilers over the past few years. The default size for `int` is 32 bits, pointers are either 32-bit or 64-bit, enumerations default to 32-bit values, and `auto` usually resolves to 32-bit integers.

Both tables show more than two thirds of conditional control flow is branches on the zero flag. This makes sense since most conditionals are compiled from C/C++ statements that check for NULL pointers, compare (i.e. subtract) two values in loop headers, or use boolean expressions for switch- or if-statements.

The x87 FPU is used only by legacy code, and 80-bit floats are used only by hand crafted and specialized assembly code. For best performance, 32-bit `float` and 64-bit `double` floating point types are compiled into SSE instructions rather than x87 code, and therefore actual x87 instructions are almost non-existent today.

With these observations, we can now examine and understand how existing functional simulators and their execution engines work, and why they sometimes do not deliver the performance that they could.

3. Existing and Related Frameworks

Some frameworks focus on individual user-mode applications and on instrumentation or on dynamic optimization through binary translation. Recent examples are Dynamo [BDB00], DynamoRIO [BGA03], and Pin [kLCM$^+$05]. Other frameworks focus on executing a complete operating system stack, and thus run an entire ISA including protected mode instructions, and do device and memory modeling. Examples include Bochs [Boc10, MS08], QEMU [Bel05] and Zsim [LCL$^+$11].

Dynamo was originally developed to investigate dynamic optimization of single user-mode applications on PA-RISC. It was merged with RIO and ported to Linux and Windows on Intel's IA32 architecture, and renamed DynamoRIO. Using instrumentation, both frameworks detect hot code and decode it to an intermediate representation, optimize the IR, then write optimized host code to a code cache. The compiled code allocates guest values to host registers, so entering and exiting the code cache requires expensive save and restore. For applications where

entry/exit is common, save/restore overhead may be large.

Pin is an instrumentation framework for user-mode code. Pin injects arbitrary functions, written in C/C++, into native execution of an application. Through a well defined API, instrumentation can implement tools like profilers, memory leak detectors, trace generators, and so forth. Like DynamoRIO, Pin recompiles and instruments guest code on the fly and stores it in a code cache for execution; it also has overhead for save/restore when switching between instrumented guest code and Pin core code.

Several other frameworks use similar techniques, including HDTrans [SSB05], Walkabout [CLU02] and Yirr-Ma [Jen02].

In contrast, full system simulators execute an entire operating system stack. System simulators are more complex, but can collect more authentic data about guest code execution. For example, user-mode optimizers and instrumentation tools go to great effort to maintain control of sandboxed applications in the face of exceptions, signals, kernel calls, callbacks, and asynchronous interrupts. System simulators sit underneath the guest operating system and can thus observe a complete picture of events and code execution without intrusion or interception.

Bochs is a system simulator for IA32 and Intel64 architectures. It hosts various flavors of Windows, Linux, BSD, and other operating systems built for Intel architectures. Bochs' core execution engine is a classical interpreter: the decoder is table-driven and decodes a single byte at a time, then dispatches to coarse handlers which then select the correct implementation for the specific guest instruction. By implementing handler functions with conditional control flow, however, Bochs introduces branch mis-prediction stalls on almost every path that interprets guest instructions. Recent versions of Bochs use a trace cache and lazy flag evaluation, to achieve over 100 guest MIPS.

QEMU is a binary translator which can be used as a stand-alone application, but it also sits at the core of system simulators and virtualizers like VirtualBox. QEMU shares close copies of Bochs' device model implementation. QEMU is a dynamic compiler rather than an interpreter and supports various guest architectures on different hosts and a simple lazy flag implementation. Guest values are allocated to host registers, introducing save/restore overhead when entering/exiting the code cache. Integer instructions execute at a 2x to 6x slowdown compared to native execution, but control flow instructions like function calls and indirect branches may be a factor of 100x slower.

Zsim is an ISA interpreter for functional system simulation. It uses binary translation to improve performance of frequently-executed and simple-enough guest instruction sequences. It also implements a more direct dispatcher and a form of lazy flag execution, and delivers performance of 40 guest MIPS in interpretation mode, and 150+ guest MIPS running compiled code.

Transmeta's Crusoe and Efficeon processors are system simulators [Deh03, Kep09]. They use custom VLIW processors and dynamic binary translation. These processors run about 1x slowdown. The translator performs generic and host-specific optimizations, but also relies on the host processor having transactional commit/abort and some guest functionality, including flags, some segmented addressing modes, endianness, and some data types.

4. The Fast Interpreter

This section describes the design and implementation of a fast interpreter with transactional commit/abort, easy guest retargetability, host portability, and good performance. The "take away" message from this section is these goals can all be met, but doing so requires careful attention to many seemingly-small details.

4.1. Building Blocks

A functional system simulator consists of several components, each of which implements a model of the hardware architecture being simulated. To boot an operating system stack, the simulator requires at least the following components:

- an execution engine for the operational semantics of guest ISA;
- delivery of synchronous and asynchronous interrupts and exceptions to the execution engine;
- virtual and physical addressing, guest memory protection, TLB, optional caches, and a conforming guest memory architecture;
- device models, including at least a timer device (e.g., APIC) and input and output devices (e.g., a serial port keyboard and display).

More accurate guest simulation generally means slower guest performance. For functional simulation, the goal is enough accuracy to run unmodified operating systems and applications. Response latencies, signal routing, and other hardware details are not needed. Our simulator uses interfaces for a physical memory implementation with store logging and a device manager for IO dispatch with a generic device model interface, but they are not subject of this paper.

The remainder of the paper investigates how simulator design and guest ISA implementation decisions affect

guest performance. Many of the techniques apply to both our interpreter and to other frameworks using binary or byte-code compilation, optimization, and instrumentation.

4.2. Performance Limitations

There are different levels on which we can improve simulation performance. [Trö04] outlines and formalizes these. One of the most popular and researched approaches to improve performance is dynamic compilation of guest code into host code, called just-in-time or "JIT" compilation or dynamic binary translation. However, this approach has pitfalls that can limit performance improvements. Notably, generated code quality is limited by the need to accurately implement complex guest behavior on the host, and by the need to use data layouts and code sequences that use the processor microarchitecture efficiently.

By rethinking the implementation of the simulated guest state and by carefully tuning the core of our interpreter, we work with several limiting boundaries at the same time to improve interpreter performance. Some of the same techniques can also help a dynamic compiler.

4.3. Working the Constraints

As shown above, roughly twenty x86 guest instructions make up over 60% of the dynamic instruction count of many of today's real world scenarios. These instructions operate mainly on 32-bit and 64-bit integers. Call, return and conditional branch occur every dozen or so dynamic instructions. Code makes limited use of guest arithmetic flags. Floating-point arithmetic is implemented using 32-bit and 64-bit SSE.

A first performance limitation is modeling and implementing guest architectural state. Figure 2 shows memory loads and stores, if cached and properly aligned, are very cheap. That allows us to organize guest register files as a carefully laid out data structure in memory. Doing so relieves register pressure for host code, and avoids save/restore overhead. Furthermore, aligning and structuring guest state avoids penalties caused by misaligned accesses or store-forwarding stalls during member access.

A second performance limitation is modeling guest arithmetic flags. X86 code execution produces flags with almost every instruction, but flags are consumed rarely. Mapping the guest's flags to host flags may seem cheap at first, but access to the host flags can be costly (figure 2). Worse, host flags may vary across differing architectures, for example "carry" vs. "borrow" semantics for subtraction. And, different implementations of the same architecture may set "undefined" flags differently, and some applications rely on specific behaviors for undefined flags.

It is often cheaper to use a portable implementation based on carry-out vectors [MT10]: instead of reading host flags after each flag producing arithmetic instruction, we store the result of the operation and a vector of carry-out bits, used to infer some or all guest flag values on demand, also called "lazy flags reconstruction".

A third performance limitation is the main interpreter loop and interpreter functions. A given guest instruction rarely appears twice in sequence, so an "obvious" decode-and-dispatch interpreter often has high branch mispredict penalties. As shown above, though, branch-predicted calls and returns are very cheap. Therefore, instead of using centralized dispatching and general instruction interpretation functions, we spread out call sites and chain together very specialized and simple interpretation functions. We elaborate on this below (section 4.4.3).

4.4. Implementation

4.4.1. Features

The design of our interpreter was guided by several project constraints and feature requests:

Portability. To run on varying host architectures, the simulator avoids assembly code. The entire framework is written in platform-independent C/C++ only.

Atomicity. Execution and rollback of an interpreter sequence must be atomic and side-effect free.

Performance. Speed is one of our highest concerns.

Large Memory Support. Modern operating systems use 64-bit ISAs and large memories. Our simulator supports 32-bit and 64-bit guests. It runs on 32-bit hosts, but is optimized for 64-bit host architectures.

Multi-threading. Operating systems and applications increasingly use multiple software threads and cores to improve parallelism. Our interpreter should be able to implement several guest cores and assign them to multiple host cores, thus achieving true multi-core simulation.

Virtual Time. Many system simulators, optimizers, and virtualizers base guest time on the host timer. In contrast, we need our guest time to be completely decoupled and purely virtual to implement reliable and deterministic playback, and to avoid synchronization problems when halting, debugging, instrumenting, or otherwise interfering with guest code execution.

Instrumentation. Instrumentation is a required feature, should come at low overhead, and should have no overhead when disabled. Instrumentation must also be adjustable during a simulation run; it cannot be a compile time feature which is statically "compiled in" on demand.

Specification-Driven ISA design for Experimentation. To support guest ISA experimentation, an interpreter must

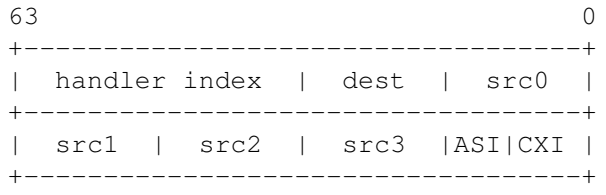

Figure 5: 128-bit scode instruction.

be easy to retarget, and should use abstract interfaces to memory, uncore, and device models so they are independent of the guest speficiation.

Self-Modifying Code The simulator must implement guest self-modifying code fast enough to run common JITs, and must support device DMA that overwrites code.

We designed the interpreter around these constraints and feature requests.

4.4.2. Scode and Traces

Our portable microcode interpreter uses a version of threaded interpretation, based on a special interpreter virtual ISA we call *scode*, or simulator code, and carefully-defined interpretation functions.

The decoder is generated from an ISA encoding/decoding specification [RF95]. By design, the generated decoder accesses every field of a guest opcode only once.

A generated decoder fetches guest instructions into an internal prefetch buffer. It then breaks individual guest instructions into a sequence or "trace" of scode instructions. Scode resembles RISC instructions, but with added attributes. A decoder generates traces from individual guest instructions and also merges multiple guest instructions into one trace. Merging reduces outer-loop dispatch costs and allows optimizations between guest instructions.

Many scode instructions are guest-neutral and are reused across guests. Further, scode instructions are typically simple, so retargeting can focus on just key differences. For example, `add` is guest-neutral, with condition codes modeled by a separate handler. Retargeting to new condition code behavior thus only needs new handlers for condition codes.

An scode instruction is represented as an aligned 128-bit wide data structure. Not all handlers need all combinations of state, so the structure is a union, to limit total size. The most common format, shown in Figure 5, consists of a unique 16-bit opcode value, one destination register, up to three source registers, a shift or rotate count value, an execution context identifier (for hyper-thread or pipeline contexts). Other formats include a segment selector for memory accesses, a 32-bit or 64-bit immediate operand, and an additional immediate value or registers. Commonly-used fields are byte-aligned for quick access at only a slight space cost, while uncommon fields are bit-aligned for compactness.

Scode uses a handler index rather than a pointer. Using an index requires double indirection on every call, but limits the scode size [MS94] and makes scode independent of whether host pointers are 32-bit or 64-bit. 256 handlers is not enough, so the handler index is 16 bits.

It is always desirable to make scode as small as possible, to minimize host cache footprint and thus scode miss costs. However, host performance timings, above, show it is also vital that common fields are naturally aligned for fast access, as misaligned accesses that cross cache line or page boundaries may be quite slow. Since the handler index is 16 bits, scode instructions must be at least 16-bit aligned, or any savings in memory pressure is more than than offset by the higher cost of misaligned accesses. In practice, guest code has substantial locality, so host code also has good locality. In our experience so far, the 128-bit format is cached well, making fast field extraction a good tradeoff.

The scode interpreter itself has no concept of the guest architecture but instead interprets over an internal state of 256 64-bit general purpose integer registers, 64 64-bit general purpose floating-point registers, and 16 predicate registers. For every actual guest architecture that we support (e.g. x86) the interpreter also provides a mapping from guest architectural registers to internal registers, and additional implementations of the guest architecture's special-purpose registers. The mapping between guest and internal state is implemented by a class derived from the internal Core class. Figure 6 illustrates the architecture.

The interpreter's scode instruction set is RISC-like. It has no concept of flags or datatypes other than 32-bit or 64-bit integer and floating point. This is one of the most fundamental design decisions, and is based on the observations outlined above, that there is very little need for any more but these datatypes in most of today's scenarios.

Guest arithmetic flags are implemented using a lazy flags approach and scode instructions that compute a carry-out vector of an operation in addition to the actual result of an arithmetic operation. Using the result value and the carry-out vector of an arithmetic instruction, we can derive any other flag value on demand. This approach speeds up retrieval and computation of guest arithmetic flags, and is portable because it is implemented solely in C/C++, rather than relying on custom fast assembly to compute flags or multiplexing on top of the host condition codes.

Integer datatypes smaller than 32 bits are sign and zero extended as needed, then operated on. The relatively small portion of guest scenarios using smaller datatypes means little added overhead for extension to the scode datatypes.

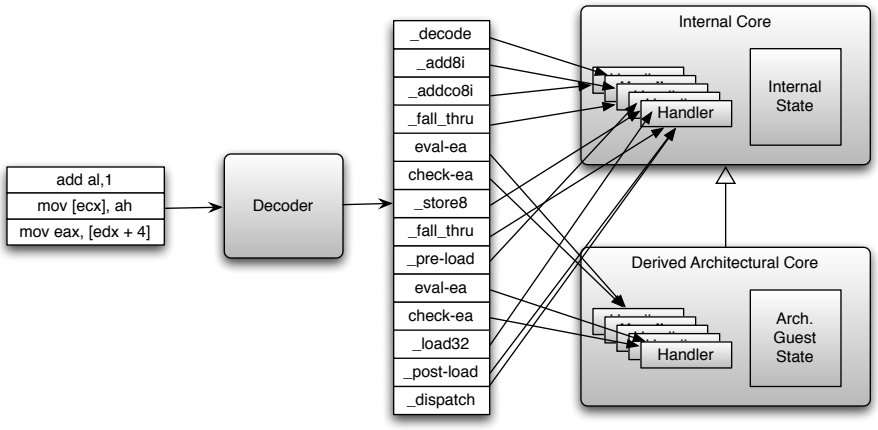

Figure 6: Internal architecture of the interpreter. Guest instructions are decoded into a trace of scode instructions, each of which is interpreted by a handler function. The simulator implements a private Core execution engine, and a derived Core implements further guest architectural functionality. Here, guest loads are instrumented.

4.4.3. Handler Functions

A decoded trace is committed into a trace cache, a large buffer indexed by guest physical addresses. The high physical address bits subscript an index table, `tot`, and the low 22 bits subscript a particular trace cache. Guest instructions are decoded on demand, so the trace cache holds traces only for instructions actually executed.

Each scode instruction is implemented by a carefully designed and placed interpreter function, a so-called handler. A handler is a C++ member function either of the interpreter's internal Core class or of a derived guest architectural Core class. Handlers use four register parameters:

- the `this` pointer provides internal and guest context for the handler. The context data layout is aligned and flat – it does not contain further pointers, so each context access is a single load, often with an immediate offset relative to `this`;
- a pointer to the aligned 128-bit scode structure with operand values of the current instruction;
- a pointer to the Core handler table, holding function pointers to all handlers, indexed by 16-bit opcode;
- a 64-bit operand often used as an immediate or pointer to data that one handler passes to the next, used for data flow internal only to the trace.

The last operand is used to decompose guest operations efficiently into multiple handlers. For example, for a single guest instruction, some memory addressing computations are performed in one handler, the actual load or store is in another, and the address is passed between them in the last operand. This organization allows generic handlers to implement many parts of the guest ISA, which reduces retargeting effort. It also improves dynamic code reuse, which helps control the simulator's instruction cache footprint.

Finally, a handler function always returns a pointer to the next scode instruction to interpret. Using four parameters per handler function is intentional, as the calling convention of 64-bit host systems, both Linux and Windows, provides for passing these parameters in registers, thus improving performance.

As a general rule, handlers avoid conditional statements, to minimize conditional branches which may not predict reliably. If a guest instruction like `add` has different forms for 8, 16, 32, and 64-bit datatypes, with and without flag recording, the decoder selects a specialized handler based on the datatype. That is, there is no single "add" handler that selects the correct implementation when invoked. In addition, flags computations are handled by a dedicated handler that consumes the results of the previous arithmetic handler. Figure 7 illustrates the approach.

4.4.4. Atomicity

To implement atomic execution of traces, every handler executes in three sequential phases:

- The *Operation* phase prepares operands and, if necessary, normalizes operand values to a 64-bit width.
- The *Execution* phase fetches values from the guest execution context, computes side-effect free results, and then calls to the next handler of the trace. Thus, calls to the various handlers of a trace are spread out across all handler functions of the trace.

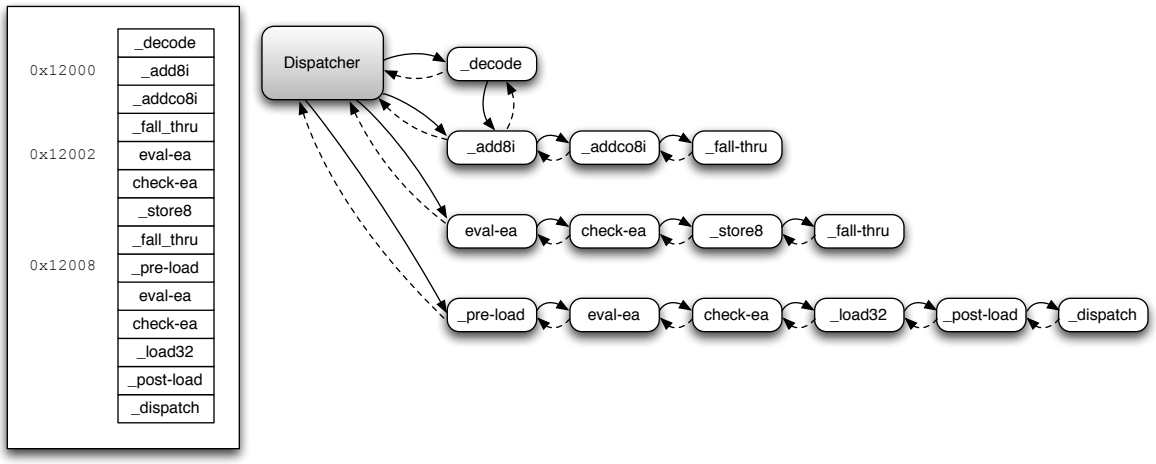

Figure 7: Dispatch and Trace execution with almost perfect branch prediction. The Dispatcher invokes call chains of handler functions as defined by the traces of scode instructions.

- Finally, *Writeback* saves all buffered results and other side-effects into the internal and guest architectural state, thus committing the handler.

Figure 8 shows a handler schematically. This handler structure in effect implements a logging mechanism that saves temporary values and then copies them to committed state only once all execute actions have succeeded. However, using the call/return mechanisms instead of an explicit log makes efficient use of compiler call/return optimizations and processor out-of-order mechanisms. In practice, it also allows for a simple and natural expression of handlers without the clutter of explicit logging.

In this context, the definition "trace" is up to the user of the interpreter. One may choose to decode single scalar or microcode instructions into an atomic trace of scode instructions, or may instead decode entire guest basic blocks. This approach gives a handy vehicle to experiment with transactional memory.

4.4.5. Handler Chaining

One of the most performance-critical paths is the interpreter's core loop. Most interpreters use a centralized dispatcher that causes a host's branch predictor to mispredict almost every time it calls an interpretation function [EG03]. To avoid this penalty, our interpreter call-chains handlers in a trace, and, if possible, also chains consecutive traces. Each handler computes the next one by simply incrementing its own scode pointer (2nd operand), then fetching the new address from the handler function table (3rd operand) by indexing into it using the next scode's opcode. The new handler is then called directly.

This structure improves interpreter dispatch prediction by spreading branches over many call sites. This makes each call site much more predictable and also gives more information to the host's branch predictor. Using call/return structure for handling also takes advantage of host processor call/return prediction. In addition, the hottest handlers often stay in the host instruction cache for a long time. Although this handler structure may at first seem unintuitive, it avoids mispredicted branches, where the cost of a single mispredicted branch is sometimes more than the total cost of a handler.

Traces always either return for commit, or exit via `longjmp()` for abort, thus avoiding indefinite growth of the interpreter's call stack.

4.4.6. Dispatch and Interpretation

The main loop of the core interpreter is different than most interpreters: instead of doing decode and dispatch, the loop calls via a "global actions" pointer which is usually a function that returns its handler argument, but which may be updated by handlers, for example to handle interrupts. After global actions, the dispatcher simply calls the current trace. Figure 9 shows the main loop schematically.

Initially, the trace cache is zeroed. Given a guest instruction pointer, dispatch finds only a "zero scode instruction" which is the "Decode" scode instruction, and thus calls the decoder.

The Decode handler is just an ordinary handler. The Execute phase fetches the instruction from guest memory, decodes it and breaks it up into an scode trace, which it then stores into the trace cache. During Writeback, the Decode handler calls the new trace, and on Writeback the new trace returns to the Decode handler with a pointer to the

```
handler_t *schematic_handler(context, hdlr_table, 128b_struct, arg) {
Operate:
  next = 128b_struct+1
  next_handler = hdlr_table[next.handler_index]
  ...
Execute:
  ...
  successor = (*next_handler)(context, hdlr_table, next, arg)
Writeback:
  ...
  return successor
}
```

Figure 8: Schematic scode handler.

```
mainloop(...) {
  setjmp(... for exceptions ...)
  128b = tot[paddr:42..22].table[paddr:21..0]

  loop:
    128b = (*global_actions)(..., 128b, ...)
    next = hdlr_table[128b.handler_index]
    128b = (*h)(context, hdlr_table, 128b, 0)
}
```

Figure 9: Schematic main loop.

next microcode instruction in the trace cache. When Decode returns to the dispatcher, the dispatcher calls the handler function of that returned microcode instruction, without further lookup.

Aside from Decode, the interpreter is independent of the guest ISA. This allows construction of hybrid ISAs. For example, modern x86 processors use a combination of fixed logic for simple instructions and wide microcoded instructions for complex instructions. Wherever a complex guest instruction appears, the decoder can emit scode traces of the microcode, allowing design, measurement, and validation of microcode in the context of a full system.

Every trace is terminated by a TraceEnd handler that increments the guest program counter, advances guest time, increments its own scode instruction pointer and returns to the caller, which executes its Writeback phase and returns to its caller. The call stack unwind repeats until the dispatcher is reached. Due to host call/return hardware, the returns have perfect branch prediction.

Handlers that implement guest control flow modify their return value during Writeback. The return value is computed with a very fast table lookup, which, in contrast to traditional hashing techniques, operates over flattened sparse dispatch tables that map guest physical instruction pointers to an internal trace cache index. Figure 7 illustrates trace dispatching.

Some simple trap/fault cases are placed at the beginning of the trace, and simply return the handler for the trap/fault. More complex cases perform a C `longjmp()` to unwind back to the main loop. This adds 100 or more cycles in the case of taken traps/faults, but means zero overhead for the common case of no trap/fault. This structure also allows use of host faults, again unwinding to the main loop. Using host faults can be cheaper than testing explicitly in handlers. In either case, up to the occurrence of a fault or the end of a trace, no architectural guest state is committed, so the trace appears to commit or rollback atomically.

Traces are edited dynamically to implement guest self-modifying code, DMA that writes code, and demand paging that overwrites old code with new code. The first scode in a trace is overwritten with zero, so on reexecution the simulator will decode and execute the new guest code.

Trace editing could be used to implement VLIW "restart under mask", where functional units are selectively disabled after exceptions: handlers for disabled actions would be NOP'd and the 0'th scode set to "Decode", or `global_actions` could restore NOP'd scodes.

Traces are also edited dynamically to implement some

simulator controls. For example, tracing uses TRACE scodes in a trace, and tracing is disabled quickly and selectively by overwriting a TRACE scode with a NOP scode, or by removing TRACE scodes.

Note, in contrast, writing host code means evicting code from the host's instruction cache then re-fetching the modified code from the host's data cache. Further, dynamic translators often regenerate translations instead of patching them. These costs limit where translators can use patching.

4.4.7. Debugging and Instrumentation

To simplify debugging and to improve performance, the interpreter itself does not allocate memory dynamically. All resources are acquired before the interpreter enters its main execution loop, to prevent runtime failures and to save allocation overhead. The simulator allocates some very large tables that are used sparsely but in "dense clumps", for example, the large flat dispatch table. Although much of these tables may go unused, they have essentially zero overhead and are managed efficiently by the host virtual memory system.

On Windows hosts we have implemented a debug extension for the Windows kernel debugger. The debugger can attach to a running simulator process in a non-invasive way, and the debug extension allows us to inspect and manipulate a simulated guest transparently, just as it does with the actual host architecture.

Low overhead instrumentation is a runtime feature, implemented by injecting dedicated handler functions arbitrarily into a trace. To provide maximum flexibility, the interpreter can invoke callback functions from within a trace through a dedicated Hook scode instruction. This approach allows us to enable and disable instrumentation dynamically without recompiling the interpreter code and without paying a penalty for dummy instrumentation hooks.

5. Initial Results and Future Work

After more than two programmer-years of design and development, our fast interpreter is currently in its initial deployment. It is a linkable library with an SDK, so customers can write various tools that integrate the interpreter into existing functional simulation frameworks. We have disclosed source code and the SDK to Intel internal customers like the Simics and Zsim teams, and we are also working on our own simulator infrastructure.

The interpreter has been partly retargeted to several guest instruction sets, example an experimental 64-bit microcode architecture as well as IA32, thus demonstrating easy retargatability. We are not quite able to boot an entire guest operating system stack, but initial results show improved performance compared to full-system interpreters.

The simulator provides a scalable number of internal cores for (currently un-synchronized) multi-core guests, a linear guest physical address space with store logging for transactional memory; and a device manager and first generic device models and interfaces. Guest virtual memory support is implemented via the derived guest architecture Core class, as are TLB and optional caches.

The entire interpreter SDK compiles and runs on Windows, Linux, and MacOS X on both 32-bit and 64-bit hosts.

The scode interpreter supports arithmetic, logic, and shift and rotate instructions with automatic integer datatype widening, lazy flags, conditional and unconditional control flow, a complete segmented load-store architecture with TLB and store logging, synchronous and asynchronous exceptions and interrupts, IOIO and MMIO instructions and device model interfaces, and instrumentation of the interpreted scode instruction traces.

Table 10 shows performance numbers for some isolated benchmarks, listing the number of individual scode instructions and traces (i.e. guest instructions) executed by each benchmark, the execution time, and the number of scode instructions and traces retired per second. The benchmarks were compiled with off-the-shelf VisualStudio and gcc compilers, and measured on a 3.0GHz Xeon running 64-bit Windows7.

The first test is a simple HelloWorld program, and the second and third tests implement various loops that format and print out text and numbers. The two arithmetic tests are loops over pure integer arithmetic. We experimented with different trace lengths: longer traces execute faster, with the reduction in scode instructions due to fewer TraceEnd handlers. The last test implements a LaPlace transformation of a 120x80 image.

Table 11 shows performance of a few micro-benchmarks. Here, the interpreter ran on a 3.4GHz Sandy Bridge machine under 64-bit Windows7. The first two NOP benchmarks merge four guest NOP instructions into a single trace vs. running the NOPs individually. Again, increased trace length improves runtime performance measurably. Executing over 300 million NOP handlers per second demonstrates the efficiency of our core interpreter engine. The next two ADD benchmarks show the overhead of arithmetic flags. Even though the lazy flag implementation is faster than reading host flags, it still incurs significant overhead. The last two benchmarks measure guest load/store performance, where a single memory access is broken into as many as six scode instructions.

In all tests the interpreter dispatches at least 127 million scode handlers per second, with a peak of 307 million

	Runtime	Scode Insn (MSPS)	Guest Insn (MIPS)
Simple HelloWorld	14 ms	2,100,216 (150)	526,807 (38)
printf Loop	44,571 ms	5,670,027,272 (127)	4,830,023,516 (108)
Nested Integer Loop	9,196 ms	1,471,248,646 (159)	1,260,650,284 (137)
Pure Integer Arith 1	3,022 ms	629,145,610 (208)	209,715,205 (69)
Pure Integer Arith 2	2,233 ms	524,288,009 (234)	209,715,206 (94)
LaPlace Transform	5 ms	744,207 (148)	400,272 (80)

Figure 10: Initial results of test benchmarks. The first column shows the test runtime, the second column the number of scode instructions executed, and the third column the number of guest instructions executed.

	Runtime	Scode Insn (MSPS)	Guest Insn (MIPS)
Merged NOP	844.6 ms	260,000,006 (307.85)	190,000,004 (224.97)
NOP	1,787.0 ms	380,000,006 (212.66)	190,000,004 (106.33)
ADD/32	1,326.9 ms	300,000,006 (226.10)	190,000,004 (143.20)
ADD/32 flags	2,312.2 ms	540,000,006 (233.54)	190,000,004 (82.17)
MOV phys	5,032.6 ms	1,140,000,006 (226.52)	190,000,004 (37.75)
MOV TLB	8,525.8 ms	1,140,000,006 (133.71)	190,000,004 (22.29)

Figure 11: Guest micro-benchmarks shows how guest instructions can be merged into single traces (Merged NOP vs. NOP); how computing flags incurs additional runtime (ADD tests); and how complex guest instructions are broken into several sometimes expensive handlers (MOV tests).

when the scode is simple enough and when trace length is increased. Depending on the complexity of the guest instructions this results in 22 to 225 million guest instructions per second, where a single guest instruction maps to a trace of scode instructions. These numbers demonstrate the importance of designing the critical paths of the interpreter around the host architecture's features.

We are currently still in initial deployment, and are well on the way to boot an entire operating system stack, with necessary internal and architectural handlers, fully connected device models for the guest, and accurate virtual timers for checkpoint and deterministic replay.

Summary

In this paper we identified and evaluated real-world workloads and micro-benchmarks, and introduced the design and implementation of a fast interpreter for functional architecture simulation.

In the first part of the paper we describe real-world workloads, software scenarios that average customers of laptop and desktop computers run today. These scenarios involve executing an operating system stack like Windows7 or Linux, and running several applications simultaneously. Investigating these scenarios shows that around 60% of the dynamic instruction count consist of only 15 to 20 different instructions, mainly loads and stores, conditional control flow and function calls and returns, and basic integer arithmetic. Some of these common instructions, however, can be expensive to run if simulated naively.

We then introduce the design and implementation of our portable and fast interpreter. The interpreter gains its performance by

- aligning and flattening critical data structures such as execution context, dispatch tables, and scode traces;
- separating guest and host execution contexts to avoid context switches;
- spreading out call sites across handlers;
- avoiding conditional control flow inside of handlers;
- avoiding high-latency host instructions on critical paths; and
- passing context information to handler functions in registers

We gain further improvements through host-independent lazy flags for guest arithmetic flags, and handler execution in phases for atomic trace commit, rollback, and fast exception handling. Our interpreter supports retargeting through specification-driven instruction decode and guest-independent handlers that can be used as building blocks. It currently runs on 32-bit and 64-bit Windows, Linux, and MacOS X hosts. Initial results show it dispatches 130 to 300 million handler functions per second and 20 to 220 million guest instructions per second, thus exceeding the performance of comparable frameworks.

References

[BDB00] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a Transparent Dynamic Optimization System. In *PLDI: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2000.

[Bel05] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference*, 2005.

[BGA03] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An Infrastructure for Adaptive Dynamic Optimization. In *CGO: International Symposium on Code Generation and Optimization*, 2003.

[Boc10] Bochs. Bochs. Open Source Community Software, 2010.

[CLU02] Cristina Cifuentes, Brian Lewis, and David Ung. Walkabout - A Retargetable Dynamic Binary Translation Framework. Technical Report TR-2002-106, Sun Microsystems Laboratories, 2002.

[Cor10] Standard Performance Evaluation Corporation. The SPEC Benchmark Suite. http://www.spec.org/, 1988-2010.

[Deh03] Jim Dehnert. Transmeta Crusoe and Efficeon: Embedded VLIW as a CISC Implementation. In *SCOPES: 7th International Workshop on Software and Compilers for Embedded Systems*, 2003.

[EG03] M. Anton Ertl and David Gregg. The structure and performance of efficient interpreters. *Journal of Instruction-Level Parallelism*, (5):1–25, 2003.

[Jen02] Jens Tröger and John Gough. Fast Dynamic Binary Translation – The Yirr-Ma Framework. In *Workshop on Binary Translation*, 2002.

[Kep09] David Keppel. Transmeta Crusoe Hardware, Software, and Development. In *ISCA/AMAS-BT: 2nd Workshop on Architectural and Microarchitectural Support for Binary Translation*, 2009.

[kLCM+05] Chi keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa, and Reddi Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *In Programming Language Design and Implementation*, 2005.

[LCL+11] Yair Lifshitz, Robert Cohn, Inbal Livni, Omer Tabach, Mark Charney, and Kim Hazelwood. Zsim: A Fast Architectural Simulator for ISA Design-Space Exploration. In *CGO: International Symposium on Code Generation and Optimization*, 2011.

[MS94] Peter S. Magnusson and David Samuelsson. A compact intermediate format for simics. Technical Report R94:17, Swedish Institute of Computer Science, 1994.

[MS08] Darek Mihočka and Stanislav Shwartsman. Virtualization without Direct Execution or Jitting: Designing a Portable Virtual Machine Infrastructure. In *ISCA/AMAS-BT: 1st Workshop on Architectural and Microarchitectural Support for Binary Translation*, 2008.

[MT10] Darek Mihočka and Jens Tröger. A Proposal for Hardware-Assisted Arithmetic Overflow Detection for Array and Bitfield Operations. In *CGO/WISH: Workshop on Infrastructures for Software/Hardware co-design*, Toronto, April 2010.

[RF95] Norman Ramsey and Mary F. Fernandez. The New Jersey Machine-Code Toolkit, 1995.

[SSB05] Swaroop Sridhar, Jonathan S. Shapiro, and Prashanth P. Bungale. HDTrans: A Low-Overhead Dynamic Translator. In *In Proceedings of the 2005 Workshop on Binary Instrumentation and Applications, IEEE Computer Society*, 2005.

[Trö04] Jens Tröger. *Specification-Driven Dynamic Binary Translation*. PhD thesis, Queensland University of Technology, 2004.