Zsim: A Fast Architectural Simulator for ISA Design-Space Exploration

Yair Lifshitz
Intel Corporation

Robert Cohn
Intel Corporation

Inbal Livni
Intel Corporation

Omer Tabach
Intel Corporation

Mark Charney
Intel Corporation

Kim Hazelwood
University of Virginia

ABSTRACT

Moore's law has enabled next generation CPUs to integrate more functionality from software and peripheral logic – be it graphics, virtualization, or encryption. As integration brings more functionality into the main core, architecting new extensions, quantifying their impact, and validating them becomes more complex.

One way to mitigate challenges arising from this complexity increase is by providing simulation tools. Zsim is an `x86` instruction-set simulator designed to enable rapid prototyping, evaluation, and validation of architectural extensions. It is fast enough to execute full platform workloads – a modern OS can boot in several minutes - thus enabling research, evaluation and validation of complex functionalities related to multi-core configurations, virtualization, security and more. To reach such high speeds, Zsim employs a mix between a simple just-in-time (JIT) compiler that helps simulate simple instructions efficiently, with a fast interpreter used for simulating new or complex instructions.

This paper presents some of the key techniques used to optimize the Zsim interpreter for high performance, including the use of a JIT compiler and several software caches. After presenting an overview of the fast interpreter design, we break down the contribution of each optimization to the overall performance, which results in simulation speeds on the order of 100x faster than a naïve implementation.

Categories and Subject Descriptors

C.0 [**Computer Systems Organization**]: General; B.8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids

General Terms

Design, Experimentation, Measurement, Performance

WISH '11 Chamonix, France

Keywords

simulation, design-space exploration, architectural extensions, rapid prototyping, interpreter optimizations, caching

1. INTRODUCTION

As the ability to scale frequency has diminished, innovation has increasingly relied on changes to ISA and system architecture to deliver improved performance and features. We can no longer evaluate ideas by measuring a new design against an old trace library. Software to exercise new features must be developed and analyzed long before hardware or even RTL is available.

Examples of ISA changes are instructions that increase the floating-point bandwidth of the processor. The compilation tools must be extended to generate the new instructions; runtime libraries must be rewritten to use them, and new applications must be created to evaluate their benefit. Hardware/software co-design features may add new operating modes to host a binary translator and to transition between native code and dynamically optimized code. This requires new instructions and changes to the system architecture. For these types of changes, new software must be developed to exercise the features as part of the evaluation. After design is complete, operating systems, compilers, and debuggers must support the new features as soon as working hardware is available.

Virtual Platforms [3, 10, 7] such as Simics simulate an entire computer system including CPU and peripherals in software. By accurately modeling all parts of a system, it is possible to run system software and a complete application stack on a simulated system before hardware is available. As these systems typically accurately model the operational behavior but not the timing, they are called functional models. The heart of a virtual platform is the CPU model, which simulates instruction execution and system behavior such as address translation and interrupts.

Zsim is a CPU model designed to support ISA experimentation. We believe three key ingredients are required to provide a good framework:

High performance. Architectural extensions usually interact with the operating system. The simulator must be fast enough to boot an operating system and execute a full application stack. Zsim can boot a Linux derivative in 50 seconds.

Extensibility. The simulator must be simple to understand and change so engineers outside of the core develop-

ment groups can make non-trivial extensions. This makes it possible for simulation to be part of the early stage of pathfinding projects. We maintain extensibility by careful partitioning so features that provide high performance do not interfere with understanding.

Interoperability. Zsim leverages the Intel functional simulation ecosystem. It uses Intel's XED[1] encoder/decoder and employs XED's instruction description to generate the instruction-specific glue code of the simulator. It was also able to leverage XED's database to autogenerate a large amount of marshalling code used to prepare the emulation state, reducing the implementation burden. Zsim can share instruction emulation routines with Intel's SDE [7] process-level emulator and Simics's CPU model [10]. Zsim can be plugged into the Simics platform simulator, booting Windows or Linux, and can also execute snapshots of workloads that have been captured for simulation.

The paper focuses on the optimizations employed in the Zsim interpreter. Many optimizations exploit caches to save the result of earlier execution. We ensure the cache has a high hit rate and then try to move all work so it is performed when the cache is filled, and not in the hit path. Optimizations increase the performance by greater than 100x over a naïve implementation.

Section 2 gives an overview of Zsim architecture. A description of the optimizations can be found in Section 3. Section 4 evaluates the performance of Zsim and characterizes the relative contribution of the optimizations. The paper concludes with a summary in Section 5.

2. ZSIM ARCHITECTURE

In this section, we describe the high-level architecture of Zsim. We begin by describing how the Zsim CPU connects to a platform, followed by a description of the basic operation of simulation.

2.1 Platform Simulation

Zsim models the execution of the CPU core. It connects to a platform simulator to provide system memory and devices such as disk, keyboard, and display. Zplay is a simple platform that only provides memory and is sufficient to execute trace-based workloads. Zsim also connects to complete platform simulators like Simics [10] and Bochs [9] and can boot unmodified operating system images. These are all examples of hosted simulations that run inside an operating system user mode process. However, the core of Zsim does not require operating system services and can run on bare metal. or as an execution layer that runs beneath the operating system. This interface allows the platform to request Zsim to execute instructions, inject interrupts, and inspect/modify CPU state. The CPU can read and write system memory, perform I/O, and respond to interrupts.

2.2 Hybrid Interpreter/Just-in-time Compiler

Zsim is a hybrid of an interpreter and just-in-time (JIT) compiler. The interpreter is a complete implementation of a functional simulator written in C. 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 1-1 relationship between simulated instructions and host instructions, and keeping simulated registers in host registers. As the JIT compiler's only purpose is higher simulation speed, it only needs to model the frequently-executed instructions and falls back to the interpreter for complex, less frequent events, such as interrupts.

Zsim's hybrid solution provides maximum flexibility, debuggability, and performance. The Zsim interpreter is portable and easy to understand, extend, and debug. It is structured to provide high performance and can simulate 40 million instructions per second (MIPS). When the JIT compiler is active, performance improves to 150+ MIPS. Since the interpreter is C code, developers can debug at the source level with GDB or Visual Studio. Debugging errors in simulation caused by the JIT compiler is more challenging because developers must debug `x86` code that is generated while the program runs. Extending the JIT compiler is also more difficult because it may require writing `x86` code templates. Compared to the interpreter, the JIT compiler incurs a higher fixed cost the first time it executes an instruction. When the cost is amortized over hundreds of executions, JIT compilation provides a net benefit.

2.3 Interpreter Structure

The interpreter structure closely mirrors the implementation of a non-pipelined, single issue CPU. The front-end fetches and decodes instructions, then advances the instruction pointer for the following instruction. The output of decoding 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 various addressing modes of `x86` are expressed in a canonical form:

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

Operands can come from a register, memory, and immediate. An emulation function specific to the instruction operation and operands is selected.

The back-end dispatches execution to the emulation function. It reads operands from registers or memory and writes back updated values of registers or memory according to the semantics of the instruction. Most emulation functions are simple because the irregularity of `x86` has been factored out by the SIIS. After execution, control passes back to the front-end to fetch the next instruction.

3. OPTIMIZATIONS

In this section, we present the performance optimizations we implemented in the interpreter. We provide the rationale behind each optimization, and discuss any mitigation actions that were needed in the case where the optimization broke any other assumption in Zsim. The optimizations are categorized into *front-end optimizations* which focus on fetching and decoding instructions, and *back-end optimizations* which focus on acquiring necessary values from registers or memory and executing the instructions. Later, Section 4 will present the effectiveness of each optimization.

A standard principle is to make the common path fast. We use an inexpensive test to validate our assumptions about the execution frequency, then execute a highly optimized

[1]XED [8] stands for `x86` encoder/decoder.

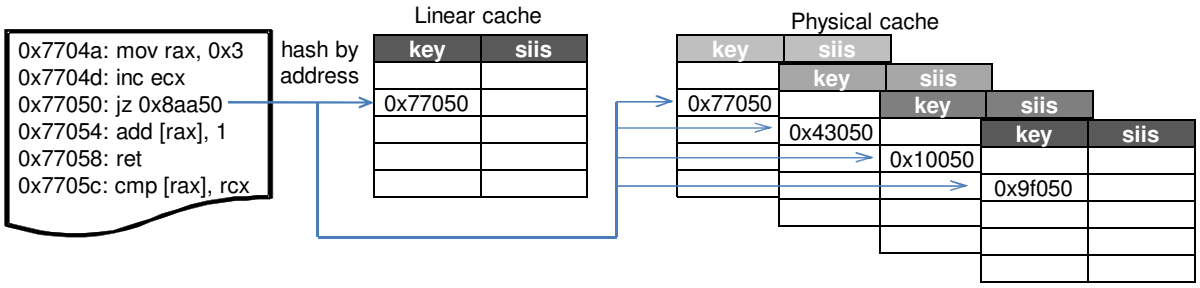

Figure 1: Linear and associative physical instruction caches are used to improve temporal locality. Zsim searches the linear cache first, and upon a miss, it searches the physical cache.

Table 1: Miss rates for various linear and physical cache configurations. Input application: Linux.

Linear	Physical	$Miss_{lin}$	$Miss_{phys}$	$Miss_{tot}$
64K, direct		2.520%	3.150%	0.0793%
4K, direct	64K, 4-way	4.860%	1.620%	0.0787%
1K, direct		10.010%	0.790%	0.0790%
64K, direct		2.540%	4.380%	0.1135%
4K, direct	64K, 2-way	4.880%	2.260%	0.1103%
1K, direct		9.950%	1.130%	0.1124%
64K, direct		2.730%	16.110%	0.4398%
4K, direct	64K, direct	5.010%	8.930%	0.4474%
1K, direct		10.230%	4.290%	0.4389%
	64K, 2-way	3.740%	2.960%	0.1107%
8K, direct	8K, 2-way	3.910%	19.210%	0.7511%
	1K, 2-way	7.420%	67.770%	5.0285%

Table 2: Miss rates for various linear and physical cache configurations. Input application: Gzip.

Linear	Physical	$Miss_{lin}$	$Miss_{phys}$	$Miss_{tot}$
64K, direct		0.007%	3.980%	0.0003%
4K, direct	64K, 4-way	0.074%	0.382%	0.0003%
1K, direct		1.098%	0.026%	0.0003%
64K, direct		0.007%	5.280%	0.0004%
4K, direct	64K, 2-way	0.080%	0.515%	0.0004%
1K, direct		1.109%	0.036%	0.0004%
64K, direct		0.009%	24.308%	0.0021%
4K, direct	64K, direct	0.064%	2.665%	0.0017%
1K, direct		1.180%	0.143%	0.0017%
	64K, 2-way	0.027%	1.440%	0.0004%
8K, direct	8K, 2-way	0.035%	10.520%	0.0037%
	1K, 2-way	0.126%	78.388%	0.0988%

implementation for the common cases. For situations where an event might require a longer path to be taken, we bail-out and perform the full operation, without optimizations.

3.1 Front-End Optimizations

The Zsim front-end fetches and decodes the `x86` instruction byte stream. As described in Section 2.3, it inputs an instruction pointer and outputs a handle to the SIIS, which is passed to the back-end. A straightforward implementation takes ~1000 cycles, with most of the overhead coming from decoding an instruction and the rest from address translation and copying the instructions from memory. This overhead would limit simulation speed to a ~1000-fold slowdown over native (~3 MIPS). Zsim exploits temporal and spatial locality in the executed code stream to reduce the overhead. The temporal locality comes from executing the same instruction multiple times and is exploited by the linear and physical caches. Spatial locality comes from executing the same sequence of instructions repeatedly and is exploited by the trace generation.

3.1.1 The Linear Cache

The linear cache stores the result of previous mappings from the linear address to the SIIS. A linear cache hit avoids address translation, fetch, and decode. To optimize the hit path, we use a direct-mapped cache. The cache size is a power of two, and Zsim uses the result of masking off high bits in the linear address to index the cache. By default, there are 64K entries.

Any change to linear to physical address mapping (e.g. writes to CR3) requires the entire cache to be invalidated. As a consequence, the cache incurs a large number of compulsory misses after an operating system process swap. A partial invalidate is required when there is a conflict miss or anything that overwrites the original `x86` code in memory. This can occur because of operating system paging or self modifying code. A software TLB detects when instructions have been overwritten, as described in Section 3.2.2.

3.1.2 The Physical Cache

The physical cache is similar to the linear cache, except that it is indexed by physical address, requiring the instruction pointer be translated from linear to physical first. A physically indexed cache does not need to be invalidated when there is an address space change and can be shared among multiple cores. When there is a miss in the linear cache, we try to fill from the physical cache. Linear cache misses caused by address space changes or cache conflicts usually hit in the physical cache. Like the linear cache, partial invalidates are required on conflict misses and anything that overwrites cached instructions. A miss in the physical cache requires a very expensive decode so it is worthwhile to use a more complex lookup and reduce misses caused by conflicts. Zsim uses a 4-way associative 64K entry cache with LRU replacement.

3.1.3 Cache Size Interactions

The fact that Zsim has both a linear and physical cache is a unique feature. Tables 1 and 2 demonstrate the complex interactions between the two caches with respect to cache

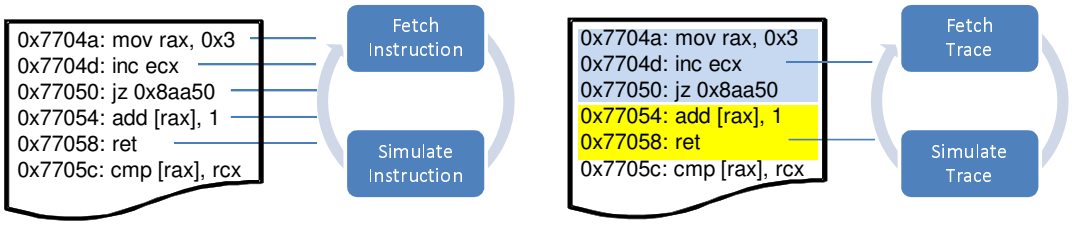

Figure 2: Fetch/simulate loop for regular (left) and trace cache (right). The use of a trace cache reduces the number of fetches from 5 to 2.

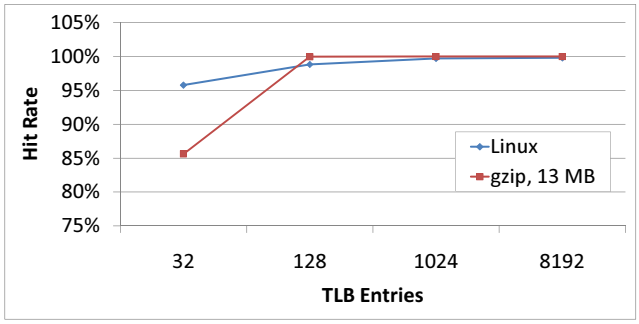

Figure 3: Hit rate for various TLB sizes.

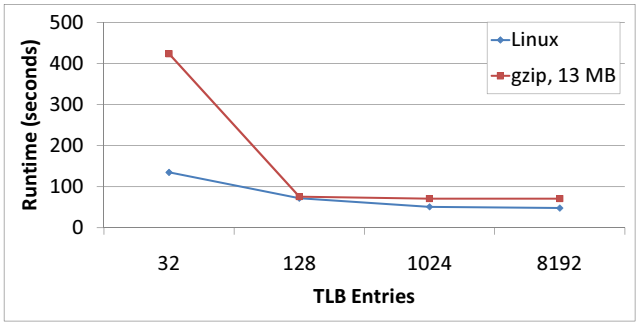

Figure 4: Resulting runtimes for various TLB sizes.

sizing and organization. The tables report the miss rates of the linear cache, the physical cache, and the combined miss rate. Note that a table entry that reads "64K 4-way" set associative has a total of 256 cache entries. An interesting trend from the tables is that the physical cache miss rates improve when the linear cache size is reduced.

3.1.4 Forming Traces

Tracing consists of storing a sequence (trace) of executed instructions and performing the decode-simulate cycle on the entire grouping, rather than a single instruction at a time. Tracing exploits spatial locality by retrieving a sequence of SIIS with a single lookup. The linear and physical cache entries contain a trace instead of a single SIIS, so cache management does not change. Fetching an entire trace of instructions amortizes the overhead of cache lookup over the execution of several instructions. Traces are terminated whenever an instruction in the trace modifies the RIP – thus breaking the ability to safely determine the next instruction – or when the trace become excessively long (usually >20 instructions). Table 3 shows that traces average just under 5 instructions in length, varying slightly by benchmark.

Table 3: The average length of the instruction traces generated by Zsim.

Application	Average Trace Length
Linux boot	4.979 instructions
gzip	4.695 instructions

3.2 Back-End Optimizations

After an instruction is fetched and decoded, the Zsim back-end performs the actual heavy-lifting. This typically involves fetching the operands from registers or memory, performing the actual operation, updating condition code flags, and storing the result to memory or registers. Since each instruction has different logic, it is hard to optimize each instruction manually. Optimizing a specific 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.

3.2.1 Direct Memory Interface

When a simulated instruction references memory, Zsim may need to invoke the platform to complete the operation. The address may be for system memory or could be connected to a peripheral device for memory-mapped I/O. By default, Zsim calls out to the platform for every memory read and write operation, passing the address and data pointer. To avoid the cost of the function call, Zsim 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 Zsim uses it to directly access the memory. The platform returns an error if Zsim requests a host pointer for memory-mapped I/O, and Zsim falls back to using the function interface to memory. Zsim caches host pointers for future references to the same memory page.

3.2.2 Translation Look-Aside Buffer

Address translation is the most common operation performed by the CPU. The CPU must translate the linear instruction pointer to a physical address before fetching the next instruction. Many instructions operate on data from memory, which requires another address translation. The linear instruction cache presented in Section 3.1.1 eliminates most translations for instruction references, leaving the data references to be executed by the back-end.

While a hardware TLB translates linear addresses to phys-

ical, the Zsim TLB must translate to both physical addresses and host pointers discussed in Section 3.2.1. The Zsim TLB efficiently translates addresses from linear to physical and host in ~25 cycles. It contains 1024 entries and uses direct indexing. We arrived at this number of TLB entries by finding the knee in the hit rate and performance curves, as shown in Figures 3 and 4.

The `x86` ISA supports multiple page sizes, but a software implementation of variable page sizes would have required a more expensive lookup function. Instead, Zsim TLB entries only map 4k pages in the address space. The number of Zsim TLB entries used for a large page depend on how much of the page is referenced. While the address translation algorithm itself is straightforward, caching the results efficiently to allow fast translations is trickier, as several checks need to be performed for every access:

1. Check if a cached translation exists
2. Validate permissions (read/write/execute/privilege-level)
3. Detect page-crossings that would require cross-page memory access

In order to perform all checks and the actual translation efficiently, each TLB entry contains the base linear address of the page; a host_offset which, when added to the linear address, marks the actual location in host memory; and a `sizes` two-dimensional array, consisting of privilege level and access type. The `sizes` array contains a series of "valid" bits that contain 0 when the current entry is invalid, or 4096 (page-size) when the entry is valid. When address translation is requested by the simulator, the following uber-check is performed:

```
if (linear - entry->linear + size <= entry->size[cpl][access])
    return PTR_CAST(void*, entry->host_offset + linear);
```

This single check essentially wraps all three checks together: if the entry is invalid, entry->size[cpl]access[] will be 0, and since the size passed to the function is always >= 1 the check fails. If there are no appropriate permissions to the given page, entry->size[cpl][access] is again 0, leading to the same failure. For page crossing, (linear + size) would cross over to the next page, and so the difference would be bigger than a page size (4096), whereas entry->size[cpl][access] is either 0 or 4096, and so the check would fail again.

If the above check fails, a more detailed routine is run to identify the exact failure reason and decide whether the TLB simply needs to be filled (missing entry), there is an access violation, or a page-crossing memory reference. Just like hardware, Zsim TLB entries must be invalidated when CR3 changes or pages are explicitly invalidated by an instruction.

As described in Section 3.1, the linear and physical instruction caches must be invalidated when `x86` code is overwritten in system memory. We use the Zsim TLB to detect when that occurs. When an entry is filled in the instruction caches, write permission for the containing page is removed in the Zsim TLB. A write access that fails the above check causes Zsim to invalidate any entries in the linear and physical instruction caches for the same page. This is similar to the technique used in binary translation [1, 5]. Although Zsim does not generate code, the SIIS is in effect a representation of the instruction stream that needs to be kept consistent with memory.

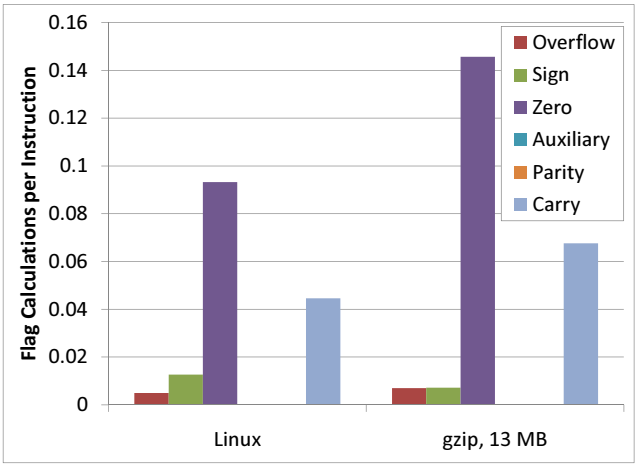

Figure 5: Status flags usage. This figure reports the average number of flags calculated/used per instruction executed. The parity flag was rarely used, and the auxiliary flag was never used.

3.2.3 *Lazy Status Flags Evaluation*

Status flags are another big contributor to simulation runtime. Most arithmetic and logic instructions – ADD, SUB, AND, XOR, etc. – update most or all status flags. These are sometimes used later, mostly to determine branch targets. The six status flags are sometimes referred to as OSZAPC: Overflow, Sign, Zero, Auxiliary carry, Parity, and Carry. Most flags have straightforward behavior, with the zero flag raised when an instruction's result is zero, sign flag raised when an arithmetic instruction's high bit is set to 1, etc.

While most flags are easy to calculate, some are more complex (e.g., the overflow flag). Moreover, arithmetic and logic instructions are commonly executed, and calculating six flags per instruction is very expensive. An earlier version of Zsim used the underlying hardware to perform the instruction and then read the status flags from the actual register. We switched to another technique called lazy flags because Zsim needs to be able to simulate 64-bit mode while executing in a 32-bit process. In that situation, you cannot execute a 64-bit instruction to generate the flags result. Using the direct hardware technique also obscures the logic to calculate the flags, defeating the purpose of having an easy-to-read specification for architects and software developers.

The concept of lazy flags [12] evaluation leverages the fact that most flags are rarely used, so calculating them is unnecessary in most cases. To avoid calculating the status flags, relevant instructions were modified to write source operands, operation, and the final result into a temporary structure. Whenever a flag value is requested, either the result is used for flags which are not affected by the input operands (e.g. zero flag), or else the full calculation is performed for the more complicated flags (e.g. overflow flag). Figure 5 shows the relative number of flag calculations per instruction for two workloads: booting Linux, and zipping a 13MB file: while the zero flag is used for 10%-14% of the time (~1 out of 7 instructions), the carry flag – which is next in line – is calculated for only about 1 out of 15 instructions. All other flags are rare, being used for less than 1 out of 100 instructions.

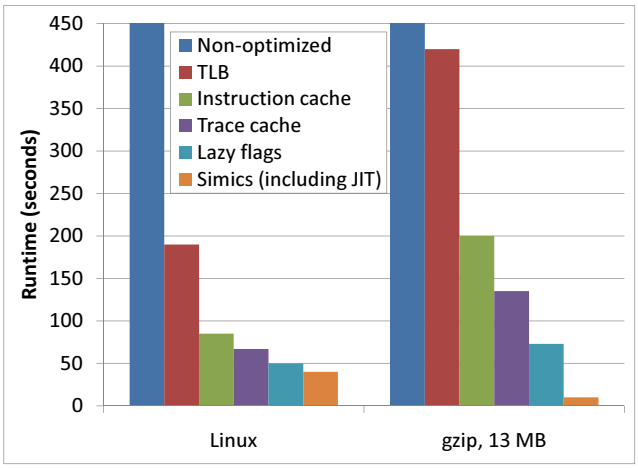

Figure 6: CPU time in seconds. Non-optimized bars are limited to 450 seconds for scale. Actual values: 3630 (Linux), >7200 (gzip)

4. EVALUATION

All front-end and back-end optimizations from Section 3 were implemented in Zsim. To measure their impact, we turned off all optimizations except for Zsim's decode cache, and then enabled them one-by-one, in the order they were implemented: TLB, instruction cache, trace cache, and finally lazy flags. For reference, we have also included the performance of the Simics instruction set simulator.

Figure 6 provides the runtime, in seconds, of two workloads: Linux boot, and zipping a 13MB file. The TLB improves runtime dramatically. Note that we did not run the non-optimized gzip to completion, it was interrupted after 7200 seconds.

The benefit of each optimization changes depending on the workload: a boot workload tends to have a large code footprint with less repetition, while compression is loop intensive with a small code footprint. As all the optimizations mentioned rely on locality (data locality for the TLB, code locality for the instruction cache), code-block length (trace cache) or the amount of arithmetic operations performed, compression lends itself better to each of the optimizations. The instruction cache provides approximately a 2.1x improvement over the TLB-only implementation; the trace cache provides a 1.26x improvement for Linux, and 1.48x improvement for gzip; and the lazy flags approach provides a 1.34x improvement for Linux and 1.84x for gzip. For reference, we have also shown the Simics instruction set simulator. Note that we are comparing our interpreter to the Simics just-in-time compiler. Interestingly, our interpreter is only slightly slower than Simics on the boot workload, but significantly slower for gzip. Our hope is that the interpreter is fast enough for most experimentation because it is easier to extend and analyze than a JIT compiler.

Overall, implementing all of the above improvements provides a 72x performance improvement for Linux boot, and over 100x improvement for gzip, providing acceptable performance for interpreted-mode `x86` simulation.

Many simulators are geared toward architectural experimentation [2, 13, 14, 4, 6, 11]. The system most similar to Zsim is Bochs [9]. Unlike Bochs, Zsim features a JIT, which relieves some of the pressure on interpreter performance. Zsim is a pure CPU with an abstract interface that allows it to work with any platform that implements that interface. Zsim can also run on bare metal. Our work builds upon that of Mihocka and Shwartsman [12], but we use both linear and physical caches. We also provide a detailed measure of the relative contribution of optimizations and analyze sensitivity to cache size. QEMU [3] has a JIT but no interpreter, which makes it more difficult to change and debug.

Because Intel prototypes its new instructions in XED and SDE, Zsim can leverage that commonality for architecture exploration and enabling. Zsim already supports the cutting edge instruction set features and confidential features. The design allows for layering so that we can make public or private versions with different extensions.

5. SUMMARY

As the ability to scale frequency diminishes, the necessity to provide value through architectural innovations has grown significantly. Unlike frequency scaling, these innovations require a different set of capabilities in order to mature, including the abilities to prototype new architectural extensions, develop and debug content for non-existent hardware, and analyze the platform-level behavior of the CPU.

Zsim provides a fast `x86` core model that uses a hybrid interpreter/JIT compiler approach to provide fast simulation for both current and future ISA extensions; the JIT compiler takes care of older/simpler instructions, while the interpreter handles complex instructions, or ones not yet supported by the host hardware. The interpreter is also used for prototyping, as each instruction is coded in C, thus making it easy to understand and modify for anyone outside the Zsim core development team – as opposed to the JIT compiler which requires more time and deeper understanding of JIT compilation. Therefore, solid interpreter performance is key to enabling a decentralized architectural exploration model.

The front-end of the Zsim interpreter needs to generate a SIIS handle given the instruction pointer. We use a multi-level cache to exploit spatial and temporal locality in the executed instruction stream. The back-end does the actual execution, and our optimizations focus on the work that is common to most instructions – address translation, memory access, and condition code evaluation. For repetitive workloads such as gzip, the JIT compiler is significantly more efficient than the current Zsim interpreter. For more complex workloads, both are in the same ballpark, and Zsim's interpreter performance is good enough to boot a modern OS quickly, allowing any type of workload to be run.

To summarize, Zsim provides a high-performance, extensible solution for `x86` core simulation. Optimizations implemented in Zsim's interpreter allow it to run anything from simple arithmetic workloads to a full OS with adequate performance; and as Zsim's interpreter provides the basis for evaluating new architectural extensions, these are a key enabler in driving Zsim to the architectural exploration world.

Acknowledgments

We would like to thank the entire Zsim team for their extensive development work that has made this paper possible, including but certainly not limited to Shmulik Brantz and Alex Raginsky. We also wish to thank the anonymous reviewers for their feedback and suggestions.

6. REFERENCES

[1] Apple. Rosetta. http://www.apple.com/rosetta/.

[2] T. Austin, E. Larson, and D. Ernst. Simplescalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, February 2002.

[3] F. Bellard. Qemu: a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference*, ATEC '05, pages 41–46, Anaheim, California, USA, 2005.

[4] B. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 128–137, Nashville, Tennessee, USA, May 1994.

[5] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The transmeta code morphing software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proceedings of the 1st Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '03, pages 15–24, San Francisco, California, USA, March 2003.

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

[7] Intel. Intel software development emulator. http://www.intel.com/software/sde.

[8] Intel. Xed: An x86 encoder/decoder. http://www.pintool.org/docs/39599/Xed/html.

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

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

[11] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Computer Architecture News*, 33(4):92–99, November 2005.

[12] D. Mihocka and S. Shwartsman. Virtualization without direct execution or jitting: Designing a portable virtual machine infrastructure. In *Proceedings of the 1st Workshop on Architectural and Microarchitectural Support for Binary Translation*, AMAS-BT '08, Beijing, China, June 2008.

[13] M. Yourst. Ptlsim: A cycle-accurate full system x86-64 microarchitectural simulator. In *Proceedings of the ACM International Symposium on Performance Analysis of Systems and Software*, ISPASS '07, San Jose, California, USA, April 2007.

[14] H. Zeng, M. Yourst, K. Ghose, and D. Ponomarev. Mptlsim: a simulator for x86 multicore processors. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pages 226–231, San Francisco, California, USA, 2009.