# Portable Trace Compression through Instruction Interpretation

Svilen Kanev
Harvard University
*skanev@eecs.harvard.edu*

Robert Cohn
Intel Corp.
*robert.s.cohn@intel.com*

***Abstract*—Execution traces are a useful tool in studying processor and program behavior. However, the amount of information that needs to be stored makes them impractical in uncompressed form. This is especially true for full-state traces that can capture up to kilobytes of processor state for every instruction. In this paper we present Zcompr – a compression scheme that allows practical usage of full-state traces that are billions of instructions long. It allows complete state reproducibility, sufficient even for validation purposes, that is fully portable between different operating systems and host platforms. The compression scheme exploits the general similarity between compression and prediction. A simplified functional simulator is used to predict instruction effects in a repeatable manner. Its predictions can be used to reproduce those effects at decompression time, limiting the amount of information that needs to be stored per instruction. Final trace densities achieved by our scheme are on the order of two bits per instruction, with typical decompression speeds of 300 KIPS.**

## I. Introduction

Full-state traces contain the entire architectural state of a computer system after each execution of an instruction. Architectural state includes all general purpose and floating point registers, control registers, model-specific registers, and memory. A full-state trace is useful for a variety of tasks in computer system design. Performance models use traces to drive simulation. An instruction cache model can use the sequence of instruction pointers to predict an instruction cache hit rate, while a cycle accurate model needs the full register and memory information contained in a full-state trace. A microprocessor design can be validated by comparing the full-state trace of a reference model against the observed behavior of a system under test. Differences in architectural state between the two are likely to be defects. While dedicated formats currently exist for these trace uses, being able to derive the data for a specialized format from a full-state image is certainly beneficial.

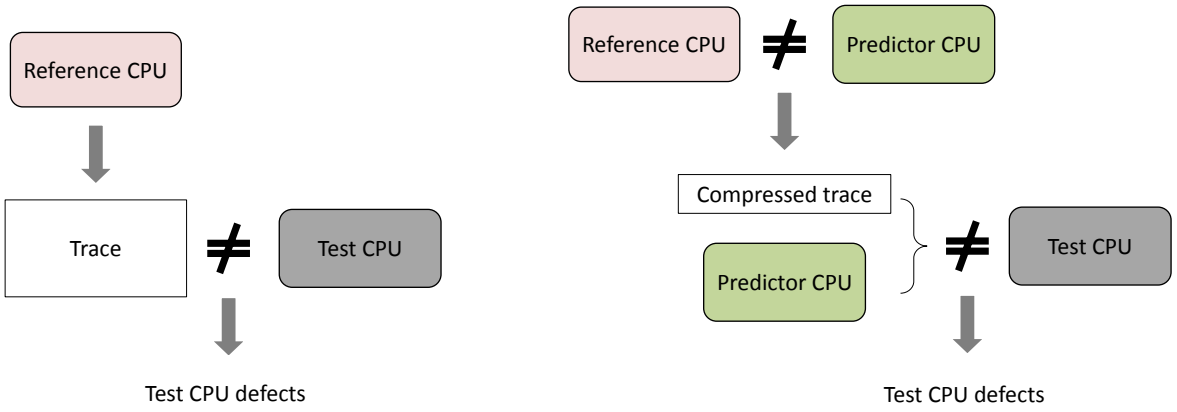
Full-state traces decouple the collection of behavior and analysis. Collecting the behavior may require special hardware such as a logic analyzer. Generating the behavior may need access to a specific hardware/software combination that cannot easily be replicated, such as in a transaction processing workload that requires thousands of disks. Generating the behavior may be very expensive, for example an RTL simulator that can only simulate a few instructions per second. By decoupling collection and analysis, we eliminate dependence on reproducing the environment that generated it.

Full-state traces are efficient to analyze because they can be broken into independent chunks. A trillion instruction trace can be divided into slices of a billion instructions each and distributed to a thousand computers to be analyzed in parallel. Traces make fast-forwarding easy, allowing analysis to instantly skip any part of the execution. However, a direct encoding of a full-state trace is impractical because it must include the state of all registers and memory after each instruction. Even disregarding memory, a current-generation IA32 architecture [1] would require storing kilobytes of register state per instruction.

Simulators are an alternative for compactly representing the full execution state by relying on simulated execution to reproduce register and memory values. However, they can only faithfully reproduce behavior of microprocessors they were designed to implement; they continually need to be updated to implement new model-specific features like new instructions.

In this paper, we describe a technique for compressing full-state traces. It provides the storage efficiency and encapsulation of simulators, can exactly reproduce any model-specific behavior, and allows rapid decompression. We show that a typical full-state trace consumes 2 bits per executed instruction and can be decompressed at a rate of 300k instructions per second.

Like a simulator, we use an initial state and microprocessor simulation to reproduce a sequence of states, rather than storing the results. Unlike simulation, we do not assume that our model can exactly reproduce the behavior. We instead use the simulation as a *prediction* of the behavior and only need to record differences between actual and predicted behavior. A predictor doesn't need to be perfect, allowing us to use a much simpler simulator that only accurately models the frequently executed instructions. Misprediction reduces compression, but does not affect correctness. Complex and model-specific behavior like exceptions can be completely




(a) A full-state trace decouples running reference CPU from validating test CPU.

(b) The same full-state trace compressed/decompressed using interpreter-based prediction.

Figure 1. Sample co-simulation workflow that validates a *test CPU* model against a *reference CPU*.

omitted since it does not occur frequently.

During decompression, we use the simulation and the recorded differences to exactly reproduce the original behavior. A simpler simulator enables fast decompression because it can eliminate checks in the critical path for events that rarely occur.

This paper starts by presenting simulator *co-simulation* as an environment that requires compact full-state traces (Section II). It goes on to describe *interpretation-based compression* as an approach that allows such compaction (Section III), its limits (Section IV) as well as details about our implementation of such a system (Section V). This is followed by effectiveness evaluation (Section VI) and specific optimizations to our baseline implementation (Section VII). The paper concludes after discussing alternative systems (Section VIII).

## II. Motivation

In this section, we introduce a specific use-case for full-state traces: validation of simulators through co-simulation. After providing background, we describe the requirements for a portable, efficient full-state tracing system.

### A. Full-state Traces for Co-simulation

A functional model for a microprocessor simulates the execution of instructions but not necessarily the exact timing. It is used throughout the microprocessor design cycle. Early in the design, functional models are used to drive timing models for performance projections. Software needs to be developed for the microprocessor before the hardware becomes available, so functional models are incorporated into a virtual platform that can execute the full software stack including an operating system.

Requirements for functional models vary by use. For example, a RTL model is developed as an intermediate step before physical implementation and can also serve as functional model. An RTL model executes very slowly (10 IPS) and is labor-intensive to create because it models the exact hardware. At the other end of the spectrum, virtual platforms for software development systems must be very fast (100 MIPS) and do not need to model hardware features that are not visible to software. Performance modeling lies between the two in terms of performance requirements and level of detail.

The existence of multiple models makes it possible and desirable to use *co-simulation* to test one model for correctness by comparing its behavior against another. In its simplest form, we single step two models and compare the entire state of each system after executing an instruction. Any differences are likely to be a defect in one of the models. After a difference is detected, the two models can be synchronized to the same state and continue execution. Resynchronization makes it possible to continue execution past benign differences caused by undefined behavior as well as bugs that are to be fixed later. This is especially beneficial in initial development stages where bugs are numerous and not ordered by ease of fixing.

A straightforward approach to co-simulation is to simultaneously execute the two models. However, an online comparison is not always practical. If an RTL model can only execute a few instructions per second, recording a full-state trace of an execution and consuming it multiple times is more efficient than repeatedly running the RTL simulation. Online comparison of a workload running in a virtual platform can be unwieldy because the virtual platform workload may not be easily automated (mouse clicks or keys pressed), may require

```
# Initial State
REG RAX 0x7f
REG RIP 0x80400000
SYNC
# add rax, qword ptr [rcx]
MEM_READ 0xc0000000 0x1
REG RAX 0x80
REG RIP 0x80400004
STEP
# sub rax, qword ptr [rcx]
REG RAX 0x7f
REG RIP 0x80400008
STEP
```

Figure 2. Sample difference log entry for the effects of an `add` and a `sub` instruction. Note that a *STEP* command is used to explicitly define instruction boundaries.

10's of gigabytes of virtual disk, or may be dependent on a specific host operating system or processor features.

Collecting a full-state trace of a workload running in a virtual platform solves these issues by removing all dependence on the environment that generated it. Figure 1a visualizes this decoupling.

### B. Full-state Trace Compression Requirements

The characteristics of co-simulation testing define some requirements for full-state trace compression. The reference behavior can come from multiple sources such as RTL models or high speed instruction set simulators, so we don't want to be tied to a specific system. The speed of decompression must be high enough so it does not limit the amount of testing done in a fixed time. Compression speed is less important because we typically collect a reference trace once and consume it multiple times in a regression test. Traces are distributed over the network and transfer time can be a bottleneck for a large trace.

## III. Interpretation-Based Compression

In this section, we introduce interpretation-based compression. It is a compression scheme designed with co-simulation in mind. A co-simulation workflow with compressed traces is shown in Figure 1b. The compression process takes three steps. A *collector* single steps a *reference* system, recording a difference log of the effects of executing each instruction. Next, an interpretation-based predictor compresses the difference log. Finally, a general-purpose compressor compresses the output of the interpretation-based compressor. After a similar decompression sequence, the trace can be used as a substitute for the reference system execution.

#### *Step 1: Collection*

Instead of recording full state after every instruction, the trace only contains changes to the state caused by the reference CPU executing an instruction, which we call *difference logs*. Difference logs are collected by observing the execution of the reference CPU. The trace collector maintains a shadow state containing registers and memory. The shadow state is initially set to all zeros. Before starting execution, the collector copies the reference CPU register values to the shadow state, recording the values that are non-zero as shown in Figure 2. Next, the collector emits a *SYNC* command, indicating that the shadow state and reference CPU registers are synchronized and execution can begin.

The collector single steps the reference CPU, observing memory reads and writes. Memory read and write values are copied to the shadow state and recorded in the difference log if they differ from the previous value for the respective location. After the reference CPU completes execution of an instruction, the collector copies the shadow state registers to the reference CPU registers, recording any values that change. A *STEP* command indicates an instruction boundary. In the example in Figure 2, the `ADD` instruction reads 0x1 from memory location 0xc0000000. The initial shadow state of this memory location is 0 so the new value is recorded. Register *RAX* and the instruction pointer change values; the collector records the new values in the log.

With this scheme memory reads are not recorded if the shadow state already contains the correct value, which occurs when value is the result of a previous write. This is a similar approach to [2] and [3] – rather than keeping a full memory image, only values that are actually used by the workload and are not the result of earlier computation are recorded.

With this approach the collector is intentionally simple; it has no understanding of the semantics of execution, only recording values that change. Most instructions change the instruction pointer and a general purpose register, requiring at least 2 registers to be recorded. A straightforward binary encoding of a difference log averages 20.5 bytes per instruction.

#### *Step 2: Interpretation-based Prediction*

Prediction and compression are closely related. If a byte stream can be correctly predicted, then a compressor can note that the prediction is correct and omit the data. At decompression time, the predictor regenerates the data. Data that cannot be predicted is included in the compressed file.

This similarity has been exploited for branch prediction and prefetching [4], [5], [6]. Burtscher et al. [7] use value prediction as a way to compress simple trace formats. Our scheme, Zcompr, also uses this approach, but

```
# Initial State
REG RAX 0x7f
REG RIP 0x80400000
SYNC
# add rax, qword ptr [rcx]
MEM_READ 0xc0000000 0x1
REG RAX 0x80
REG RIP 0x80400004
STEP
# sub rax, qword ptr [rcx]
FINE 1
```

Figure 3. Result after interpretation-based compression. Predicted side effects are replaced with a *FINE* command.

applied to full-state logs and on instruction granularity. We call such an operation *instruction effects prediction*.

The compressor starts with a full-state difference log as an input. Its output is a difference log, extended with commands for compression. A simple *Predictor model* makes a heuristic guess about the register and memory side effects of executing the next instruction. The guess is checked against the register and memory values from the input log. If the prediction is correct, the compressor discards the side effects in the difference log and emits a *FINE* command. Otherwise, the full instruction effects are transferred directly to the compressed log. As expected, long sequences of instructions can be correctly predicted and the *FINE* command takes a count of the number of correctly predicted instructions. Figure 3 shows a log that has been compressed with a *FINE* command.

When the decompressor reads a *FINE* command from the compressed log, it executes the predictor model to reproduce the effects of each instruction in the *FINE* block. For other instructions, the effects are simply read off the log.

In our framework, the predictor model can be any instruction set simulator that meets the following requirements:

- supports single-stepping and inspection of register/memory values
- reports memory values read/written by an instruction
- for a given input state, always produces the same output state, even if the effect of executing an instruction is undefined or model-specific

The last requirement ensures that the decompressor exactly reproduces the input stream even if compressor and decompressor are executed on different systems.

In this respect, any change in behavior in the CPU model, even to correct incorrect execution, potentially invalidates all previously collected traces because there are no guarantees that decompression will reproduce the original full-state trace. We have adapted existing simulators to be predictor models. However we have also implemented a CPU model specifically designed for compression to avoid the need to change the predictor and invalidate already collected traces.

By designing a predictor rather than an accurate simulator, we can develop a customized model that better controls reproducibility, portability, and complexity. If the predictor returns inaccurate results for a given instruction, the compression ratio is reduced but compression is still correct. For the IA32 architecture, much of the complexity in developing a simulator is the combination of segmentation, paging, interrupts, faults, tasking, and a large CISC instruction set. Our model only needs to implement basic segmentation and paging support and totally omits interrupts, faults, and tasking. The predictor only implements the instructions that are frequently executed. It is difficult to make a simulator fast while properly handling exceptions and self modifying code; it is possible to get good performance without heroics when the complex parts of the CPU can be ignored.

*Step 3: General Purpose Compression*

The output of the interpretation-based compressor is dramatically smaller. However, there is still redundancy in its output. The gzip compressor is effective in reducing the output further and is the final step in compression. We use gzip, rather than better-compressing algorithms like bzip2, mostly because of faster compression and decompression speeds. In this way, the speed overheads of two-stage compression are less than 1% and we don't include them in subsequent timing data.

## IV. Limitations on Compression

In this section, we characterize some of the factors that limit the compression in order to evaluate our design decisions. Incorrect predictions limit compression and can either be caused by external events or by inaccurate modeling. We examine the two causes separately.

### A. Unpredictable External Events

Events that originate outside the model are inherently unpredictable and limit the effectiveness of compression. We chose to only model the CPU in our predictor, so external events like IO, interrupts, and initial memory state cannot be predicted, even with a very precise CPU model. Analysis of some workloads quantifies the effect of that decision.

Figure 4 shows a breakdown of commands in an uncompressed difference log. The *IO_READ* and *MEM_READ* commands are caused by instructions that read IO and memory and are the major source of external events. The distribution in Figure 4 shows that, on

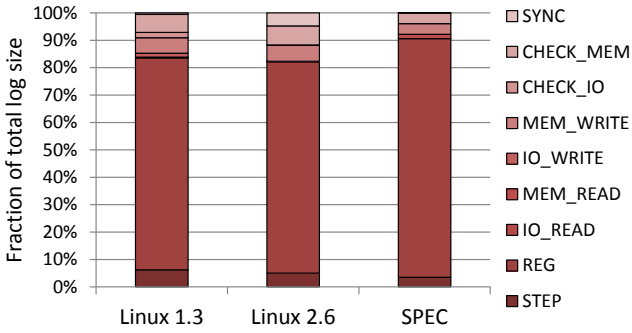



Figure 4. Distribution of trace commands. Over 97% of the log size is taken up by predictable effects.

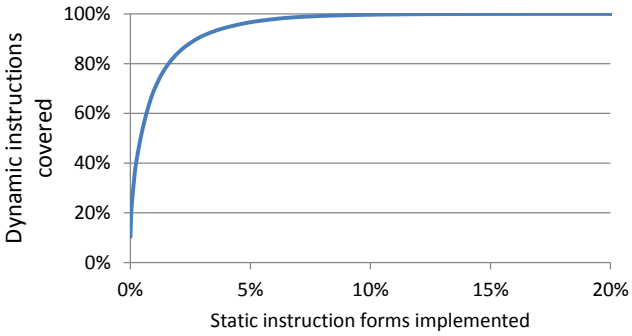



Figure 5. Cumulative histogram of runtime instructions over instruction forms.

average, 2.8% of the uncompressed log size consists of unpredictable events. Therefore, compression ratios of $30-40\times$ are possible using our scheme. Interrupts are not accounted for in this graph, but since interrupts typically occur less than once every 10,000 instructions, their contribution to log size is insignificant compared to IO and memory.

### B. Inaccurate Modeling

The compression ratios in the previous section optimistically assume that the predictor functional model can produce accurate results for all CPU events. Faults are even less frequent than interrupts and not modeling them does not change the compression ratio. The IA32 instruction set is large, complex, and evolves over time and it is difficult to accurately model the entire instruction set. According to our design principles, the ideal predictor model should be fast and simple (not model-specific). Additional workload analysis implies that designing a simple CPU model that covers a significant fraction of dynamic instructions is possible. Figure 5 shows the cumulative distribution of dynamic instructions over static instruction forms. By instruction form, we mean the combination of instruction semantics, type, and size of operands. For example, `MOVSX eax, al`, `MOVSX eax, ax` and `MOVSX [eax], al` are all considered different instruction forms. The very fast rise of the cumulative distribution suggests that a functional model that supports a small fraction of the instruction forms defined in the ISA can cover the majority of dynamic instructions. In detail, to achieve 99.9% coverage, the model only needs to implement less than 10% of the instruction set. The intuition behind these results is that most code is not executed in a straight line sequence and instructions are often repeated in loops. It is essential for a general-purpose functional model to accurately implement the entire ISA, but we can simply rely on the collector to include the effects of infrequent instructions in the difference log.

## V. Implementation

This section describes relevant implementation details for the above approach. First, we discuss the design of the predictor CPU model created for Zcompr. Then, we move on to its input – focusing on the approach we use to identify and handle unpredictable events.

### A. Predictor model

The predictor model used by Zcompr is in essence a custom-built functional simulator. Internally, the predictor follows the classical model of an instruction interpreter. First, in a fetch stage, memory requests are generated and a byte queue is filled with the response. Then, instructions are decoded from the byte queue. For efficient and up-to-date IA32 decoding Zcompr uses XED [8]. Finally, based on the instruction form, execution is forwarded to the appropriate emulation routine, which updates architectural state based on the instruction semantics.

As already stated, the predictor's main aims are neither completeness, nor 100% accuracy. It can be thought of as an instruction effects cache – predicting the effects of simple, general and frequent instructions, but omitting complex and rare ones. Thus, there is a large room for various optimizations that would not be applicable if fidelity had to be guaranteed.

Incomplete instruction support is such an optimization. In Section IV-B we showed that covering only 10% of the static instruction forms can result in significant coverage of the dynamic instructions executed. The current version of our predictor core recognizes ∼80 instructions and ∼500 instruction forms (compared to more than 2400 defined by the ISA). Instruction support was added based on the frequency in the examined workloads.

In a similar manner, the predictor does not support the complete IA32 architectural state. Modeling infrequently used registers (such as debug or model-specific ones) or even execution modes (such as Virtual-8086 mode) is intentionally omitted. As a general design principle, we optimized for the common case, modelling inaccurately

```
# mov rax, qword ptr [0xff]
MEM_READ 0xff 1
# REG RAX 1
# STEP
# add rax, 2
# REG RAX 3
# STEP
FINE 2
```

(a) An instruction sequence that can be compressed despite the unpredictable memory read in its beginning.

```
# mov rax, qword ptr [0xff]
MEM_READ 0xff 1
# REG RAX 1
# STEP
FINE 1
# mov rax, qword ptr [0xff]
MEM_READ 0xff 2
# REG RAX 2
# STEP
FINE 1
```

(b) Splitting unpredictable memory reads from the same address with a *FINE* command.

Figure 6. Handling of unpredictable events. Commands preceded by # are removed from the uncompressed log.

or even omitting infrequent or exceptional behavior. This allows simpler and more compact code, as well as speed improvements because checks for infrequent behavior are not conducted.

On the other hand, other widely used optimizations cannot be safely used for our predictor. For example, Xen [9], VMWare [10], and KVM [11] achieve high performance by assuming that a guest instruction can be simulated by executing the same instruction on host hardware. This creates the possibility that the predictor may behave differently if compressor and decompressor are run on different hosts, leading to incorrect decompression.

The differences in behavior come from model-specific and undefined behavior. Every processor generation adds new instructions. Thus, if the compressor host provides the instruction, its effects are eliminated from the difference log. However, if the decompressor host does not have the instruction, then it is not able to regenerate the results. A more subtle problem occurs for undefined behavior. For example, the overflow condition code is undefined after executing a shift right instruction with a shift value greater than 1. Undefined behavior like this is typically deterministic, but can vary with each generation or even stepping of a processor. When full-state traces are used for validation, we must be able to exactly reproduce the same state every time, independent of the hosts used for compression and decompression. To avoid dependence on model-specific or undefined behavior of host instructions, the Zcompr model implements the semantics of instructions in C, not assembly code.

### B. Unpredictable Events

Unpredictable events, as discussed in Section IV-A, consist of any effects external to the core. In our log format they are mostly IO and initial memory reads. They cannot be predicted by Zcompr, which does not model IO.

In the initial uncompressed log format, read events are treated in the same way as other instruction effects and recorded along with the rest of the effects of an instruction. If the memory reads in the difference log did not match previously read or written values, they cannot be predicted; they must always copied to the compressed trace. For example, in Figure 6a, the rest of the side effects of the first instruction can be predicted if we first set the memory location to 1 and then execute the `MOV` with the predictor. In order to compress, we delete the *REG RAX 1;STEP* sequence and replace it with a *FINE* command since there is no need to correct any other register or memory state in the predictor. Instructions that follow and are correctly predicted can be combined with a *FINE 2*.

Note that consecutive *MEM_READ* or *IO_READ* commands (usually originating from IO devices or other processor cores) must always be separated with a *FINE* or *STEP* to ensure that each instruction sees the correct value (Figure 6b).

## VI. Results

We evaluate Zcompr with a collection of workloads, noting the achieved log sizes and compression/decompression speeds. All sizes are taken as reported by the operating system and execution times include user and system time, as measured by the Unix *time* command, thus ignoring disk IO. The workloads include two operating system boots – embedded Linux distributions based on version 1.3 and 2.6 kernels – and the SPEC CPU2006 suite [12] with test inputs. The initial uncompressed logs were captured with different functional simulators. This mix covers a variety of workloads: the OS boots use memory and IO extensively, while SPEC is CPU-bound and covers both integer and floating point code. The two OS boots are comprised of 32-bit and 16-bit code, while the SPEC runs were gathered with a simulator in 64-bit mode. Therefore, we

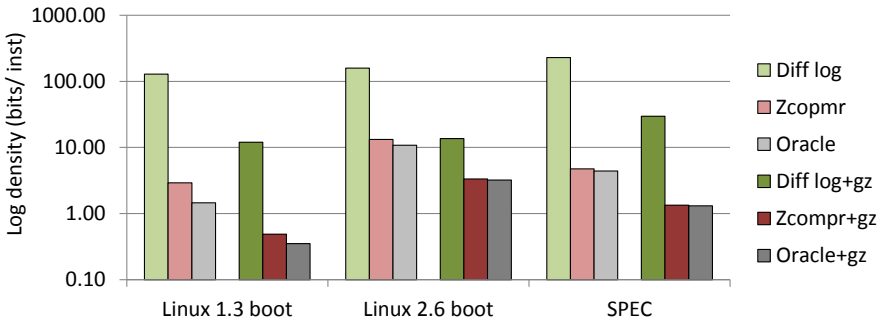



Figure 7. Log density in the compressed logs. Light bars show single-stage compression, while darker ones use gzip as a second stage compressor. Bars marked as *oracle* use a fully-functional CPU model that is able to achieve virtually perfect prediction.

| Benchmark | Length ($10^6$ Inst) | Compression speed (KIPS) | Decompression speed (KIPS) | Compression ratio |
|---|---|---|---|---|
| Linux 1.3 boot | 107 | 113 | 315 | 265× |
| Linux 2.6 boot | 1420 | 96 | 344 | 48× |
| SPEC 2006 | 1750 | 67 | 294 | 171× |

Table I
SPEED AND ABSOLUTE COMPRESSION RATIO WHEN USING ZCOMPR.

expect that Zcompr achieves similar results for a large range of workloads.

Figure 7 shows log densities achieved by Zcompr (raw data are presented in Table I). The first set of 3 bars represents single-stage compression, while in the second set of 3 an additional compression stage with gzip is applied. *Oracle* results are obtained with a complete functional IA32 model, rather than our specialized predictor CPU model. Although such a solution is not suitable for compression mainly because of lack of portability, on a specific platform it can achieve near-perfect compression and we include it here for comparison.

There are three important points to note in Figure 7. First, after gzip the average compressed log density is 2.2 bits per instruction, which is close to our compression design goal. The Linux 2.6 workload is a clear outlier. We note that in Figure 4 unpredictable instruction effects comprise 5% of the uncompressed log size for its execution, compared to $< 1\%$ for the other workloads. Second, two-stage compression is definitely a useful optimization – for a 1% speed overhead it achieves almost an order of magnitude lower log densities. Finally, there is very little difference between log densities achieved by the oracle CPU model and our specialized predictor model. This further verifies the claim from Section IV-B that a simple model suffices for covering a significant fraction of predictable instruction effects.

In order to understand the factors limiting obtainable compression, we look into the size distribution of log commands in the compressed logs before gzip is applied. The results are shown in Figure 8. As in Section IV-A,

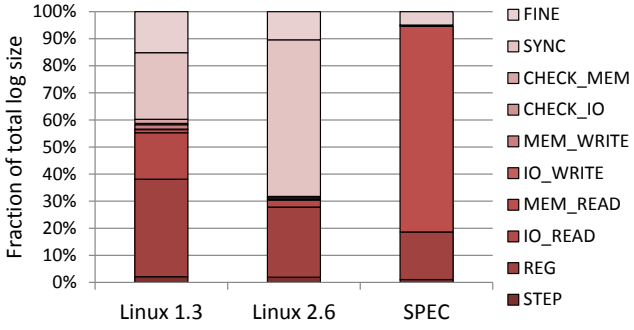



Figure 8. Distribution of trace commands in logs compressed using our scheme. Unpredictable commands take up 75% of final size.

*SYNC* represents the synchronization blocks in each log file and is comprised mostly of memory reads. Note that the *FINE* command introduced to manage decompression comprises a non-ignorable fraction of the log size – a 8.7% overhead. In contrast to Figure 4, after compression predictable events are a minority – 25% of the overall compressed size. This again confirms that a simple functional model can predict a significant fraction of the log commands. While there is still room for improvement in terms of better prediction, Amdahl's law limits the potential benefit.

## VII. OPTIMIZATIONS

In order to achieve the results presented above, Zcompr implements specific optimizations to the approach and implementation described in the previous sections. In this section we describe two categories

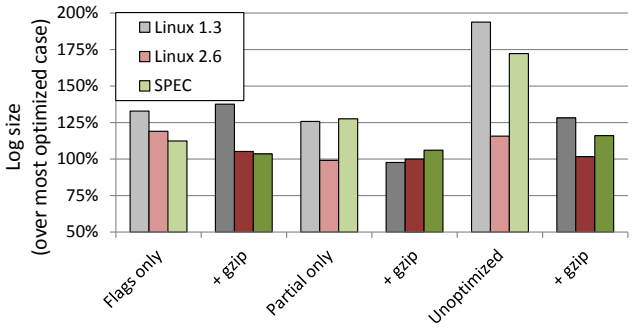



Figure 9. Effectiveness of strategies to deal with undefined flags. Results are normalized to the case with both optimizations on.

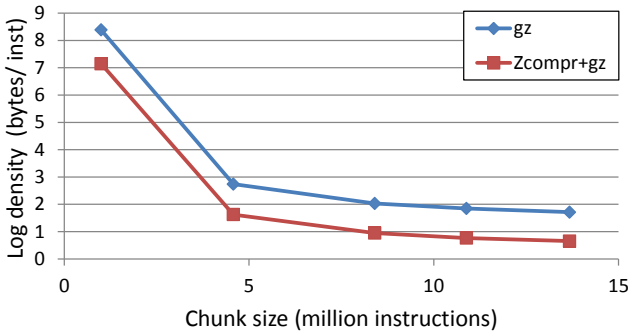



Figure 10. Effect of log chunk size on compressed log density for the Linux 2.6 workload.

of such optimizations. The first category targets better compression by limiting the unpredictable effects of undefined flags and varying the log chunk size. The second combines a number of optimizations aimed at increasing decompression speeds.

### A. Compression

**Flags handling.** Undefined condition flags are a major source of unpredictability. The IA32 architecture [1] does not specify the effects of all instructions on the flags register and different implementations vary. This affects even most common instructions classes – logic instructions, shifts and some arithmetic. Our analysis shows that, in some workloads, instructions that generate undefined flags can comprise up to 10% of the instruction stream. This can be a limiting factor in the compression achievable by Zcompr, since it insists on reproducing undefined flags behavior for validation purposes.

To address this issue, we use a two-step approach. First, our predictor model uses a simple set of heuristics to predict the results of an instruction with undefined flags. Undefined flags are usually either left unchanged, set, or cleared. Then, if the heuristics are not correct for the behavior in the trace, we mark the instruction as partially predicted and only emit changes to the *EFLAGS* register, but not the other results of the instruction.

The effectiveness of both approaches is shown in Figure 9. *Flags only* refers to the case when only the first optimization is on, while *Partial only* has only the second one. All results are normalized to the case with both optimizations on (which is also the one shown in Section VI). In the case of single-stage compression, each combination of optimizations can provide significant size reductions, reaching almost a 2× improvement for Linux 1.3. However, the second stage bridges a large portion of the gap between the unoptimized and most optimized case and the maximal effect of both optimizations acting together is only slightly over 25%. Note that for *Partial only* and Linux 1.3, the result after applying two-stage compression is even better than with both optimizations. Similar effects of two-stage compression are observed and exploited in [7], but do not appear to be prevalent in Zcompr.

An alternative approach to dealing with undefined behavior can be based on learning from the original trace. The predictor can note which of the heuristics are correct for a particular trace and base future flag handling on these heuristics. We leave this to future work.

**Log chunk size.** Splitting the trace for a full workload into smaller-sized chunks can be useful – the chunks can be analyzed in parallel, or used as checkpoints for skipping irrelevant initialization code. However, each chunk must contain initial values for the memory and IO addresses it references. Values that are not changed and read multiple times will appear in multiple chunks, increasing the total size.

To isolate this effect, we evaluated the resulting log density for different chunk sizes. Results from this sweep are shown in Figure 10. The workload examined is our Linux 2.6-based distribution[1]. The knee of the curve is close to a 7 million instruction chunk size. For smaller sizes, the replication effects can easily dominate the total log size. The trends for gzip only and Zcompr+gzip are identical because Zcompr essentially does not compress such unpredictable data. Based on these results, we always used chunk sizes above 10 million instructions when evaluating achievable compression.

### B. Decompression Speed

**Persistent decode cache.** Mihocka and Shwartsman [13] show that decoding is a major bottleneck in interpreter-based x86 simulators. In order to maximize decompression speeds, we completely eliminate decoding when the predictor CPU is operating in decompression mode. Instead, during decompression the simulator operates off a decode cache. It is created

[1]This particular experiment has been conducted with a different version of Zcompr, so absolute values may be different than those presented in Section VI.

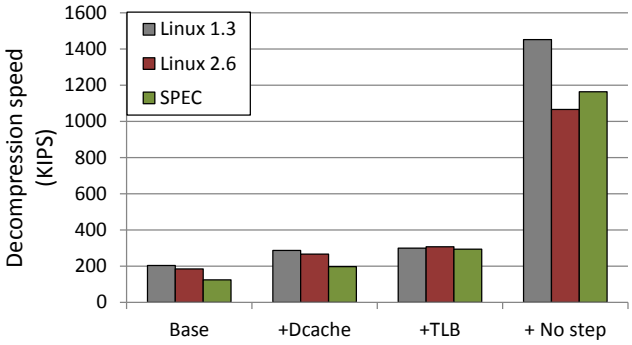



Figure 11. The effect of dedicated optimizations targeting decompression speed. The last three sets of bars represent incrementally enabling the three optimizations described.

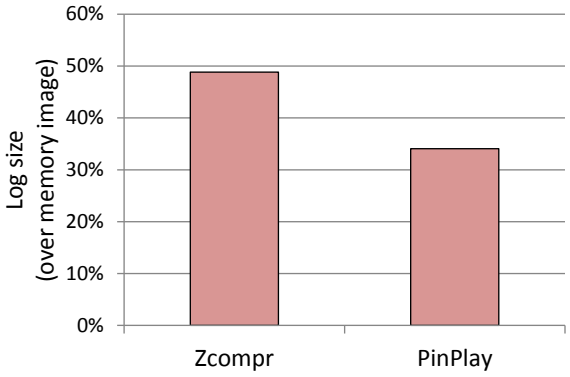



Figure 12. Comparison between Zcompr and PinPlay on user-mode SPEC workload. Y-axis shows compressed log size, normalized to the size of the complete memory image, compressed with gzip.

by the same simulator during compression and stores information about instruction semantics and operands that is sufficient to execute each compressed instruction. This information is in a less compact, but easier to parse form than regular IA32 encoding. Thus, decode time is significantly reduced at the cost of the space needed to persist the decode cache between compression and decompression. The cache itself is implemented as a hash table, indexed by the physical address of the program counter and is split in chunks in the same way as the input log files. This is done to ensure that collisions inside a single chunk (different instructions on the same physical address due to, for example, dynamic loading of a program by the OS) are extremely rare. In case such collisions do occur, they are detected during compression and the instructions that generated them are simply not compressed.

We found that persisting the decode cache adds an overhead of less than 3% to the compressed log size. We consider this negligible and do not add it to the results in Section VI. On the other hand, as seen in the second set of bars in Figure 11, being able to remove the decode stage results in a total speedup of 53% over our baseline implementation.

**Software TLB.** The next low hanging fruit in decompression speed was eliminating most virtual to physical address translations. Translation-lookaside buffers (TLBs) are a well-established solution to this problem. We added a 1024-element software TLB to our predictor model to save address translations. When coupled with the decode cache, this results in a 2× total decode speed improvement over the baseline. Achieved decompression speeds are approximately 300 KIPS (see Table I).

**Block execution.** After applying the above two optimizations, profiling results indicate that the main decompression speed bottleneck is not in the predictor CPU model, but in its interaction with the decompression harness. By default, the harness gets instruction effects for each decompressed instruction from the CPU model. Such functionality is essential in a validation context where errors should be easily identified with the instruction that produced them. However, communicating the instruction effects is expensive both because of the large amount of data and the fact that such transfers occur with every instruction. Then, a simple way to mitigate this overhead is to extract instruction effects more rarely and effectively decompress the execution of blocks of instructions together. We leave this as an option to the user.

To evaluate the effectiveness of this scheme, we used the largest possible block size – that of the respective *FINE* block. The last set of bars in Figure 11 shows that the potential for improvement is impressive – 6.7× the baseline speed, resulting in a total average decompression rate of 1.1 MIPS.

## VIII. Related Work

### A. Repeatable execution platforms

*PinPlay* [2] is a deterministic replay framework built on top of the Pin dynamic instrumentation system [14]. It enables reproducing parallel program execution by recording synchronization points and other unpredictable behavior. However, since it is based on Pin, it is limited to user-level code and relies on host execution for replay capabilities. Zcompr's design was inspired by PinPlay's approach and targets overcoming these limitations for its use as a co-simulation platform. To compare both schemes, Figure 12 shows their compression effectiveness on the SPEC workload. The resulting PinPlay logs are 30% smaller than their Zcompr analogs. This is expected since Zcompr is able to reproduce the original execution more closely – complete full-state reproducibility, while possible, comes at a cost.

*iDNA* [15] is a record/replay mechanism developed by Microsoft. It is similar to PinPlay – using a binary

translation framework and working on the user level. However, it is an internal tool, so comparison results are not included.

*ReTrace* [16] and *ReVirt* [17] are replay frameworks that rely on system-level virtual machines. In this case, virtual machines simulate the execution of a instruction in the guest machine by executing the same instruction on the host. This can limit their use to modeling behavior of CPU's that already exist. ReTrace specifically shares most of Zcompr's design goals. The authors report log densities one order of magnitude lower than Zcompr's. However, the densities presented in [16] do not include the input data (the virtual disk images), which are required for re-execution. A minimal virtual disk for a modern operating system is 2 gigabytes and can easily be 50 gigabytes or more. Our system can efficiently handle traces as short as 10 million instructions, but when the large fixed cost of a virtual disk is factored in ReTrace needs traces of length 1 billion or more.

### B. Trace compression

The *VPC* algorithm family [7] uses multiple simple value predictors to recompute trace data based on observed patterns. The different algorithm revisions vary the number and type of predictors, as well as the granularity they operate on. The main difference from the proposed approach is that the only semantic fact the VPC family makes use of is that every entry is tagged by a PC value. Also, the algorithms require a fixed, pre-determined format for each instruction entry, which does not trivially map to full-state traces, where each instruction can generate a variable number of architectural effects.

### C. Execution caching

FastSim [18] by Schnarr and Larus also uses prediction-based techniques to short-circuit execution. FastSim is a performance simulator which relies on *fast-forwarding*. It caches actions (corresponding to pipeline state) and their appropriate timing. In case of correct prediction (cache hits), timing is readily available in this cache. On a misprediction, a detailed pipeline model is invoked.

## IX. Conclusion

With trace-driven simulation being a major vehicle in pre-silicon development and research, we addressed the need for a trace format that contains full architectural state. In order to make storing such logs reasonable, we developed Zcompr – a compression framework that relies on reinterpreting instructions in a stable and predictable manner. Zcompr is able to compress billion-length traces to several bits per instruction, while retaining validation-quality reproducibility. It does so without any assumptions about the underlying operating system or host architecture revision.

## References


[1] "Intel 64 and IA-32 architectures software developer's manuals." [Online]. Available: http://www.intel.com/products/processor/manuals/

[2] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie, "Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs," in *CGO '10: Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. New York, NY, USA: ACM, 2010.

[3] S. Narayanasamy *et al.*, "Bugnet: Continuously recording program execution for deterministic replay debugging," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 2, 2005.

[4] E. Federovsky, M. Feder, and S. Weiss, "Branch prediction based on universal data compression algorithms," in *Proceedings of the 25th annual international symposium on Computer architecture*. IEEE Computer Society, 1998, pp. 62–72.

[5] I. Chen *et al.*, "Analysis of branch prediction via data compression," *ACM SIGPLAN Notices*, vol. 31, no. 9, p. 137, 1996.

[6] K. Curewitz *et al.*, "Practical prefetching via data compression," *ACM SIGMOD Record*, vol. 22, no. 2, p. 266, 1993.

[7] M. Burtscher *et al.*, "The VPC Trace-Compression Algorithms," *IEEE Transactions on Computers*, vol. 54, no. 11, pp. 1329–1344, Nov. 2005.

[8] M. Charney, "x86 Endcoder-Decoder (XED)." [Online]. Available: http://www.pintool.org/docs/36111/Xed/html/

[9] P. Barham *et al.*, "Xen and the art of virtualization," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*. ACM, 2003, p. 177.

[10] J. Sugerman, G. Venkitachalam, and B. Lim, "Virtualizing I/O devices on VMware workstations hosted virtual machine monitor," *Boston, MA, USA*, 2001.

[11] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "KVM: the Linux virtual machine monitor," in *Proceedings of the Linux Symposium*, vol. 1, 2007, pp. 225–230.

[12] "SPEC CPU2006." [Online]. Available: http://www.spec.org/cpu2006/

[13] D. Mihocka and S. Shwartsman, "Virtualization Without Direct Execution or Jitting: Designing a Portable Virtual Machine Infrastructure," in *1st Workshop on Architectural and Microarchitectural Support for Binary Translation in ISCA-35*, 2008.

[14] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. ACM, 2005.

[15] S. Bhansali *et al.*, "Framework for instruction-level tracing and analysis of program executions," in *Proceedings of the 2nd international conference on Virtual execution environments*. ACM, 2006.

[16] M. Xu *et al.*, "ReTrace : Collecting Execution Trace with Virtual Machine Deterministic Replay," in *Proceedings of the Third Annual Workshop on Modeling, Benchmarking and Simulation (MoBS 2007)*.

[17] G. Dunlap *et al.*, "ReVirt: Enabling intrusion analysis through virtual-machine logging and replay," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, p. 224, 2002.

[18] E. Schnarr and J. Larus, "Fast out-of-order processor simulation using memoization," in *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*. ACM, 1998, pp. 283–294.