Generalized Just-In-Time Trace Compilation using a Parallel Task Farm in a Dynamic Binary Translator

Igor Böhm   Tobias J.K. Edler von Koch   Stephen Kyle   Björn Franke   Nigel Topham

Institute for Computing Systems Architecture
School of Informatics, University of Edinburgh
Informatics Forum, 10 Crichton Street, Edinburgh, EH8 9AB, United Kingdom
I.Bohm@sms.ed.ac.uk, T.J.K.Edler-Von-Koch@sms.ed.ac.uk, S.C.Kyle@sms.ed.ac.uk, bfranke@inf.ed.ac.uk,
npt@inf.ed.ac.uk

Abstract

Dynamic Binary Translation (DBT) is the key technology behind cross-platform virtualization and allows software compiled for one Instruction Set Architecture (ISA) to be executed on a processor supporting a different ISA. Under the hood, DBT is typically implemented using Just-In-Time (JIT) compilation of frequently executed program regions, also called *traces*. The main challenge is translating frequently executed program regions as fast as possible into highly efficient native code. As time for JIT compilation adds to the overall execution time, the JIT compiler is often decoupled and operates in a separate thread independent from the main simulation loop to reduce the overhead of JIT compilation. In this paper we present *two* innovative contributions. The first contribution is a *generalized* trace compilation approach that considers *all* frequently executed paths in a program for JIT compilation, as opposed to previous approaches where trace compilation is restricted to paths through loops. The second contribution reduces JIT compilation cost by compiling several hot traces in a concurrent task farm. Altogether we combine generalized light-weight tracing, large translation units, parallel JIT compilation and dynamic work scheduling to ensure timely and efficient processing of hot traces. We have evaluated our industry-strength, LLVM-based parallel DBT implementing the ARCompact ISA against three benchmark suites (EEMBC, BIOPERF and SPEC CPU2006) and demonstrate speedups of up to 2.08 on a standard quad-core Intel Xeon machine. Across short- and long-running benchmarks our scheme is robust and never results in a slowdown. In fact, using four processors total execution time can be reduced by on average 11.5% over state-of-the-art decoupled, parallel (or *asynchronous*) JIT compilation.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Incremental Compilers

General Terms Design, experimentation, measurement, performance

Keywords Dynamic binary translation, just-in-time compilation, parallelization, task farm, dynamic work scheduling

1. Introduction

DBT is a widely used technology that makes it possible to run code compiled for a target platform on a host platform with a different ISA. With DBT, machine instructions of a program for the target platform are translated to machine instructions for the host platform during the execution of the program. Among the main uses of DBT are *cross-platform virtualization* for the migration of legacy applications to different hardware platforms (e.g. DEC FX!32 [9], Apple Rosetta and IBM POWERVM LX86 [32] both based on Transitive's QuickTransit, or HP ARIES [44]) and the provision of *virtual platforms* for convenient software development for embedded systems (e.g. Virtual Prototype by Synopsys). Other current and emerging uses of DBT include, but are not limited to, generation of cycle-accurate architecture simulators [6, 12], dynamic instrumentation [17], program analysis, cache modeling, and workload characterization [23], software security [42], and transparent software support for heterogeneous embedded platforms [11].

Efficient DBT heavily relies on Just-in-Time (JIT) compilation for the translation of target machine instructions to host machine instructions. Although JIT compiled code generally runs much faster than interpreted code, JIT compilation incurs an additional overhead. For this reason, only the most frequently executed code regions are translated to native code whereas less frequently executed code is still interpreted. Using a single-threaded execution model, the interpreter pauses until the JIT compiler has translated its assigned code block and the generated native code is executed directly. However, it has been noted earlier [1, 16, 20] that program execution does not need to be paused to permit compilation, as a JIT compiler can operate in a separate thread while the program executes concurrently. This *decoupled* or *asynchronous* execution of the JIT compiler increases complexity of the DBT, but is very effective in hiding the compilation latency – especially if the JIT compiler can run on a separate processor.

Our main contribution is to demonstrate how to effectively reduce dynamic compilation overhead and speedup execution by doing *parallel* JIT compilation, exploiting the broad proliferation of multi-core processors. The key idea is to detect *independent*, large translation units in execution traces and to *farm out* work to *multiple, concurrent* JIT compilation workers. To ensure that the latest and most frequently executed code traces are compiled first, we apply a priority queue based dynamic work scheduling strategy where the most recent, hottest traces are given highest priority.

We have integrated this novel, concurrent JIT compilation methodology into our LLVM-based state-of-the-art ARCSIM [19, 38] DBT implementing the ARCompact ISA and evaluated its performance using three benchmark suites: EEMBC, BIOPERF and

PLDI'11, June 4–8, 2011, San Jose, California, USA.

SPEC CPU2006. We demonstrate that our parallel approach yields an average speedup of 1.17 across all 61 benchmarks – and up to 2.08 for individual benchmarks – over decoupled JIT compilation using only a single compilation worker thread on a standard quad-core Intel Xeon host platform. At the same time our scheme is robust and never results in a slowdown even for very short- and long-running applications.

1.1 Trace-based JIT Compilation

Before we take a more detailed look at our approach to generalization of JIT trace compilation using a parallel task farm, we provide a comparison of state-of-the-art trace-based JIT compilation approaches to highlight our key concepts (see Figure 1).

Trace-based JIT compilation systems start executing in interpreted mode until a special structure (e.g. loop header, method entry) is detected. This causes the system to switch from interpreted mode to trace mode. In trace mode executed paths within that particular structure are recorded to form a trace data structure. Once the traced code region's execution count reaches a threshold, the recorded trace is compiled just-in-time and control flow is diverted from the interpreter to the faster, native code. In general, trace-based JIT compilation approaches can be categorized based on the following criteria:

- *Trace Boundaries* - One popular [14–16, 26, 27, 29, 37] approach is to only consider paths within natural loops (①②③ in Figure 1). However, there are also paths outside of loops (e.g. paths crossing method or function boundaries, IRQ handlers, irreducible loops) which are executed frequently enough to justify their compilation to machine code. In this work we propose a *generalized* approach to tracing based on traced control-flow-graphs (CFG) of basic blocks, that can start and end almost anywhere in the program (④ in Figure 1). As a result we can discover more hot traces and capture larger regions of frequently executed code than just loop bodies.
- JIT *Compilation Strategy* - Some state-of-the-art JIT trace compilers [26, 27, 37] are sequential and wait for JIT compilation of a trace to finish before continuing execution (① in Figure 1). An early approach to parallel JIT compilation [14] proposed to split trace compilation into multiple, parallel pipeline stages operating at instruction granularity (② in Figure 1). While the compiler pipeline can – in principle – be parallelized, this approach is fundamentally flawed as execution is stopped until compilation of a complete trace is finished. This results in consistent slowdowns over all benchmarks in [14]. Synchronization of up to 19 pipeline stages for every compiled instruction adds significantly to the overall compilation time while no new translation units are discovered. The HOTSPOT JVM [29] and [16] implement a concurrent JIT compilation strategy by running the JIT compiler in *one* separate helper thread whilst the master thread continues to interpret code (③ in Figure 1). We build on this approach and extend it to run *several* JIT compilers in a parallel task farm whilst execution continues. We do this to further hide JIT compilation cost and switch to native execution of hot traces even sooner and for longer (④ in Figure 1) than previous approaches.

The objective of our trace-based JIT compilation strategy ④ outlined in Figure 1 is translating frequently executed program regions into native code faster when compared to e.g. ①②③ [14–16, 26, 27, 29, 37]. Figure 1 also illustrates that task parallel JIT compilation ④ reduces the time until native code execution starts in comparison to current approaches ①②③.

Our generalized approach to tracing enables us to start tracing right from the first instruction, avoiding a period of interpretation until specific structures (e.g. loop headers) are encountered. Consequently we can find more opportunities for JIT compilation (i.e. hot traces) earlier. Being able to discover more hot traces that can be compiled independently, we take the idea of decoupled JIT compilation a step further, and propose a truly parallel and scalable JIT compilation scheme based on the parallel task farm design pattern.

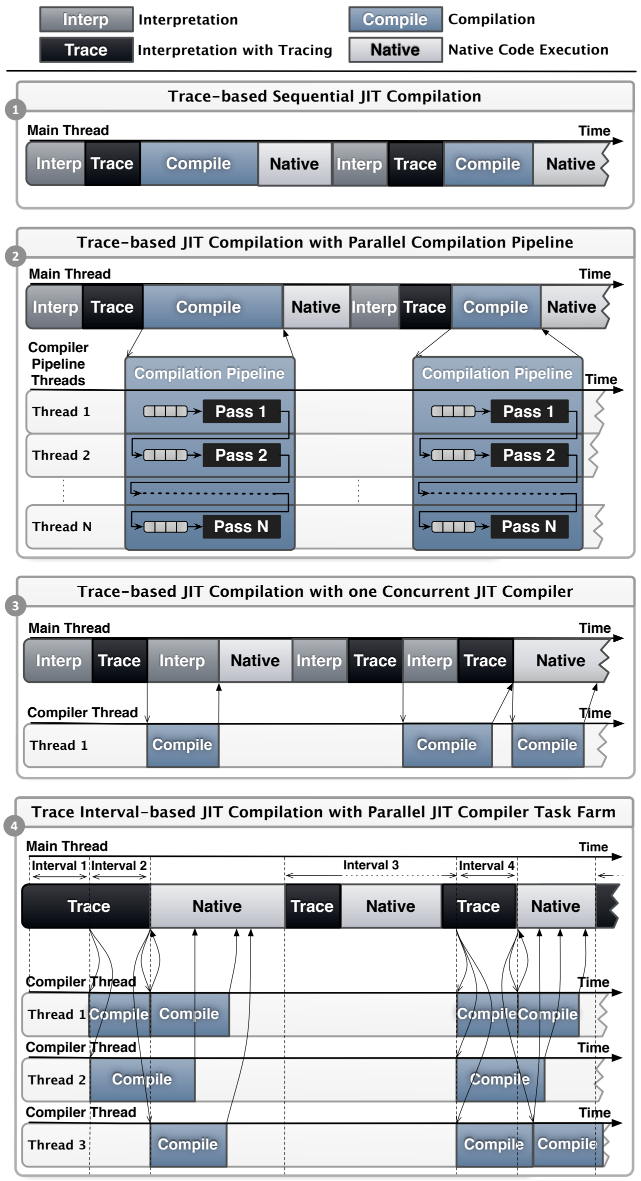

Interp Interpretation
Compile Compilation
Trace Interpretation with Tracing
Native Native Code Execution
1 Trace-based Sequential JIT Compilation
Main Thread
Time
Interp Trace Compile Native Interp Trace Compile Native
2 Trace-based JIT Compilation with Parallel Compilation Pipeline
Compiler Pipeline Threads
Compilation Pipeline
Thread 1
Thread 2
Thread N
Pass 1
Pass 2
Pass N
3 Trace-based JIT Compilation with one Concurrent JIT Compiler
Compiler Thread
4 Trace Interval-based JIT Compilation with Parallel JIT Compiler Task Farm
Interval 1
Interval 2
Interval 3
Interval 4
Thread 3

Figure 1. Comparison of trace-based JIT compilation approaches: ① sequential approach [15, 26, 27, 37], ② parallel compilation pipeline [14], ③ decoupled JIT compilation using *one* thread [16, 29]. Finally, ④ depicts our parallel JIT compilation task farm.

1.2 Motivating Example

Consider the full-system simulation of a Linux OS configured and built for the ARC 700 processor family (RISC ISA). On a standard quad-core Intel Xeon machine we simulate the complete boot-up sequence, the automated execution of a set of commands simulating interactive user input at the console, followed by the full shut-down sequence. This example includes rare events such as boot-up and shut-down comparable to the initialization phase in an application,

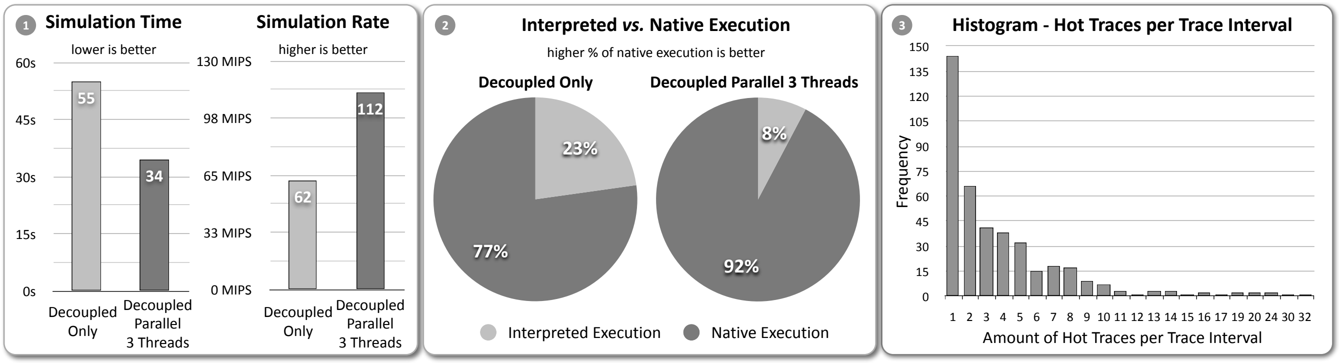


Figure 2. Full-system Linux simulation benchmark - comparison of ① simulation *time* in seconds, *rate* in MIPS, and ② interpreted *vs.* natively executed instructions in %, between *decoupled only* and *decoupled parallel* JIT compilation using three JIT compilation worker threads. Histogram ③ demonstrates how often more than one *hot* trace per trace interval is found.

but it also includes very frequent events occurring after the boot-up sequence during interactive user mode. In a full-system OS simulation there are frequent calls to interrupt service routines that must be simulated. Our generalized tracing approach can easily identify frequent traces including interrupt service routines that would otherwise be missed if we would restrict tracing to loop or function boundaries.

Our dynamic binary translator speeds up the simulation by identifying and translating hot traces to native `x86` code during simulation using the sequence of steps illustrated in Figure 3. As simulation runs in parallel to JIT compilation we continue to discover and dispatch more hot traces to the JIT compiler. In fact, it is possible that JIT compilation of the previous trace has not yet been completed by the time new traces are discovered. In this case, work is distributed over several JIT compilation threads. To ensure that the most *profitable* traces are compiled first, we have implemented a dynamic work scheduling strategy that dynamically prioritizes compilation tasks according to their *heat* and *recency*.

For the purpose of this motivating example we compare a simulation using only one decoupled JIT compiler thread (current state-of-the-art) with a simulation using multiple decoupled JIT compiler threads in parallel (see ④ in Figure 1). In Chart ① of Figure 2 we compare the overall simulation time in seconds and the simulation rate in MIPS for both approaches. Our new approach using three decoupled JIT compilers in parallel completes the previously outlined sample application 21 seconds earlier. This results in an improvement of 38% and, thus, achieves a speedup of **1.6** when compared to using only one decoupled JIT compiler. The overall simulation rate (in MIPS) improves from 62 to 112 MIPS by using our new approach. As several JIT compilers work on hot traces in parallel, native translations are available much earlier than using a single decoupled JIT compiler (see ③ in Figure 1), leading to a substantial increase from 77% to 92% of natively executed target instructions (see Chart ② of Figure 2).

The obvious question to ask is where does the speedup come from? Histogram ③ in Figure 2 shows how often a certain amount of frequently executed traces is found per trace interval. The fact that 65% of the time there are at least two or more hot traces discovered per interval clearly demonstrates the benefits of having more than one JIT compiler available on today's multicore machines. Even if only one hot trace per interval is discovered, the JIT compilation of the previous hot trace might not have finished. Having several JIT compilers that can already start working on the newly discovered hot traces before others have finished helps to effectively hide most of the JIT compilation latency (see Box ① in Figure 5).

1.3 Contributions

Among the contributions of this paper are:

1. The introduction of a light-weight and generalized JIT trace compilation approach considering frequently executed paths (i.e. hot traces) that can start and end almost anywhere in a program and are not restricted to loop or function boundaries,
2. the introduction of an innovative *parallel task farming* strategy for truly concurrent JIT compilation of hot traces,
3. the development of *dynamic work scheduling* and *adaptive hotspot threshold selection* approaches that give priority to the most recent, hottest traces in order to reduce time spent in interpreted simulation mode, and
4. an extensive evaluation of our LLVM-based DBT targeting the ARCompact ISA using three full benchmark suites, EEMBC, BIOPERF and SPEC CPU2006.

1.4 Overview

The remainder of this paper is structured as follows. In section 2 we give an overview of the LLVM-based ARCSIM dynamic binary translator. This is followed by a presentation of our tracing approach and parallel JIT compilation scheme in section 3. We then explain the evaluation methodology and show our empirical results in section 4. Related work is discussed in section 5 and, finally, we summarize and conclude in section 6.

2. Background

In our work we extended our DBT ARCSIM, a target adaptable simulator with extensive support of the ARCompact ISA. It is a full-system simulator, implementing the processor, its memory sub-system (including MMU), and sufficient interrupt-driven peripherals to simulate the boot-up and interactive operation of a complete Linux-based system. The DBT has a very fast and highly-optimized interpreted simulation mode [38]. By using a trace-based JIT compiler to translate frequently interpreted instructions into native code, our DBT is capable of simulating applications at speeds approaching or even exceeding that of a silicon ASIP whilst faithfully modeling the processor's architectural state [38].

The JIT compiler is based on release 2.7 of the LLVM compiler infrastructure [22] and executes in a concurrent thread to the main interpretation loop, reducing the overhead caused by JIT compilation [16, 29]. The LLVM JIT compilation engine implementation provides a mature and competitive JIT compilation environment

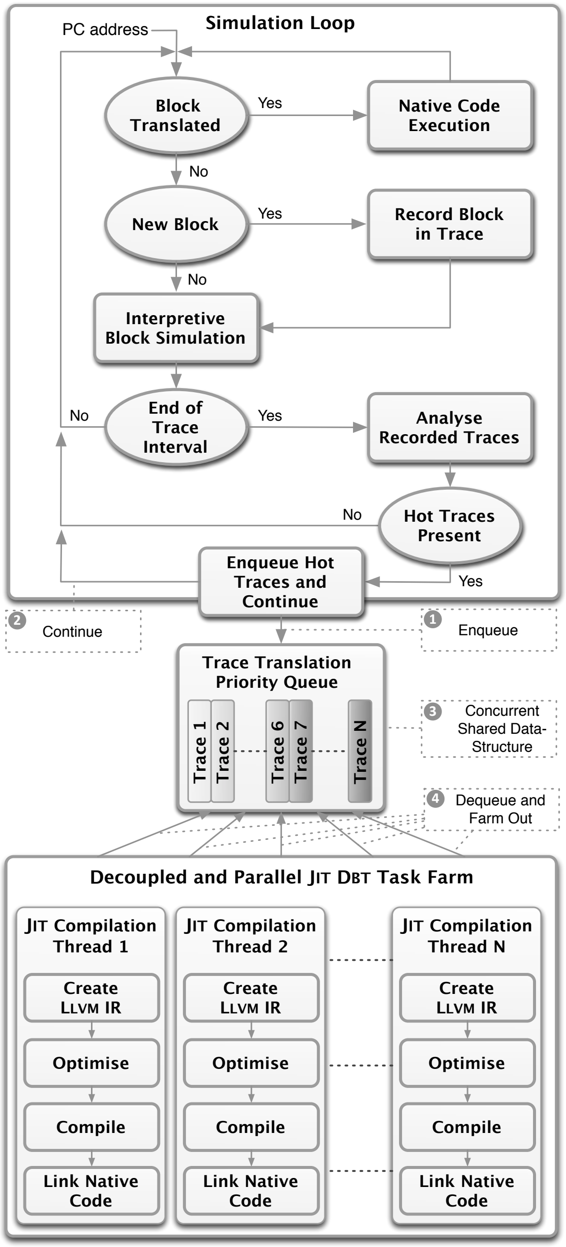


Figure 3. Trace-based JIT DBT flow showing main simulation loop at the top, priority queue scheduling in the middle, and parallel task farm at the bottom.

used in many major products by companies such as Apple, Adobe, Nvidia, and Sony, to name just a few [36].

State-of-the-art trace-based JIT compilers typically use natural loops as boundaries for traces, resulting in relatively small units of translation [14–16, 26, 27, 29, 37]. Our DBT deliberately considers larger traces by allowing traces to start and end almost anywhere in a program. This approach is comparable to the techniques evaluated in [3, 19] and has the benefit of providing greater scope for optimizations to the JIT compiler as it can operate on larger traces. Furthermore our DBT includes various software caches [3, 38] (e.g. decode cache to avoid re-decoding recently decoded instructions, translation cache to improve lookup times for locations of native code) to improve simulation speed.

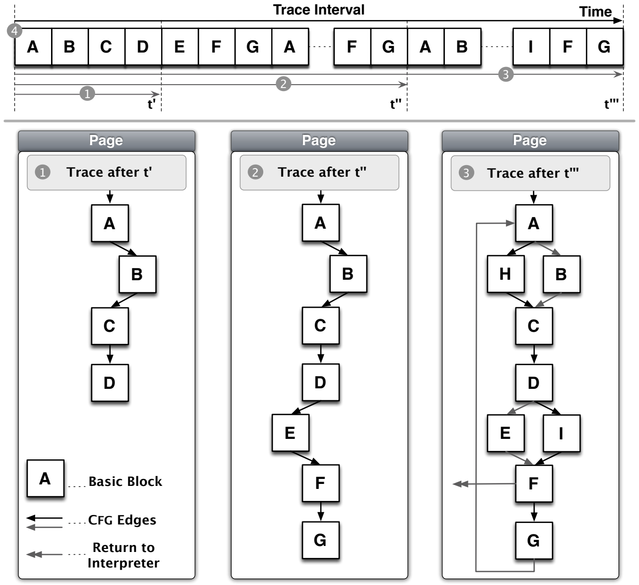


Figure 4. Incremental ①②③ trace construction from sequence of interpreted basic blocks ④ during one trace interval.

3. Methodology

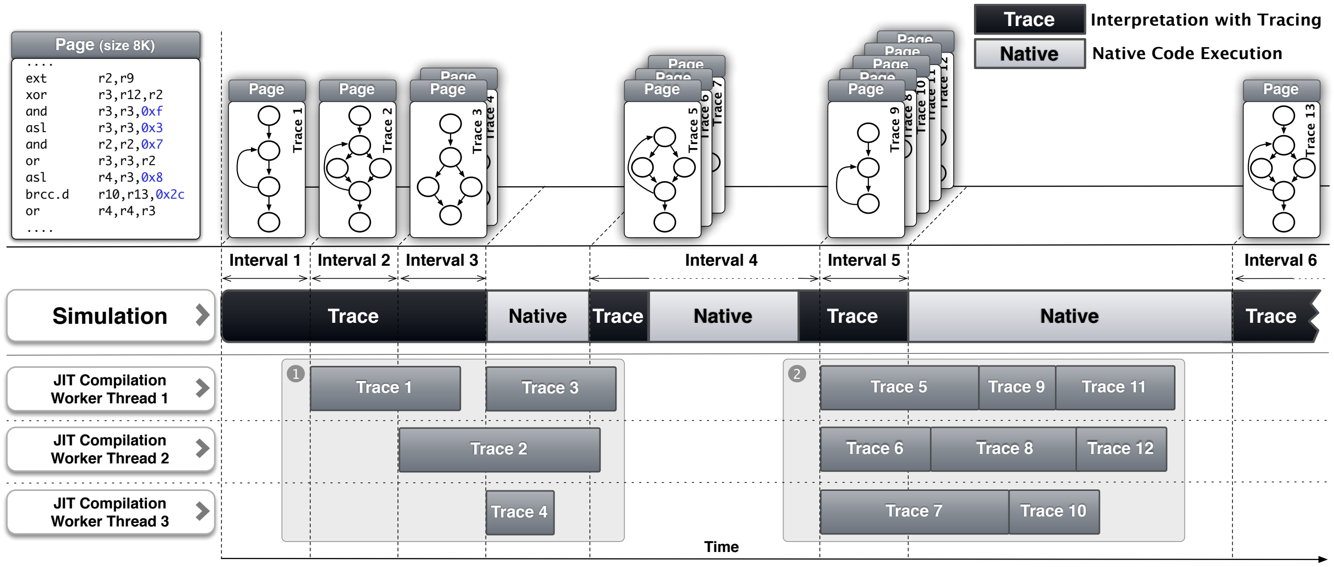
Our approach to parallel and decoupled JIT compilation in a dynamic binary translator significantly speeds up execution whilst maintaining full architectural observability of the target architecture. We use a light-weight approach to discover and construct traces eligible for JIT compilation. A trace reflects the control flow exhibited by the source program at run-time. Newly discovered hot traces are then translated using multiple decoupled JIT compiler threads in parallel. Decoupling the simulation (i.e. interpretation) loop from JIT compilation prevents any slowdown incurred by JIT compilation as the simulation continues while hot traces are being translated [16, 34]. By adding more JIT compilation threads, working on independent hot traces, JIT compilation time is further reduced resulting in increased simulation speed as native code becomes available earlier.

In the following sections we outline our trace generation and hotspot detection technique and explain why this technique discovers more relevant hot traces for JIT compilation than previous approaches [14–16, 26, 27, 29, 37]. We also discuss the decoupling and parallelization of JIT compilation and demonstrate the importance of scheduling hot traces for JIT compilation. The key idea behind our scheduling scheme is to consider heat and recency of traces in order to prioritize hot traces that are being simulated right now and to JIT compile them first.

3.1 Trace Construction and Hotspot Detection

Interpreted simulation time is partitioned into *trace intervals* (see Figures 3, 4 and 5) whose *length* is determined by a user-defined number of interpreted instructions. After each trace interval, the hottest recorded traces are dispatched to a priority queue for JIT compilation before the simulation loop continues. Decoupled from this simulation loop, JIT compilation workers dequeue and compile the dispatched traces. The *heat* of a trace is defined as the sum of the execution frequencies of its constituent basic blocks.

Our DBT must maintain an accurate memory model (see Section 2) to preserve full architectural observability. This means that traces generated during a trace interval are separated at page boundaries, where a page can contain up to 8 KB of target instructions. For



```
....
ext       r2,r9
xor       r3,r12,r2
and       r3,r3,0xf
asl       r3,r3,0x3
and       r2,r2,0x7
or        r3,r3,r2
asl       r4,r3,0x8
brcc.d    r10,r13,0x2c
or        r4,r4,r3
....
```

Figure 5. Concurrent JIT compilation of hot traces - Box ② demonstrates how we exploit *task parallelism* by JIT compiling independent hot traces in parallel. Box ① shows how several concurrent JIT compilation threads can effectively hide most of the JIT compilation latency by overlapping compilation of hot traces.

each page a trace recording interpreted basic blocks contained in that page is maintained (see Figure 5). Tracing is light-weight because we only record basic block entry points (i.e. memory addresses) as nodes, and pairs of source and target entry points as edges in the per-page CFG traces. Figure 4 shows the incremental trace construction for interpreted basic blocks within one page during one trace interval and highlights our trace boundary or unit of compilation, namely a trace within a page of target memory instructions.

Typically JIT compilation schemes [14–16, 26, 27, 29, 37] are based on compilation units reaching threshold frequencies of execution to determine when to trigger compilation. Our approach is based around the concept of trace intervals during which traces are recorded and execution frequencies are maintained. At the end of a trace interval we analyze generated traces for each page that has been touched during that interval and dispatch frequently executed traces to a trace translation priority queue (see ①②③ in Figure 3). We decided to use an interval based scheme because we found it generates more uniformly sized compilation units, resulting in better and more predictable load balance between compilation and execution. Our trace intervals are somewhat related to the concept of bounded history buffers in DYNAMO [3] but impose fewer restrictions on their use.

3.2 Parallel JIT Compilation

Our main contribution is the demonstration, through practical implementation, of the effectiveness of a parallel multi-threaded JIT compilation task farm based on release 2.7 of the LLVM [22] compiler infrastructure. By parallelizing JIT compilation we effectively hide JIT compilation latency and exploit the available parallelism exposed by our generalized trace construction scheme.

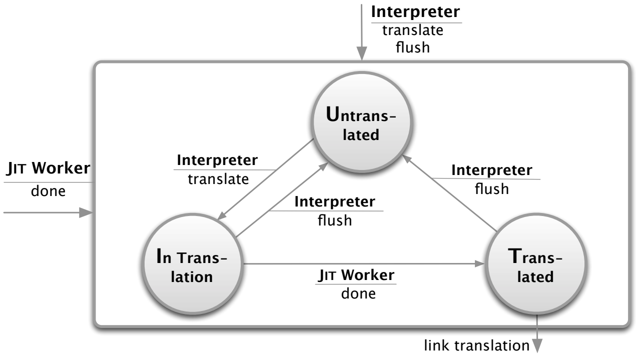


Figure 6. Trace translation state variable transitions.

A concurrent trace translation priority queue acts as the main interface between the simulation loop and multiple JIT compilation workers (see ③ and ④ in Figure 3). Each JIT compilation worker thread dequeues a trace and generates a corresponding LLVM intermediate representation (IR). Subsequently, a sequence of standard LLVM optimization passes is applied to optimize the generated LLVM-IR. We use LLVM's `ExecutionEngine` to JIT-compile and link the generated LLVM-IR code. Each JIT compilation worker thread receives its own private `ExecutionEngine` instance upon thread creation.

In our parallel trace-based JIT compiler the interpreter uses a *translation state variable* to mark unseen traces for JIT compilation and to determine if a native translation for a trace already exists. While traces are JIT compiled, the interpreter continues to execute the program. As soon as JIT compilation of a trace is finished, the JIT compilation worker immediately modifies its translation state variable such that the interpreter is notified about its availability. Thus, a trace can be in one of the following three states:

- UNTRANSLATED - When a trace is seen for the first time, its state is set to *untranslated* by the interpreter.
- IN TRANSLATION - When a trace is dispatched for JIT compilation the interpreter changes its state from *untranslated* to *in translation*.
- TRANSLATED - When a JIT compilation worker is finished with the translation of a trace it changes its state from *in translation* to *translated*.

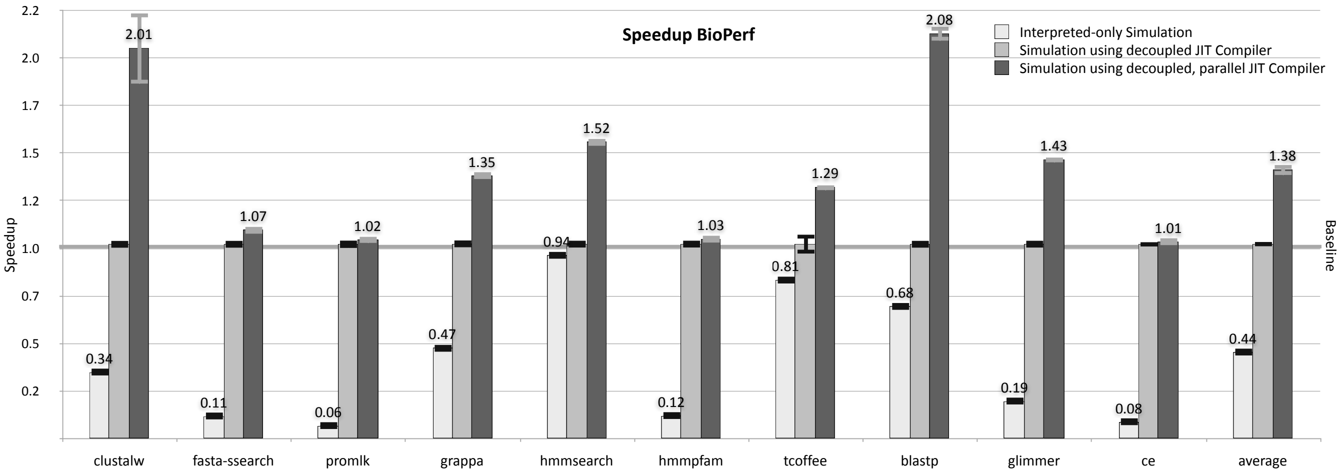


Figure 7. Speedups using the BIOPERF benchmark suite comparing (a) interpreted-only simulation, (b) simulation using a decoupled JIT compiler, and (c) simulation using our novel, parallel JIT compiler with three JIT compilation worker threads including dynamic work scheduling.

This particular use of a state variable indicating the translation state of a trace allows for *lock-free* synchronization between JIT compilation threads and the simulation loop (see Figure 6). This approach is somewhat similar to the compiled state variable concept implemented in [16].

3.3 Dynamic Work Scheduling

In any kind of JIT compilation environment it is paramount to translate hot code regions as fast as possible into highly efficient native code. Thus having discovered multiple traces across several trace intervals we would like to JIT compile the most recently and frequently executed traces first.

To efficiently implement a dynamic work scheduling scheme based on both recency and frequency of interpreted traces, we have chosen a priority queue as an abstract data type using a binary heap as the backbone data-structure. We chose a binary heap because of its worst case complexity of $O(log(n))$ for inserts and removals. Our sorting criteria insert the most frequently executed trace from the most recent trace interval at the front of the priority queue.

3.4 Adaptive Hotspot Threshold Selection

It is possible that our trace-selection and dispatch system can produce more tasks than the JIT workers can reasonably handle. The aforementioned dynamic work scheduling mitigates this problem by ensuring that the hottest and most recent traces are serviced first, leaving relatively colder and older tasks waiting until the important work has been completed. However, it would also be beneficial to reduce the number of tasks actually being dispatched to the trace translation queue in the event of an overloaded JIT compilation task farm.

In order to avoid large amounts of waiting trace translation tasks, we implemented an adaptive hotspot threshold scheme. Initially, the hotspot threshold is set to a constant value based on the number of JIT workers available – as the number of workers increases, the threshold can be set more aggressively. This threshold is then adjusted based on the priority queue's current length, where a longer queue raises the threshold at which new potential traces are considered to be hot enough. The threshold can either be tied directly to the length of the queue, or a certain queue length can trigger an increase in the hotspot threshold.

4. Empirical Evaluation

We have evaluated our parallel trace-based just-in-time compilation and dynamic work scheduling approach across more than 60 industry standard benchmarks, including BIOPERF, SPEC, EEMBC, and COREMARK, from various domains. In this section we describe our experimental setup and methodology before we present and discuss our results.

4.1 Evaluation Methodology

We have evaluated our parallel trace-based JIT compilation approach using the BIOPERF benchmark suite that comprises a comprehensive set of computationally-intensive life science applications [2]. It is well suited for evaluating our parallel trace-based JIT compiler as it exhibits many different potential hotspots per application for most of its benchmarks. We also used the industry standard EEMBC 1.1, and COREMARK [35] embedded benchmark suites. These benchmarks represent small and relatively short running applications with complex algorithmic kernels. An evaluation using SPEC CPU 2006 benchmarks [18] is included as they are widely used and considered to be representative of a broad spectrum of application domains.

The BIOPERF benchmarks were run with "class-A" input datasets available from the BIOPERF web site. The EEMBC 1.1 benchmarks were run for the default number of iterations and COREMARK was run for 1000 iterations. For practical reasons we used the largest possible data set for each of the SPEC CPU 2006 benchmarks such that simulation time does not become excessive.

Our main focus has been on simulation speedup by reducing the overall simulation time. Therefore we have measured the elapsed real time between invocation and termination of our simulator using the UNIX `time` command. We used the average elapsed wall clock time across 10 runs for each benchmark and configuration (i.e. interpreted-only, decoupled, decoupled parallel) in order to calculate speedups. Additionally, we provide error bars for each benchmark result denoting the standard deviation to show how much variation there is between different program runs.

We use a strong and competitive baseline for our comparisons, namely a decoupled trace-based JIT compiler using one asynchronous thread for JIT compilation [16]. Relative to that baseline we plot the speedups achieved by our parallel trace-based JIT compiler using three asynchronous JIT compilation threads. Fur-

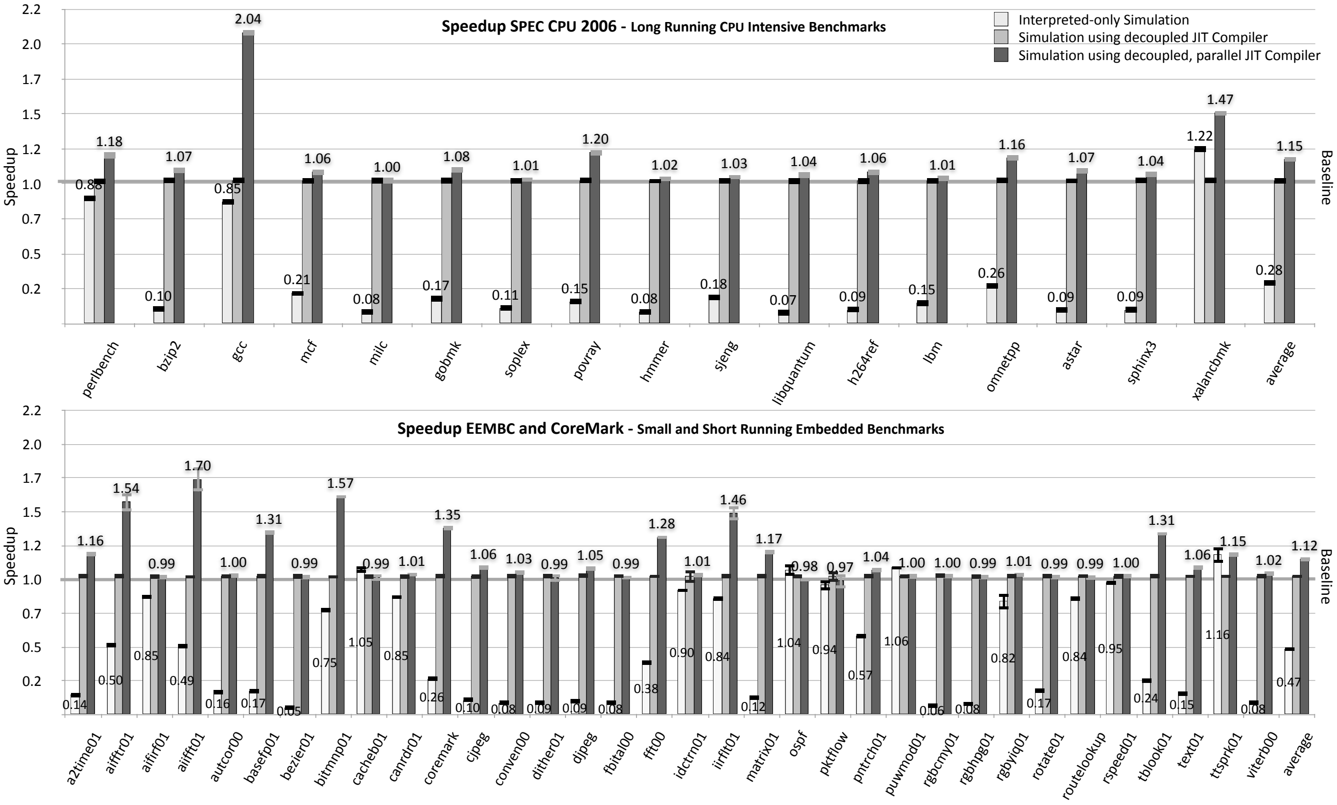


Figure 8. Speedups using SPEC CPU 2006, EEMBC and COREMARK benchmark suites comparing (a) interpreted-only simulation, (b) simulation using a decoupled JIT compiler, and (c) simulation using our novel, parallel JIT compiler with three JIT compilation worker threads including dynamic work scheduling.

thermore, we also present speedups (i.e. slowdowns) relative to our baseline when using interpreted simulation only (i.e. disabling trace-based JIT compilation).

All measurements were performed on a standard x86 DELL™ POWEREDGE™ quad-core outlined in Table 1 under conditions of low system load. To evaluate the scalability of our approach, when adding more cores, we performed additional measurements on a parallel symmetric multiprocessing machine with 16 2.6 GHz AMD Opteron™ (AMD64e) processors running Scientific Linux 5.0 (see Section 4.4).

4.2 Summary of Key Results

Our novel parallel trace-based JIT compilation approach is *always* faster than the baseline decoupled JIT compiler and achieves an average speedup of **1.38** equivalent to an average execution time reduction of **22.8**% for the BIOPERF benchmark suite. This corresponds directly to an average increase of **14.7**% in the number of natively executed instructions compared to the baseline.

For some benchmarks (e.g. `blastp`, `clustalw`) our proposed scheme is more than *twice* as fast as the baseline. This can be explained by the fact that 59% of the time we find more than one hot trace per trace interval for `blastp`. From this it follows that `blastp` exhibits a large amount of task parallelism (see Box ② in Figure 5). `Clustalw` mainly benefits from hiding compilation latency by using several JIT compilation worker threads (see Box ① in Figure 5).

Shorter running BIOPERF benchmarks perform particularly well with our scheme (e.g. `tcoffee`, `hmmsearch`, `clustalw`) because more JIT compilation workers can deliver translations much quicker as they can split the workload (i.e. hide compilation latency). Especially for `hmmsearch` where the baseline decoupled JIT compiler performs worse than the interpreted-only version, our scheme can significantly boost execution speed and reduce overall simulation time by 34.4%. Even for very long running BIOPERF benchmarks (e.g. `fasta-ssearch`, `promlk`, `hmmer-hmmpfam`, `ce`), where JIT compilation time typically represents only a small fraction of the overall execution time, our scheme achieves a reduction of execution times of up to 6.8%.

4.3 Worst-Case Scenarios

The BIOPERF benchmark suite is well suited to show the efficacy of our parallel trace-based JIT compiler. Additionally we also demonstrate its favorable impact on benchmarks where we would not expect to see significant speedups from our technique - so called *worst-case scenarios*.

For this analysis we have considered short running embedded benchmarks (EEMBC, COREMARK) containing few application hotspots (i.e. algorithmic kernels). Some of these benchmarks are so short that interpreted only execution takes less than two seconds, leaving very little scope for improvement by a JIT compiler. At the other end of the scale are very long running and CPU intensive benchmarks (SPEC CPU 2006) where JIT compilation time contributes only a marginal fraction to the overall execution time.

Across the SPEC CPU 2006 benchmarks our parallel trace-based JIT compiler is *never* slower than the baseline and achieves an average speedup of **1.15**, corresponding to an average increase of **4.2**% in the number of natively executed instructions. The best speedups are achieved for `gcc` (2.04x), `xalancbmk` (1.47x),

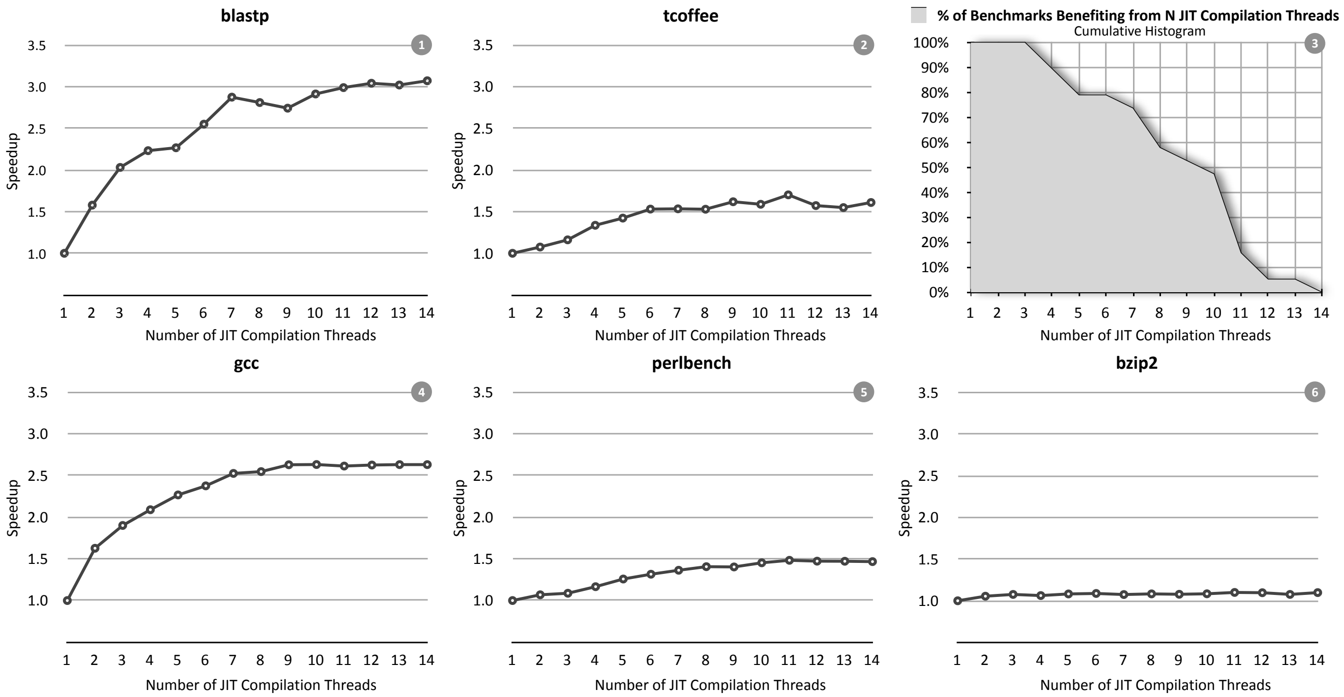


Figure 9. Scalability charts for selected benchmarks from BioPerf ①② and Spec Cpu 2006 ④⑤⑥ demonstrating the effect of additional Jit compilation workers on speedup. Top right cumulative histogram ③ shows % of benchmarks benefiting from given number of Jit compilation threads.

Vendor & Model	Dell™PowerEdge™1950
Number Cpus	4 (quad-core)
Processor Type	Intel©Xeon™ processor E5430
Clock/Fsb Frequency	2.66/1.33 Ghz
L1-Cache	32K Instruction/Data caches
L2-Cache	12 Mb
Operating System	Scientific Linux 5.5 (64-bit)

Table 1. Simulation Host Configuration.

povray (1.2x), and perlbench (1.18x). For perlbench and gcc the number of natively executed instructions improves by 19.5% and 38.0%, respectively, when using our parallel trace-based Jit compiler. The gcc benchmark greatly benefits from our approach as it runs a compiler with many optimization flags enabled resulting in a multitude of application hotspots representing compilation phases.

Xalancbmk is one of the shorter running Spec Cpu benchmarks performing Xml transformations. Due to its short runtime and abundance of application hotspots the tracing overhead causes the baseline decoupled Jit compiler to be slower than the interpreted-only version. Our parallel trace-based Jit can easily recover this overhead resulting in a speedup of 1.47 when compared to the baseline, and a speedup of 1.21 when compared to interpreted-only simulation. Povray represents a long running benchmark where we achieve a speedup of 1.2. This is again due to an abundance of application hotspots across the runtime of the povray benchmark.

For the Eembc and CoreMark benchmark suites our approach achieves an average speedup of 1.12 over the baseline, and an average improvement of 3.8% in the number of natively executed instructions. Small embedded benchmarks do not often benefit from task parallelism because they rarely exhibit more than one hot trace per trace interval. The speedups are mostly due to the fact that Jit compilation workers can already start working on newly discovered traces while previous traces are being translated by other workers. Also tracing must be light-weight, causing only very little overhead, to enable speedups for small benchmarks like Eembc and CoreMark.

For all benchmarks performing Fast Fourier Transforms (i.e. aifftr01, aiifft01, fft00) speedups ranging from 1.46 to 1.7 are achieved by our parallel trace-based Jit compiler. The bit manipulation (bitmnp01) and infinite impulse response filter (iirflt01) benchmarks also show speedups of 1.57 and 1.46 using our scheme. Three Eembc benchmarks (cacheb01, puwmod01, ospf) yield very short runtimes using interpreted-only mode (i.e. below one second) causing a small slowdown of the baseline decoupled Jit compiler and our parallel Jit compiler. This is entirely due to the fact that for very short benchmarks a Jit compiler has almost no chance to speed up execution, actually causing a slight slowdown due to the overheads caused by tracing and Jit compiler thread creation.

4.4 Available Parallelism and Scalability

According to Amdahl's law we expected only marginal improvements with increasing numbers of Jit compilation worker threads, so it is remarkable how well some benchmarks scale (see Figure 9) on a 16-core system. For blastp ① from BioPerf the maximum speedup of 3.1 is reached with 14 Jit compilation workers. The gcc ④ and perlbench ⑤ benchmarks from Spec Cpu reach their maximum speedup of 2.6 and 1.5 with 10 and 11 Jit compilation workers, respectively. Not all benchmarks show benefits from adding more Jit compilation threads. For bzip2 ⑥ from Spec Cpu the peak speedup is reached with 3 Jit compilation threads, thus adding more threads does not improve execution time.

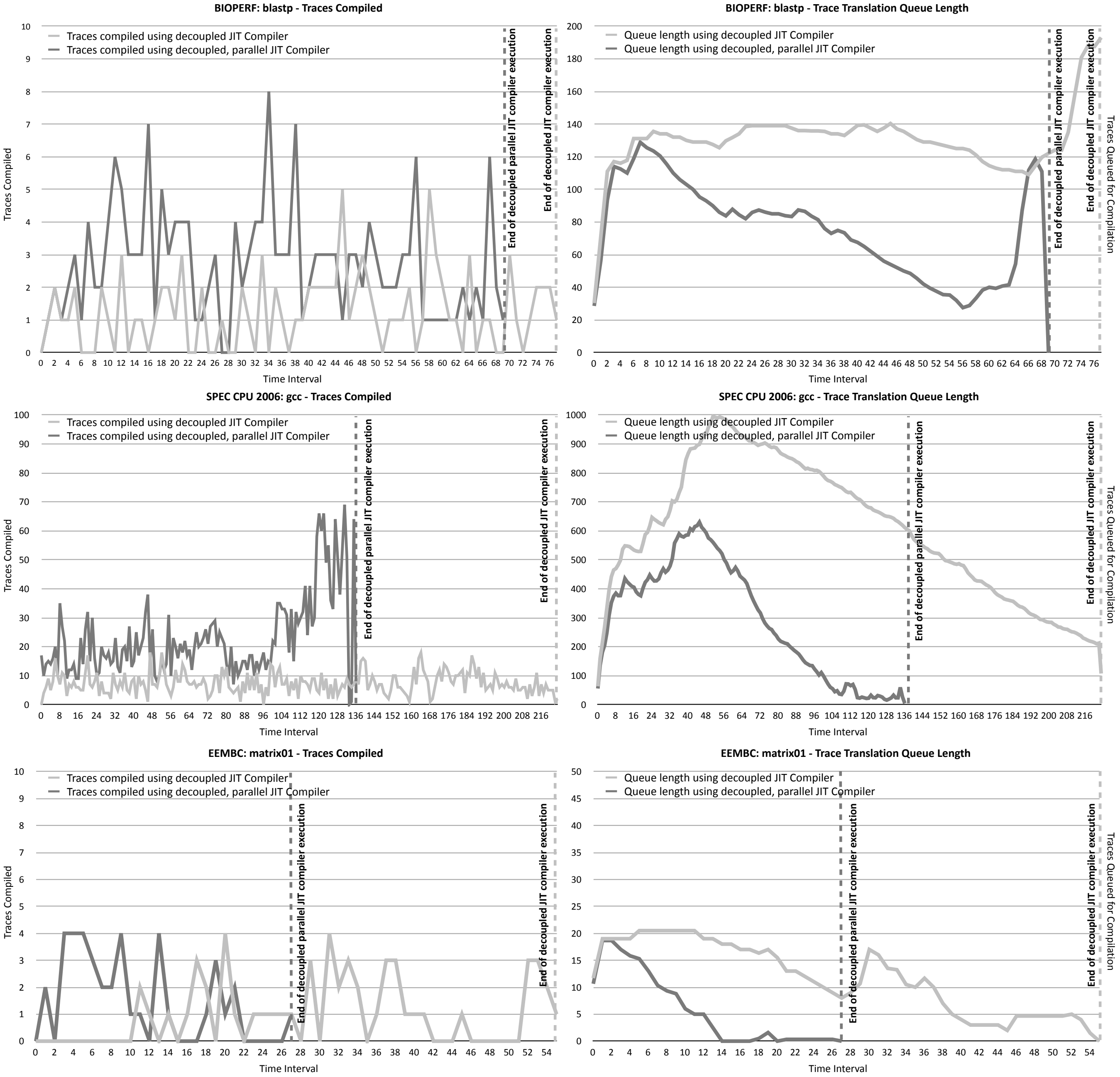


Figure 10. Comparing compiled traces per time interval and trace translation queue length per time interval using an *aggressive* threshold for hot trace selection for (a) simulation using a decoupled JIT compiler, and (b) simulation using our novel, parallel JIT compiler with three JIT compilation worker threads.

As shown in the scalability charts in Figure 9, the peak speedup is reached with different numbers of JIT compilation worker threads. So what is the maximum number of JIT compilation threads such that 100% of all benchmarks show speedup, i.e. how far does it scale? The cumulative histogram ③ in Figure 9 answers this question by depicting the number of benchmarks (in %) that show an improvement by adding more JIT compilation threads. From the histogram we can see *all* benchmarks show speedups with 3 JIT compilation worker threads and 50% of all benchmarks benefit from 9 or more JIT compilation threads.

4.5 Dynamic Work Scheduling

We have also evaluated the impact of our novel dynamic work scheduling scheme in comparison to a simpler approach that schedules traces based on their order of creation for JIT compilation. For BIOPERF, we measured an average improvement of 8.9%, which is equivalent to an average speedup of 1.13. For the SPEC CPU 2006 benchmarks, the average improvement and speedup are 3.5% and 1.04, respectively.

4.6 Compiled Traces and Trace Translation Queue Length over Time

The key performance indicator of our parallel trace compilation scheme is the reduction of overall simulation time. In this section we give a more detailed overview of two additional performance indicators, namely (1) the number of compiled traces per time interval (i.e. rate at which work is completed), and (2) the resulting trace translation queue lengths over the runtime of three selected benchmarks (see Figure 10). We chose one benchmark from each benchmark suite exhibiting a variety of application hotspots over time, and used a rather aggressive threshold for hot trace selection to highlight some of the main advantages of using more than one JIT compilation thread.

A comparison of the number of compiled traces and trace translation queue lengths over time indicates that our parallel JIT compilation task farm is able to translate traces significantly faster than the decoupled scheme which relies on a single JIT compilation thread. On average, three parallel JIT compilation threads can translate 2.1, 3.1 and 1.5 times as many traces per time interval than the decoupled JIT compiler for `blastp`, `gcc` and `matrix01`, respectively. Consequently, the *average* trace translation queue length is 43%, 52% and 51% shorter, and the *maximum* observed queue sizes are 33%, 37%, and 9% shorter for the same three benchmarks, respectively. At the same time, the queue length grows at a noticeably slower rate.

Using a very aggressive initial hotspot threshold, the amount of hot traces identified per trace interval and the resulting translation queue lengths quickly exceed the number of available JIT translation workers, even for a short benchmark such as `matrix01`. This demonstrates the need for our dynamic work scheduling and adaptive hotspot threshold selection schemes. Dynamic work scheduling ensures that the most recent and hottest traces are scheduled for translation first, whereas adapting the hotspot threshold based on the translation queue length aims at improving the utilisation of available resources.

4.7 Memory Overhead of Parallel JIT Compilation

The use of multiple LLVM JIT compiler threads incurs a memory overhead. However, we found this overhead to be modest compared to the baseline. Internal memory consumption (i.e. memory which is not application data or target binary code) never exceeds 150 MB for any of the benchmarks.

5. Related Work

5.1 Dynamic Binary Translation

Dynamic translation techniques are used to overcome the lack of flexibility inherent in statically-compiled simulators. The MIMIC simulator [24] simulates IBM SYSTEM/370 instructions on the IBM RT PC and translates groups of target basic blocks into host instructions. SHADE [10] and EMBRA [21] use DBT with translation caching techniques in order to increase simulation speeds. The Ultra-fast Instruction Set Simulator [45] improves the performance of statically-compiled simulation by using low-level binary translation techniques to take full advantage of the host architecture.

Just-In-Time Cache Compiled Simulation (JIT-CCS) [28] executes and caches pre-compiled instruction-operation functions for each function fetched. The Instruction Set Compiled Simulation (IC-CS) simulator [31] was designed to be a high performance and flexible functional simulator. To achieve this the time-consuming instruction decode process is performed during the compile stage, whilst interpretation is enabled at simulation time. The SIMICS [31] full system simulator translates the target machine-code instructions into an intermediate format before interpretation. During simulation the intermediate instructions are processed by the interpreter which calls the corresponding service routines. QEMU [5] is a fast simulator using an original dynamic translator. Each target instruction is divided into a simple sequence of micro-operations, the set of micro-operations having been pre-compiled offline into an object file. During simulation the code generator accesses the object file and concatenates micro-operations to form a host function that emulates the target instructions within a block. More recent approaches to JIT DBT ISS are presented in [7, 19, 30, 38]. Apart from different target platforms these approaches differ in the granularity of translation units (basic blocks *vs.* pages or CFG regions) and their JIT code generation target language (ANSI-C *vs.* LLVM IR).

Parallel simulation of multi-core target platforms on multi-core host platforms has been demonstrated in [21, 25, 40]. These approaches, however, are mainly concerned with the mapping of target to host processors and do not consider the parallelization of the embedded JIT compiler.

5.2 Trace-based JIT Optimization/Compilation

Tracing is a well established technique for dynamic profile guided optimization of native binaries. Bala et al. [3] introduced tracing as a method for runtime optimization of native program binaries in their DYNAMO system. They used backward branch targets as candidates for the start of a trace, but did not attempt to capture traces of loops. Zaleski et al. [43] used DYNAMO-like tracing in order to achieve inlining, indirect jump elimination, and other optimizations for Java. Their primary goal was to build an interpreter that could be extended to a tracing VM.

Whaley [41] uses partial method compilation to reduce the granularity of compilation to the sub-method level. His system uses profile information to detect never or rarely executed parts of a method and to ignore them during compilation. If such a part gets executed later, execution continues in the interpreter. Compilation still starts at the beginning of a method. Similarly, Suganuma et al. [33] propose region-based compilation to overcome the limitations of method-based compilation. They use heuristics and profiles to identify and eliminate rarely executed sections of code, but rely on expensive runtime code instrumentation for trace identification.

Gal et al. [13, 15] proposed an approach to building dynamic compilers in which no CFG is ever constructed, and no source code level compilation units such as methods are used. Instead, they use runtime profiling to detect frequently executed cyclic code paths in the program. The compiler then records and generates code from dynamically recorded code traces along these paths. It assembles these traces dynamically into a trace tree, a tree-like data-structure, that covers frequently executed (and thus compilation worthy) code paths through hot code regions. Trace trees suffer from the problem of code explosion when many control-flow paths are present in a loop, causing them to grow to very large sizes due to excessive tail duplication as outlined in [4]. To solve this problem Bebenita et al. [4] propose to use trace-regions as a data-structure for tracing in their implementation of Hotpath, a trace-based Java JIT compiler in the Maxine VM. Trace-regions are an extension to trace trees as they can include join nodes instead of using tail duplication. Locations where trace recording can be enabled, so called *anchors*, are determined statically during byte-code verification, and trace regions are restricted to method boundaries. In contrast, our approach does not rely on statically determined anchors for tracing and trace regions are not confined to method boundaries.

5.3 Parallel JIT Compilation

JIT compilation has a long history [1] dating back to the 1960s. The possibility of reducing the overhead of dynamic compilation by decoupling the JIT compiler from the main simulation loop and executing it in a separate thread has been suggested by several researchers, e.g. [16, 20, 39].

Parallel JIT compilation is not an entirely new concept. Some approaches [8, 14] have attempted to exploit *pipeline parallelism* in the JIT compiler. However, pipelining of the JIT compiler has significant drawbacks. First, compiler stages are typically not well balanced and the overall throughput is limited by the slowest pipeline stage – this is often the front-end or IR generation stage. Second, unlike method based compilers, trace-based JIT compilers operate on relatively small translation units in order to reduce the compilation overhead to a bare minimum [15]. Small translation units and long compilation pipelines, however, increase the relative synchronization costs between pipeline stages and, again, limit the achievable compiler throughput. Third, compilation pipelines are static and do not scale with the available *task parallelism* in inherently independent translation units.

Most relevant to our work is the approach presented in [30]. This paper pioneered the concept of concurrent JIT compilation workers to speed up DBT, but suffers from a number of flaws. First, rather than taking a trace-based compilation approach entire pages are translated – this is unnecessarily wasteful in a time-critical JIT environment. Second, there are no provisions for a dynamic work scheduling scheme that prioritizes compilation of hot traces – this may defer compilation of critical traces and lower overall efficiency. Third, JIT compilers reside in separate processes on remote machines – this significantly increases the communication overhead and limits scalability. This last point is critical, as results shown in [30] are based solely on CPU time of the main simulation process rather than the more relevant wall clock time that includes CPU time, I/O time and communication channel delay.

6. Summary and Conclusions

In this paper we have presented a generalized trace construction scheme enabling parallel JIT compilation based on a task farm design pattern. By combining parallel JIT compilation with lightweight tracing, large translation units and dynamic work scheduling, we not only minimize and hide JIT compilation overhead, but fully exploit the available hardware parallelism in standard multicore desktop PCs. Across three full benchmark suites comprising non-trivial and long-running applications from various domains we achieve an average reduction in total execution time of 11.5% – and up to 51.9% – for four processors. Our innovative, parallel JIT scheme is robust and never results in a slowdown. Given that only a small fraction of the overall execution time is spent on JIT compilation, and the majority of time is spent executing natively-compiled code, these results are more than remarkable.

While primarily developed for DBT, the concept of concurrent JIT compilation may also be exploited elsewhere to effectively reduce dynamic compilation overheads and speedup execution. Prime examples are JIT-compiled Java Virtual Machines (JVM) or JavaScript engines. In our future work, we plan to extend our DBT for the simulation of multi-core architectures, and explore alternative dynamic work scheduling strategies.

References

[1] J. Aycock. A brief history of just-in-time. *ACM Comput. Surv.*, 35: 97–113, June 2003.

[2] D. Bader, Y. Li, T. Li, and V. Sachdeva. Bioperf: a benchmark suite to evaluate high-performance computer architecture on bioinformatics applications. In *Proceedings of the IEEE International Workload Characterization Symposium*, IISWC'05, pages 163–173, 2005.

[3] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 1–12, New York, NY, USA, 2000. ACM.

[4] M. Bebenita, M. Chang, G. Wagner, A. Gal, C. Wimmer, and M. Franz. Trace-based compilation in execution environments without interpreters. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, PPPJ '10, pages 59–68, New York, NY, USA, 2010. ACM.

[5] F. Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference*, ATEC '05, page 41, Berkeley, CA, USA, 2005. USENIX Association.

[6] I. Böhm, B. Franke, and N. Topham. Cycle-accurate performance modelling in an ultra-fast just-in-time dynamic binary translation instruction set simulator. In *Proceedings of the International Conference on Embedded Computer Systems*, IC-SAMOS'10, pages 1 – 10, 2010.

[7] F. Brandner, A. Fellnhofer, A. Krall, and D. Riegler. Fast and accurate simulation using the llvm compiler framework. In *Proceedings of the 1st Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, RAPIDO'09, pages 1–6, 2009.

[8] S. Campanoni, G. Agosta, and S. C. Reghizzi. A parallel dynamic compiler for CIL bytecode. *SIGPLAN Not.*, 43:11–20, April 2008.

[9] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. Bharadwaj Yadavalli, and J. Yates. FX!32: a profile-directed binary translator. *Micro, IEEE*, 18(2):56–64, 1998.

[10] B. Cmelik and D. Keppel. Shade: a fast instruction-set simulator for execution profiling. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '94, pages 128–137, New York, NY, USA, 1994. ACM.

[11] A. Cohen and E. Rohou. Processor virtualization and split compilation for heterogeneous multicore embedded systems. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 102–107, New York, NY, USA, 2010. ACM.

[12] S. Farfeleder, A. Krall, and N. Horspool. Ultra fast cycle-accurate compiled emulation of inorder pipelined architectures. *J. Syst. Archit.*, 53:501–510, August 2007.

[13] A. Gal, C. W. Probst, and M. Franz. HotpathVM: an effective JIT compiler for resource-constrained devices. In *Proceedings of the 2nd International Conference on Virtual Execution Environments*, VEE '06, pages 144–153, New York, NY, USA, 2006. ACM.

[14] A. Gal, M. Bebenita, M. Chang, and M. Franz. Making the compilation "pipeline" explicit: Dynamic compilation using trace tree serialization. Technical Report 07-12, University of California, Irvine, 2007.

[15] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 465–478, New York, NY, USA, 2009. ACM.

[16] J. Ha, M. Haghighat, S. Cong, and K. McKinley. A concurrent trace-based just-in-time compiler for single-threaded JavaScript. In *Proceedings of the Workshop on Parallel Execution of Sequential Programs on Multicore Architectures*, PESPMA'09, 2009.

[17] M. Hauswirth, P. F. Sweeney, A. Diwan, and M. Hind. Vertical profiling: understanding the behavior of object-priented applications. In *Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 251–269, New York, NY, USA, 2004. ACM.

[18] J. L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34:1–17, September 2006.

[19] D. Jones and N. Topham. High speed CPU simulation using LTU dynamic binary translation. In *High Performance Embedded Architectures and Compilers*, volume 5409 of *Lecture Notes in Computer Science*, pages 50–64. Springer Berlin Heidelberg, 2009.

[20] C. J. Krintz, D. Grove, V. Sarkar, and B. Calder. Reducing the overhead of dynamic compilation. *Software: Practice and Experience*, 31(8):717–738, July 2001.

[21] R. Lantz. Fast functional simulation with parallel Embra. In *Proceedings of the 4th Annual Workshop on Modeling, Benchmarking and Simulation*, MOBS'08, 2008.

[22] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '04, page 75, Washington, DC, USA, 2004. IEEE Computer Society.

[23] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. PIN: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[24] C. May. MIMIC: a fast System/370 simulator. In *Papers of the Symposium on Interpreters and Interpretive Techniques*, SIGPLAN '87, pages 1–13, New York, NY, USA, 1987. ACM.

[25] J. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A distributed parallel simulator for multicores. In *Proceedings of the IEEE 16th International Symposium on High Performance Computer Architecture*, HPCA'10, pages 1–12, 2010.

[26] Mozilla Foundation. Tamarin project, 17 November 2010. URL `http://www.mozilla.org/projects/tamarin/`.

[27] Mozilla Foundation. Tracemonkey, 17 November 2010. URL `https://wiki.mozilla.org/JavaScript:TraceMonkey`.

[28] A. Nohl, G. Braun, O. Schliebusch, R. Leupers, H. Meyr, and A. Hoffmann. A universal technique for fast and flexible instruction-set architecture simulation. In *Proceedings of the 39th annual Design Automation Conference*, DAC '02, pages 22–27, New York, NY, USA, 2002. ACM.

[29] Oracle Corporation. HotSpot VM, 17 November 2010. URL `http://java.sun.com/performance/reference/whitepapers/`.

[30] W. Qin, J. D'Errico, and X. Zhu. A multiprocessing approach to accelerate retargetable and portable dynamic-compiled instruction-set simulation. In *Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '06, pages 193–198, New York, NY, USA, 2006. ACM.

[31] M. Reshadi, P. Mishra, and N. Dutt. Instruction set compiled simulation: a technique for fast and flexible instruction set simulation. In *Proceedings of the 40th annual Design Automation Conference*, DAC '03, pages 758–763, New York, NY, USA, 2003. ACM.

[32] E. Stahl and M. Anand. A comparison of PowerVM and x86-based virtualization performance. Technical Report WP101574, IBM Techdocs White Papers, 2010.

[33] T. Suganuma, T. Yasue, and T. Nakatani. A region-based compilation technique for dynamic compilers. *ACM Trans. Program. Lang. Syst.*, 28:134–174, January 2006.

[34] V. Tan. Asynchronous just-in-time compilation. *United States Patent Application*, WO/2007/055883, 2007.

[35] The Embedded Microprocessor Benchmark Consortium. EEMBC benchmark suite, 12 August 2009. URL `http://www.eembc.org`.

[36] The LLVM Compiler Infrastructure. Users of LLVM JIT compilation engine, 17 November 2010. URL `http://llvm.org/Users.html`.

[37] The WebKit Open Source Project. SquirrelFish, 17 November 2010. URL `http://trac.webkit.org/wiki/SquirrelFish`.

[38] N. Topham and D. Jones. High speed CPU simulation using JIT binary translation. In *Proceedings of the Annual Workshop on Modelling, Benchmarking and Simulation*, MOBS '07, 2007.

[39] P. Unnikrishnan, M. Kandemir, and F. Li. Reducing dynamic compilation overhead by overlapping compilation and execution. In *Proceedings of the 2006 Asia and South Pacific Design Automation Conference*, ASP-DAC '06, pages 929–934, Piscataway, NJ, USA, 2006. IEEE Press.

[40] K. Wang, Y. Zhang, H. Wang, and X. Shen. Parallelization of IBM Mambo system simulator in functional modes. *SIGOPS Oper. Syst. Rev.*, 42:71–76, January 2008.

[41] J. Whaley. Partial method compilation using dynamic profile information. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '01, pages 166–179, New York, NY, USA, 2001. ACM.

[42] B. Yee, D. Sehr, G. Dardyk, J. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. In *30th IEEE Symposium on Security and Privacy*, pages 79–93, May 2009.

[43] M. Zaleski, A. D. Brown, and K. Stoodley. YETI: a gradually extensible trace interpreter. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE '07, pages 83–93, New York, NY, USA, 2007. ACM.

[44] C. Zheng and C. Thompson. PA-RISC to IA-64: transparent execution, no recompilation. *Computer*, 33(3):47 –52, Mar. 2000.

[45] J. Zhu and D. D. Gajski. A retargetable, ultra-fast instruction set simulator. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '99, New York, NY, USA, 1999. ACM.