The Dissertation Committee for Jung Woo Ha
certifies that this is the approved version of the following dissertation:

Scaling Managed Runtime Systems for Future Multicore Hardware

Committee:

Kathryn S. McKinley, Supervisor

Matthew Arnold

Stephen M. Blackburn

Stephen W. Keckler

Emmett Witchel

Scaling Managed Runtime Systems for Future Multicore Hardware

by

Jung Woo Ha, B.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

December 2009

Dedicated to my wife Songhee.

Acknowledgments

I would like to express my deepest appreciation for my advisor Kathryn McKinley. Kathryn not only provided me with technical expertise and positive feedback, but she also motivated, inspired, and made the research exciting. Kathryn not only helped me produce a good research output, but she also focused on improving my research skills, and she spent more of her time than was necessary. Kathryn always showed more by example than words. For example, I learned to make the best use of my time and to balance both research and raising kids. I feel very fortunate to be her student.

Steve Blackburn and Matthew Arnold have been enthusiastic supporters, providing lots of good ideas, advice, and helped with writing. They were always available for meetings and discussions, and encouraged me to improve the quality of my research. I am especially grateful to Steve for attending meetings at an inconvenient time in his time zone.

Emmett Witchel advised me in a new research area, and his support was always helpful. Part of my recent research was motivated and influenced from our earlier collaborations. Steve Keckler gave helpful feedback and spent a lot of time reading my thesis and attending talks.

I would also like to thank the graduate students at UT: Mike Bond, Justin Brickell, Katherine Coons, Jason Davis, Maria Jump, Byeongcheol Lee,

Bert Maher, Don Porter, Dimitris Prountzos, Christopher Rossbach, Hany Ramadan, Indrajit Roy, Jennifer Sartor, and Suriya Subramaniam. I am grateful for many discussions with them, and that they spent a tremendous amount of time helping to improve my papers and talks.

I would like to thank my wife Songhee for her selfless support. She was tolerant and sacrificed herself by taking care of the family. I could not have completed this work without her excellent support.

Scaling Managed Runtime Systems for Future Multicore Hardware

Publication No. _____

Jung Woo Ha, Ph.D.
The University of Texas at Austin, 2009

Supervisor: Kathryn S. McKinley

The exponential improvement in single processor performance has recently come to an end, mainly because clock frequency has reached its limit due to power constraints. Thus, processor manufacturers are choosing to enhance computing capabilities by placing multiple cores into a single chip, which can improve performance given parallel software. This paradigm shift to chip multiprocessors (also called multicore) requires scalable parallel applications that execute tasks on each core, otherwise the additional cores are worthless.

Making an application scalable requires more than simply parallelizing the application code itself. Modern applications are written in *managed languages*, which require automatic memory management, type and memory abstractions, dynamic analysis and just-in-time (JIT) compilation. These managed runtime systems monitor and interact frequently with the executing application. Hence, the managed runtime itself must be scalable, and the

instrumentation that monitors the application should not perturb its scalability.

While multicore hardware forces a redesign of managed runtimes for scalability, it also provides opportunities when applications do not fully utilize all of the cores. Using available cores for concurrent helper threads that enhance the software, with debugging, security, and software support will make the runtime itself more capable and more scalable.

This dissertation presents two novel techniques that improve the scalability of managed runtimes by utilizing unused cores. The first technique is a *concurrent dynamic analysis framework* that provides a low-overhead buffering mechanism called *Cache-friendly Asymmetric Buffering (CAB)* that quickly offloads data from the application to helper threads that perform specific dynamic analyses. Our framework minimizes application instrumentation overhead, prevents microarchitectural side-effects, and supports a variety of dynamic analysis clients, ranging from call graph and path profiling to cache simulation. The use of this framework ensures that helper threads perturb the performance of application as little as possible.

Our second technique is *concurrent trace-based just-in-time compilation*, which exploits available cores for the JavaScript runtime. The JavaScript language limits applications to a single-thread, so extra cores are worthless unless they are used by the runtime components. We redesigned a production trace-based JIT compiler to run concurrently with the interpreter, and our technique is the first to improve both responsiveness and throughput in a

trace-based JIT compiler.

This thesis presents the design and implementation of both techniques and shows that they improve scalability and core utilization when running applications in managed runtimes. Industry is already adopting our approaches, which demonstrates the urgency of the scalable runtime problem and the utility of these techniques.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Recent trends in modern software towards managed and virtualized programming systems, and trends in hardware towards chip multiprocessor (also known as multicore) have created an immediate and urgent need for (1) parallel applications and (2) runtime systems that are themselves parallel and that can reason about and optimize parallel applications on multicore hardware.

Multicore Hardware Era. *Exponential* improvements in computer hardware performance over the past few decades have created an ecosystem of ever more capable software that has revolutionized science, communication, entertainment, business, and government. Hardware performance improvements have been mostly the result of scaling up the clock frequency of single processors combined with a *sequential* and *portable* software programming model. *Sequential* programs execute one task at a time. *Portable* software operates on multiple generations of different hardware platforms. Unfortunately, frequency scaling has reached its limit due to power, wire, and other technology constraints. As a result, hardware vendors are instead seeking to improve performance by putting multiple cores on a single chip. This type of hardware

is referred to as *chip multiprocessors* or *multicore*. Multicore processors are already available in the mainstream market and are found in a wide range of computing environments including servers, desktops, mobile, and embedded computers.

Modern Applications Use Managed Runtime Systems. The rapid improvements in processor speed have resulted in programmers more frequently choosing high-level *managed* sequential languages and programming environments to help them create large, correct, capable, and sequential applications. *Managed* languages such as Java, JavaScript, Ruby, and C# encourage a modular object-oriented programming style, enforce rich static or dynamic type system, and provide automatic memory management. *A managed runtime* then performs dynamic analysis, dynamic just-in-time (JIT) compilation, optimization, scheduling, and automatic memory management together with the running application. Managed runtime systems also include dynamic binary translators and hypervisors. Although there are various types of managed runtime systems, they all generally provide applications with rich features such as dynamic optimization [3, 35], dynamic analysis [14], profiling [4], memory management, enhanced debugging, and improved security [47]. Many modern applications run inside a managed runtime system to obtain these benefits.

Challenges and Opportunities of Multicores for Managed Runtime Systems The performance promise of multicore can only be fulfilled when

software executes in parallel, and software scalability is now the most important performance concern. It is not possible to improve software performance by increasing the number of cores without application scalability. For managed applications, the whole system, including both the applications and the managed runtime, must be scalable. Parallelizing both the application and each managed runtime component is thus necessary. In particular, communication between the application and the runtime components must be carefully orchestrated to achieve overall scalability. The managed runtime typically interacts with applications through instrumented code, which pauses the application and performs runtime system-level tasks such as dynamic analysis, dynamic compilation, and garbage collection. Many of these components are either fully or partially sequential, because until now, parallel hardware was not widespread, and short frequent communication was sufficiently efficient and exploited memory locality. In the multicore era, redesign of the managed runtime is critical to reduce the pause time incurred by instrumented code and improve application and runtime scalability.

Although multicore hardware presents a scalability challenge to managed runtimes, it also provides new opportunities. When a single application does not fully utilize all available cores, moving or adding managed runtime functionality to concurrent helper threads will hide their overhead. This functionality includes dynamic analysis for performance optimization, debugging, security, and software support. These analyses can collect the necessary data from the application and process it on a separate core perturbing the applica-

tion less. Such fine-grained helper threads were not practical when processors shared data only through high latency off-chip memory, but multicore processors significantly reduce memory latency by on-chip communication. However, extra cores are not free and the performance of the runtime system is related to microarchitectural factors such as cache configuration and sharing, which require careful design for successful deployment.

In summary, performance on multicore processors require application scalability, which in turn requires scalable managed runtime components. Multicore processors also present new opportunities in the runtime layer. The runtime can enhance existing services and add new ones by using extra cycles with concurrent helper threads.

1.1 Improving Managed Runtime Systems on Multicore Processors

We present two novel techniques for enhancing scalability and reducing application overhead using concurrent threads to improve managed runtime systems on multicore processors: (1) *a concurrent dynamic analysis framework* and (2) *a concurrent trace-based JIT compiler*.

1.1.1 Utilizing Extra Cycles with Helper Threads

Managed runtime system components perform various dynamic analyses. For example, JIT compilers collect call graph or edge profiling data at runtime, and many data race detectors collect the happens-before relationships

of loads and stores. Most of these dynamic analyses are sequential, (i.e., they pause the application thread to perform the analysis) and generally perform sampling to reduce the overhead. However, depending on the analysis, sampling is not always feasible [28], and the sequential part of dynamic analysis reduces the scalability of the application. Our research explores the possibility of executing the analysis concurrently to exploit under-utilized cores and improve scalability. Designing a concurrent analysis scheme is challenging because micro-architectural side-effects and the synchronization between the application and the analysis thread can incur more overhead than sequential analysis.

We introduce a novel *concurrent dynamic analysis framework* to facilitate the implementation of low-overhead concurrent dynamic analysis [25]. The key to our framework is a novel and general producer, consumer communication mechanism, we call *Cache-friendly Asymmetric Buffering (CAB)*. CAB provides a communication channel between the application and the analysis thread so that the application can quickly transfer analysis data off the critical path with minimal overhead. CAB transfers data between the application and analysis thread in a way that proactively prevents micro-architectural side-effects. With our framework, dynamic analysis writers can easily achieve scalability by utilizing extra cycles without having to consider low-level micro-architectural optimization. We show that the CAB communication mechanism is sufficiently general and efficient on a variety of multicore hardware platforms for a wide variety of dynamic analysis problems, from light-weight call graph

profiling to path profiling and cache simulation. Our results demonstrate the success of our design as well as the need for careful concurrent communication design.

1.1.2 Improving Concurrency of Emerging Runtime Systems

Industry recently embraced JIT compilation to improve the performance of scripting languages such as JavaScript and ActionScript. Developers are increasingly implementing sophisticated web applications in AJAX (shorthand for asynchronous JavaScript and XML), which uses these scripting languages heavily, and interpretation cannot keep up with the increasing demand for performance. Nonetheless, naively applying traditional method-based JIT compilation by evolving Java virtual machine technology has not worked, because these languages are dynamically typed. Because the type of an object is determined at run-time, the compiler must either generate machine code for all combinations of types or perform type specialization, neither of which has proven effective.

Recent work has introduced trace-based JIT compilation as an alternative method of type specialization for dynamic languages [18]. Instead of compiling the whole method, trace-based JIT focuses on compiling a hot path within a loop. The interpreter performs loop back-edge profiling, and traces the sequence of instructions on the hot loop path. It only compiles the traced instructions later when they become hot. Naturally, the type of an object is specialized during tracing, which makes the code efficient. Firefox, one of

the most popular web browsers, uses this technique, which has proved to be practical.

While trace-based JIT has led to huge performance improvements in JavaScript applications, the performance does not scale on multicore processors because current JavaScript runtimes are all sequential. Because the JavaScript language is sequential – it executes only in one thread, more than one core is useless, unless the managed runtime exploits them. While running a compiler concurrently is trivial for a method-based JIT, previous attempts to make trace-based JIT concurrent have not been successful [17] because of the complex tracing and update states it requires.

We introduce a novel *concurrent trace-based JIT compiler* that executes concurrently with the application on a separate core [27]. Our technique improves application responsiveness because it reduces synchronization with the application to the bare minimum, a single compare and swap. Moreover, the compiler improves the overall throughput because it delivers native code more quickly. While current sequential trace compilers perform a minimal amount of optimization to keep the compilation pause time small, our technique provides an opportunity to reduce the time and improve the code quality without slowing down the application.

1.2 Meaning and Impact

Our concurrent dynamic analysis framework provides the basis for low overhead communication between applications and other managed runtime

components, which decouples any micro-architectural optimization from the development of the managed runtime service. Any managed runtime service developer can use our framework to collect data from other cores to design a concurrent service. This framework is not limited to dynamic analyses, but can be applied to any system component that requires offloading data from application instrumented code, such as binary translators and hypervisors. Furthermore, our CAB mechanism is generally applicable to any parallel producer, consumer algorithm that seeks to maximize the performance of the producer.

Both of our techniques have immediate practical impact because they can be deployed now to improve managed runtime scalability. For example, Mozilla, Adobe, and Intel are considering incorporating our concurrent trace-based JIT design principles into their products. ISI East is considering using our concurrent dynamic analysis framework for their mission-critical software.

Software scalability is growing in importance as the number of cores in a processor increases. Use of these techniques and others, such as more scalable and memory efficient garbage collection, are essential to provide a basis for application scalability.

Chapter 2

A Concurrent Dynamic Analysis Framework for Multicore Processors

Dynamic analysis is a base technology for performance optimization [1, 12, 45], debugging [29, 39, 44], software support [28, 56], and security [36, 40]. Binary rewriting systems and Just-In-Time (JIT) compilers in managed runtimes need dynamic information about the program to optimize it. They often employ techniques for reducing the overhead, such as sampling, that trade accuracy for performance. However, dynamic analyses used for debugging, software support, and security often require fully accurate analysis. The overhead of more expensive analyses limit their use.

Multicore architectures offer an opportunity to improve the design and performance of dynamic analysis. As the number of cores on commodity hardware continues to increase and application developers are struggling to parallelize application tasks, exploiting unused processors to perform dynamic analysis in parallel with the application becomes an increasingly appealing option.

This chapter explores the design and implementation of a dynamic analysis framework that exploits under-utilized cores by executing analysis

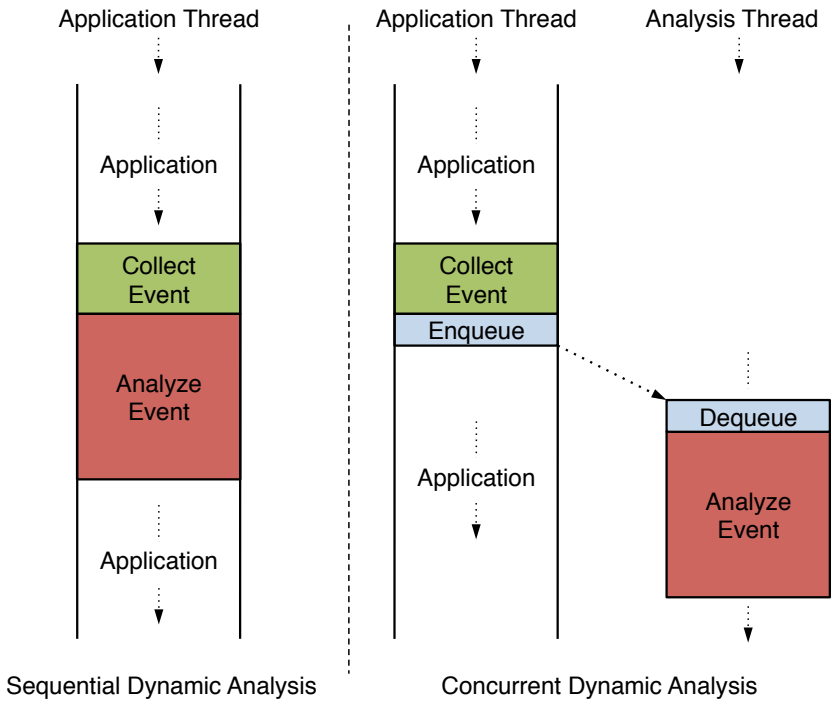


Figure 2.1: Generic sequential dynamic analysis versus concurrent dynamic analysis.

concurrently with the application. In the framework, an application produces events, such as paths executed or memory operations performed, and a separate concurrent analysis thread consumes and analyzes them. Figure 2.1 compares sequential and concurrent dynamic analysis. Whereas traditional dynamic analysis is performed sequentially when the application produces one or a group of events, in our framework, the application queues events in a buffer, and a concurrent analysis thread dequeues and analyzes them.

The ability to communicate data efficiently from one core to another is critical to the success of a concurrent dynamic analysis implementation. Un-

fortunately, the complexity and variety of multicore architectures and memory hierarchies pose substantial challenges to the design of an efficient communication mechanism. We found that a number of variables influence performance, such as hardware variation, communication costs, bandwidth between cores, false sharing between caches, coherence traffic, and synchronization between the producer and consumer threads.

Our main contribution is a new buffering design that we call *Cache-friendly Asymmetric Buffering (CAB)*, which provides an efficient mechanism for communicating event data from application threads to analyzer threads on multicore hardware. CAB is asymmetric because we bias the implementation to minimize impact on the application; the application rarely synchronizes with the analysis thread. The design is cache friendly because it exploits shared caches, carefully limits synchronization, and avoids coherence traffic and contention on shared state between private caches.

We present the design and implementation of a concurrent dynamic analysis framework that uses CAB as its communication mechanism between the application and analysis. We implement the framework in Jikes RVM [1], a high performance research Java Virtual Machine and perform experiments on three Intel processors with very different cache organizations: Pentium 4, Core 2 Quad, and Core i7. We show that compared to two highly optimized state-of-the-art alternative buffering mechanisms: N-way buffering [57] and FastForward concurrent lock-free queues [22], that CAB reduces overhead for path profiling on average by 8 and 41% respectively.

To evaluate the framework, we implement a variety of popular dynamic analyses: *method counting, call graph profiling, call tree profiling, path profiling,* and *cache simulation*. We build and compare sequential and concurrent versions of these analyses.

We demonstrate the framework in an exhaustive mode, for analyses that require fully accurate event records, and in a sampling mode for analyses that can trade accuracy for overhead via sampling. Experimental results for exhaustive mode demonstrate that this framework provides performance improvements for dynamic analysis when the analysis work is greater than the buffering overhead, such as for call graph, call tree, and path profiling. For example, compared to sequential profiling, we reduce the overhead of exhaustive call tree and path profiling between 10 to 70%, depending on the architecture.

In sampling mode, the framework reduces overhead even further. For example, sampling achieves greater than 97% accuracy at a 5% sampling rate, while reducing the overhead by more than half for call graph and path profiling with `hsqldb` benchmark.

In summary, the contributions of this chapter are as follows.

- The design, implementation, and evaluation of CAB, a novel efficient communication mechanism that is easily tuned for various multicore processors.
- The design, implementation, and evaluation of a novel framework for concurrent dynamic analysis using CAB for exhaustive and sampling

analyses. The framework is analysis-neutral and it is easy to add analyses.

- A demonstration of the framework with a range of analyses: method counting, call graph profiling, call tree profiling, path profiling, and cache simulation.

- A demonstration of the framework with a range of threading model: N:M threading, and native threading.

- A cost model that characterizes dynamic analyses amenable to concurrent implementation and that guides the performance analysis.

We believe that the design issues addressed here transcend the framework as these same issues and solutions are applicable more generally to software design for multicore hardware. The CAB design, which carefully manages communication, coherency traffic, false sharing, and cache residency, offers a building block to future software designers tasked with parallelizing managed runtime services and applications with modest to large communication requirements.

2.1 Related Work

We focus here on differences with the most closely related research on dynamic analysis, which exploits parallelism to reduce dynamic analysis overhead.

PiPA (Pipelined Profiling and Analysis) describes a technique for parallelizing dynamic analysis on multicore systems and uses multiple profiling threads per application thread [57]. PiPA is implemented in a dynamic binary translator and collects execution profiles to drive a parallel cache simulator. PiPA uses symmetric N-way buffering and locks to exchange buffers between producers and consumers. Their buffering overhead grows with respect to the size of the buffer, and a small buffer size, e.g., 16KB, achieves the lowest overhead. However, some of their profiling clients require larger buffers for high frequency events. As we show in the results section, CAB is on average 8% faster and up to 16% faster than this organization, and the overhead is consistently low with a large buffer. In our work, the analysis is concurrent (runs in parallel with the application) and parallel (multiple analysis threads run at the same time), but is different from PiPA in that we currently support at most one analysis thread per application thread. This configuration is just for our current implementation, and is not a fundamental limitation of CAB. Our work focuses on efficiently transferring data between cores, and we believe that PiPA would benefit from using CAB.

FastForward is a software-only concurrent lock-free queue implementation for multicore hardware [22]. It uses a sentinel value (`NULL`) to avoid concurrent access of the queue head and tail indices, and forces a delay between the consumer and producer to avoid cache line thrashing. While their design is reasonable for a general purpose queue, CAB is more suitable for use in concurrent dynamic analysis for two reasons. First, CAB's enqueueing code

is more efficient for handling a large number of events, such as those produced by dynamic instrumentation. Second, CAB's dequeueing operation spins only at the beginning of each chunk while FastForward dequeueing operates at a finer granularity, spinning on single events (i.e., one memory location). It thus synchronizes with the producer much more frequently than is necessary with CAB. We compare CAB to FastForward queueing and show that CAB improves performance by 41% on average, and up to 117%.

Shadow Profiling and SuperPin are profiling techniques that fork a shadow process, which runs concurrently with original application process [43, 52]. The shadow process executes instrumented code, while the original application runs uninstrumented. Currently, these approaches are limited to single-threaded applications, because implementations of fork on most thread libraries only fork from the current thread. Unlike our framework, the shadow processes cannot cover the whole program execution, because events around fork and unsafe operations may be lost.

Aftersight decouples profiling at the virtual machine layer using record and replay technology [14]. During one execution of the application, Aftersight uses VM recording to replay execution and then performs profiling on subsequent replayed executions. The profiling executions can be performed concurrently with the recording run, or offline at a later time. In our framework, the application and analysis are decoupled, but the application is executed only once and dynamic analysis is performed online.

Continuous profiling was the first to reduce the profiling overhead due

to micro-architectural side-effects such as cache misses [2]. Continuous profiling collects sampled profile events using hardware performance monitors and interrupts. On every 64k events, the processor raises the interrupt and collects the sample. By using per-processor hash-table, they significantly reduce both synchronization overhead and cache miss penalty, which allows profiling on deployed systems. Most recent processors, support this hardware event-based sampling, and top-notch system event profiling tools such as Intel VTune [15] uses similar event-based sampling. However, this technique is limited to sampling events that are supported by the hardware. Our framework supports both sampling and exhaustive collection, and events need not be determined by the hardware.

Recent work suggests hardware support for low-overhead dynamic analysis. For example, HeapMon uses an extra helper thread to decouple memory bug monitoring [48]. The idea of offloading the data to another thread is similar to our framework. However, HeapMon achieves low-overhead because of hardware buffering and instrumentation support. The hardware support is specifically for heap memory bugs. We achieve performance without any special hardware support, and we assume less about the type of analysis.

iWatcher leverages hardware assisted thread-level speculation to reduce the overhead of monitoring program locations [58]. The platform offers general debugging analysis, but low-overhead is only guaranteed with hardware support. Current multicore processors do not support thread-level speculation.

Our dynamic analysis framework supports both exhaustive and sam-

pling analysis of events. Prior work presented designs for low-overhead sampling of instrumentation [4, 5, 10, 31], where sampling logic is executed in the application thread to determine when a sample should be taken. These approaches are orthogonal and complimentary to our work; our framework could perform sampling in the application thread to reduce the amount of data sent to another core. However, our framework also enables a new methodology for sampling, where data is written into a buffer exhaustively and is then optionally consumed (sampled) by the analyzer thread(s). By enabling concurrent execution of the analyzer and the application thread, our technique is likely to outperform traditional sampling techniques when a higher sample rate is used and time in the analyzer increases. However, even with low sample rates this new approach can be beneficial because it moves the sampling logic off the fast path (out of the application thread) and into the analyzer thread. Thus, this new approach is likely to be beneficial if a profiler's communication cost between cores is less than the cost of the sampling logic. CAB reduces communication costs, making this form of sampling more viable.

2.2 Concurrent Dynamic Analysis Framework

Figure 2.2 shows our software architecture how it is intended to map to modern multicore hardware. Our dynamic analysis system includes an *event producer* (the instrumented application), an *event consumer* (an analyzer), and an *event handling mechanism*, which links the first two. The application and analyzer may be folded together to execute within the same thread, or

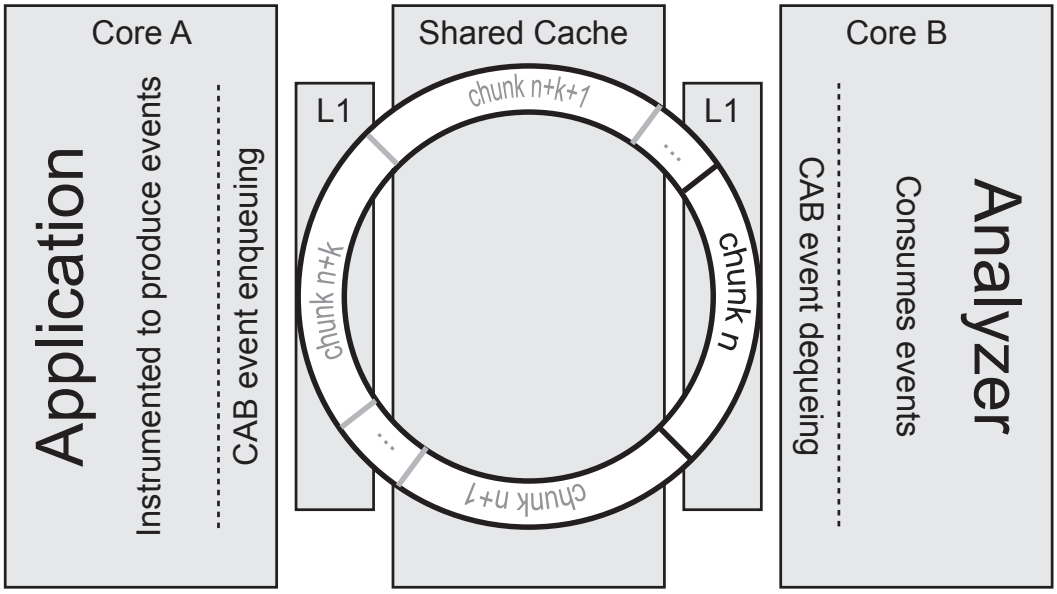


Figure 2.2: Cache-friendly Asymmetric Buffering (CAB) in a concurrent dynamic analysis framework.

they may be distinct, executing concurrently in separate threads as shown in Figure 2.1. We focus on the design and implementation of a generic event handling mechanism that supports concurrent dynamic analysis on multicore platforms. The goal of this framework is to exploit underutilized computational resources and fast on-chip communications to minimize the observed overhead of dynamic analysis.

Figure 2.2 also shows the CAB event handling mechanism. An application thread and a dynamic analyzer thread execute on separate cores. The application produces analysis events at injected instrumentation points, and CAB transfers the events to the analysis thread. Since CAB is generic and yet cache-friendly, the analysis writer is: a) freed from low-level micro-

architectural optimization concerns when offloading the event, and b) can implement the analysis logic independently of the application instrumentation.

By constructing a framework, many analyses may reuse the highly tuned mechanisms. The framework is flexible and general. It supports an *exhaustive mode* that collects and analyzes all events, and a *sampling mode*, in which the analysis samples a subset of the events.

2.2.1 CAB: Cache-friendly Asymmetric Buffering

CAB provides a communication channel between application and analysis threads. Two objectives guide the design of CAB: 1) minimizing application instrumentation overhead, and 2) minimizing producer-consumer communication overhead. We use three tactics to address these goals: a) we bias the design toward very low overhead enqueueing, b) we use lock-free synchronization, and c) we partition access to the ring buffer to avoid costly micro-architectural overheads due to cache contention.

At the center of CAB is a single-producer, single-consumer lock-free ring buffer, in which an application thread produces events and an analysis thread consumes them. Since each CAB has only one producer and consumer pair, we can optimize for fast, lock-free access to this shared buffer. Our approach is asymmetric. The application views the buffer as a continuous ring into which it enqueues individual events. By contrast, the analyzer views the buffer as a partitioned ring of fixed sized *chunks*, and each dequeue operation yields an entire chunk.

Although producer-consumer algorithms are well studied [22, 37, 41, 42], none of these approaches exploit asymmetry, nor do they consider memory system impact on algorithms. As we show, these CAB features are key to scalable performance on multicore.

2.2.1.1 Lock-free Synchronization

The special case where a communication buffer is shared by just a single producer and a single consumer has the distinct advantage of avoiding intra-producer and intra-consumer coordination, and is well-studied for general purpose concurrent queue implementations [21, 22, 37]. Specifically, the common case enqueue and dequeue operations can be implemented without locks, as wait-free operations [30]. Of course, the operations are not actually wait-free if the desired semantics require that the producer block on a full buffer and the consumer block on an empty buffer. However for dynamic analysis, the common case is high frequency enqueueing and dequeueing, so blocking is exceptional with a reasonably sized buffer. Although requiring CAB to be single producer, single consumer is restrictive, the simplicity and performance of the lock-free implementation it yields is attractive given the importance of minimizing perturbation of the application. However, this does not preclude building a multiple producer or consumer system on top of the lock-free CAB, as discussed in Section 2.4.

```
while (*bufptr != CLEAR) {
  if (*bufptr == MAGIC)
    bufptr = &buffer;   // wrap back to start
  if (*bufptr != CLEAR)
    block();            // busy, back off
}
*bufptr++ = data;       // enqueue data
```

(a) Enqueueing events in application code

```
block() {
  spin_wait();
  pollptr = SKIP(bufptr, CHUNK_SIZE * 2);
  while (*pollptr != CLEAR) {
    if (isInvokedGC())
      thread_yield();   // must cooperate
    else
      sleep(n);
  }
}
```

(b) Blocking the application

```
while (isApplicationRunning()) {
  /* keep distance of 2 chunks from producer */
  index = ((chunk_num + 2) * chunk_size)
      % buffer_size;
  while (buffer[index] == CLEAR)
    spin_or_sleep();
  /* consume & clear entire chunk */
  consume_chunk(chunk_num);
  chunk_num = next(chunk_num)
}
```

(c) Dequeueing events in analysis code

Figure 2.3: Enqueueing and dequeueing pseudo-code.

2.2.1.2 Queue Operations

CAB can be used for both exhaustive and sampled event collection. We start by describing queuing operations for exhaustive mode. In exhaustive

mode, every event is enqueued, dequeued, and analyzed.

Enqueueing The detailed design of CAB's enqueueing operation is guided by three goals: 1) the design should minimally perturb the application; 2) it needs to accommodate dynamically allocated and dynamically sized event buffers; and 3) if an enqueue operation causes an application thread to block, it must cooperate with the garbage collector and any other scheduling requirements to prevent deadlock.

To minimize perturbation of the application thread, the common case for enqueueing must be fast, and the injection of enqueueing operations should minimally inflate the total code size. Figure 2.3(a) shows the pseudocode for the enqueueing operation. The common case for enqueueing consists of just two lines (1 and 7). When there is space in the buffer, the test at line 1 evaluates to false and execution falls directly through to line 7. The exceptional case may occur either because the end of the buffer has been reached or because the buffer is full. These cases are dealt with by lines 3 and 5 respectively. If the buffer is full, the blocking code in Figure 2.3(b) is executed via a call. Note that all of this code is lock-free, and in the common case, just a single conditional branch is executed (line 1 of Figure 2.3(a)). As shown later in Figure 2.7(a) and Figure 2.7(b), the compiler or binary translator can push lines 2–6 out of the hot code block, keeping the code small and the length of the critical path short.

The control flow in the enqueueing operation depends only on `*bufptr`

and two constants: `CLEAR` and `MAGIC` (lines 1, 2 and 4 of Figure 2.3(a)). This design is very efficient while also supporting variable sized, dynamically allocated buffers. Dynamic allocation is essential since the number of buffers is established at run-time, and dynamic sizing is valuable since the system may respond to the particular requirements and resource constraints of a given application.

The idea is that the producer will only ever write into buffer fields which have been cleared by the consumer: the producer guards in line 1 of Figure 2.3(a), and the consumer sets the sentinel `CLEAR` when it consumes the chunk in line 8 of Figure 2.3(c). By using a special sentinel value (`MAGIC`) to mark the end of the ring buffer, a single test for `CLEAR` in line 1 will guard against both the end of the buffer being reached (line 2) and a full buffer (line 4). When the end of the buffer is reached, `bufptr` is reset to point to the start of the buffer, `&buffer` (line 3). The buffer address is only required in line 3, and is held in a variable. Furthermore, the code path has no explicit test against the buffer size or end of buffer, which is implicitly identified via the `MAGIC` marker. We can therefore dynamically allocate and size the buffer. This design requires that `CLEAR` and `MAGIC` are illegal values for analysis events. In practice, it is easy to choose `CLEAR` and `MAGIC` suitably to avoid imposing on the needs of the analyzer.

The exceptional case where the producer thread must block because the buffer is full (line 5) is handled out of line (Figure 2.3(b)). In general, when the producer thread blocks, it must remain preemptible, otherwise it could lead to

deadlock. Specifically, if the consumer invoked a garbage collection while the producer thread was blocked, and the producer thread were unpreemptible, deadlock would ensue. For this reason, the producer thread spins briefly (line 2 of Figure 2.3(b)) before re-testing whether the buffer is full (line 4) and yielding to GC (line 6) or sleeping (line 8). Note that the code checks the contents of `pollptr`, a point two chunks *ahead* of `bufptr` (`pollptr` is set in line 2). By doing this, we effectively back off the producer, giving the consumer time to work and ensuring that upon return there will be at least two chunks of free space available in the buffer.

Dequeueing The design of CAB's dequeueing operation is guided by two goals: 1) the design should minimize producer-consumer communication overhead, and 2) similar to enqueueing, it needs to accommodate dynamically allocated and sized buffers. We address the second goal by avoiding any static reference to the buffer address or buffer size, as we described above for enqueueing. To meet the first goal, the analysis thread synchronizes at a coarse grain by consuming a large number of events at once (i.e., a chunk). Furthermore, the design does not induce unnecessary cache coherence traffic on shared or private caches, because CAB never accesses the chunk into which the producer is writing.

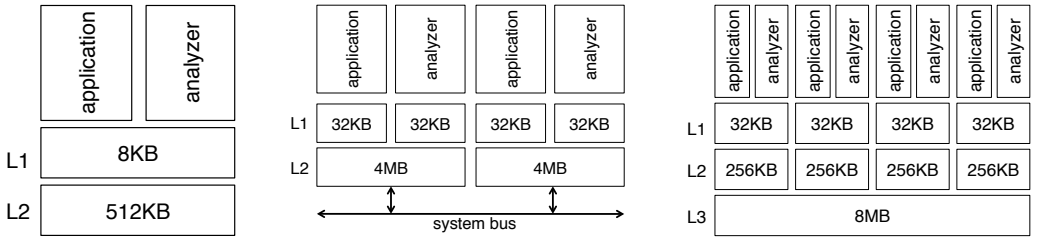
CAB prevents the producer and consumer from accessing the same cache lines at once by logically partitioning the ring buffer into large fixed-size chunks, and then ensuring that the consumer remains at least one complete

chunk behind the producer (line 5 of Figure 2.3(c)). The size of a chunk is a dynamically configurable option (`chunk_size` in Figure 2.3(c)). Recall that the producer is largely oblivious to this partitioning of the ring buffer; it enqueues events regardless of chunk boundaries. However, if the buffer becomes full, the producer waits until there are at least two empty chunks available to it (lines 3 and 4 of Figure 2.3(b)).

In this design, the consumer minimizes overhead and synchronization by dequeueing and processing one chunk at a time (line 8 of Figure 2.3(c)), reducing spinning and checking without affecting the fine-grained producer activity. The analysis happens in the call to `consume_chunk()` at line 8. If the analyzer itself is multi-threaded, it may dispatch analysis events to multiple threads. The analyzer clears the buffer immediately after it processes each event as part of `consume_chunk()`. Clearing is essential, since it communicates to the producer that the buffer is available (line 1 of Figure 2.3(a)). Clearing immediately after processing each event maximizes temporal locality. In the special case when the producer terminates, it is usually desirable for the consumer to process the remaining entries. Since the consumer normally may not read from the same chunk as the producer, we include in our API the facility for the producer to explicitly flush residual events to the consumer.

2.2.1.3 Optimizing CAB For Multicore Processors

We tune CAB's chunk-based ring buffer design to reduce microarchitectural side-effects due to producer-consumer contention. However, we make



(a) Pentium 4 with Hyper-threading (b) Core 2 Quad (c) Core i7

Figure 2.4: Experimental processors data cache structure. Instruction or trace cache is omitted. Application and analyzer's mapping to the cores is idealized, and it is not a requirement.

only minimal assumptions about the multicore architecture. We assume that the hardware can execute multiple software threads simultaneously on separate cores or on the same core. We do not require any specific cache hierarchy. The design works for both private and shared cache designs, but benefits from shared lower-level caches. For example, Figure 2.4 shows three Intel hardware generations, which comprise our experimental platforms. (Section 2.6 has more details on each.) These designs are quite different, yet CAB works well with all of them.

The CAB design ensures that 1) the producer and consumer never access the same cache line simultaneously, 2) the producer and consumer can exploit a shared cache, and 3) the producer and consumer exhibit spatial locality that is amenable to hardware prefetching. The first two design goals avoid cache thrashing, the second also minimizes memory latency, and the third seeks to hide cache miss penalties.

To avoid cache thrashing when the producer and consumer do not share an L1 cache, the chunk size should be large enough that by the time the producer is writing to chunk $n + 2$, chunk n has been fully evicted from the producer's L1 cache. If we assume a strict LRU cache replacement policy, this criteria is satisfied with a chunk size that is greater than or equal to the L1 cache size. In practice, cache replacement policies are not always strict LRU. Thus a larger chunk size is better. Furthermore, since producer-consumer synchronization occur on chunk boundaries, smaller chunks are generally more expensive. Thus, when the producer and consumer share an L1 cache, the synchronization overhead of small chunks still outweighs any locality advantage, which is why large chunks are effective on shared L1 caches as well. This design easily generalizes for more levels of private cache. Our evaluation uses a chunk size of four times the L1 size.

If the runtime uses native threads, we control producer-consumer affinity via the POSIX `sched_setaffinity()` API. On the other hand, if the runtime employs a user-level scheduler, we may require modest changes to the scheduler (see Section 2.5.1). We do not require special operating system support or modifications to the operating system's scheduler.

By using a ring buffer, the producer and consumer's memory operations are almost strictly sequential (except when the ring buffer infrequently wraps around). It is hard to test directly the hypothesis that CAB addresses our locality objective, but we measured L1 and L2 miss rates and found that they were not correlated with buffering overhead when we varied the buffer size

on both shared and private L1 cache architectures. We also experimented with special Intel non-temporal memory operations but found they degraded performance compared with our straightforward sequential baseline. Worse, the current Intel implementations of non-temporal store operations bypass the entire cache hierarchy, forcing the consumer to go to memory rather than the shared last level cache. CAB would benefit from previously proposed hardware instructions, such as the *evict-me*, or some other mechanism that marks cache lines LRU [38, 53]. CAB could then reduce its cache footprint and thus its influence on the application, while still benefiting from sharing.

2.2.2 Sampling

If the analysis thread is unable to keep up with the application, the buffer will eventually fill up and the application thread will block (line 5 of Figure 2.3(a)). Depending on the analysis, this application slowdown may be unavoidable. For example, security analyses and cache simulation profilers typically require fully accurate traces. Other analyses, such as those designed for performance analysis, often tolerate reduced accuracy to gain reduced overhead. In such cases, the profilers in CAB may sample to prevent the application from blocking.

In our sampling framework, the producer still enqueues all the events and then the consumer samples the buffer, analyzing only a subset of the recorded data, skipping over the rest. Other sampling designs, such as timer-based sampling [5], reduce the number of events. However, client analyses that

are control-flow-sensitive (e.g., path profiling) and context-sensitive analyses (e.g., call trees), must still insert pervasive instrumentation and maintain their state even if the instrumentation does not store the events. In contrast, our sampling framework eases the burden of implementing these more advanced forms of sampling because the sample decisions are made in the analysis thread; the logic is off the fast path of the application thread so it can be written in a high-level language (rather than inlined into compiled code) and with less concern over efficiency.

Enqueueing In sampling mode, the producer *never* checks whether the buffer is full. If the consumer cannot keep up with the producer, the producer simply continues writing to the buffer and data is lost. This design obviously trades accuracy for performance. Figure 2.5(a) shows pseudocode for the application thread when in sampling mode, and should be compared to Figure 2.3(a). The code consists of the minimal instructions required to insert an element into a CAB buffer.

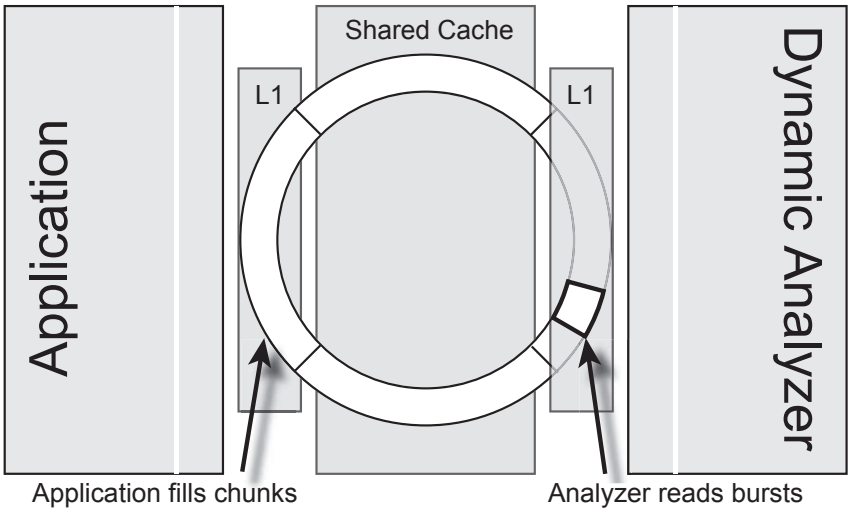
Dequeueing Figure 2.5(b) shows pseudocode for dequeueing in sampling mode. Compared to exhaustive mode dequeueing (Figure 2.3(c)), there are two differences. First, each chunk is sampled, according to the value of `sampling_rate` (line 8). Second, only the first element in each chunk is cleared (line 10), rather than the entire chunk. We now describe these points in more detail.

```
1  if (*bufptr == MAGIC) // end buffer
2    bufptr = &buffer;
3  *bufptr++ = data;
```

(a) Enqueueing pseudo-code for sampling mode.

```
1  while (isApplicationRunning()) {
2    /* keep distance of 2 chunks from producer */
3    index = ((chunk_num + 2) * chunk_size)
4        % buffer_size;
5    while (buffer[index] == CLEAR)
6      spin_or_sleep();
7    /* analyze some fraction of the chunk */
8    sample_chunk(chunk_num, sampling_rate);
9    /* clear only the first entry */
10   buffer[chunk_num*chunk_size] = CLEAR;
11   chunk_num = next(chunk_num)
12 }
```

(b) Dequeueing pseudo-code for sampling mode.



(c) Bursty Sampling. The analyzer samples a burst from each chunk.

Figure 2.5: Sampling mode. The application does not block and the profiler may use bursty sampling.

Consumers read events in bursts to maximize cache locality, as shown in Figure 2.5(c). The size of the burst is arbitrary within the scope of a chunk. However, L1 cache performance is likely to benefit when the burst is cache line-aligned. Our evaluation shows that sampling accuracy is maximized when we keep the sample rate sufficiently low such that the consumer keeps up with the producer, which avoids the producer overwriting data before it can be sampled.

The consumer does not need to clear every element in the chunk after it is read (line 10 of Figure 2.5(b)), because the producer is no longer checking for `CLEAR`. The first entry of each chunk still needs to be cleared by the consumer to allow it to observe when a chunk has been refilled, and thereby avoid re-processing old data.

Note that because the application thread logic no longer checks for a full buffer, the application thread may *catch up* and overwrite a chunk that the analysis is sampling, causing accuracy to drop. An alternate design could make the producer skip over a chunk if the analyzer is still working on it. We instead keep the sampling rate low and use simple chunk logic. Our evaluation shows that we can achieve high accuracy with a very low sampling rate.

To reduce memory bandwidth requirements, even with exhaustive event recording, a more sophisticated buffer assignment could reuse a chunk when the consumer is sampling another chunk. It could also bias its choice to a chunk that is still likely to be resident in cache. This design would pay for the reduced memory bandwidth with increased producer-consumer synchronization.

2.2.3 Interaction with the Garbage Collector

The enqueueing operation in sampling mode is wait-free, i.e., it is guaranteed to finish in a finite number of instructions, but exhaustive mode enqueuing may block the application. When the garbage collector is triggered, the application must yield to the garbage collector thread. Otherwise, deadlock may happen because analysis thread may have yield to the GC and would not process the buffer. Therefore, slow path in Figure 2.3(b) checks the garbage collector state.

To transfer the program control to the garbage collector, it must be a GC safe point where all the heap objects are stored in memory. However, not all instrumentation locations are GC safe points, and having a `block()` call is unsafe. To prevent unsafe operations, instrumentation at unsafe points include the sampling mode enqueueing, which must check the buffer space availability at each prologue and loop backedge yieldpoint. The interval of these yieldpoint is finite, so it is always possible to keep the buffer from overrunning with the sampling mode enqueueing in the exhaustive mode.

2.3 A Model For Analysis Overhead

The performance benefit of offloading dynamic analysis work in a separate thread depends on a number of factors, such as the amount of time spent in the application versus the analysis code, and the amount of data the application must transfer to another core for processing. If the amount of time spent transferring data far exceeds the time spent processing that data, the

concurrent analysis is unlikely to show significant benefit.

This section describes a basic cost model for overheads in concurrent and single-threaded dynamic analysis systems. The model provides a detailed look at which performance characteristics determine the success of a concurrent implementation, and thus help identify the types of analysis for which a concurrent implementation is beneficial.

The model presented below compares a single-threaded scenario, where the application and analysis execute in the same thread, to a concurrent scenario, where the application and analysis execute in separate threads and communicate through shared memory. We start with the following definitions:

A	Isolated application execution time.
P	Isolated analyzer execution time.
E_s	Execution time with instrumentation and analysis inline in the **s**ame thread as the application.
E_c	Execution time with a **c**oncurrent analyzer.

and a simple model of overheads:

A_i	Application overhead due to **i**nstrumentation to produce events.
I_{ap}	Interference overhead due to **a**pplication and analyzer (**p**rofiler) running in same thread *(when single-threaded).*
A_q	Application thread overhead due to **q**ueueing *(when concurrent).*
P_c	Analyzer thread overhead due to **c**ommunication and dequeueing *(when concurrent)*

The event instrumentation overhead A_i is idealized, since in practice it is hard to isolate the cost of instrumentation for extracting events from the surrounding code which processes those events. In a single threaded system, I_{ap} is the *indirect* overhead due to resource contention between the application and the analyzer sharing common hardware. The effects will depend on the nature of P and may include memory contention, cache displacement, register pressure, etc. We define E_s, the cost of single-threaded analyzer as:

$$E_s = A + A_i + P + I_{ap} \tag{2.1}$$

In a concurrent system, the queuing overhead A_q reflects time spent by the application enqueueing items and blocking on communication to the analyzer, plus the indirect effect enqueueing has of displacing the application's cache. P_c reflects the cost to the analyzer of dequeueing events, which includes the communication overhead of loading data from a shared cache. A_i, A_q and P_c are each a function of the *event rate*: the rate at which the application generates analysis events. To define the cost of concurrent analysis, E_c, we start with the cost of each of the two threads, E_c^A and E_c^P, and consider each thread separately with the assumption that the given thread is dominant (i.e. it never waits for the other). When the application dominates:

$$E_c^A = A + A_i + A_q \tag{2.2}$$

and when the analyzer dominates:

$$E_c^P = P + P_c \tag{2.3}$$

Since the application E_c^A and the analyzer E_c^P are concurrent, one may dominate the other. For simplicity, we assume that for a given analyzer, *either* the application *or* analyzer will uniformly dominate. In practice, the application and analyzer may exhibit phased behavior, but event bursts should be somewhat smoothed by buffering. In any case, the simplification helps illuminate the nature of the problem. Under these assumptions, execution time for concurrent analysis can be defined as:

$$E_c = max(E_c^A, E_c^P) \tag{2.4}$$

We now discuss the conditions that make concurrent analysis worthwhile, looking at the two cases separately when either the application or analyzer dominates.

Application Thread Dominates. The application dominates when $E_c^A \geq E_c^P$, i.e., the application takes longer than the analysis:

$$A + A_i + A_q \geq P + P_c \tag{2.5}$$

For concurrent analysis to improve performance in this scenario, it must maximize $E_s - E_c^A$:

$$E_s - E_c^A = P + I_{ap} - A_q \quad \geq \quad 0 \tag{2.6}$$

$$P + I_{ap} \quad \geq \quad A_q \tag{2.7}$$

Concurrent analysis will improve performance as long as the application queuing costs, A_q, are small relative to analysis costs, $P + I_{ap}$. As we show in Section 2.6.2, A_q is typically small. I_{ap} is a function of P, and thus a very lightweight analyzer, where $P + I_{ap}$ is smaller than A_q, will not benefit from concurrent analysis. We show this case holds for method counting, but in all the other cases we tested, which includes the only slightly more expensive call graph construction, the cost of $P + I_{ap}$ is greater than A_q, and the framework provides performance benefits. However, because the benefit $E_s - E_c^A = I_{ap} + P - A_q$, and from Equation 2.5,

$$P - A_q \leq A + A_i - P_c \tag{2.8}$$

in the case the application dominates, the benefit of concurrent analysis is limited.

Profiler Thread Dominates. In scenario 2, where $E_c^A < E_c^P$, waiting for the analyzer becomes the bottleneck. For a concurrent analyzer to improve performance, it must maximize: $E_s - E_c^P$:

$$E_s - E_c^P = A + A_i + I_{ap} - P_c \;\geq\; 0 \tag{2.9}$$

$$A + A_i + I_{ap} \;\geq\; P_c \tag{2.10}$$

Concurrent analysis will improve performance as long as the communication cost is small relative to the application thread and associated overhead (A+A_i+ I_{ap}). Note that once the analyzer dominates, the performance improvement,

$E_s - E_c^P$, is independent of the analyzer's execution time, P. The speedup of the concurrent analyzer is determined by the analyzer's buffering and communication costs P_c; an analyzer with higher cost P does not provide more incentive for a concurrent implementation.

Extensions and Lessons. We can draw a number of lessons from the analysis above. When the application thread dominates and $E_c^A \geq E_c^P$, the performance improvement from concurrent analysis is limited by the single-threaded analyzer cost $P + I_{ap}$. When the analyzer thread dominates, the performance improvement from concurrent analysis is limited by time spent in the application thread and associated overhead ($A + A_i + I_{ap}$).

In all cases, communication performance (A_q and P_c) is key because it determines whether the theoretical improvements of concurrent analysis can be realized in practice. The goal of CAB is to reduce these communication costs as far as possible, thus allowing concurrent analysis to be effective for a wider class of analyzers than is possible today.

The model assumes that the analysis executes concurrently with the application, but is not itself parallelized (subdivided into multiple worker threads). Once an analysis adopts a concurrent model, parallelizing the analysis becomes relatively easier and has the potential to significantly improve performance when analysis time dominates application time. We do not investigate parallelizing the analyzers themselves here because this process is highly dependent on the particular analyses. Instead, we focus on minimizing

communication overhead as part of a general framework.

2.4 Threading Model Impact on Design and Implementation

CAB's lock-free design is predicated on each CAB having a single producer and a single consumer. This design allows for fast, low overhead queueing, but has a number of consequences, which we discuss in detail now.

2.4.1 N:M threading model with per-processor CAB

Maintaining the single producer property implies allocating one CAB for each application thread. When application threads are mapped directly to kernel threads ("1:1 threading"), we allocate the CAB in thread-local storage. For some user-level thread models ("N:M threads" also called "green threads"), true concurrency only exists at the level of underlying kernel threads, so we allocate one CAB per kernel thread and multiplex it among user threads. With this model, user threads time-share CABs and may migrate from CAB to CAB according to the user-level scheduler, but in all cases, there is only one user thread mapped to a CAB at any given time. With multiplexing, events from different producer threads will be interleaved. Since some analyses are context-sensitive, the producer must add special events which communicate thread switches to the consumer, and the consumer must de-multiplex the interleaved events to regain the context that would otherwise be lost.

Our design explicitly supports dynamic sizing of CABs, which should

be sized according to the rate of event production and the available memory. Presently we configure CAB sizes via the command line. We leave to future work extending the framework to adaptively size each CAB based on its usage characteristics at run-time. The framework could use small buffers for threads that produce very few events and larger ones for prolific threads. Thus the total space requirements for all CABs in a system would scale with the total event production rate in the system, rather than the absolute number of threads.

The requirement of a single consumer per CAB does not preclude either a single consumer thread from servicing multiple CABs, or the consumer thread from dispatching analysis work to multiple threads. In a setting with a low event rate and lightweight analysis, a single analysis thread may be able to service all CABs, processing them in a round-robin fashion. By contrast, in a setting where analysis is very heavyweight and the analysis is conducive to parallelization, multiple threads can perform the analysis. However both scenarios observe the requirement that a given CAB is only ever accessed by one consumer thread, satisfying the precondition of our lock-free implementation.

2.4.2 Native threading model with per-thread CAB

In the N:M threading model, allocating one CAB on each kernel thread is feasible because the user-level scheduler guarantees that each kernel thread only runs at most one user thread at a time. However, the native threading model does not have a user-level scheduler, and the operating systems scheduler is solely responsible for the thread scheduling. Without changing the

design, CAB would be placed on each thread's local storage, which makes it a per-thread buffer, which is the default implementation for native threading model. However, per-thread buffering does not scale with the hardware when the number of threads are far greater than the number of cores. We next provide a solution to this problem.

2.4.3 Native threading model with per-processor CAB

To improve the locality and scalability in memory usage, we introduce a new design that multiplexes CAB similar to N:M threading model, and improves the affinity of each buffer to a single core.

Enqueue Figure 2.6(a)–2.6(c) shows the pseudo-code for a per-processor CAB. We initialize the beginning of each chunk to `MAGIC` upon buffer creation. Unlike the above models, the application operates at chunk granularity. Each application thread requests a chunk from the buffer associated with its current core. Once an application thread obtains a chunk, it is guaranteed that it owns the chunk until it is full. The application detects if it fills the chunk when it reaches `MAGIC`. When the chunk is filled, the application must notify the analyzer by putting the chunk into the *history queue* and then obtain a new chunk. Explicit synchronization is inevitable on obtaining a chunk. However, this operation happens in the slow path, which occurs once in the whole chunk write. Moreover, since the lock is a per-processor lock, synchronization on obtaining the buffer will not be commonly contended. Upon obtaining

```
1  while (*bufptr != CLEAR) {
2    if (*bufptr == MAGIC)
3      bufptr = obtain_chunk();
4  }
5  *bufptr++ = data;        // enqueue data
```

(a) Enqueueing events in application code

```
1  obtain_chunk() {
2    buffer = buffers[current_cpu]
3    if (buffer.full()) // deadlock prevention
4      buffer[current_cpu] = realloc(current_size * factor);
5    synchronized {
6      current_cpu.history.put(chunk);
7      chunk = current_cpu.getChunk();
8      chunk[1] = thread_id;
9      return &chunk[2];
10   }
11 }
```

(b) Obtaining chunk

```
1  while (isApplicationRunning()) {
2    /* keep distance of 2 chunks from producer */
3    foreach CAB in assigned_buffer {
4      chunk = CAB.history.get()
5      consume_chunk(chunk_num);
6    }
7  }
```

(c) Dequeueing events in analysis code

Figure 2.6: Enqueueing and dequeueing pseudo-code for per-processor buffering on native threading model.

a new chunk, the application thread writes its thread id next to `MAGIC` to communicate its context to the analysis thread for context sensitive analysis.

In this design, positioning the buffer in the hardware caches is difficult, but our per-processor buffering design keeps the data produced on its core's buffer as much as possible by assigning chunks from the current core's buffer.

Furthermore, each application thread will write sequentially until the chunk is full. Therefore, we expect that spacial locality is enhanced while keeping temporal locality as same as per-thread buffering, and our evaluation supports that the amount of overhead is similar or better than per-thread buffering on native and N:M threading model.

Dequeue Each analysis thread operates one buffer at a time. Each analysis thread owns a set of buffers to avoid unnecessary synchronization among analysis threads. Figure 2.6(c) shows the pseudo-code for dequeueing. Since the application may finish chunks out-of-order, the analysis thread must search a full chunk. In order to avoid linear scan on the buffers, the analysis thread processes chunks from the history queue. Chunks in the history queue are added by each application thread when its chunk is full. This queue also needs synchronization, and we choose Lamport's lock-free single-producer single-consumer queue [37] for simplicity.

Deadlock Prevention When a thread tries to obtain a chunk, no chunk may be available if all the chunks in the buffer are assigned to other threads. Simple approaches, such as migrating the thread to another core or voluntarily yielding the thread until any chunk is available, do not solve the problem and deadlock could occur. For example, all other threads may algorithmically depend on a new thread blocked until the new thread finishes running, while the new thread cannot proceed because there is no available chunk. To prevent

and escape from the deadlock, we dynamically increase the size of the buffer when there is no chunk available. To simplify checking chunk availability, we only consider the current core's buffer. Therefore, even when there are available chunks on other core's buffer, the buffer size will still increase if the local core's buffer runs out of chunks.

2.5 Implementation

We next discuss implementation details that are specific to our particular environment, and then we briefly describe each of the five dynamic analyses that we implemented in our framework.

2.5.1 Platform-Specific Implementation Details

We implemented our framework in Jikes RVM [1]. Jikes RVM is an open source high performance Java Virtual Machine (VM) written almost entirely in a slightly extended Java. This setting affected our implementation only in that we needed to take care to ensure the enqueueing operations avoid locking out the garbage collector.

We implement our framework using two threading models: N:M and native. While we were developing this concurrent analysis framework, researchers changed from N:M threads implemented in Jikes RVM 2.9.2 to native threads implemented in Jikes RVM 3.1.0. This transition was imposed upon us, but it serves as an opportunity to demonstrate the generality of CAB with respect to fundamentally different threading models.

Native threads improve average performance over N:M threads in the base system. Jikes RVM version 3.1.0 however improves over 2.9.2 in many other ways as well, which makes it difficult to compare their performance directly. For example, biased locking has reduced thread synchronization overhead, the Immix garbage collector improves locality and garbage collection times [9], and the compiler generates better code.

We implemented our framework in Jikes RVM 2.9.2 with N:M threads and then ported it to native threads in Jikes RVM 3.1.0. Except for the changes in how we map the analysis thread and the user threads, which we describe below, our instrumenting and analysis code remained the same. We have not yet however ported our *experimental infrastructure*, which teases apart the different overheads and reports cache behaviors to explain our results. We thus report overall performance results for all the client analyses for both N:M and native threading models, but a detailed breakdown analysis is presented for N:M threads only. The trends are the same for both models.

N:M Threading Version 2.9.2 of Jikes RVM uses an N:M threading model (also known as *"green threads"*), which multiplexes N user-level threads onto M *virtual processors* via a simple timer-based scheduler that the system triggers at yield points in the application. The Jikes RVM compilers inject yield points in method prologues, epilogues, and control-flow back edges. Each of the M virtual processors maps directly to a single native thread that the operating system manages. Jikes RVM chooses M to match the number of available

hardware threads. Jikes RVM uses a `Processor` data structure for per-virtual-processor state.

Since true concurrency only exists among the M virtual processors, we implemented CABs at this level, associating one CAB with each `Processor`. Many user threads may share a given virtual processor, but only one thread can ever be executing on a virtual processor at any time. The scheduler may migrate user threads among virtual processors as it schedules them. We modified Jikes RVM's scheduler to: a) prescribe the affinity between virtual processors and the underlying hardware, b) prevent the migration of application threads onto analysis virtual processors, and c) record thread scheduling events in the CAB. We use the first two modifications to schedule producer and consumer threads in pairs on distinct cores with a common last level cache. The producer uses the third modification to inform the consumer of the changing affinity between producer threads and CABs.

Native Threads with Per-thread Buffering Version 3.1.0 of Jikes RVM uses a native threading model, which maps each user and VM thread onto one operating system thread (also known as a "pthread"). Jikes RVM does not control the thread scheduling. It instead relies on the operating system scheduler. Timer-based sampling may still trigger thread yield points, as in N:M threads. The OS may migrate the user thread to different cores transparently to Jikes RVM. Therefore, this implementation does not control the affinity between the user and analysis threads.

Unlike the N:M implementation, with native threads the framework takes the number of analysis threads as a parameter. The framework assigns each user thread a thread-local CAB buffer and an analysis thread. When the user thread terminates, the framework processes any remaining chunks by moving them to a pending buffer queue on the associated analysis thread for processing. We assume that thread creation and termination is infrequent and thus synchronize accesses to the pending queue.

Native Threads with Per-processor Buffering Similar to per-thread buffering, we use version 3.1.0 of Jikes RVM. Instead of putting CAB buffers in thread local memory, we set the number of CAB buffers to the number of cores, and access them globally. Unlike other implementations, CAB buffer memory is allocated out-side the VM space, i.e., buffers do not occupy the VM heap space. By taking the buffer memory out of VM space, the whole heap space can be used by the application, and the frequency of garbage collection invocation is less influenced by using our framework. We believe this is a realistic configuration because production JVMs implement VM components including dynamic analyses outside the VM space. We currently only have call graph profiling for this implementation.

Instrumentation and Enqueueing Each dynamic analysis in our implementation has four parts: 1) program instrumentation that produces an event, 2) enqueueing operations, 3) dequeueing operations, and 4) analysis. Parts 1)

```
1     mov  eax $BUFPTR[esi]
2     cmp  [eax], CLEAR
3     jne  B
4  A: mov  [eax], $DATA
5     add  eax, 4
6     mov  $BUFPTR[esi], eax
```

(a) Precise Mode (Fast Path).

```
1  B: cmp  [eax], MAGIC
2     jne  C
3     mov  eax, $BUFADDR[esi]
4     cmp  [eax], CLEAR
5     jeq  A
6  C: call block()
7     jmp  A
```

(b) Precise Mode (Slow Path).

```
1     mov  eax $BUFPTR[esi]
2     cmp  [eax], MAGIC
3     jne  A
4     mov  eax, $BUFADDR[esi]
5  A: mov  [eax], $DATA
6     add  eax, 4
7     mov  $BUFPTR[esi], eax
```

(c) Sampling Mode (Slow path is just line 4).

Figure 2.7: x86 assembly code for CAB enqueueing operations. The `esi` register is used as a base register for the `Processor` object in Jikes RVM, and `eax` is a register allocated by the compiler.

and 4) are analysis-specific and are described below in Section 2.5.2. We use a straightforward implementation of part 3) from the design section. The remainder of this section presents further details on our enqueueing implementation.

Figure 2.7 shows x86 assembly code for enqueueing in both exhaustive and sampling modes. Note the simplicity of the common case code (Fig-

ures 2.7(a) and 2.7(c)). In sampling mode, the slow path comprises just a single instruction (line 4). The rest of this section describes how we implement the call to `block()` (line 6 of Figure 2.7(b)) to avoid locking out other threads and to avoid introducing unsafe thread switches.

In exhaustive mode, the producer may block while the consumer catches up. It is essential that this blocking exhibits correct scheduling behavior and does not: a) lock out other threads, or b) allow garbage collection to occur at unsafe points.

It is only correct (i.e., *safe*) to perform garbage collection when the runtime system can correctly enumerate all the pointer references into the heap. These references reside in the statics, registers, and stack locations. To reduce the burden on the compiler, which generates this enumeration, a *garbage collection (GC) safe point* is typically a subset of all possible instructions in the program. In Jikes RVM, each method call, method return, loop back edge, allocation, and potentially exception generating instruction is a GC safe point. The Jikes RVM compiler guarantees that all compiled code will reach reach a GC-safe point in a timely manner by injecting conditional yield points on each loop back edge and method prologue, and producing a map for enumerating references at each one of these points.

Thus if a producer does not yield to GC when blocked, the garbage collector will not be able to proceed, and then all application threads will block waiting for GC the next time they try to allocate a new object, leading to deadlock. For this reason, our producer, rather than simply spinning, calls

the `block()` method (line 6, Figure 2.7(b)), which explicitly checks whether a GC yield is necessary (line 5, Figure 2.3(b)). This protocol ensures that a blocked producer does not lock out other threads.

However, if an analysis requires instrumentation on an instruction that is not a GC safe point, it may block at a GC unsafe point. For example, cache simulation requires instrumentation at every load and store, which are not, in general, GC safe points. We address this problem by using a non-blocking enqueue (Figure 2.7(c)) for instrumentation at non-GC safe points. We add checks at each loop back edge and method prologue to ensure there is sufficient memory in the buffer before the next GC safe point for any potential enqueues. Our current implementation uses the generous heuristic of ensuring that there is a full chunk available at each check. While this heuristic works very well in practice, it cannot guarantee correctness since it does not count the maximum number of potential enqueues. Although almost certainly straight line code will produce many fewer analysis events than an entire chunk can hold, an industrial strength solution would not be particularly difficult to engineer; the compiler could estimate the number of potential enqueues and compare with the buffer size, or guarantee a fixed limit by enforcing a maximum on the length of non-GC-safe paths by injecting occasional safe-points (at the expense of generating the appropriate GC maps). However, we leave such an implementation to future work.

2.5.2 Dynamic Analyses

To evaluate our concurrent dynamic analysis framework, we prototyped five popular profiling algorithms taken from the literature: method counting, call graph profiling, call tree profiling, path profiling, and a cache simulator. In each case, we instrument the application and implement the event processing logic, and bind the two with our dynamic analysis framework providing the event handling glue. All instrumentation is performed after inlining, so inlined method calls are not instrumented. Path profiling produces 64 bit event records, whereas the other clients produce 32 bit event records. In each case, we implement a sequential and concurrent version of the analysis for comparison.

Method counting On entry to each method, the method counting instrumentation writes a 32 bit method identifier into the event buffer. The analysis uses an array of method indexes initialized to zero. For each entry in the event buffer, the profile thread reads the entry and increments the corresponding method counter. The single-threaded version simply increments the appropriate element in the array upon each method entry.

Call graph profiling On entry to each method, call graph profiling instrumentation produces a 32 bit profile event which includes the current method and its caller. To identify the caller, the instrumentation must walk up the stalk, which requires three memory loads in Jikes RVM. We are able to pack

both the caller and callee into 32 bits since 16 bits is sufficient to identify all methods in our programs (as well as many larger ones). Thus, the profile event rate is exactly the same as for method counting. The analysis reads the events, computes a hash, and increments the corresponding hash table entry indexed by the event. The single-threaded implementation performs a hash table look-up and increment on each method entry.

Call tree profiling Call tree profiling is a dynamic analysis tool to classify a user program's behavior for automated support [28]. It summarizes a subtree of depth two in the dynamic call tree to represent the software execution. To reduce the overhead of call tree profiling, Ha et al. used a bit vector on the stack to mark the set of the calls. Unlike their implementation, we construct the dynamic call tree on the analysis threads using the trace of method calls sent over the CAB to capture the subtree pattern. The instrumentation is exactly the same as call graph profiling, which is necessary to construct the dynamic call tree.

This design is an interesting use of the concurrent dynamic analysis, because it makes the heavy-weight optimization for the instrumentation unnecessary, it simplifies the implementation the analysis code, and yet it achieves good performance.

Path profiling We inject full path profiling instrumentation into the application [6]. Path profiling assigns a unique number to all possible acyclic

paths through a method and stores each executed path during execution. We adapt Bond and McKinley's basic implementation from the Jikes RVM research archive [10]. This version does not include optimizations that: a) eliminate increments to the path register on some edges and b) use arrays instead of hash tables when methods are small, which is much more efficient. While a production implementation would include these optimizations, they are not key to our evaluation.

The application instrumentation for path profiling is the most invasive of all the clients. On entry to each method, the path profiling instrumentation clears the path register. On each branch, it increments the path register by some value. On each back-edge and method exit, the instrumentation stores the path number in the profile event buffer and resets it to zero. Path profiling uses a 64 bit record since the path numbers are often larger than 32 bits in Java [10]. For each path number, the profile thread computes a hash and increments the corresponding entry in the hash table. The single-threaded version performs this same work, but on each back-edge and method exit.

Thus, path profiling is more invasive, produces more and larger entries, and uses a larger hash table to store its entries compared to method counting, call graph profiling, and call tree profiling. Prior work finds, and our results confirm, that sequential exhaustive path profiling can add overheads ranging from 20% to over 100% of execution time.

Cache simulation We also implement a set associative cache simulator, which is similar to `dcache` implemented in Pin [39]. For each load and store instruction, the application instrumentation writes the 32 bit address into the buffer, using the low order bit to indicate whether the operation was load or store. Because these addresses are already in registers, this instrumentation, while prolific, is cheap. It does have a very high event rate, and therefore stresses the communication mechanisms in our framework. The analysis thread consumes each entry, computing a new state for the cache. It stores cache state in arrays. For our experiments, the cache simulator models a 32KB 4-way set associative L1 and a 512KB 8-way set associative L2. The L1 and L2 have a line size of 64B, are inclusive and have an LRU policy. In the single-threaded implementation, the analysis code calls out to a routine that updates the cache state at every load and store. Fully accurate cache simulation is expensive; it adds overheads ranging from 200 to 4500% to application time.

These five clients thus insert a wide range of types of instrumentation and perform light to heavy weight analysis.

2.6 Evaluation

This section describes our benchmarks, hardware, operating system, experimental design, and data analysis methodology. Section 2.6.1 compares CAB to other buffering mechanisms. Section 2.6.2 presents experimental results for our five analyses in our concurrent framework and compares them to

a sequential implementation that does *not* write event data to a buffer. Section 2.6.3 evaluates buffer size scalability and Section 2.6.4 evaluates sampling mode. Finally, Section 2.6.5 explores the importance of modern shared cache architectures to concurrent dynamic analysis.

Benchmarks. We use the SPECjvm98 benchmarks [49] and 9 of 11 DaCapo Java (v. 2006-10-MR2) benchmarks [7]. The DaCapo suite is a recently developed suite of substantial real-world open source Java applications. The characteristics of both are described elsewhere [8]. We focus on the multi-threaded benchmarks: `mtrt` in SPECjvm98 and `hsqldb`, `lusearch`, and `xalan` in DaCapo, but consider all of them. Due to a problem in our cache simulation implementation, we omit `xalan` on the Core 2, and `luindex`, `lusearch`, and `xalan` on the P4 in the cache simulation results. We omit `chart` and `eclipse` from DaCapo in all our results because the older version of Jikes RVM we use does not always run them correctly.

Hardware and Operating System. We evaluated the framework on three generations of Intel processors depicted in Figure 2.4. The Intel Pentium 4 has a single core with 2 hardware threads that share an 8KB data cache, and a 12kuops trace cache. The Intel Core 2 Quad has 4 cores on two dies, each core has 8-way 32KB L1 data and instruction caches. The pair of cores on each die share a 4MB 16-way L2 cache, for a total of 8MB of L2 cache. The Intel Core i7 has 4 cores, each of which has 2 simultaneous multi-threading (SMT)

threads, a private 32KB L1, and 256KB L2 cache. All of the cores share a single 8MB L3 cache.

We run a 2.6.24 Linux kernel with Pettersson's performance counter patch [46] on all of the processors. We used PAPI for performance counter measurements [11]. We have 4GB of physical memory in all systems.

Experimental Design and Data Analysis. Jikes RVM's timer-driven adaptive optimization system results in non-deterministic compiler and garbage collection activity. We use Jikes RVM's *replay* system to control this non-determinism (see Blackburn et al. for the detailed methodological justification [7]). In order to reflect steady state performance, before running any experiments, we first execute each benchmark fifteen iterations within the same invocation of the VM, and record a compiler advice file. The file specifies the optimization level and profile information, e.g., method and edge frequencies, for each method. We repeated this five times and chose the best performing run. Later, when the VM is run in replay mode, it immediately optimizes each method to its final optimization level based on the profile. This both delivers determinism and short circuits the normal code warm-up. Thus all methods are optimized to their final level by the end of the first iteration of a benchmark. Before starting the second iteration, we perform a full heap garbage collection. We report timing measurements for the second iteration. For each experiment, we report the average of 30 runs to eliminate noise. Our default configuration uses 2MB buffer size and 128KB of chunk size. We set the heap

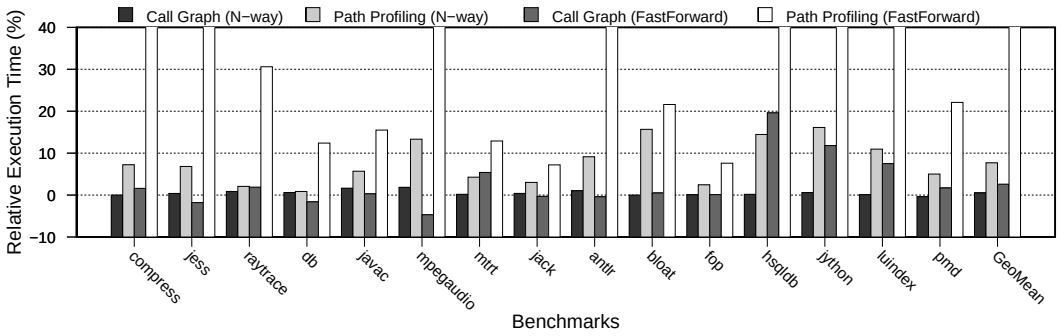


Figure 2.8: Performance of N-way buffering and FastForward relative to CAB. We use N:M threading on an Intel Core 2 Quad. The Y-axis is normalized to CAB's execution time.

size to 4 times the minimum required for the uninstrumented benchmark.

2.6.1 CAB versus Other Buffering Mechanisms

We start by comparing CAB with conventional N-way buffering and FastForward's concurrent lock-free queue [22]. We carefully optimized both algorithms for a fair comparison. For FastForward, enqueueing and dequeueing do not split the buffer into chunks, instead they operate on individual event records, which is equivalent to CAB using a chunk size of one event. We follow their recommendation and set their *dangerous distance* parameter to two cache lines: when the consumer becomes this close to the producer, the consumer waits. Once there is a *safe distance* of six cache lines, the consumer begins processing events again. We use the same algorithm and parameters as specified in the FastForward paper [22].

Our implementation of N-way buffering improves in two ways over PiPA [57]. First, we replaced semaphores for buffer switching to lock-free

synchronization as in CAB. Second, the buffer size is aligned to a power of two such that the end of each buffer is evaluated by a modulo operation and `test` instruction. For fair comparison with CAB, we do not pin the buffer into a fixed memory location which would remove a memory load, since a fixed memory location is incompatible with multi-threaded analysis. Note that a buffer in N-way buffering is essentially the same as a chunk in CAB, but they are accessed differently; CAB's operation is asymmetric while N-way buffering is symmetric.

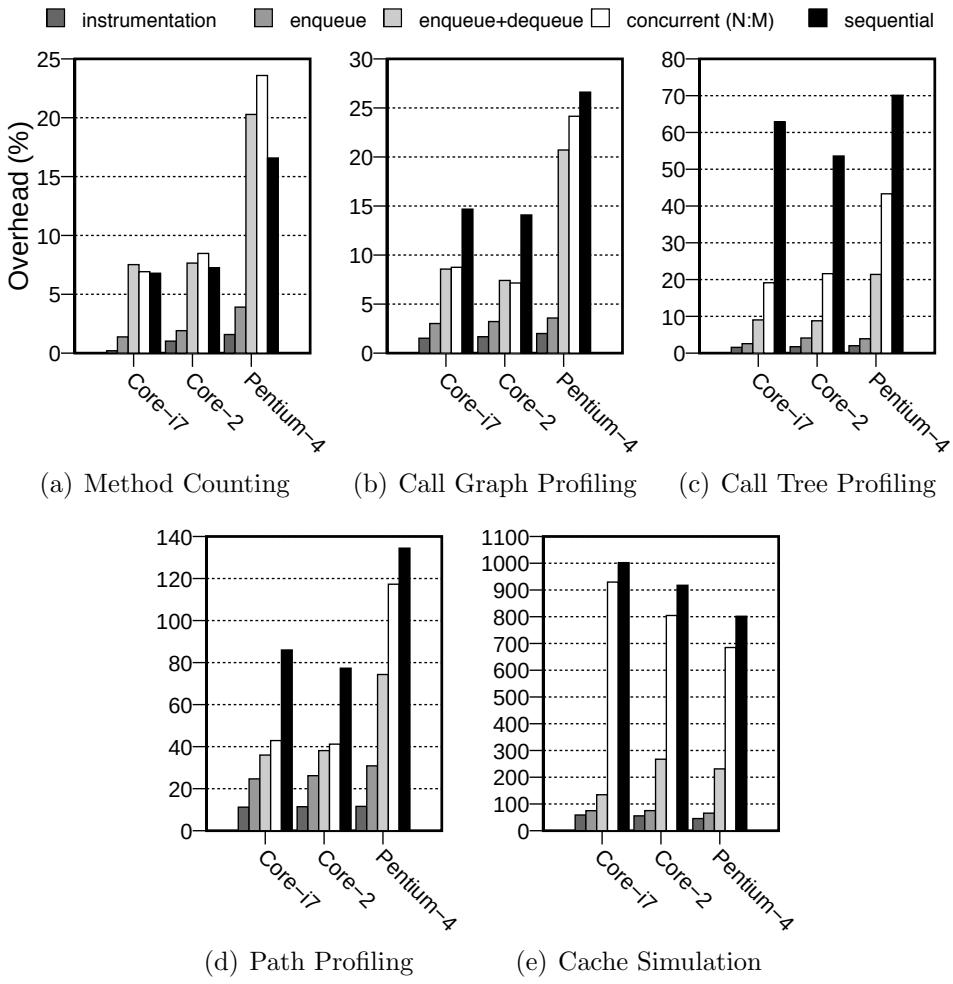
Figure 2.8 compares the performance of these buffering mechanisms to CAB on two representative analyses: call graph and path profiling. Call graph analysis requires far less communication compared to path profiling; the communication overhead of these clients is discussed in detail in Section 2.6.2. The performance of N-way buffering and FastForward queue is reported relative to CAB's execution time, where higher than zero means worse than CAB. For dynamic analyses that perform less communication, like call graph profiling, there is no significant difference between the three buffering designs. However, for path profiling where the data sharing cost is high, CAB outperforms FastForward by a significant margin. The FastForward queue is well designed as a general purpose queue, but the absence of chunks and batch processing causes significant overhead when used for concurrent dynamic analysis. CAB performs better than heavily optimized N-way buffering by ∼8% on average, and up to 16%. The performance improvements of CAB are most significant on benchmarks that produce events more frequently, such as `jython` and `hsqldb`.

The results show that CAB is more efficient in transferring events from one thread to another than other buffering designs, especially when there is significant data communication between the producer and consumer. This result suggests that existing dynamic analyses that use buffering can achieve a speed-up transparently by using CAB.

2.6.2 Exhaustive Mode Overhead

We now examine the performance of CAB in more detail, starting with exhaustive mode. Figure 2.9 shows the exhaustive mode overhead for each concurrent analyzer and processor combination with N:M threading. The results report the average over all benchmarks. Results for individual benchmarks are in Section 2.7. All measurements are relative to the application without any instrumentation or analysis, i.e., the application time A from our model in Section 2.3. Lower bars are better. We break down the overhead as follows.

In each set of bars, the fourth white bar ("concurrent analyzer") shows CAB in exhaustive, concurrent analysis mode. The fifth black bar ("sequential analyzer") is the same analysis, but the instrumentation and analysis are inline in the same thread as the application (E_s). The differences between the fourth bar and the fifth bar show the performance benefit of a concurrent implementation using CAB compared to sequential analysis. The first to third bars break down the overhead of the concurrent analysis. The first bar ("instrumentation") is pure instrumentation overhead; the application produces the event and writes it to a single word in memory, but the analyzer thread is



(a) Method Counting (b) Call Graph Profiling (c) Call Tree Profiling

(d) Path Profiling (e) Cache Simulation

Figure 2.9: Exhaustive mode overhead with performance break-down, averaged over all benchmarks (N:M threading model).

not running. The second bar ("enqueue") is the enqueueing overhead where the application enqueues to the buffer, but the analyzer thread is still not running. The third bar isolates the communication overhead; the application thread performs full CAB functionality while the analyzer dequeues and writes to a single word, but does not process the event. Thus, data is transferred

through the cache, but not analyzed.

Method counting (Figure 2.9(a)) is very lightweight with minimal analysis overhead and is thus not a compelling candidate for a concurrent implementation. In spite of its minimal analysis, concurrent method counting performs nearly as well as sequential method counting. Leveraging reduced memory latency in recent multicore hardware, the concurrent method counting is only slower by 0.1% and 1.2% on the Core i7 and Core 2 respectively, while it was 7% slower on the hyper-threaded Pentium 4.

Call graph profiling performs only slightly more analysis computation than method counting, yet the concurrent call graph performs better than the sequential version. Concurrent call graph profiling has approximately half the overhead of the sequential implementation in Core i7 and Core 2. For call graph profiling, and the other heavier-weight clients, concurrent execution on the P4 shows benefit, but less than the other architectures, because the application and analyzer share the core, and thus there is less true concurrency.

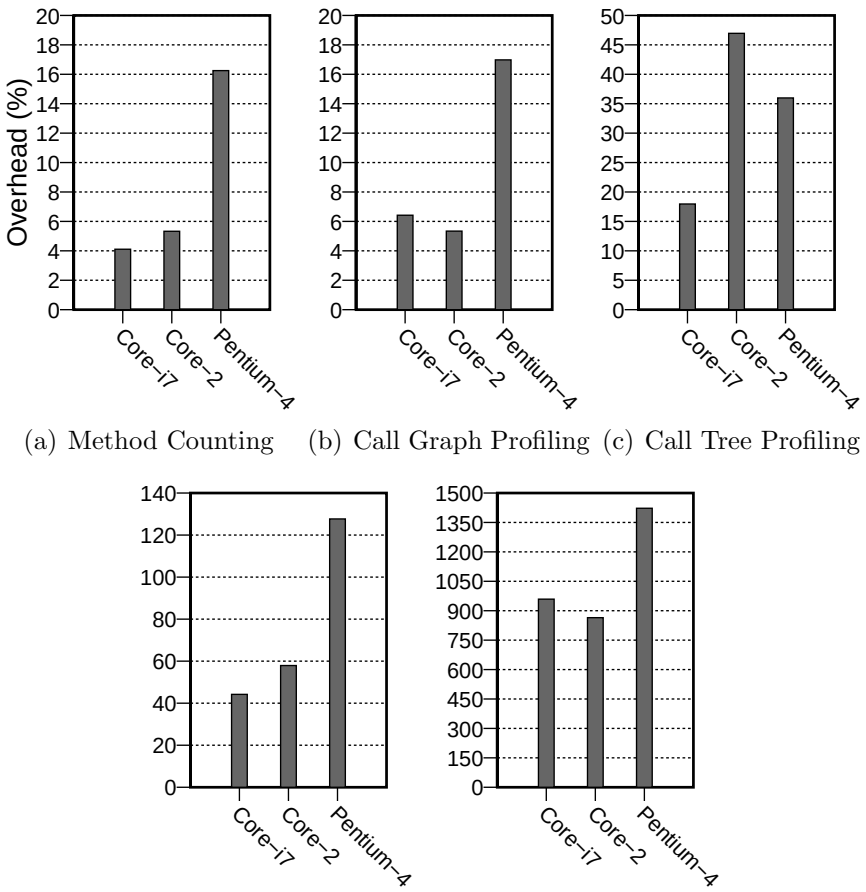
Call tree profiling has the same amount of communication as call graph profiling, but performs more analysis. This analysis time is still less than the application time, and thus concurrent dynamic analysis improves further over sequential. For example, concurrent call tree profiling's overhead is 60% less than sequential profiling on the Core i7.

Path profiling has more communication overhead than call graph profiling, but the computation required for each record is similar (updating a hash

table). This increase results in a higher relative enqueueing cost (second bar) compared to the other analyzers. On the P4, path profiling time often dominates application time, which limits performance improvements, as our model predicts, but concurrent analysis still reduces overhead by on average 17%. On the Core 2 and Core i7, concurrent path profiling decreases the overhead by about half compared to sequential profiling. These results show that CAB is efficiently offloading the profile data to the other core.

The cache simulator is an extreme case of heavy-weight analysis. The analyzer itself is an order of magnitude slower than the application. Thus, even if all the event data were transferred to the other core with zero overhead, the benefit of the concurrent cache simulator is limited to a 100% reduction, i.e., eliminating the application execution time. However, we measured faster critical path execution because CAB offloads load and store data from the critical path. CAB thus sometimes reduces the overhead by more than the application time. Our results show that concurrent cache simulator was faster by 73% on the Core i7, and 110% on the Core 2, and 193% on the Pentium 4.

Native threading Figure 2.10 presents the average concurrent analysis overhead using our native thread implementation. These results show similar overheads compared to the N:M threading results discussed above. The native thread implementation is relative to a better baseline; without concurrent analysis, native threads and other enhancements improve performance over the Jikes RVM version with N:M threading by 15 to 20%. Our native



(a) Method Counting (b) Call Graph Profiling (c) Call Tree Profiling

(d) Path Profiling (e) Cache Simulation

Figure 2.10: Exhaustive mode overhead, average over all benchmarks (native threading model with per-thread CAB).

thread implementation of concurrent analysis improves over the N:M thread version for method counting and call graph profiling, and is a bit slower for path profiling and cache simulation. These results confirm that the threading model is not central to our results.

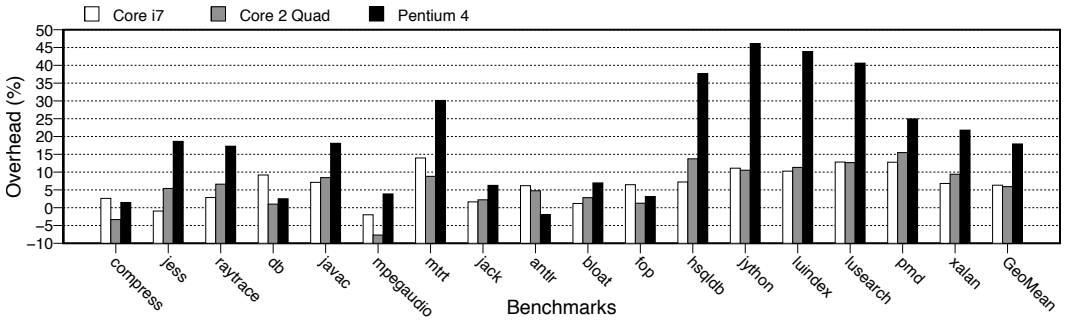


Figure 2.11: Call graph profiling overhead on native threading model with per-processor CAB.

Native threading with per-processor CAB Figure 2.11 presents the overhead of call graph profiling on native threading with per-processor CAB. Geometric means for Core i7, Core 2, and Pentium 4 are 6.3%, 5.9%, and 17.9%, respectively, which are less than 1% difference to native threading with per-thread CAB, i.e., 6.4%, 5.3%, and 17.0%, respectively. These results prove that our framework successfully maintains scalability in buffer memory usage. However, we choose the best buffer and chunk size for each benchmark by hill climbing because the overhead is more sensitive to these parameters than other implementations. This sensitivity suggests future work to explore a mechanism that dynamically self-tunes these parameters. We believe that per-thread CAB is useful in general for simplicity, but that per-processor CAB are required for memory scalability for applications with many threads.

Exhaustive Mode Summary Our concurrent dynamic analysis framework improves performance on three generations of hardware, from the Pentium 4 to the Core i7. All of our results, except for cache simulator on Core i7,

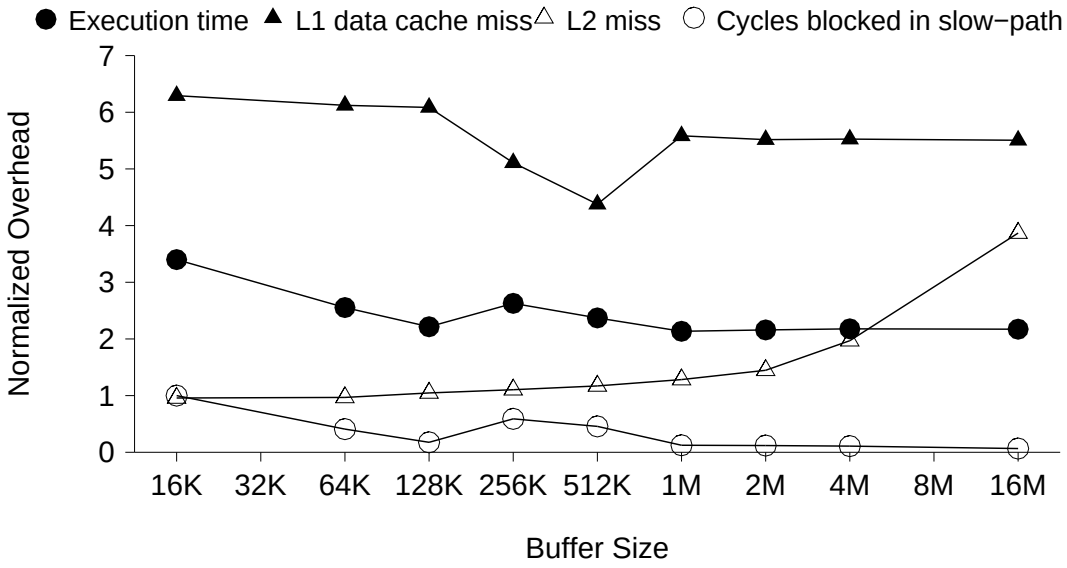


Figure 2.12: Performance as buffer size varies for path profiling on DaCapo `hsqldb` (using N:M threading on an Intel Core 2 Quad).

show that newer generation multicores yield the largest improvements. This trend supports our contention that concurrent dynamic analysis will be more important for future architectures, and that our framework can be the basis for this and other applications that require offloading work to other cores.

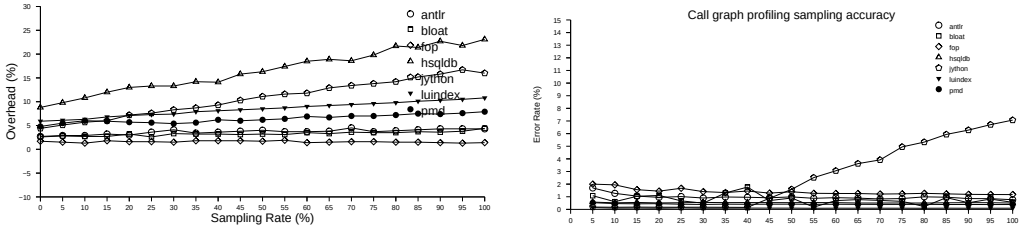
2.6.3 Buffer Size Scalability

One of the strengths of our concurrent dynamic analysis framework is the scalability of the buffer size in CAB. The particular benchmark and analysis together determine a minimal buffer size that is sufficient to minimize the overhead that comes from a variable event rate. If large buffers cause performance degradations, as reported for PiPA [57], the increased headroom of the larger buffer will come at the cost of degraded average performance.

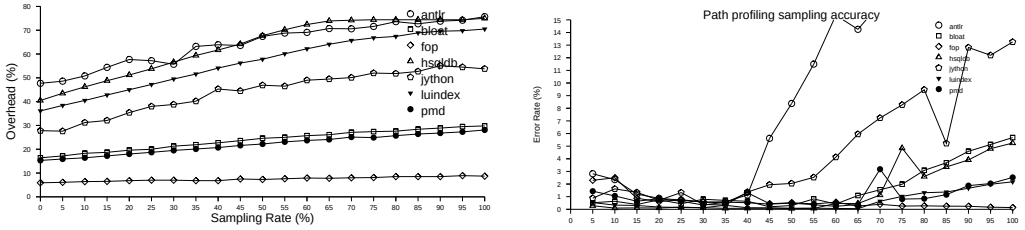
Figure 2.12 presents buffer size scalability with path profiling and shows L1 and L2 misses, as well as the cycles blocked on the slow-path of the CAB enqueueing operation for `hsqldb`, a representative benchmark. Section 2.7 contains results for six more DaCapo programs. This experiment is performed on a Core 2 Quad processor because it has the most irregular memory latency of the three processors we evaluate. Each of the metrics is normalized to the measurement with no analyzer. Since the application never blocks without an analyzer, we normalize cycles blocked to the slow-path with a 16KB buffer size.

L1 misses are high for small buffers because there are more conflict misses between the application and analyzer threads while the application is blocked. L1 misses drop near the 128KB and 512KB buffer sizes, and grow again because a larger buffer size increases the memory footprint of the buffer. There are few L2 misses on small buffers because they fit into the L2 cache, and L2 misses grow as the buffer starts to contend with the application memory.

The execution time shows that the overhead is nearly constant given a sufficiently large buffer size, demonstrating that the overhead is not correlated to L1 or L2 cache misses and that larger buffers do not degrade performance. This result supports our hypothesis that the design of CAB allows the hardware prefetcher and cache subsystem to hide latency.



(a) Call Graph profiling sampling overhead



(b) Path profiling sampling overhead

Figure 2.13: Sampling overhead and error rate for call graph and path profiling (N:M threading on an Intel Core 2 Quad).

2.6.4 Sampling Mode Accuracy vs Overhead

Figure 2.13 reports the overhead and error rate of call graph profiling and path profiling in sampling mode running on a Core 2 Quad. The graphs on the left show sampling overhead and the graphs on the right show accuracy.

For the overhead graphs, the y-axis shows percent overhead, while the x-axis shows the sampling rate, expressed as the percent of samples that are processed by the analyzer thread. All sampling mode data was collected using a default burst size of 64 bytes, which is equal to the cache line size on each of the processors we evaluated. A sampling rate of zero means that no samples were processed by the analyzer thread, and thus represents the minimum

overhead possible. Note that a 100% sampling rate is not the same as exhaustive mode. The analyzer does not intentionally discard any samples, but since the application does not block, it is possible for the application to overwrite samples before they reach the analyzer.

In the accuracy graphs, the y-axis reports the error rate, which is the average error rate of each individual metric. Each individual error rate is defined as follows:

$$Error\ Rate = \left| \frac{Actual\ Frequency - \frac{Sampled\ Frequency}{Sampling\ Rate}}{Actual\ Frequency} \right|$$

For example, in call graph profiling, an individual error rate is the error rate of each caller and callee pair. The error rate on the accuracy graph is the average accuracy of all the individual error rates. This error rate treats low to high frequency events equally so that it is not biased.

The overall performance trend is not surprising; overhead increases linearly as the sample rate is increased. Sample rates ranging from 5% to 20% offer a significant reduction in overhead versus the same profile collected in exhaustive mode (from Figure 2.9), yet still produce profiles with extremely high accuracy. The average overhead reduction relative to the exhaustive profile was 55% for both call graph and path profiling at 5% sampling rate, and the error rate is less than 3% (97% accurate). Depending on the use of the profile data, this sampled profile may be indistinguishable from an exhaustive mode profile.

For some benchmarks, the error rate begins increasing rapidly when

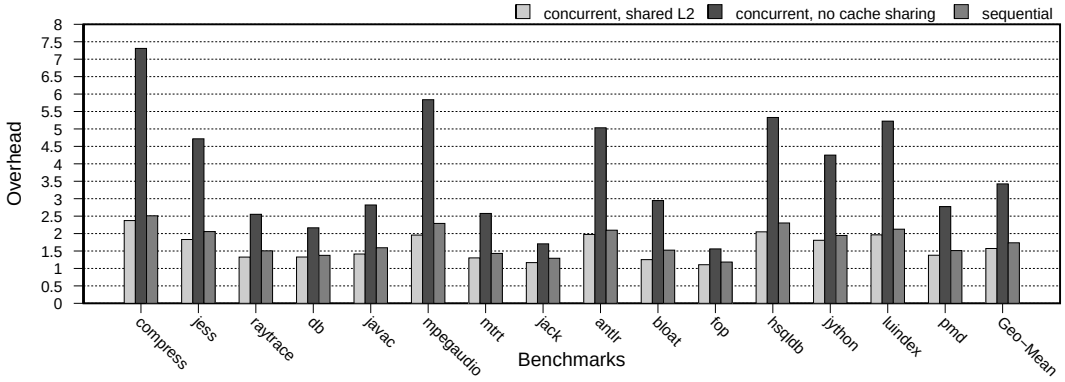


Figure 2.14: The importance of shared caches. Path profiling overhead with and without sharing between analyzer and application threads. Note that y-axis is the factor of overhead, and not a percentage.

the number of samples taken *increases* past a certain point, which is quite counter intuitive. More samples usually results in a more accurate profile. This degradation occurs when the analyzer thread cannot keep up with the application thread and the CAB buffer overflows. At this point, data is lost in large, non-random bursts, so the accuracy of the sampled profile suffers. Path profiling is more expensive, so increasing the sample rate leads to buffer overflow sooner than with call graph profiling. To avoid this degradation, our algorithm could overflow by periodically sampling the buffer head and tail, and scale back the sample rate accordingly. We leave this functionality to future work.

2.6.5 Shared cache and fine-grained parallelism

We now evaluate why concurrent analysis has become feasible with recent multicore hardware. A concurrent dynamic analysis is fine-grained par-

allelism where data sharing happens frequently, thus the performance is sensitive to the latency and the bandwidth of inter-core communication. To show how much benefit comes from the low-latency communication, we changed the affinity of the analyzer thread and the application thread to force them onto different dies on the Intel Core 2 Quad processor. In this configuration, they are much less likely to be on cores that share a cache at any level. In this experiment, we use one application and one profiler thread to avoid cache thrashing among application threads. Figure 2.14 compares the performance of this new configuration, called "no cache sharing", to the original shared L2, and single-threaded configurations. The figure reports overhead as a *factor slowdown, not as percent*. Since in this configuration the threads must communicate through memory instead of the L2, the overhead increases from an average of $\sim 50\%$ to $\sim 250\%$, confirming that in our setting, cache-aware communication is critical to good performance.

2.7 Additional Results

Exhaustive Mode Overhead Figures 2.15 and 2.16 and 2.17 report the per-benchmark breakdown of CAB's exhaustive mode overhead with N:M threading executing on the Core i7, Core 2, and Pentium 4, respectively. Figure 2.9 in Section 2.6.2 summarizes this data for all architectures.

Figure 2.16 reports the per-benchmark breakdown of CAB's exhaustive mode overhead with N:M threading executing on the Core 2 Quad processor. Please refer to [26] for complete results on the other architectures (Core i7

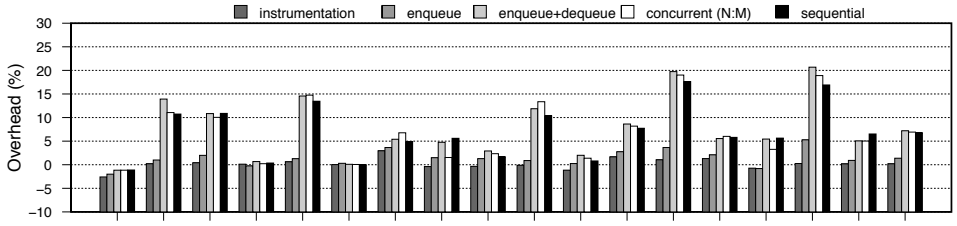
and Pentium 4). Figure 2.9 in Section 2.6.2 summarizes this data for all architectures.

Similarly, Figure 2.18 reports the per-benchmark exhaustive mode overheads for the native threading model. Figure 2.10 in Section 2.6.2 summarizes this data.

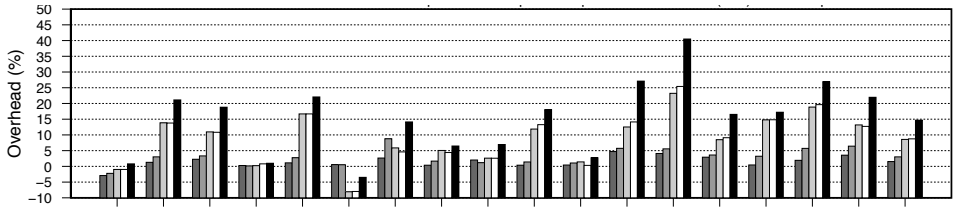
Buffer Size Scalability Figure 2.19 reports performance as buffer size increases when performing path profiling, for each of the remaining DaCapo benchmarks. Figure 2.12 of Section 2.6.3 presented this data for the `hsqldb` benchmark.

2.8 Conclusion and Interpretation

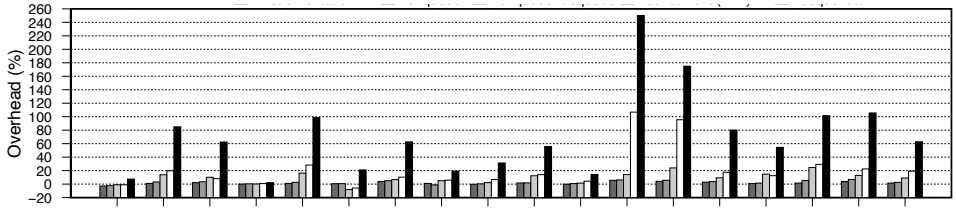
Managed languages have succeeded in part because the run up in single processor speeds from Moore's law more than compensated for the cost of abstractions, such as managed runtimes and dynamic analyses. To continue to give programmers current and future generations of powerful abstractions, we will need to construct efficient mechanisms that more carefully minimize their costs. This chapter addresses the cost of dynamic analysis. We introduce a framework that uses CAB, a new highly-optimized cache-friendly asymmetric buffering mechanism, that outperforms the prior state of the art, sometimes significantly. For extremely light weight analysis (i.e., few events and little processing) our framework is not beneficial, but for a wide class of dynamic analysis, we show that our framework improves performance. Our work on



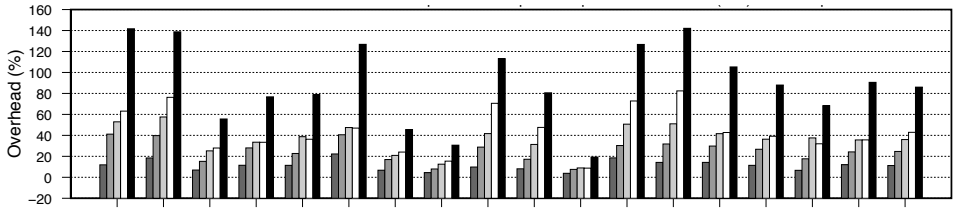
(a) Method Counting Overhead Percentage



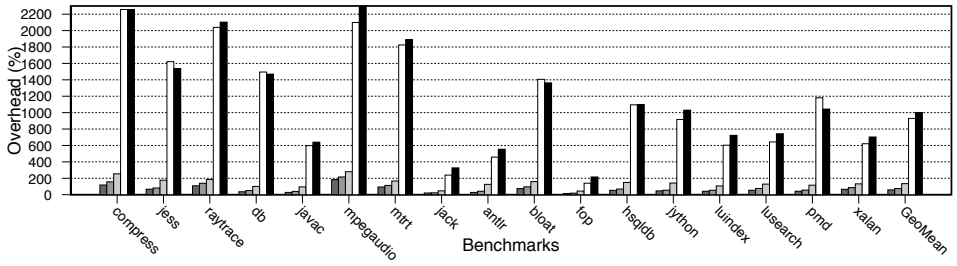
(b) Call-Graph Profiling Overhead Percentage



(c) Call-Tree Profiling Overhead Percentage

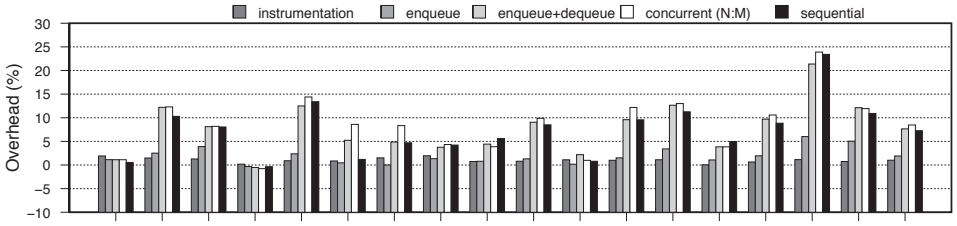


(d) Path Profiling Overhead Percentage

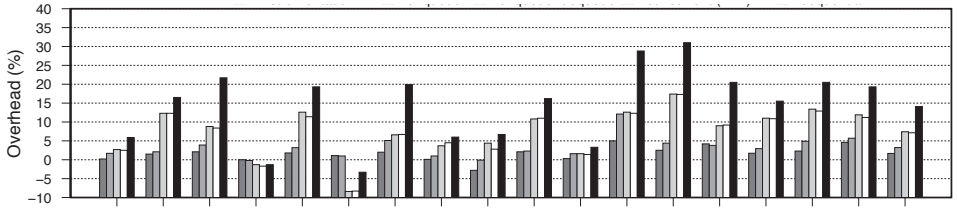


(e) Cache Simulation Overhead Percentage

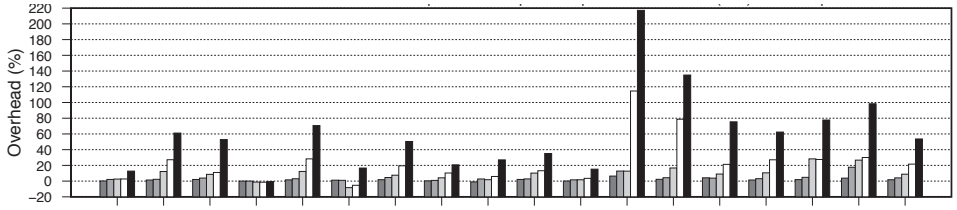
Figure 2.15: Per-benchmark exhaustive mode overhead on Core i7 (N:M threading)



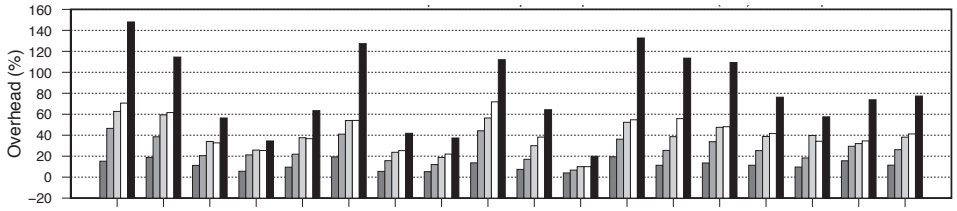
(a) Method Counting Overhead Percentage



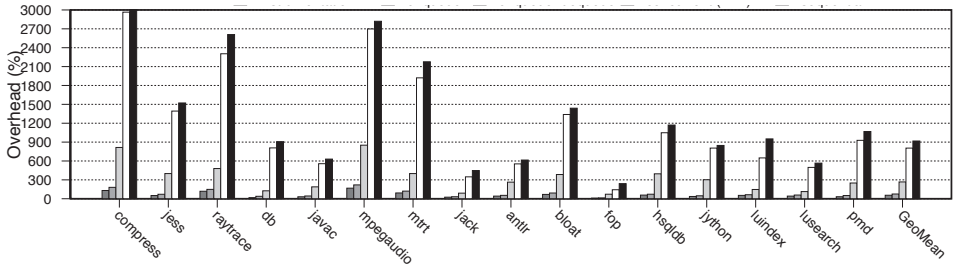
(b) Call Graph Profiling Overhead Percentage



(c) Call Tree Profiling Overhead Percentage

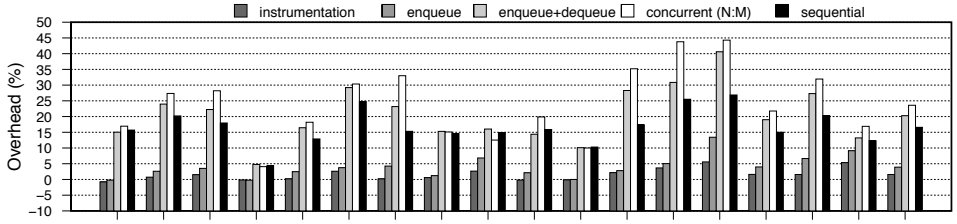


(d) Path Profiling Overhead Percentage

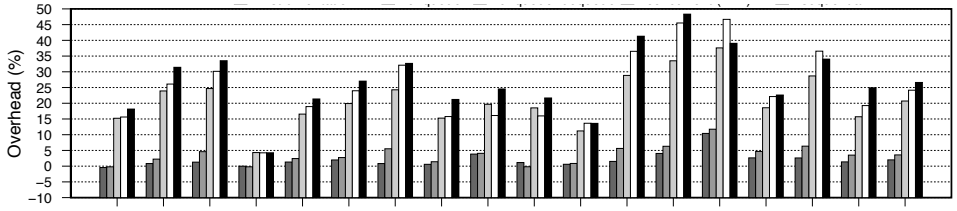


(e) Cache Simulation Overhead Percentage

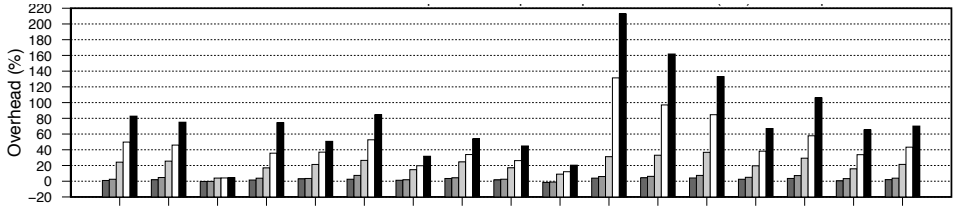
Figure 2.16: Per-benchmark exhaustive mode overhead on Core 2 (N:M threading).



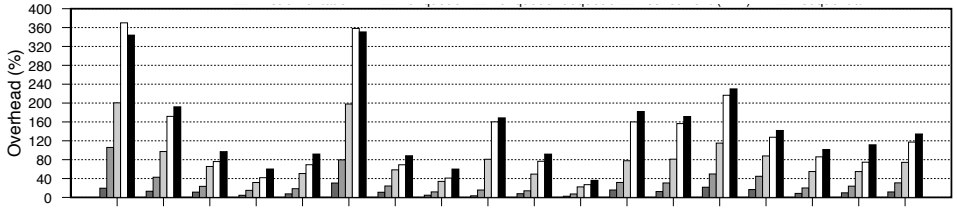
(a) Method Counting Overhead Percentage



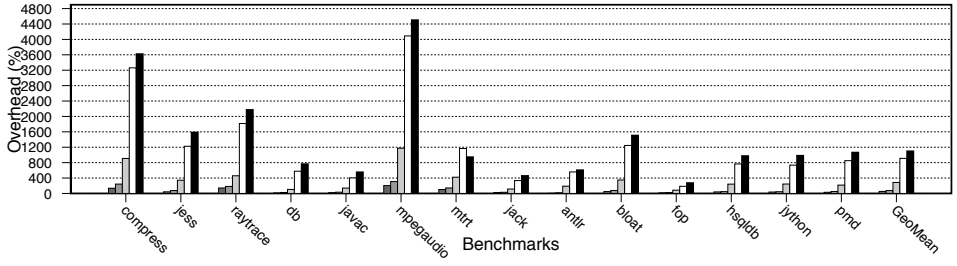
(b) Call-Graph Profiling Overhead Percentage



(c) Call-Tree Profiling Overhead Percentage

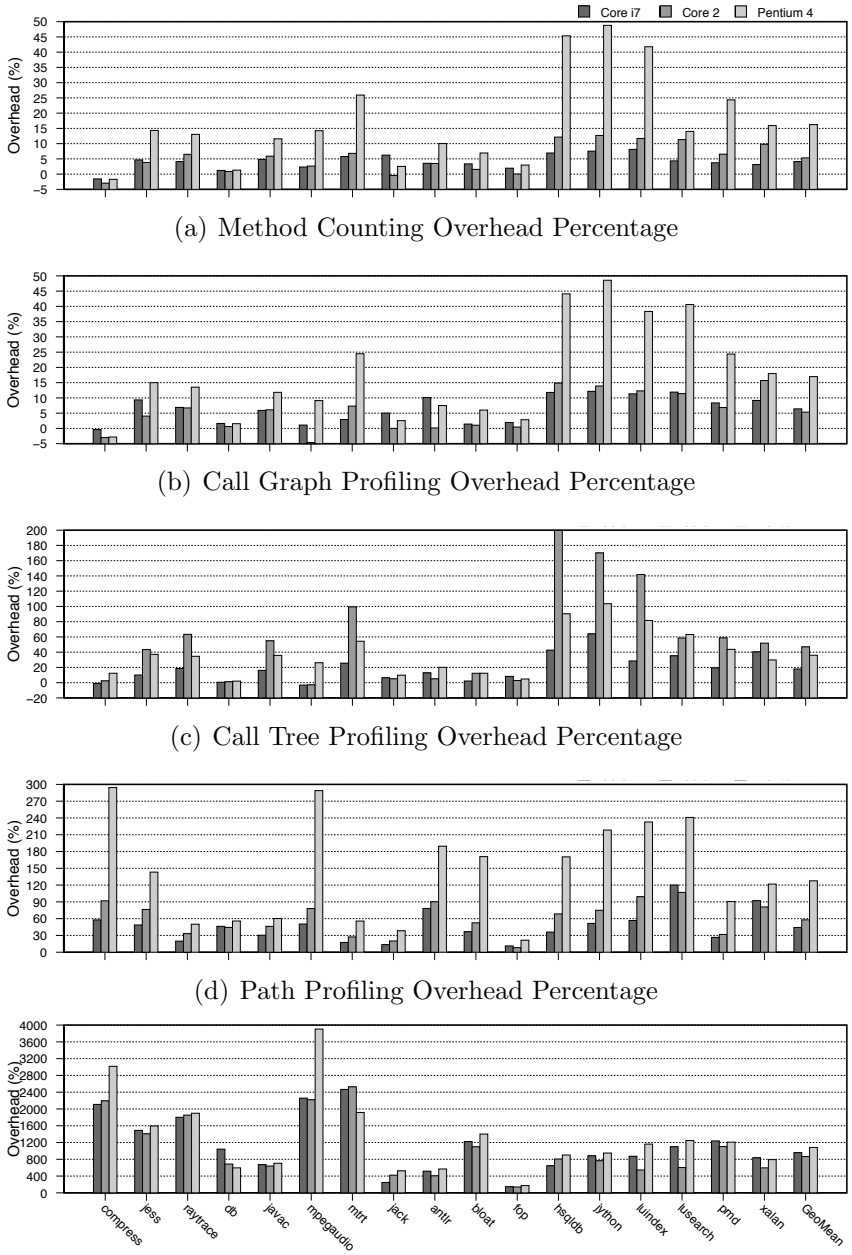


(d) Path Profiling Overhead Percentage



(e) Cache Simulation Overhead Percentage

Figure 2.17: Per-benchmark exhaustive mode overhead on Pentium 4 (N:M threading model)



(a) Method Counting Overhead Percentage

(b) Call Graph Profiling Overhead Percentage

(c) Call Tree Profiling Overhead Percentage

(d) Path Profiling Overhead Percentage

(e) Cache Simulation Overhead Percentage

Figure 2.18: Per-benchmark exhaustive mode overhead (native threading).

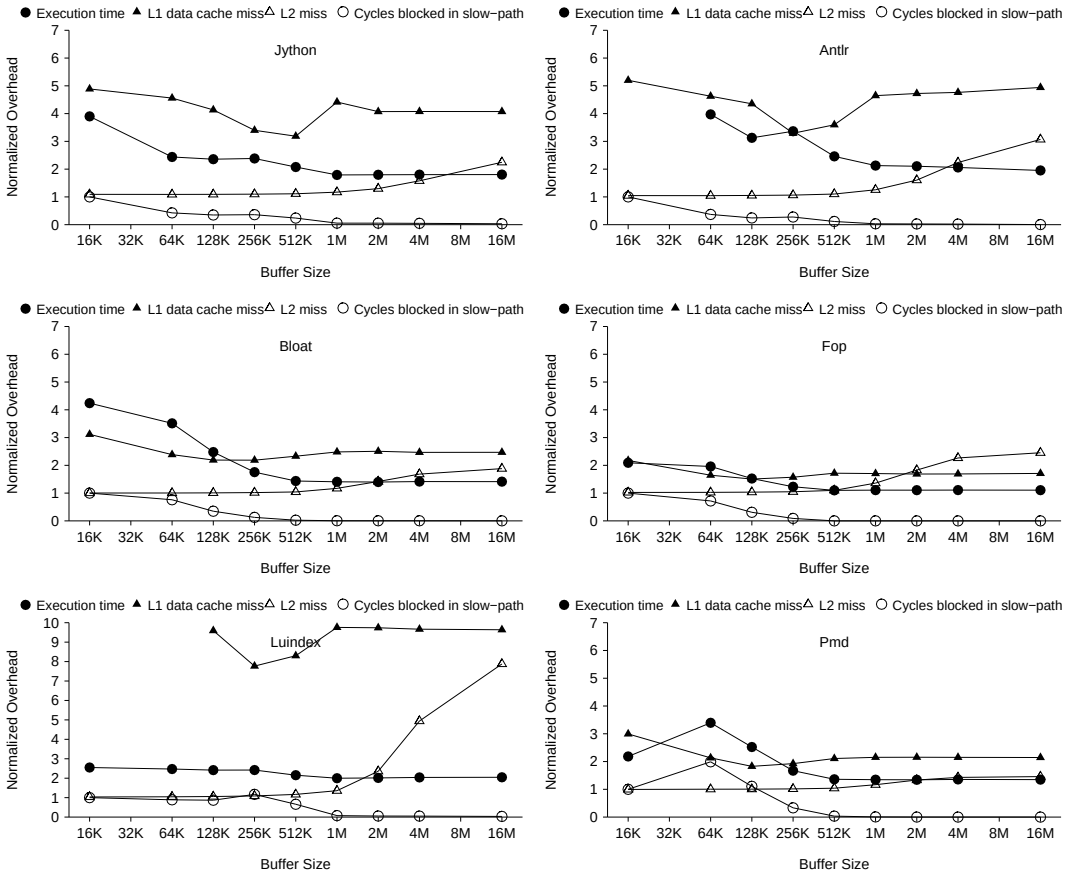


Figure 2.19: Performance as buffer size varies for each of the DaCapo benchmarks. Results are for path profiling using N:M threading on an Intel Core 2 Quad.

this chapter takes an important step towards reducing abstraction costs for dynamic analyses by utilizing otherwise idle cores in multicore systems. We believe that our optimization lessons are broadly applicable, and can help optimize more generic parallel programs with heavy inter-thread communication.

Chapter 3

A Concurrent Trace-based JIT Compiler for JavaScript

JavaScript is emerging as the scripting language of choice for client-side web browsers [23], and the number of such application is exploding. Client-side JavaScript applications initially performed simple HTML web page manipulations to aid server-side web applications, but they have since evolved to use asynchronous and XML features to perform sophisticated, interactive dynamic content manipulation on the client-side. This style of JavaScript programming is called AJAX (for Asynchronous JavaScript and XML). Companies, such as Google and Yahoo, are using it to implement interactive desktop applications such as mail, messaging, and collaborative spreadsheets, word processors, and calendars. Because Internet usage on mobile platforms is growing rapidly, the performance of JavaScript has become critical for both desktops and embedded mobile devices. To speed up the processing of JavaScript applications, many web browsers are adopting Just-In-Time (JIT) compilation, including Firefox TraceMonkey [19], Google V8 [24], and WebKit SFE [54].

Generating efficient machine code for dynamic languages, such as JavaScript, is more difficult than for statically typed languages. For dynamic lan-

guages, the compiler must generate code that correctly executes all possible runtime types. Gal et al. recently introduced a trace-based JIT compilation for dynamic languages to address this problem and to provide responsiveness (i.e., low compiler pause times and memory requirements) [18]. Responsiveness is critical, because JavaScript runs on client-side web browsers. Pause times induced by the JIT must be short enough not to disturb the end-user experience. Therefore, Gal et al.'s system interprets until it detects a hot path in a loop. The interpreter then *traces*, recording instructions and variable types along a hot path. The JIT then specializes the trace by type and translates it into native code in linear time. The JIT sacrifices code quality for linear compile times, rather than applying heavy weight optimizations. This trace-based JIT provides fast, light-weight compilation with a small memory footprint, which make it suitable for resource-constrained devices.

On the hardware side, multicore processors are prevailing from embedded to general purpose systems. The JavaScript language however lacks a thread model, and thus all JavaScript applications are single-threaded. This limitation provides the opportunity to perform the JIT and other VM services concurrently on another core, transparently to the application, since the application is guaranteed not to be using it. Unfortunately, state-of-the-art trace-based JIT compilers are sequential [18, 19, 51], and have not exploited concurrency to improve responsiveness. In fact, previous attempts failed to produce a concurrent system because of the complex state space in trace-based JIT compilation [17].

In this chapter, we present the design and implementation of a concurrent trace-based JIT compiler for JavaScript that combines responsiveness and throughput for JavaScript applications. We address the synchronization problem specific to the trace-based JIT compiler, and present novel lock-free synchronization mechanisms for wait-free communication between the interpreter and the compiler. Hence, the compiler runs concurrently with the interpreter and reduces pause times to nearly zero.

Our mechanism piggybacks a single word, we call the *compiled state variable (CSV)*, on each trace. Comparing with CSV synchronizes all of the compilation actions, including checking for native code, preventing duplicate traces, and allowing the interpreter to proceed, without using a lock.

We introduce lock-free *dynamic trace stitching* in which the compiler patches new native code to the existing code. Dynamic trace stitching prevents the compiler from waiting for trace stitching while the interpreter is executing the native code, and reduces the potential overhead of returning from native code to the interpreter.

We implement our design in the open source TamarinTracing VM, and evaluate our implementation using the SunSpider JavaScript benchmark suite [55] on three different hardware platforms. The experiments show that our concurrent trace-based JIT implementation reduces the total pause time by 88%, the maximum pause time by 93%, and the average pause time by 97% on Linux. Moreover, the design improves the throughput by an average of 2–7%, with improvements up to 36%.

Our concurrent trace-based JIT virtually eliminates compiler pause times and increases application throughput. Because tracing overlaps with compilation, the interpreter prepares the trace earlier for subsequent compilation, thus the JIT delivers the native code more quickly. This approach also opens up the possibility of increasing the code quality with compiler optimizations without sacrificing the application pause time.

3.1 Related Work

Sequential trace-based JIT has been proposed for Java [20] and JavaScript [18]. Gal et al. proposed splitting trace tree compilation steps into multiple pipeline stages to exploit parallelism [17]. This is the only work we can find seeking parallelism in the trace-based compilation. There are a total of 19 compilation pipeline stages, and each pipeline stage runs on a separate thread. Because of data dependency between each stage and the synchronization overhead, the authors failed to achieve any speedup in compilation time. We show having a parallel compiler thread operating on an independent trace provides more benefit than pipelining compilation stages. With proper synchronization mechanisms, our work successfully exploits parallelism in the trace-based JIT by using tracing concurrently with the compilation, even when using only one compiler thread.

Related work on adaptive compilation, includes the SELF-93 VM, which introduced adaptive compilation strategies for minimizing application pause time [33]. When a method is invoked for the first time, the VM compiles it

without optimizations using a light weight compiler. If the number of times a method is invoked exceeds a threshold, the VM recompiles it with a more heavy weight optimizing compiler. While the SELF-93 VM provided reasonable responsiveness, it still must pause the application thread for compilation.

Krintz et al. implemented profile-driven background compilation in the Jalapeño Virtual Machine (now called Jikes RVM) [3, 34]. In multiprocessor systems, a background compiler thread overlaps with application execution, which reduces compilation pause times. They also applied lazy compilation, where the JIT only compiles the method on demand within a class instead of compiling every method in a class. When the method is invoked for the first time before the optimized code is ready, the VM pauses the application and run the baseline compiler.

Kulkarni et al. explored maximizing throughput of background compilation by adjusting the CPU utilization level of the compiler thread [35]. This technique is useful when the number of application threads exceeds the number of physical processors and the compiler thread cannot fully utilize a processor resource. They conducted their evaluation on method-based compilation, though the same technique can be applied to trace-based compilation. However, because JavaScript is single-threaded, it is less likely that all the cores are fully utilized in today's multicore hardware. Hence, the effect of adjusting CPU usage levels will not be as significant as it is in multi-threaded Java programs.

These novel techniques have made adaptive compilation practical in

Java Virtual Machines, such as Sun HotSpot [45], IBM J9 [50], and Jikes RVM [3]. VMs without an interpreter still introduce compiler induced pause times when first executing a method. This initial compiler pause time is not an issue for most desktop and server environments, and hence has been neglected by the research community. As computing environments evolve to resource constrained devices, JIT compilation will be used for interactive applications, where start-up pause times hurt responsiveness. Our concurrent JIT technique is complementary to the prior work and shows how to use the interpreter and a concurrent compiler to limit pause times. We explore issues specific to the concurrent tracing JIT, which has not been evaluated by any previous work.

3.2 Background

3.2.1 Dynamic Typing in JavaScript

JavaScript is a *dynamically* typed language. The type of every variable is inferred from its content dynamically. Furthermore, the type of JavaScript variables can change over time as the script executes. For example, a variable may hold an integer object at one time and later hold a string object. A consequence of dynamic typing is that operations need to be dispatched dynamically. While the degree of type stability in JavaScript is the subject of current studies, our experiences and empirical results indicate that JavaScript variables are type stable in most cases. This observation suggests that type-based specialization techniques pioneered in Smalltalk [16] and later used in Self [32] and Sun HotSpot [45] have the potential for tremendously improving

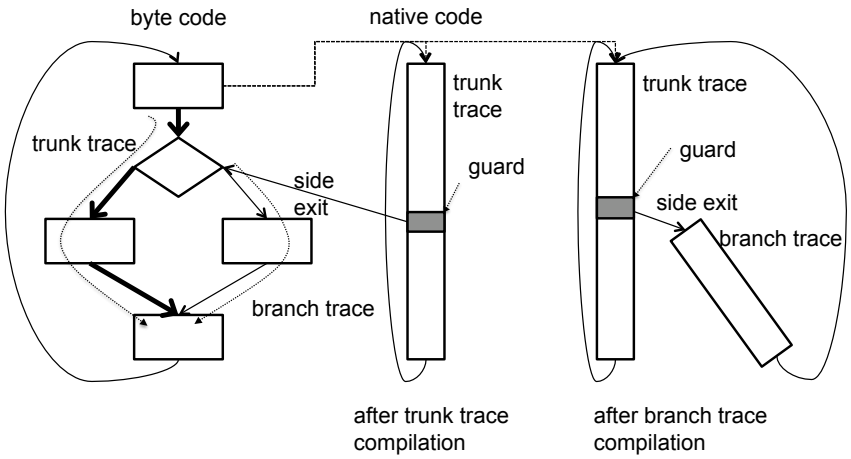


Figure 3.1: Byte code and native code transition in the trace-based JIT. Initially, the interpreter interprets on the byte code. First detected hot path (thick path) is traced forming a trunk trace. Following hot paths guarded and installed in a side-exit. The compiler attach the branch trace, which begins from the hot side-exit to the loop header, to the trunk trace.

JavaScript performance.

3.2.2 Trace-based JIT Compilation

Hotpath VM is the first trace-based JIT compilation introduced for Java applications in a resource-constrained environment [20]. The authors later explored trace-based JIT for dynamic languages, such as JavaScript [18].

The trace-based JIT compiles only frequently executed paths in a loop. Figure 3.1 shows an example of how the interpreter identifies a hot path, and expands it. Initially, the interpreter executes the byte code instructions, and identifies the hot loop with backward branch profiling as follows. When the execution reaches the backward branch, the interpreter assumes it a loop

backedge and increments the counter associated with the branch target address. When the counter reaches a threshold, the interpreter enables tracing, and records each byte code instruction to a trace buffer upon execution. When the control reaches back to the address where the tracing started, the interpreter stops tracing and the compiler compiles the trace to native code. As the interpreter is not doing an exact path profile, the traced path may or may not be the real hot path. The first trace in a loop is called a *trunk trace*.

Instructions are *guarded* if they potentially diverge from the recorded path. If a guard is triggered, the native code takes a *side-exit* back to the interpreter, and begins interpreting from the branch that caused the side-exit. The interpreter counts each side-exit to identify frequent side-exits. When a side-exit is taken beyond a threshold, it means the loop contains another hot path, and the interpreter enables tracing from the side-exit point until it reaches the address of the trunk trace. This trace is called a *branch trace*. A branch trace is compiled and the code is stitched to the trunk trace at the side-exit instruction. As the interpreter finds more hot paths, the number of branch traces grows forming a *trace tree*.

Since the compilation granularity is a trace, which is smaller than a method, the total memory footprint of the JIT is smaller than that of method-based JITs. And because no control flow analysis is required, start-up compilation time is less than that of the method-based compilers. However, as optimization opportunities are limited, the final code quality is not likely to be as good as code generated by method-based compilation. Therefore, trace

compilation is suitable for embedded environments where resources are limited, or the initial JIT cost is far more important than steady state performance.

3.2.3 Tamarin and TraceMonkey

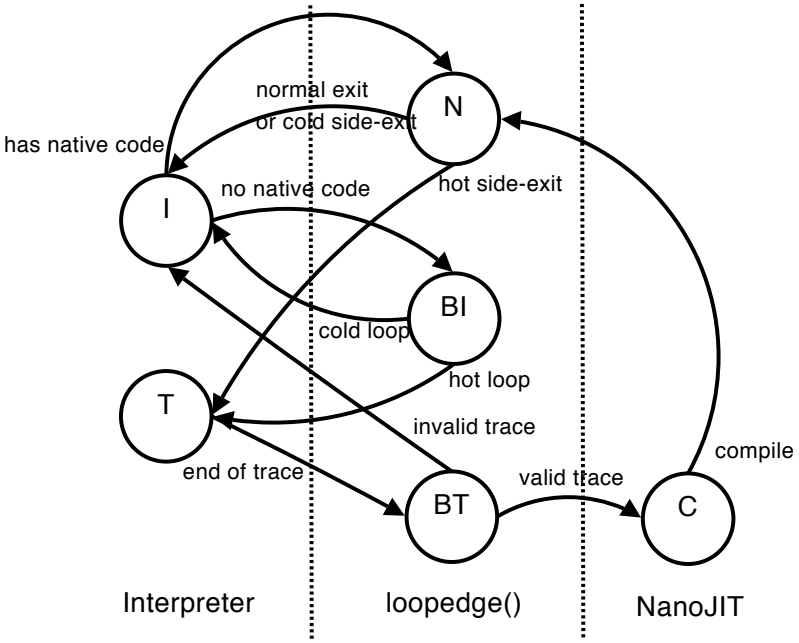
Tamarin [51] conforms to the international ECMAScript language standard. JavaScript, Adobe ActionScript, and Microsoft JScript are all dialects of the ECMAScript language. There are two branches of Tamarin VM. One is called *TamarinCentral*, which originally included a method-based JIT compiler. Tamarin Central is the ActionScript execution engine of Flash Player 10. The other one is called *TamarinTracing*, which includes a tracing JIT to enable type-specialized optimizations.

The tracing backend compiler of TamarinTracing is called *NanoJIT*. Its modular design eases migration to the other ECMAScript dialects. For example, Mozilla has integrated NanoJIT into its interpreter (SpiderMonkey) together with some enhancements. This trace-based JavaScript execution engine of Firefox is named *TraceMonkey*. It delivers a tremendous performance improvement over the previous release of Firefox, and is the state-of-the-art JavaScript engine that we improve over.

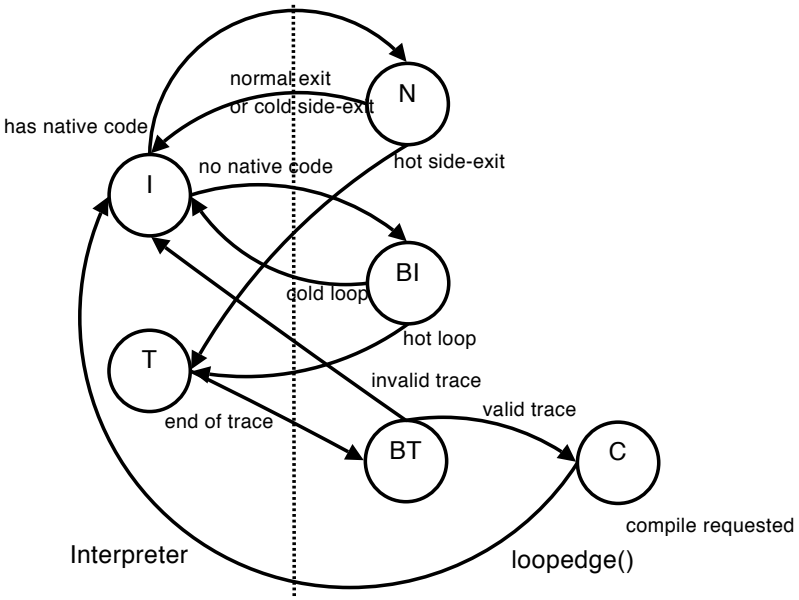
3.3 Design

3.3.1 Phase Transitions in TamarinTracing

In this section, we explain the phase transitions of the sequential JIT in TamarinTracing VM. Figure 3.2(a) depicts the phase and its transitions. It



(a) Sequential JIT



(b) Concurrent JIT

Figure 3.2: Phase transitions in Tamarin.

consists of the following six phases: I: normal interpreter, T: tracing enabled, N: running native code, BI: loop edge without trace, BT: loop edge with trace, and C: compilation or compile request.

Phase I. Initially, Tamarin starts with a standard interpretation phase I, where only the interpreter runs without tracing.

Phase T. In this phase, the interpreter is running with tracing enabled. Traced instructions are stored in a trace buffer in an SSA-based intermediate representation (IR). Inlining happens naturally since the trace is recorded beyond a method boundary. In addition to the bytecode instructions, the IR is instrumented with *guard* instructions. These guards are created with conditional branches, type checks, function dispatches, and other runtime tests to validate the preconditions of the trace. These guards guarantee an exit to the interpreter from the native code whenever the conditions are false. Tracing is performed at most one loop at a time.

Phase BI. The BI and BT phases are transitional phases between the interpreter and JIT. When program control reaches a loop back-edge, the interpreter calls the loop back-edge callback. Phase BI and BT are entered by a callback from phase I and T respectively. In phase BI, the loop back-edge counter is incremented, and upon reaching a certain threshold, i.e., *hot path*, the phase transitions to T if the loop has never been traced before. If the loop has been compiled, Tamarin invokes the native code (N).

Phase BT. This phase validates the recorded trace. The trace is valid only

if the current loop back-edge is the same as the one that initiated the tracing, i.e., from phase BI to T. If the trace was initiated by a different loop back-edge, then Tamarin discards the trace and releases its buffer. If the trace is valid, then Tamarin invokes NanoJIT (C). Because of the nature of its design, Tamarin only traces inner loops.

Phase C. NanoJIT optimizes and compiles a complete trace to native code at this phase. Since the trace is just a linear stream of statically typed SSA-based IR, it is optimized similarly to conventional compiler backends; e.g., NanoJIT performs constant propagation, CSE (common-subexpression elimination), and loop-invariant code-hoisting, and translates to machine code during linear scan register allocation. Guards are converted to conditional branches and compensation code to return to the interpreter. Once a trace has been compiled, it is registered with the loop back-edge address. Right after the compilation, Tamarin can invoke the machine code (N).

Phase N. In this phase, the compiled native code gets executed. If the trace is a true *hot* trace, the compiled code will run often enough to outweigh the cost of compilation. A guard may occasionally cause a trace to exit, for example, when a different control-flow path is taken or when a type check or bounds check fails. This condition is called a *side-exit*. The interpreter keeps a counter for each side-exit. When a side-exit is taken frequently, the interpreter starts tracing from the side-exit point. When the trace is recorded, it gets compiled and the guard instruction is patched to jump to the compiled side-exit code. If the native code is executed normally to the end of the loop, or the side-

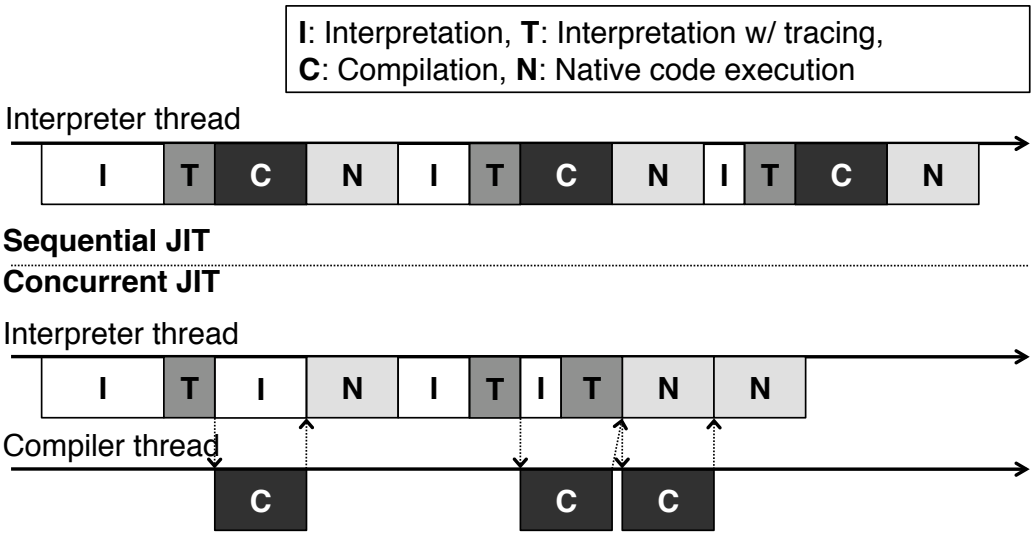


Figure 3.3: Example of sequential vs concurrent JIT execution flow.

exit was not hot enough, the interpreter phase transitions to I, i.e., normal interpretation.

3.3.2 Parallelism to Exploit

To design a proper synchronization mechanism to maximize the concurrency, we must understand what parallelism can be exploited. Figure 3.3 shows an example execution flow with a sequential and a concurrent JIT. In the concurrent JIT, as the compilation phase is offloaded to a separate thread, the interpreter is responsive and making progress while compilation happens, as is common for generic concurrent JIT compilers. For trace-based JIT, tracing must precede the compilation phase. If tracing can happen concurrently with compilation, subsequent compilation may start earlier, and deliver the native code faster. Furthermore, more hot paths can be compiled during the

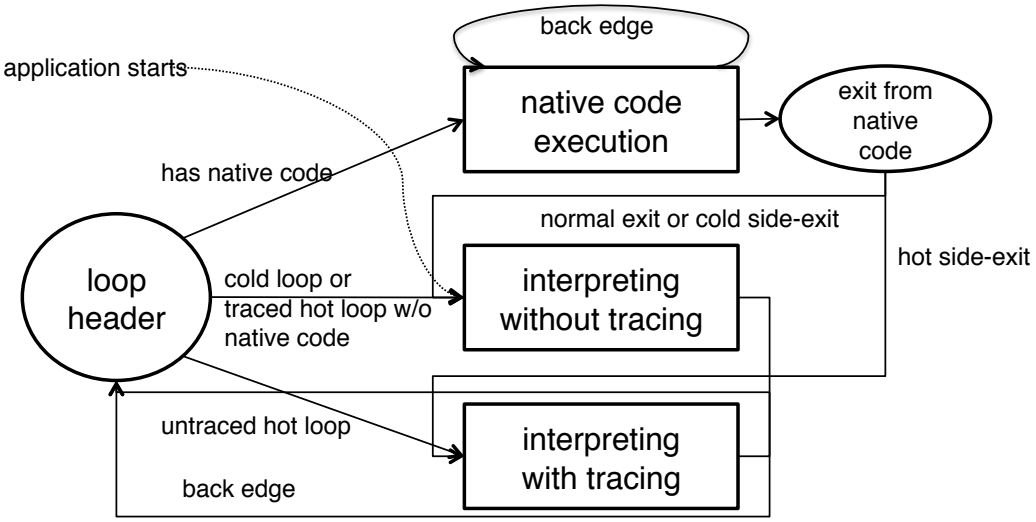


Figure 3.4: The interpreter state transition at a loop header.

execution. We can expect to achieve throughput improvements as well as a reduction in the pause time. The concurrent JIT also opens the possibility to do more aggressive optimizations without hurting pause time. The following sections explain how we designed the synchronization to achieve the parallelism shown in Figure 3.3.

3.3.3 Compiled State Variable

In this trace-based JIT compiler, the interpreter changes state at loop entry points. As shown in Figure 3.4, when the control flow reaches a loop entry point, the interpreter must identify four different states. First, if compiled native code exists for the loop, the interpreter calls it. The native code executes until the end of the loop or it takes a side-exit. Second, if the loop has never been traced and the loop is hot, i.e., it just crossed the frequency threshold,

the interpreter enables tracing and executes byte code. Identifying a hot loop path is explained in Section 3.2 in detail. Third, if tracing is currently enabled at the loop header, the interpreter disables it and requests compilation. While compiling the trace, the interpreter continues to execute the program. Fourth, if the loop is cold, the interpreter increments the associated counter and keeps on interpreting the byte codes.

Checking all these cases at a loop header requires a synchronization with the compiler thread. Otherwise, race conditions may cause overhead or incorrect execution. For example, the interpreter may make duplicate compilation requests, or trace the same loop multiple times. The simplest synchronization method is using a coarse-grained lock around the checking routine. However, the lock can easily be contended after a compilation request is made, especially with a short loop body, because the control reaches the loop header frequently. We could use a fine-grained lock for accessing each loop data structure. However, this approach is also infeasible because the native code for the loop can change as the trace tree grows, and holding a lock while executing the native code would stall the compiler too often.

To overcome these challenges, we design a lock-free synchronization technique using a *compiled state variable (CSV)*. We use a word size CSV for each loop, and align it not so as to cross the cache line. Thus, stores to it are atomic. The value of the CSV is defined as shown in Table 3.1. By following a simple but efficient protocol for incrementing the CSV value, the interpreter checks the state at the loop header *without any explicit synchronization*. The

Description	Action	CSV
Hot loop	Enable tracing	0
Cold loop	Normal interpretation	0
Trace enabled	Disable tracing and request compilation	0 to 1
Compilation already requested	Normal interpretation	1
Has native code	Call native code	2

Table 3.1: Value of Compiled State Variable(CSV) at a loop header.

initial value of CSV is zero. The interpreter traces and interprets in this state. Only the interpreter increments it from 0 to 1 when requesting a compilation. As it is a local change, the interpreter sees the value 1 on the subsequent operations before the compiler sees the value 1. The compiler changes the value from 1 to 2 to register the native code for the loop. Thus, when the interpreter reads the value 2, it is guaranteed that the native code is ready to call. Therefore, the pause time for waiting is almost zero for both the interpreter and the compiler, maximizing the concurrency.

When the compiler makes a JIT request, the trace buffer is pushed to a queue before it increments the CSV to 1. We use a simple synchronized FIFO queue for the JIT requests, because it is normally not contended. However, a generic, concurrent, lock-free queue for one producer and consumer [22] could always replace this queue.

3.3.4 Dynamic Trace Stitching

The trace-based JIT specializes types and paths, and injects guard instructions to verify the assumptions for the type and path of the trace.

Guards trigger side-exit code if the assumption is not met, and returns control back to the interpreter.

If two or more hot paths exist in a loop, the first hot path will be compiled normally, but the subsequent hot paths will frequently trigger guards. As explained in Section 3.2, the interpreter traces from the branch that caused the side-exit (branch trace), and compiles it. As more hot paths are revealed, trunk and branch traces form a trace tree. Recompiling the whole trace tree is good for the code quality, but the compilation time will grow quadratically if the whole trace tree is recompiled every time a new trace is attached to the tree. Also, this strategy would keep the trace buffer in memory for future recompilation, which is infeasible in memory constrained environments. Instead of recompiling the whole tree, we use a *trace stitching* technique. Trace stitching is a technique that compiles the new branch trace only, and patches the side-exit to jump to the branch trace native code.

Branch patching modifies code that is produced by more than one trace. Hence, it is probable that interpreter is executing the native code at the same time that the compiler wants to patch it. Naive use of a lock around the native code will incur a significant pause time on both the interpreter and the compiler. Waiting becomes a problem if time spent in the native code grows large, reducing the overall concurrency. The compiler may also make a duplicate copy of the code instead of patching, or delay the patching until the native code exits to the interpreter. Either method has inefficiencies, and we propose lock-free *dynamic trace stitching* for patching the branch. The key factor of

dynamic trace stitching is that a side-exit jump is a safe point where all variables are synchronized to the memory. We use each side-exit jump instruction as a placeholder for patching. When the compiler generates the native code for the branch trace, jumping to the previous side-exit target or jumping to the branch trace code does not change the program semantics. Therefore, if the patching is atomic, the compiler can patch the jump instruction directly without waiting for the interpreter. If the branch target operand is properly aligned, patching is done by a single store instruction. There is no harmful data race even without a lock. With these benign data races, the interpreter and the compiler run concurrently without pausing.

3.4 Implementation

We implemented our design in the open-source TamarinTracing Virtual Machine [51]. Our design implements TamarinTracing's sequential trace-based JIT and targets the 32bit x86 architecture.

Incrementing the Compiled State Variable The compiled state variable is piggybacked on both trunk and branch traces. We aligned the compiled state variable at a word granularity, and since the cache line is multiple of word size, CSV does not cross the cache line boundary. Therefore, it is safe to increment the variable without synchronization. We simply declared the CSV `volatile` to force a memory load.

Dynamic Trace Stitching A direct branch instruction(`jmp`) is 5 bytes in 32bit x86, and the last 4 bytes are the branch target operand. Since TamarinTracing VM's JIT compiler generates the machine code in reverse order, padding the branch instruction to align it with the cache line is difficult. Thus, we used compare-and-swap instruction to replace the branch target operand of the side-exit jump instruction.

GC Thread-Safety TamarinTracing uses a mark-sweep garbage collector, called *MMGC*, to manage application and VM objects. The current MMGC implementation is not thread-safe. To make the compiler concurrent, we must either make MMGC thread-safe or eliminate compiler objects from the MMGC heap. We choose the latter, and use explicit allocation and deallocation in the compiler, i.e., malloc and free. This change to explicit memory management improves the throughput by ∼10%. To isolate the improvement caused by modifying the JIT, we use this sequential JIT with explicit memory management as the baseline of our evaluation in Section 2.6. Our concurrent JIT is implemented on top of this baseline.

Implementation Correctness To validate the correctness of our implementation, we used Intel Thread Checker to detect harmful race conditions. We also ran the *acceptance test suite* used to verify sequential Tamarin VM. Our implementation passes the same test entries that Tamarin VM passes. Furthermore, our concurrent JIT runs the same set of SunSpider benchmarks

Benchmarks	Bytecode (bytes)	Compiled Traces	Compilation (%)	Native (%)	Interpreter (%)	Runtime (ms)
access-binary-trees	697	37	5.4	89.1	5.5	74
access-fannkuch	823	49	2.4	94.2	3.3	117
access-nbody	2,202	27	3.5	91.6	4.9	144
access-nsieve	543	14	1.4	96.8	1.7	56
bitops-3bit-bits-in-byte	414	6	4.0	89.7	6.3	12
bitops-bits-in-byte	385	15	1.5	96.5	2.1	40
bitops-bitwise-and	264	3	0.2	99.4	0.4	179
bitops-nsieve-bits	586	11	1.4	96.6	2.0	50
controlflow-recursive	504	35	8.3	84.5	7.3	28
crypto-aes	7,004	158	11.4	63.2	25.4	150
crypto-md5	5,470	6	24.6	17.4	58.0	120
math-cordic	832	9	1.8	95.0	3.1	32
math-partial-sums	758	11	1.3	93.2	5.5	41
math-spectral-norm	841	35	7.8	78.3	13.9	36
s3d-cube	4,918	188	8.4	41.6	50.0	155
s3d-morph	573	14	1.5	95.9	2.6	81
s3d-raytrace	7,289	147	9.3	68.1	22.6	170
string-fasta	1,426	22	1.9	95.6	2.5	141
string-validate-input	1,511	28	1.4	96.0	2.6	261

Table 3.2: Workload characterization of SunSpider benchmarks with sequential Tamarin JIT.

correctly as compared to the sequential JIT. Therefore, we believe our implementation produces the same results as the original Tamarin VM implementation.

3.5 Evaluation

3.5.1 Experiments Setup

We evaluate our implementation on three different configurations:

- Linux 2.6 and NPTL pthread library running on Intel Core 2 Quad 2.4GHz which has four cores and two 4MB shared L2 cache
- Windows and Win32 thread library running on Intel Core 2 Duo 2.4GHz which has two cores and 2MB shared L2 cache

- Mac OS X Leopard and pthread library on Intel Core 2 Duo 2.8GHz which has two cores and 4MB shared L2 cache.

Since it is easier to configure Linux to have minimal noise, we ran 50 runs and averaged the results. On other configurations, because of the inevitable perturbation of the OS services, we picked 10 best runs out of 50 and calculated the average. We present the 95% confidence interval assuming Student's t-distribution on all results to validate the statistical significance. For easy comparisons, all graphs are presented such that lower is better.

3.5.2 SunSpider Benchmarks Characterization

The SunSpider benchmark suite is a set of JavaScript programs intended to test performance [55]. It is widely used to test and compare JavaScript VMs on web browsers such as Firefox SpiderMonkey, Adobe ActionScript, and Google V8. Table 3.2 characterizes the benchmarks in the Linux configuration.

Figure 3.5 breaks down application execution time to compare the sequential and concurrent JIT. The first bar is the application with the sequential JIT, and the second bar isolates the application thread activity in the concurrent JIT: interpreter, native code, and the pause time caused by compilation request. The third bar is the compile time at the compiler thread. The y-axis value for concurrent JIT is normalized to the sequential JIT total execution time. Hence, bar 2 less than 100% is the speedup. The compilation time in bar 2 is the total pause time in concurrent JIT.

In most benchmarks, the concurrent JIT implementation improves both responsiveness and throughput. The pause time with the concurrent JIT is negligible, and speedups are noticeable in many benchmarks. In many cases, speedup is due to faster delivery of the native code. For example, in `crypto-aes`, the compilation time in both the sequential and concurrent versions is about the same, and the amount of speedup is mostly from the compiler being offloaded. The `s3d-cube` benchmark shows the most speedup, even though the compilation time in the concurrent JIT is almost twice as much. The interpreter requests more compilation and the resulting native code executes much faster. More tracing is performed concurrently with the compiler. As a result, time spent in the interpreter has reduced significantly, which accounts for the speedup.

Notice that the larger, more complex benchmarks (`crypto-aes`, `crypto-md5`, `s3d-cube`, `s3d-raytrace`) are influenced most by the concurrent JIT. This trend indicates as JavaScript programs grow in size and complexity, the concurrent JIT is likely to provide more benefits due to the need for an increase in native code execution.

3.5.3 Responsiveness

In this section, we evaluate application responsiveness using total, average, and maximum pause time. Total pause time for running a benchmark is a good indicator of how responsive the application is, and the average reflects the end-user experience. Many small pauses are better than one big pause in

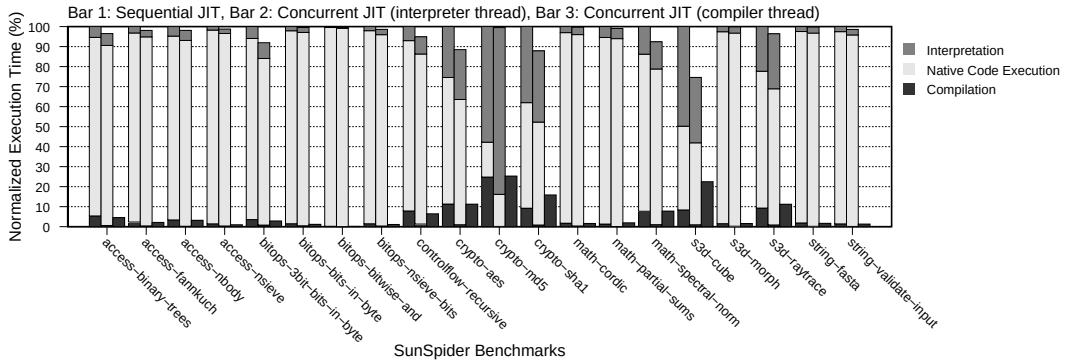


Figure 3.5: Average time break down in compilation, native code, and interpretation.

terms of responsiveness [13]. We compare maximum pause time, which is the most noticeable pause to the end-user, and we want it to be as low as possible.

Figure 3.6 demonstrates that our concurrent JIT implementation reduces both maximum and total pause time significantly. The y-axis is the pause time normalized to the pause time in the sequential JIT. A value of 1.0 means that the pause time is the same, and 0.1 means pause time is reduced by 90%. Tics at the top of each bar shows 95% confidence interval.

Geometric means in the Linux configuration show that we reduced the total pause time by 89% and 93% for the maximum, showing a huge improvement in responsiveness. Furthermore, the average pause is only 97% of the sequential JIT. Even in the worst case on Mac OS X with the `bitopts--bits-in-byte` benchmark, the total pause time is reduced by 50%. Yet, the average is reduced by 87%, which shows the implementation successfully avoided long pauses.

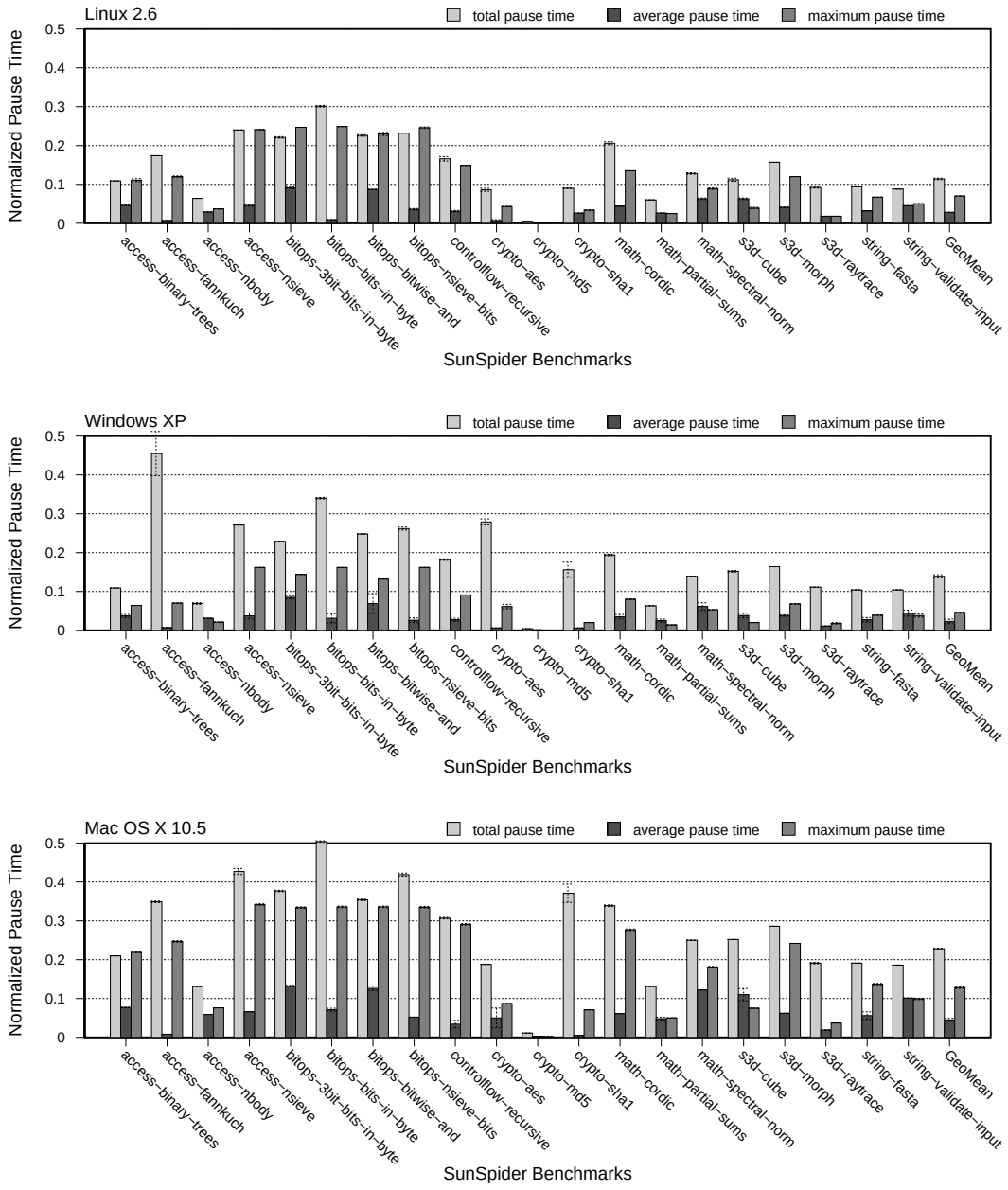


Figure 3.6: Pause time ratios of concurrent vs. sequential JITs.

As the compilation time per trace grows, it is likely that the concurrent JIT will reduce the pause time more. Table 3.2 shows that `crypto-md5` has the highest per trace compilation time-compiling 6 traces for 25% of the execution time, and achieves the best reduction in pause time: 99% for all three metrics.

3.5.4 Throughput

Improving responsiveness does not necessarily mean improving throughput. On the other hand, better responsiveness often requires sacrificing throughput, e.g., concurrent garbage collectors versus stop-the-world garbage collectors. However, we improved throughput as well as the responsiveness because our implementation executes the interpreter and tracing concurrently with the compiler.

Figure 3.7 shows the speedup for each configuration. The x-axis is the SunSpider benchmarks and the y-axis is the speedup normalized to the execution time with the sequential JIT. The concurrent JIT achieves 5–6% speedup on all platforms on average, and achieves up to 36% on `s3d-cube` on Windows. As explained in Section 3.5.2, the speedup in `s3d-cube` is due to increasing the number of compiled traces.

Another noticeable result is that the concurrent JIT is uniform in its improvement. Only two benchmarks have minor degradations on Mac OS X. The performance variation of `crypto-md5` among the platforms is due to the fact that it only spends 17% of the time in compiled native code, and only compiles 6 traces. The remaining 18 programs improve or stay the same on

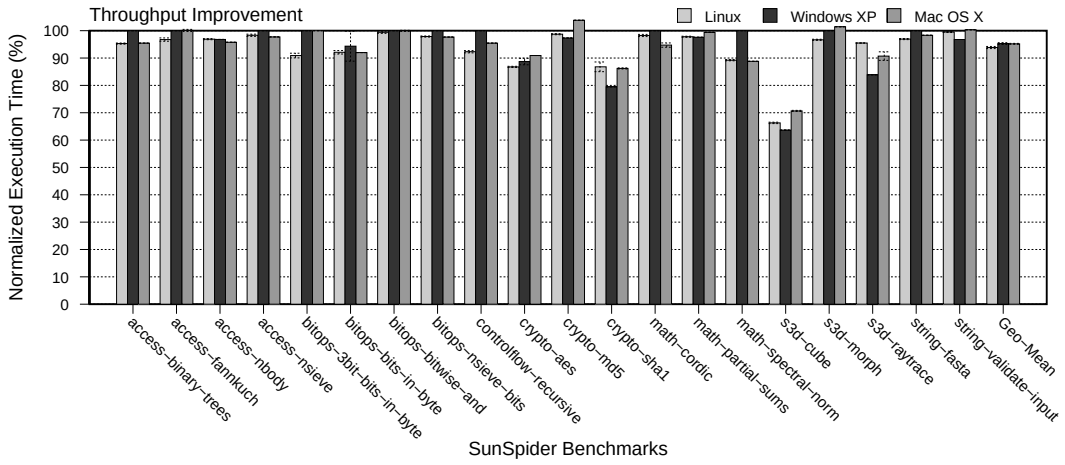


Figure 3.7: Execution time improvement with concurrent JIT.

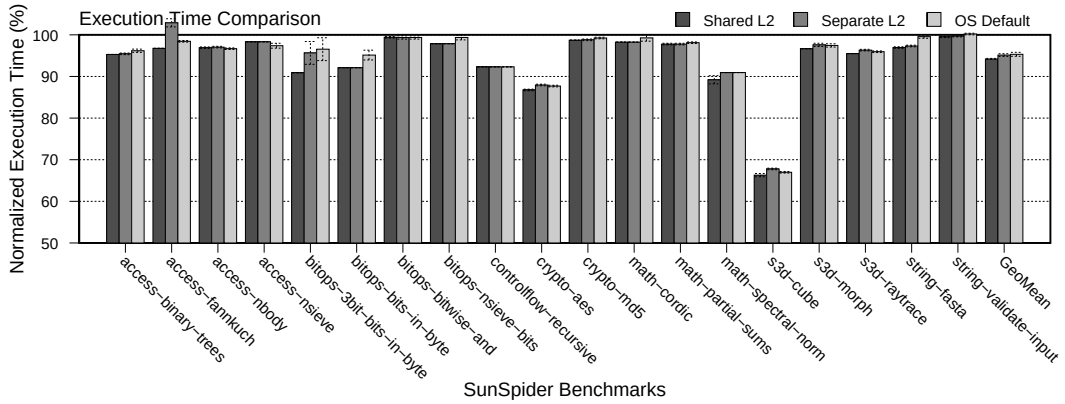


Figure 3.8: Performance impact on various core configurations.

all platforms.

3.5.5 Multicore Impact on Performance

In this section, we present the thread scheduling impact in the context of the compiler and interpreter threads in multicore systems. A major difference between multicore and traditional off-chip multiprocessors is the memory

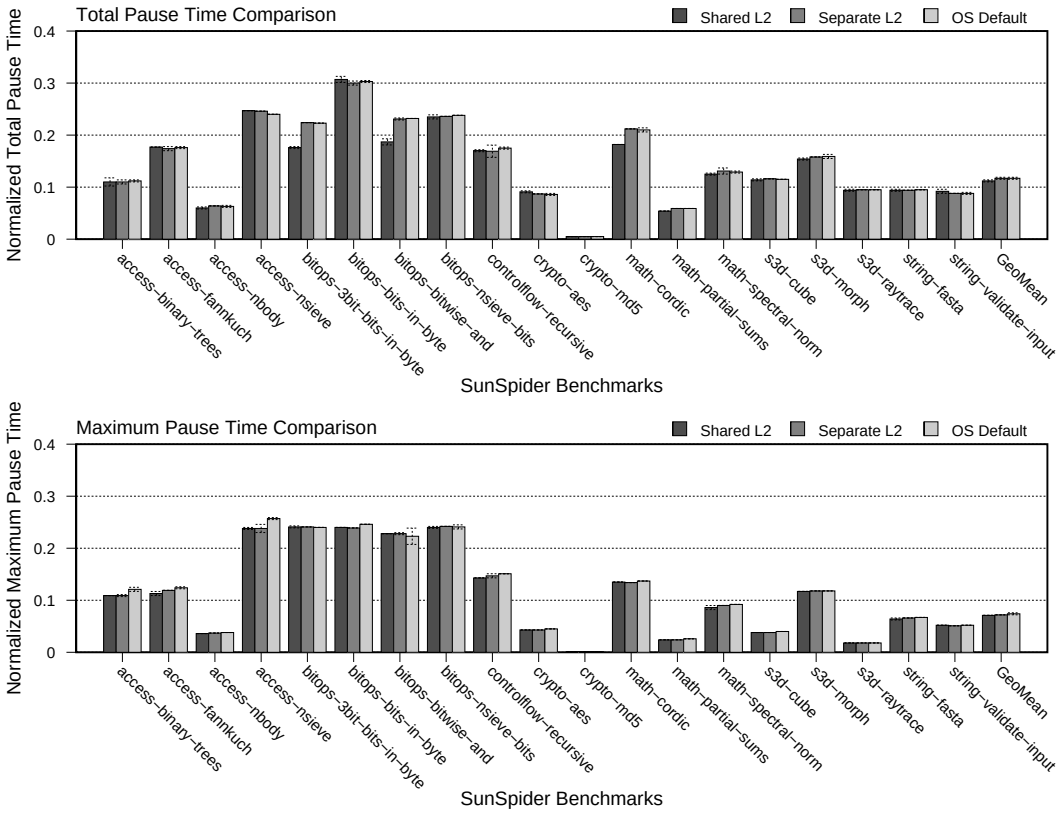


Figure 3.9: Pause time impact on various core configurations.

latency. Especially in shared-cache multicore systems, load and store latency is significantly less than off-chip multiprocessors. Moreover, shared-cache multicores add nonuniformity to load/store latency, which makes the performance less predictable.

Intel Core 2 Quad processor we used for Linux configuration is a shared-cache multicore where each pair of cores shares L2. Thus, core 0 and 1 communicate via the L2 cache, but core 0 and 2 go through the interconnect network. In the concurrent JIT, the compiler thread is a producer whose output would

be transferred to the instruction cache in the interpreter thread. Hypothetically, we can expect that lower communication latency will better improve the performance.

Figure 3.8 shows how thread assignment influences the performance. Lack of operating support for hard pinning a thread to a given core, we could only hint the OS scheduler using `pthread_setaffinity_np` provided by NPTL pthread library. The first bar corresponds to the case where compiler and interpreter threads are configured to share L2. The second bar represents the case where compiler and interpreter do not share caches. The third bar is measured without any hint to the scheduler. In all benchmarks, the shared L2 configuration performed the best, though the benefit is small for most of the benchmarks. However, in a couple of benchmarks, such as `bitops-3bit-bits-in-byte`, show a large difference. Performance is slightly stable in the shared L2 configuration. You can observe that confidence interval is the narrowest in this configuration. Therefore, it is always good to put both compiler and interpreter threads on the cores that shares cache.

We also compare total and maximum pause time in Figure 3.9. As in performance comparison, overall reduction for shared L2 configuration in total and maximum pause time is modest, and the measurements are slightly more stable. Even though improvement in overall pause time is infinitesimal, it is as significant as 5% for some benchmarks.

Our results indicate that the choice of the right pair of cores for the compiler and interpreter threads can influences performance. The impact may

increase in multicore systems with more complex memory hierarchy.

3.6 Conclusion and Interpretation

In this chapter, we showed that even though JavaScript language itself is currently single-threaded, both its throughput and responsiveness can benefit from multiple cores with our concurrent JIT compiler. This improvement is achieved by running the JIT compiler concurrently with the interpreter. Our results show that most of the compile-time pauses can be eliminated, resulting in a total, average, and maximum reduction in pause time by 88%, 97%, and 93%, respectively. Moreover, the throughput is also increased by an average of 5%, with a maximum of 36%. Our work on this chapter demonstrates a way to exploit multicore hardware to improve application performance and responsiveness by offloading system tasks.

Chapter 4

Conclusion

This dissertation concludes with a summary of the work presented and a discussion of future work.

Scalability of managed runtime systems for managed languages is an urgent problem in the multicore era. This thesis improves the managed runtime systems scalability in two ways. First, we presented a concurrent dynamic analysis framework, which demonstrates how to implement a concurrent analysis thread with very little perturbation of the application. We introduced Cache-friendly Asymmetric Buffering (CAB) that effectively offloads analysis data from the application's critical path to a separate analysis thread minimizing the microarchitectural side-effects. Second, we design and implement a concurrent trace-based just-in-time compilation that utilizes extra cores, which otherwise are not used by the application for the single-threaded JavaScript language. Our concurrent trace-based JIT achieves both throughput and responsiveness improvement. These approaches are ready for immediate adoption in real world managed runtimes.

4.1 Future Work

This dissertation contributed to making managed runtimes more scalable. However, even with these contributions, managed runtimes are not yet scalable enough. The community needs to further characterize and redesign the entire managed runtime systems to attain scalability. This problem is both an urgent and promising research direction needed to make a solid software infrastructure for applications executing on multicore hardware.

Achieving parallelism and concurrency algorithmically has been studied in some areas, such as concurrent garbage collection. As our results in this dissertation indicate, scalability is prone to microarchitectural side-effects, and for the most part, researchers have not considered these effects when designing runtime components. Blackburn et al. recently improved stop-the-world garbage collector by efficiently managing cache lines [9]. Yet, no one has designed concurrent garbage collectors, not to mention other runtime components, that minimize application perturbation due to microarchitectural side-effects.

Taking advantages of extra cores is another promising direction. Our concurrent dynamic analysis framework showed how to efficiently offload analysis data from the application. Further research should explore other clients, such as debugging, security, and software support, that can build on top of this framework. We believe that our framework will reduce the overhead and increase the accuracy for these clients.

In particular, future work should pursue improving scalability of existing managed runtime components, such as garbage collectors and thread scheduling. We believe if successful, this research would benefit managed language developers and users by increasing scalability on future processors, and this has potential to help this software enter into a new virtuous cycle in the multicore era.

Bibliography

[1] B. Alpern, D. Attanasio, J. J. Barton, M. G. Burke, P.Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. Flynn Hummel, D. Lieber, V. Litvinov, M. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. Shepherd, S. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM System Journal*, 39(1):211–238, February 2000.

[2] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, and William E. Weihl. Continuous profiling: Where have all the cycles gone? *ACM Transactions on Computer Systems*, 15(4):357–390, 1997.

[3] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive optimization in the Jalapeño JVM. In *ACM Conference on Object–Oriented Programming, Systems, Languages, and Applications*, pages 47–65, Minneapolis, MN, October 2000.

[4] Matthew Arnold and David Grove. Collecting and exploiting high-accuracy call graph profiles in virtual machines. In *ACM/IEEE International Symposium on Code Generation and Optimization*, pages 51–62, San Jose, CA, March 2005.

[5] Matthew Arnold and Barbara G. Ryder. A framework for reducing the cost of instrumented code. In *ACM Conference on Programming Language Design and Implementation*, pages 168–179, Snowbird, UT, June 2001.

[6] Thomas Ball and James R. Larus. Efficient path profiling. In *ACM/IEEE International Symposium on Microarchitecture*, pages 46–57, Paris, France, December 1996.

[7] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM Conference on Object–Oriented Programming, Systems, Languages, and Applications*, pages 83–89, Portland, OR, October 2006.

[8] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java benchmarking development and analysis (extended version). Technical Report TR-CS-06-01, Dept. of Computer Science, Australian National University, 2006.

http://www.dacapobench.org.

[9] S. M. Blackburn and K. S. McKinley. Immix: A mark-region garbage collector with space efficiency, fast collection, and mutator locality. In *ACM Conference on Programming Language Design and Implementation*, pages 22–32, Tuscon, AZ, June 2008.

[10] Michael D. Bond and Kathryn S. McKinley. Continuous path and edge profiling. In *ACM/IEEE International Symposium on Microarchitecture*, pages 130–140, Barcelona, Spain, November 2005.

[11] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *ACM/IEEE Conference on Supercomputing*, pages 1–13, Article 42, Dallas, TX, 2000.

[12] D. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Massachusetts Institute of Technology, 2004.

[13] Perry Cheng, Robert Harper, and Peter Lee. Generational stack collection and profile-driven pretenuring. In *ACM Conference on Programming Language Design and Implementation*, pages 162–173, Montreal, Canada, 1998.

[14] Jim Chow, Tal Garfinkel, and Peter M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *USENIX Annual Technical Conference*, pages 1–14, Boston, MA, 2008.

[15] Intel Corporation. Vtune: Visual tuning environment. `http://software.intel.com/en-us/intel-vtune`.

[16] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *ACM Symposium on Principles of Programming Languages*, pages 297–302, Salt Lake City, UT, 1984.

[17] Andreas Gal, Michael Bebenita, Mason Chang, and Michael Franz. Making the Compilation "Pipeline" Explicit: Dynamic Compilation Using Trace Tree Serialization. Technical Report 07-12, University of California, Irvine, 2007.

[18] Andreas Gal, Brendan Eich, Mike Shaver, Daid Anderson, Blake Kaplan, Graydon Hoare, David mandelin, Boris Zbarsky, Jason orendorff, jesse Ruderman, Edwin Smith, Rick Reitmaier, Mohammad R. Haghighat, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based just-in-time type specialization for dynamic languages. In *ACM Conference on Programming Language Design and Implementation*, Dublin, Ireland, 2009.

[19] Andreas Gal and Mozilla Foundation. TraceMonkey. `https://wiki.mozilla.org/JavaScript:TraceMonkey`.

[20] Andreas Gal, Christian W. Probst, and Michael Franz. HotpathVM: An effective JIT compiler for resource-constrained devices. In *ACM International Conference on Virtual Execution Environments*, pages 144–153, Ottawa, Canada, 2006.

[21] Kourosh Gharachorloo and Phillip B. Gibbons. Detecting violations of sequential consistency. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 316–326, Hilton Head, SC, 1991.

[22] John Giacomoni, Tipp Moseley, and Manish Vachharajani. FastForward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 43–52, Salt Lake City, UT, USA, 2008.

[23] Danny Goodman. *JavaScript Bible*. IDG Books Worldwide, Inc., Foster City, CA, 3rd, edition, 1998.

[24] Google Inc. V8. `http://code.google.com/p/v8`.

[25] Jungwoo Ha, Matthew Arnold, Stephen M. Blackburn, and Kathryn S. McKinley. A concurrent dynamic analysis framework for multicore hardware. In *ACM Conference on Object–Oriented Programming, Systems, Languages, and Applications*, Orlando, FL, 2009.

[26] Jungwoo Ha, Matthew Arnold, Stephen M. Blackburn, and Kathryn S. McKinley. A concurrent dynamic analysis framework for multicore hardware. Technical Report TR-09-24, The University of Texas at Austin, 2009.

[27] Jungwoo Ha, Mohammad Haghighat, Shengnan Cong, and Kathryn S. McKinley. A concurrent trace-based just-in-time compiler. In

Workshop on Parallel Execution of Sequential Programs on Multicore Architecture, pages 47–54, Austin, TX, 2009.

[28] Jungwoo Ha, Christopher J. Rossbach, Jason V. Davis, Indrajit Roy, Hany E. Ramadan, Donald E. Porter, David L. Chen, and Emmett Witchel. Improved error reporting for software that uses black-box components. In *ACM Conference on Programming Language Design and Implementation*, pages 101–111, San Diego, CA, 2007.

[29] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *ACM/IEEE International Conference on Software Engineering*, pages 291–301, Orlando, FL, 2002.

[30] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Language Systems*, 13(1):124–149, 1991.

[31] M. Hirzel and T. Chilimbi. Bursty tracing: A framework for low-overhead temporal profiling. In *ACM Workshop on Feedback-Directed and Dynamic Optimization*, pages 117–126, December 2001.

[32] Urs Hölzle and David Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *ACM Conference on Programming Language Design and Implementation*, pages 326–336, Orlando, FL, 1994.

[33] Urs Hölzle and David Ungar. Reconciling responsiveness with performance in pure object-oriented languages. *ACM Transactions on Programming Languages and Systems*, 18(4):355–400, 1996.

[34] Chandra Krintz, David Grove, Derek Lieber, Vivek Sarkar, and Brad Calder. Reducing the overhead of dynamic compilation. *Software: Practice and Experience*, 31:200–1, 2001.

[35] Prasad Kulkarni, Matthew Arnold, and Michael Hind. Dynamic compilation: the benefits of early investing. In *ACM International Conference on Virtual Execution Environments*, pages 94–104, San Diego, CA, 2007.

[36] Monica S. Lam, Michael Martin, Benjamin Livshits, and John Whaley. Securing web applications with static and dynamic information flow tracking. In *ACM Workshop Partial Evaluation and Semantics-Based Program Manipulation*, pages 3–12, San Francisco, CA, 2008.

[37] Leslie Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Language Systems*, 5(2):190–222, 1983.

[38] W. Lin, S. K. Reinhardt, and D. Burger. Reducing DRAM latencies with an integrated memory hierarchy design. In *IEEE International Symposium on High Performance Computer Architecture*, pages 302–312, Nuevo Leone, Mexico, January 2001.

[39] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM Conference on Programming Language Design and Implementation*, pages 190–200, Chicago, IL, 2005.

[40] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using PQL: a Program Query Language. In *ACM Conference on Object–Oriented Programming, Systems, Languages, and Applications*, pages 365–383, San Diego, CA, 2005.

[41] H. Massalin and C. Pu. Threads and input/output in the synthesis kernel. In *ACM Symposium on Operating Systems Principles*, pages 191–201, Litchfield Park, AZ, 1989.

[42] Mark Moir, Daniel Nussbaum, Ori Shalev, and Nir Shavit. Using elimination to implement scalable and lock-free fifo queues. In *ACM Symposium on Parallelism in Algorithms and Architectures*, pages 253–262, Las Vegas, NV, 2005.

[43] Tipp Moseley, Alex Shye, Vijay Janapa Reddi, Dirk Grunwald, and Ramesh Peri. Shadow Profiling: Hiding instrumentation costs with parallelism. In *ACM/IEEE International Symposium on Code Generation and Optimization*, pages 198–208, Washington, DC, 2007.

[44] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Conference on Programming Language Design and Implementation*, pages 89–100, San Diego, CA, 2007.

[45] Michael Paleczny, Christopher Vick, and Cliff Click. The Java HotSpot server compiler. In *Java Virtual Machine Research and Technology Symposium*, Monterey, CA, April 2001. Sun Microsystems.

[46] Mikael Pettersson. Linux Intel/x86 performance counters, 2003. `http://user.it.uu.se/mikpe/linux/perfctr/`.

[47] Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. McKinley, and Emmett Witchel. Laminar: Practical fine-grained decentralized information flow control. In *ACM Conference on Programming Language Design and Implementation*, pages 63–74, Dublin, Ireland, 2009.

[48] R. Shetty, M. Kharbutli, Y. Solihin, and M. Prvulovic. HeapMon: a helper-thread approach to programmable, automatic, and low-overhead memory bug detection. *IBM Journal of Research and Development*, 50(2/3):261–275, 2006.

[49] Standard Performance Evaluation Corporation. *SPECjvm98 Documentation*, release 1.03 edition, March 1999.

[50] Vijay Sundaresan, Daryl Maier, Pramod Ramarao, and Mark Stoodley. Experiences with Multi-threading and Dynamic Class Loading in a Java Just-In-Time Compiler. In *ACM/IEEE International Symposium on Code Generation and Optimization*, pages 87–97, New York, NY, 2006.

[51] Tamarin. Tamarin Project. `http://www.mozilla.org/projects/tamarin/`.

[52] Steven Wallace and Kim Hazelwood. SuperPin: Parallelizing dynamic instrumentation for real-time performance. In *ACM/IEEE International Symposium on Code Generation and Optimization*, pages 209–220, San Jose, CA, 2007.

[53] Z. Wang, K. S. McKinley, A. Rosenberg, and C. C. Weems. Using the compiler to improve cache replacement decisions. In *ACM/IEEE International Conference on Parallel Architectures and Compilation Techniques*, pages 199–208, Charlottesville, VA, September 2002.

[54] WebKit. SquirrelFish Extreme. `http://webkit.org/blog/`.

[55] WebKit. SunSpider JavaScript Benchmark. `http://webkit.org/perf/sunspider-0.9/sunspider.html`.

[56] Chun Yuan, Ni Lao, Ji-Rong Wen, Jiwei Li, Zheng Zhang, Yi-Min Wang, and Wei-Ying Ma. Automated known problem diagnosis with event traces. In *ACM European Conference on Computer Systems*, pages 375–388, Leuven, Belgium, 2006.

[57] Qin Zhao, Ioana Cutcutache, and Weng-Fai Wong. PiPA: Pipelined profiling and analysis on multi-core systems. In *ACM/IEEE International Symposium on Code Generation and Optimization*, pages 185–194, Boston, MA, 2008.

[58] Pin Zhou, Feng Qin, Wei Liu, Yuanyuan Zhou, and Josep Torrellas. iWatcher: Efficient Architectural Support for Software Debugging. In *ACM/IEEE International Symposium on Computer Architecture*, pages 224–235, München, Germany, June 2004.

Vita

Jungwoo Ha was born in Seoul, Korea on 26 Sep 1976. He is married to Songhee Lee since 2003, and they have two children, Daniel and Timothy. He graduated from Seoul Science High School in Seoul, Korea, in 1995. He received the degree of Bachelor of Science in Computer Science and Engineering from Seoul National University in 2002. He entered the Ph.D. program at the University of Texas at Austin in 2003. Prior to joining the Ph.D. program, he worked at several startup companies for 5 years as a research staff and a R&D manager.

And they that be wise shall shine as the brightness of the firmament;and they turn many to righteousness as the stars for ever and ever. (Daniel 12:3)

Permanent address: 4708 Playfield St.
Annandale, Virginia 22003

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.