A Concurrent Dynamic Analysis Framework for Multicore Hardware *

Jungwoo Ha
The University of Texas at Austin
University of Southern California
Information Sciences Institute East
jha@east.isi.edu

Matthew Arnold
IBM T.J. Watson Research
marnold@us.ibm.com

Stephen M. Blackburn
Australian National University
Steve.Blackburn@anu.edu.au

Kathryn S. McKinley
The University of Texas at Austin
mckinley@cs.utexas.edu

Abstract

Software has spent the bounty of Moore's law by solving harder problems and exploiting abstractions, such as high-level languages, virtual machine technology, binary rewriting, and dynamic analysis. Abstractions make programmers more productive and programs more portable, but usually slow them down. Since Moore's law is now delivering multiple cores instead of faster processors, future systems must either bear a relatively higher cost for abstractions or use some cores to help tolerate abstraction costs.

This paper presents the design, implementation, and evaluation of a novel concurrent, configurable dynamic analysis framework that efficiently utilizes multicore cache architectures. It introduces Cache-friendly Asymmetric Buffering (CAB), a lock-free ring buffer that implements efficient communication between application and analysis threads. We guide the design and implementation of our framework with a model of dynamic analysis overheads. The framework implements exhaustive and sampling event processing and is analysis-neutral. We evaluate the framework with five popular and diverse analyses, and show performance improvements even for lightweight, low-overhead analyses.

Efficient inter-core communication is central to high performance parallel systems and we believe the CAB design gives insight into the subtleties and difficulties of attaining it for dynamic analysis and other parallel software.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Run-time environments

General Terms Experimentation, Performance, Measurement

Keywords Dynamic Analysis, Profiling, Multicore, Instrumentation

* This work is supported by ARC DP0666059, NSF CNS-0917191, NSF CCF-0811524, NSF CNS-0719966, NSF CCF-0429859, Intel, IBM, and Google. Any opinions, findings and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

OOPSLA 2009, October 25–29, 2009, Orlando, Florida, USA.

1. Introduction

Dynamic analysis is a base technology for performance optimization [1, 9, 25], debugging [15, 21, 24], software support [13, 31], and security [18, 22]. 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 paper explores the design and implementation of a dynamic analysis framework that exploits under-utilized cores by executing 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 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. Unfortunately, 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 cost, bandwidth between cores, false sharing between caches, coherence traffic, and synchronization between the producer and consumer threads.

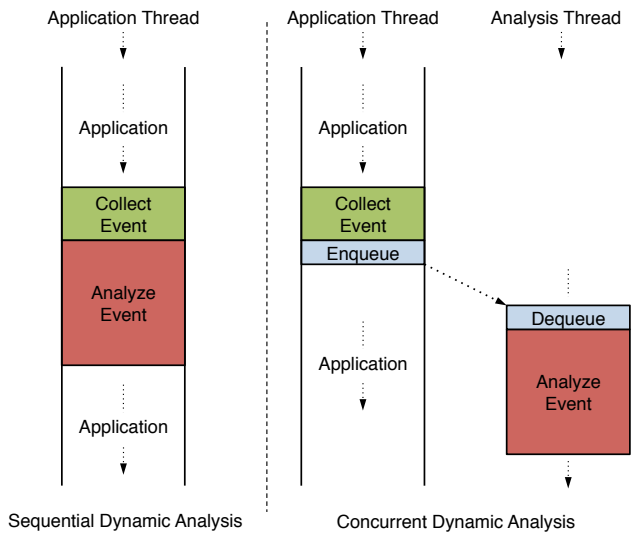


Figure 1. Generic sequential dynamic analysis versus concurrent dynamic analysis.

This paper's 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 [32] and FastForward concurrent lock-free queues [12], 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`.

In summary, the contributions of this paper 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 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. Related Work

Since there is a lot of research on dynamic analysis, we focus on differences with the most closely related research, 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 [32]. 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. This paper 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 [12]. 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 [23, 29]. 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 [10]. 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.

Recent work suggests hardware support for low-overhead dynamic analysis. HeapMon uses an extra helper thread to decouple memory bug monitoring [27]. 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 support, and we assume less about the class of analysis.

iWatcher leverages hardware assisted thread-level speculation to reduce the overhead of monitoring program locations [33]. The platform offers general debugging analysis, but low-overhead is only guaranteed with hardware support. Current multicore processors do not support thread-level speculation.

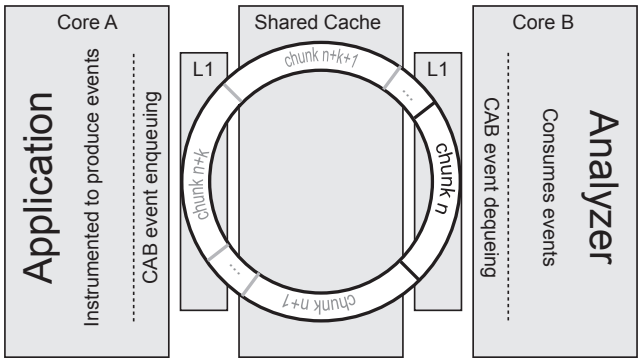


Figure 2. Cache-friendly Asymmetric Buffering (CAB) in a concurrent dynamic analysis framework.

Our dynamic analysis framework supports both exhaustive and sampling analysis of events. Prior work presented designs for low-overhead sampling of instrumentation [2, 3, 7, 17], 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.

3. Concurrent Dynamic Analysis Framework

As shown in Figure 1, dynamic analysis systems include 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 they may be distinct, executing concurrently in separate threads. 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 presents the overview of our framework and 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.

3.1 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.

3.1.1 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 [11, 12, 19]. Specifically, the common case enqueue and dequeue operations can be implemented without locks, as wait-free operations [16]. Of course, the operations are not actually wait-free if the desired semantics require that the producer block on a full buffer and that 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 reasonable 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 3.3.

```
1  while (*bufptr != CLEAR) {
2    if (*bufptr == MAGIC)
3      bufptr = &buffer;   // wrap back to start
4    if (*bufptr != CLEAR)
5      block();            // busy, back off
6  }
7  *bufptr++ = data;       // enqueue data
```

(a) Enqueueing events in application code

```
1  block() {
2    spin_wait();
3    pollptr = SKIP(bufptr, CHUNK_SIZE * 2);
4    while (*pollptr != CLEAR) {
5      if (isInvokedGC())
6        thread_yield();   // must cooperate
7      else
8        sleep(n);
9    }
10 }
```

(b) Blocking the application

```
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    /* consume & clear entire chunk */
8    consume_chunk(chunk_num);
9    chunk_num = next(chunk_num)
10 }
```

(c) Dequeueing events in analysis code

Figure 3. Enqueueing and dequeueing pseudo-code.

3.1.2 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 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 3(b) is executed via a call. Note that all of this code

is lock-free, and that in the common case, just a single conditional branch is executed (line 1 of Figure 3(a)). As shown later in Figure 6(a) and Figure 6(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 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 3(a), and the consumer sets the sentinel `CLEAR` when it consumes the chunk in line 8 of Figure 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 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 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 3(c)). The size of a chunk is a dynamically configurable option (`chunk_size` in Figure 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 3(b)).

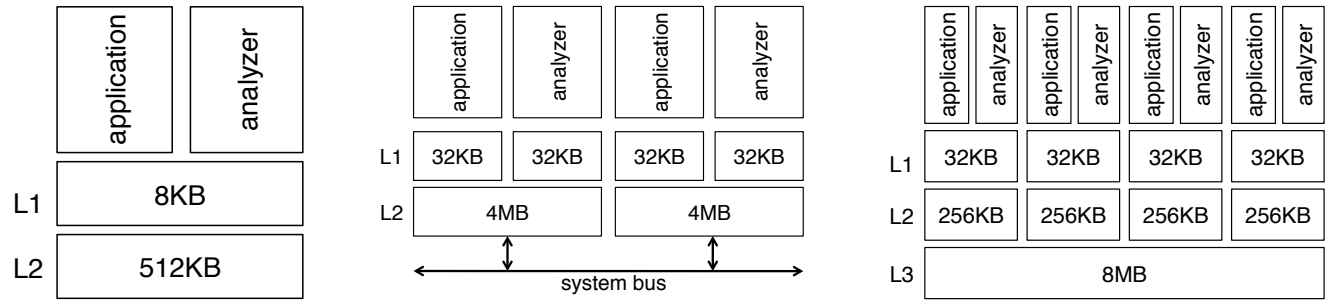
In this design, the consumer minimizes overhead and synchronization by dequeueing and processing one chunk at a time (line 8 of Figure 3(c)), reducing spinning and checking without affecting the fine-grained producer activity. The analysis happens in the call to `consume_chunk()` at line 8. If the analyzer 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 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.

3.1.3 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 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 4 shows three Intel hardware generations, which comprise our experimental platforms. (Section 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



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

Figure 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.

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 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 mechanisms that mark cache lines LRU [20, 30]. CAB could then reduce its cache footprint and thus its influence on the application, while still benefiting from sharing.

3.2 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 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 [3], 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 5(a) shows pseudocode for the application thread when in sampling mode, and should be compared to Figure 3(a). The code consists of the minimal instructions required to insert an element into a CAB buffer.

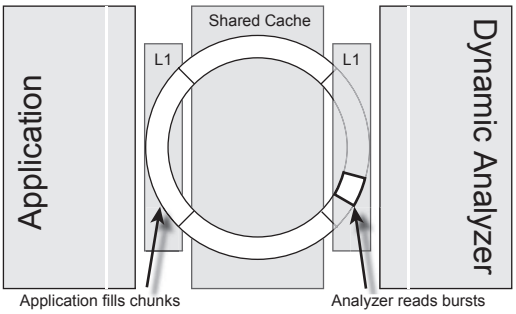
Dequeueing Figure 5(b) shows pseudocode for dequeueing in sampling mode. Compared to exhaustive mode dequeueing (Figure 3(c)), there are two differences. First, each

```
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 5. Sampling mode. The application does not block and the profiler may use bursty sampling.

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.

Consumers read events in bursts to maximize cache locality, as shown in Figure 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 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.

3.3 Impact of Single-Producer, Single-Consumer.

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.

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.

4. 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**same 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:

$$E_s = A + A_i + P + I_{ap} \tag{1}$$

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:

$$E_c^A = A + A_i + A_q \tag{2}$$

and when the analyzer dominates:

$$E_c^P = P + P_c \tag{3}$$

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:

$$E_c = max(E_c^A, E_c^P) \tag{4}$$

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:

$$A + A_i + A_q \geq P + P_c \tag{5}$$

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

$$E_s - E_c^A = P + I_{ap} - A_q \quad \geq \quad 0 \tag{6}$$

$$P + I_{ap} \quad \geq \quad A_q \tag{7}$$

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 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 5,

$$P - A_q \leq A + A_i - P_c \tag{8}$$

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: $E_s - E_c^P$:

$$E_s - E_c^P = A + A_i + I_{ap} - P_c \geq 0 \quad (9)$$

$$A + A_i + I_{ap} \geq P_c \quad (10)$$

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 is executed 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 is highly dependent on the particular analyses. Instead, we focus on minimizing communication overhead as part of a general framework.

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.

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, which we describe below. 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 and is therefore preferable. 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 garage collection times [5], 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.

```
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 6. 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.

Native Threading 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 in 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.

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) and 4) are analysis-specific and are described below in Section 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 6 shows x86 assembly code for enqueueing in both exhaustive and sampling modes. Note the simplicity of the common case code (Figures 6(a) and 6(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 6(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 a map that can enumerates 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 6(b)), which explicitly checks whether a GC yield is necessary (line 5, Figure 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 6(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. 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.

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 [13]. 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 [4]. 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 [7]. 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 [7]. 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 [21]. 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.

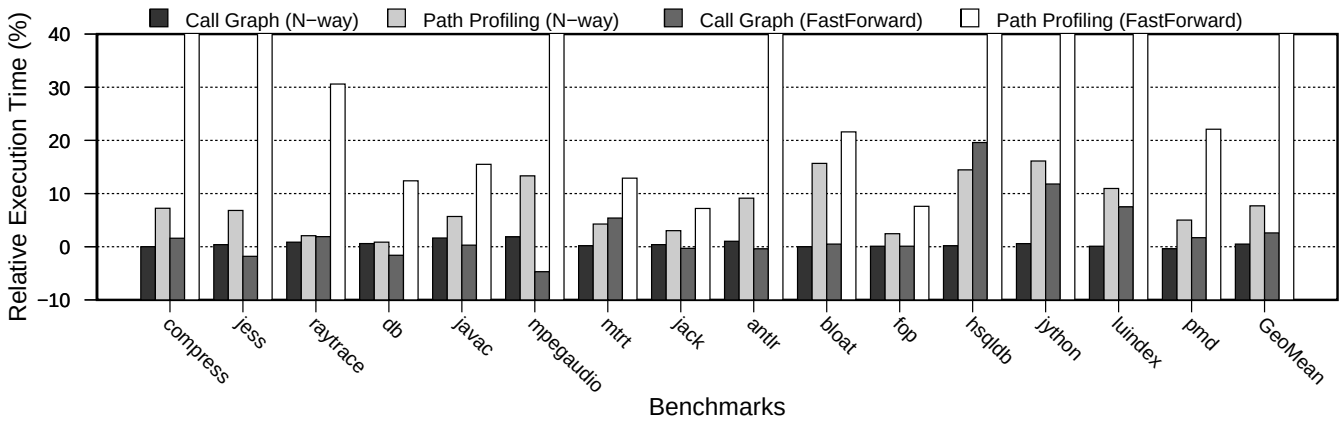


Figure 7. Performance of N-way buffering [32] and FastForward [12] relative to CAB. We use N:M threading on an Intel Core 2 Quad. The Y-axis is normalized to CAB's execution time.

6. Evaluation

We next describe our benchmarks, hardware, operating system, experimental design, and data analysis methodology. Section 6.1 compares CAB to other buffering mechanisms. Section 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 6.3 evaluates buffer size scalability and Section 6.4 evaluates sampling mode. Finally, Section 6.5 explores the importance of modern shared cache architectures to concurrent dynamic analysis.

Benchmarks. We use the SPECjvm98 benchmarks [28] and 9 of 11 DaCapo Java (v. 2006-10-MR2) benchmarks [6]. The DaCapo suite is a recently developed suite of substantial real-world open source Java applications. The characteristics of both are described elsewhere [**?**]. `mtrt` in SPECjvm98 and `hsqldb`, `lusearch`, and `xalan` in DaCapo benchmark suite are multi-threaded benchmarks. 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 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 [26] on all of the processors. We used PAPI for performance counter measurements [8]. 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 [6]). 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 size to 4 times the minimum required for the uninstrumented benchmark.

6.1 CAB versus Other Buffering Mechanisms

We start by comparing CAB with conventional N-way buffering and FastForward's concurrent lock-free queue [12]. 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 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*

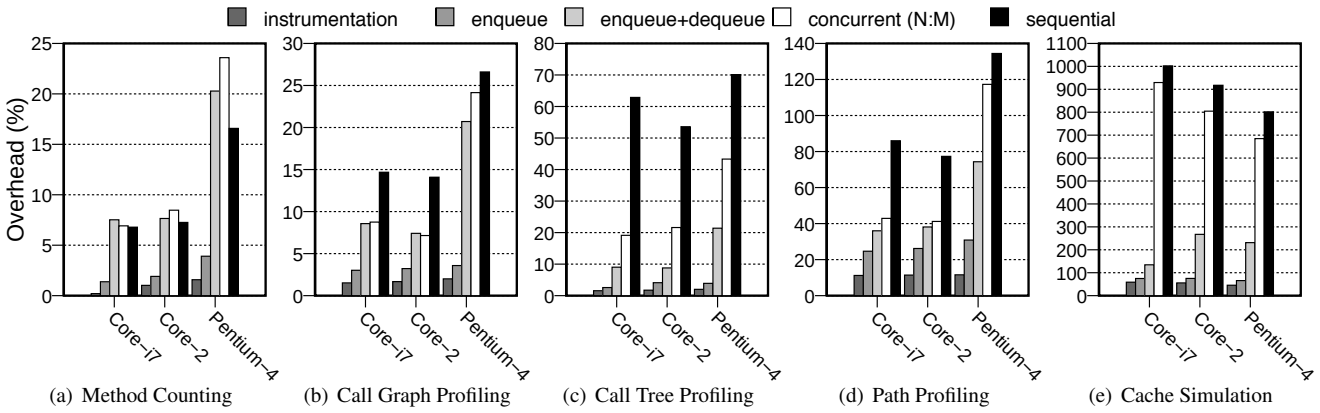


Figure 8. Exhaustive mode overhead with performance break-down, averaged over all benchmarks (N:M threading model).

distance of six cache lines, the consumer begins processing events again. We use the same algorithm and parameters as specified in the FastForward paper [12].

Our implementation of N-way buffering improves in two ways over PiPA [32]. First, we removed 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 7 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 are discussed in detail in Section 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 $\sim$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.

6.2 Exhaustive Mode Overhead

We now examine the performance of CAB in more detail, starting with exhaustive mode. Figure 8 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 the appendix. All measurements are relative to the application without any instrumentation or analysis, i.e., the application time A from our model in Section 4. 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 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 8(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

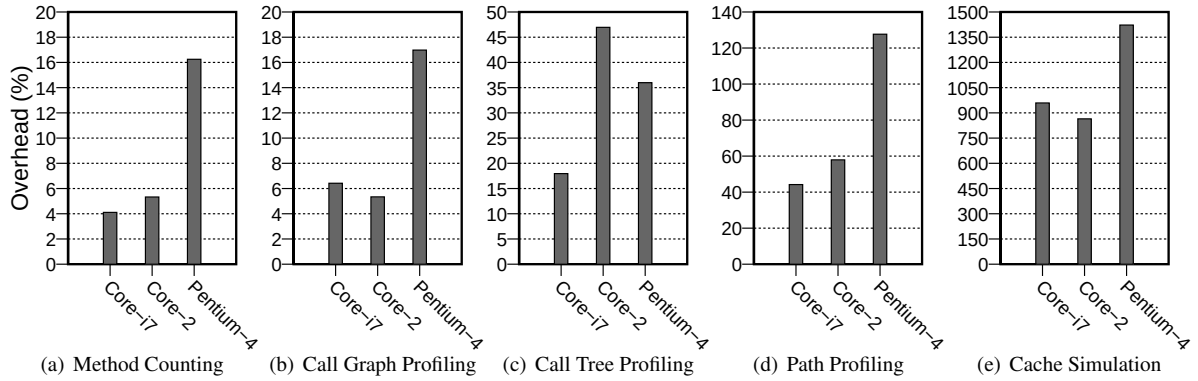


Figure 9. Exhaustive mode overhead, average over all benchmarks (native threading model).

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 9 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 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.

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, 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.

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 [32], the increased headroom of the larger buffer will come at the cost of degraded average performance.

Figure 10 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. The appendix con-

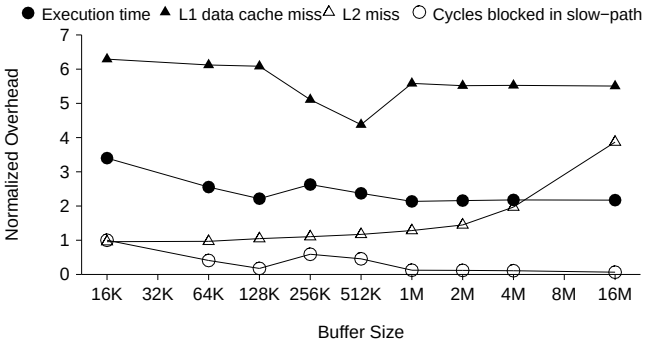


Figure 10. Performance as buffer size varies for path profiling on DaCapo `hsqldb` (using N:M threading on an Intel Core 2 Quad).

tains 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 overheads.

6.4 Sampling Mode Accuracy vs Overhead

Figure 11 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:

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

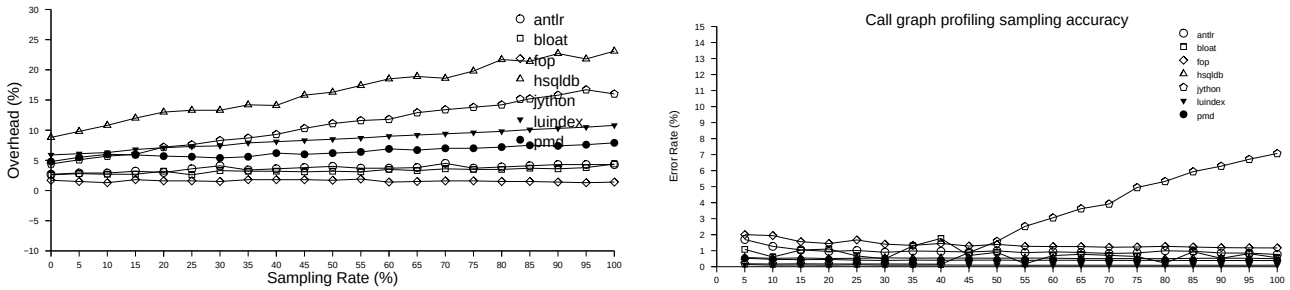
For example, in call graph profiling, an individual error rate is the error rate of each caller and callee pair. The error rate on the accuracy graph is the average accuracy of all the individual error rates. This error rate treats low to high frequency events equally so that it is not biased.

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 8), 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.

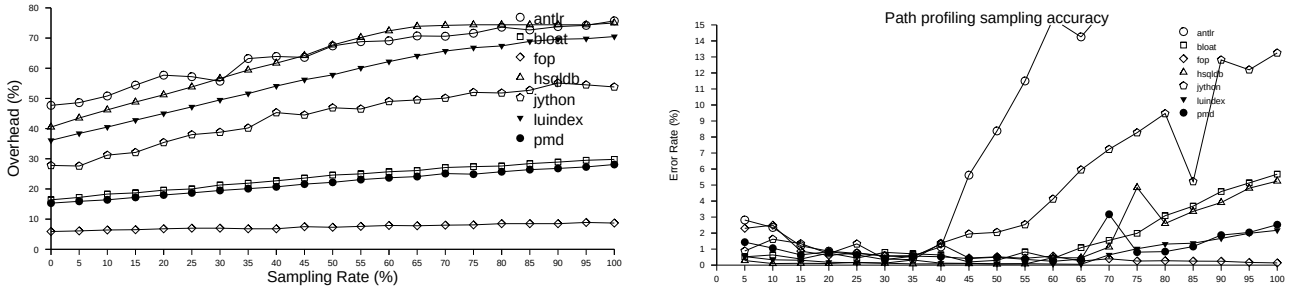
For some benchmarks, the error rate begins increasing rapidly when the number of samples taken *increases* past a certain point, which is quite counter intuitive. More samples usually results in a more accurate profile. This degradation occurs when the analyzer thread cannot keep up with the application 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.

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 parallelism 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 12 compares the performance of this new configuration, called “no cache sharing”, to the original shared L2, and single-threaded configurations. The figure re-



(a) Call Graph profiling sampling overhead and error rate



(b) Path profiling sampling overhead and error rate

Figure 11. Sampling overhead and error rate for call graph and path profiling (N:M threading on an Intel Core 2 Quad).

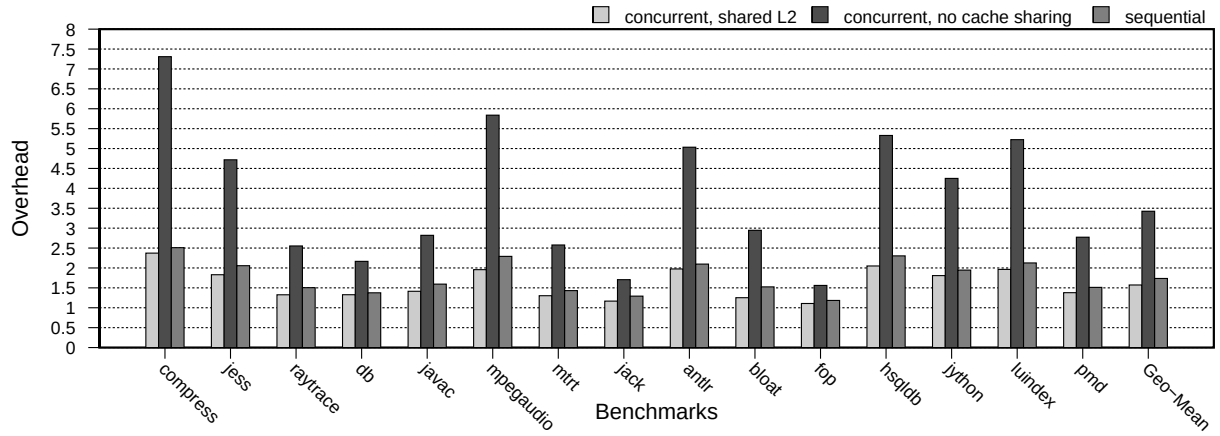


Figure 12. The importance of shared caches. Path profiling overhead with and without sharing between analyzer and application threads. Note that y-axis is the factor of overhead, and not a percentage.

ports 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.

7. Future Work

We believe that there are additional opportunities to reduce space overhead without compromising performance by automatically and dynamically adapting CAB buffer sizes, as well as the number of analysis threads. In particular, our native thread implementation keeps the CAB buffer in thread-local storage, which makes the number of buffers proportional to the number of threads. A more space-efficient designs may be possible where the number of buffers are proportional to the number of processors. These topics are being left for future work.

In addition, our current implementation binds only one analysis thread to each buffer. To speed up heavyweight analyses such as cache simulation, we plan to explore parallelizing the analysis logic itself and using multiple analysis threads to process a single buffer.

8. Conclusion

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 paper 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. This paper 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.

A. Additional Results

Exhaustive Mode Overhead Figure 13 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 [14] for complete results on the other architectures (Core i7 and Pentium 4). Figure 8 in Section 6.2 summarizes this data for all architectures. Similarly, Figure 14 reports the per-benchmark exhaustive mode overheads for the native threading model. Figure 9 in Section 6.2 summarizes this data.

Buffer Size Scalability Figure 15 reports performance as buffer size increases when performing path profiling, for each of the remaining DaCapo benchmarks. Figure 10 of Section 6.3 presented this data for the `hsqldb` benchmark.

References

[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. F. 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), Feb. 2000.

[2] M. Arnold and D. Grove. Collecting and exploiting high-accuracy call graph profiles in virtual machines. In *International Symposium on Code Generation and Optimization*, pages 51–62, 2005.

[3] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *ACM Conference on Programming Language Design and Implementation*, pages 168–179, Snowbird, Utah, 2001.

[4] T. Ball and J. R. Larus. Efficient path profiling. In *ACM/IEEE International Symposium on Microarchitecture*, pages 46–57, Paris, France, 1996.

[5] 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.

[6] 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, Oct. 2006.

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

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

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

[10] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *USENIX Annual Technical Conference*, Boston, MA, 2008.

[11] K. Gharachorloo and P. B. Gibbons. Detecting violations of sequential consistency. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 316–326, Hilton Head, South Carolina, 1991.

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

[13] J. Ha, C. J. Rossbach, J. V. Davis, I. Roy, H. E. Ramadan, D. E. Porter, D. L. Chen, and E. 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, California, 2007.

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

[15] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *International Conference on Software Engineering*, pages 291–301, Orlando, Florida, 2002.

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

[17] 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.

[18] M. S. Lam, M. Martin, B. Livshits, and J. 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, California, 2008.

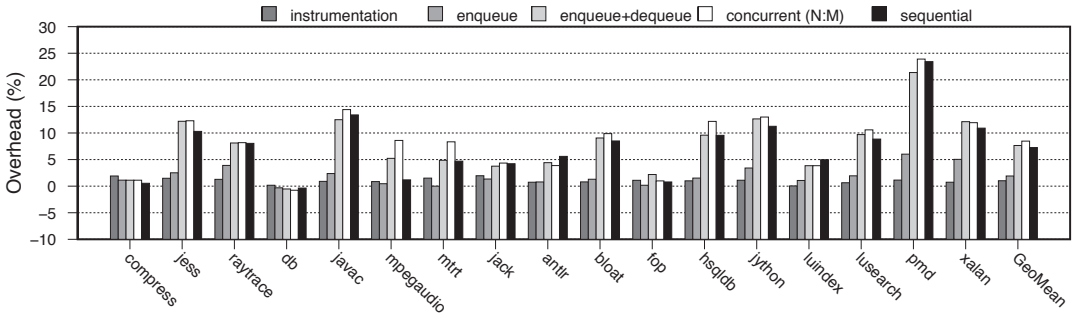
[19] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Language Systems*, 5(2):190–222, 1983.

[20] 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, 2001.

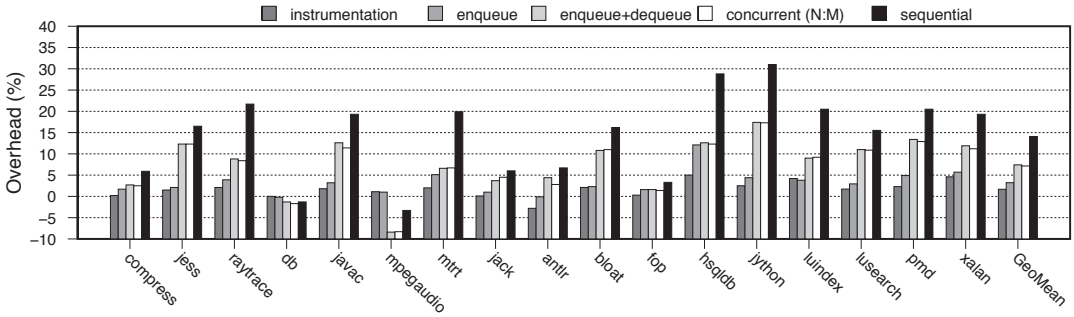
[21] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM Conference on Programming Language Design and Implementation*, pages 190–200, Chicago, IL, 2005.

[22] M. Martin, B. Livshits, and M. 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.

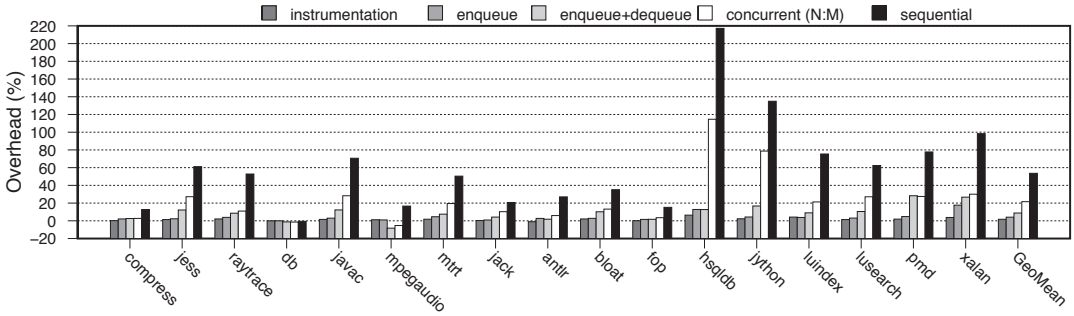
[23] T. Moseley, A. Shye, V. J. Reddi, D. Grunwald, and R. Peri. Shadow Profiling: Hiding instrumentation costs with parallelism. In *International Symposium on Code Generation and Optimization*, pages 198–208, Washington, DC, 2007.



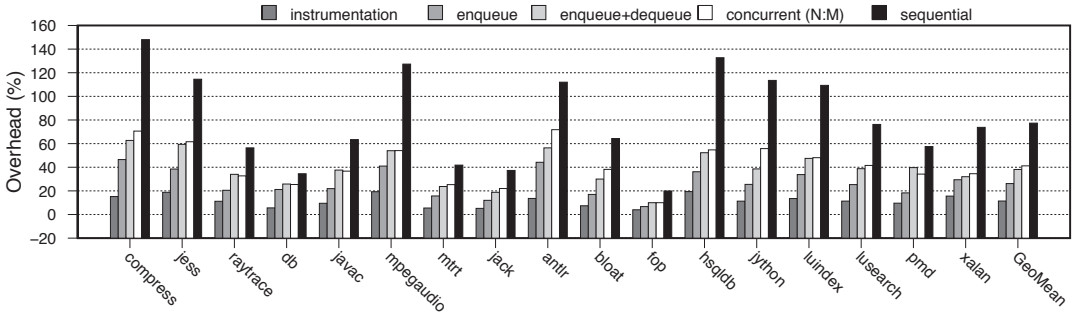
(a) Method Counting Overhead Percentage



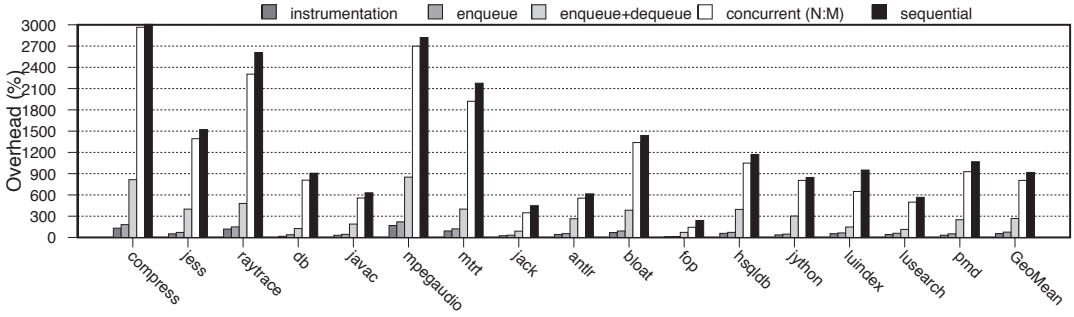
(b) Call Graph Profiling Overhead Percentage



(c) Call Tree Profiling Overhead Percentage

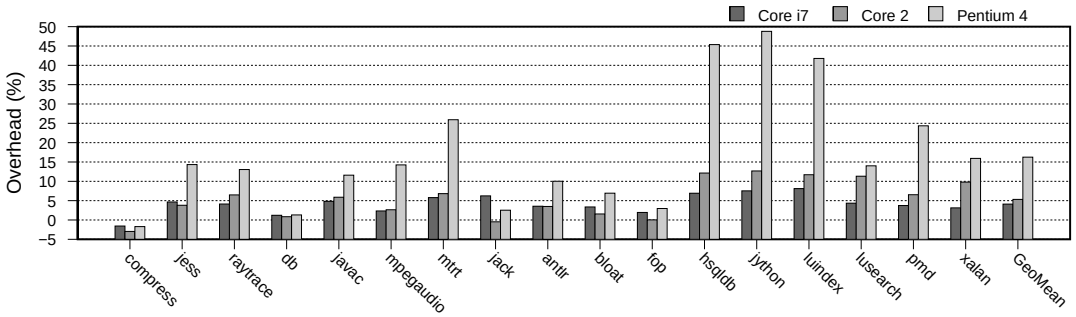


(d) Path Profiling Overhead Percentage

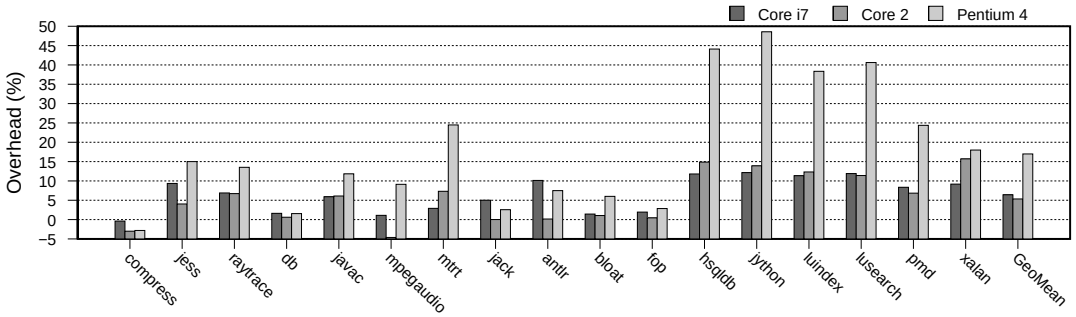


(e) Cache Simulation Overhead Percentage

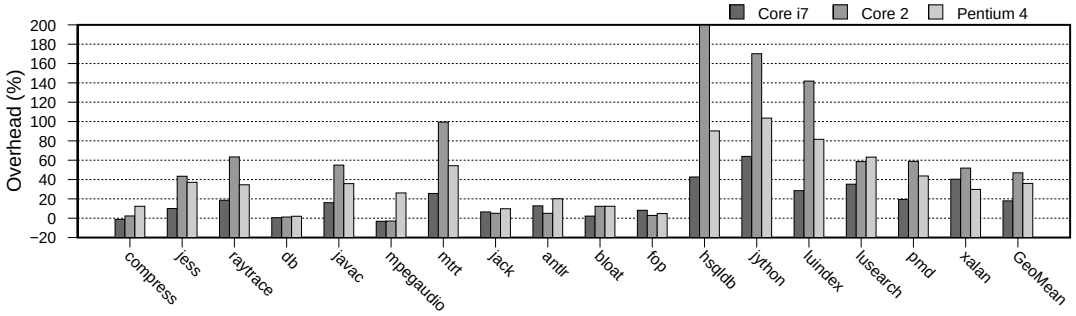
Figure 13. Per-benchmark exhaustive mode overhead on Core 2 (N:M threading).



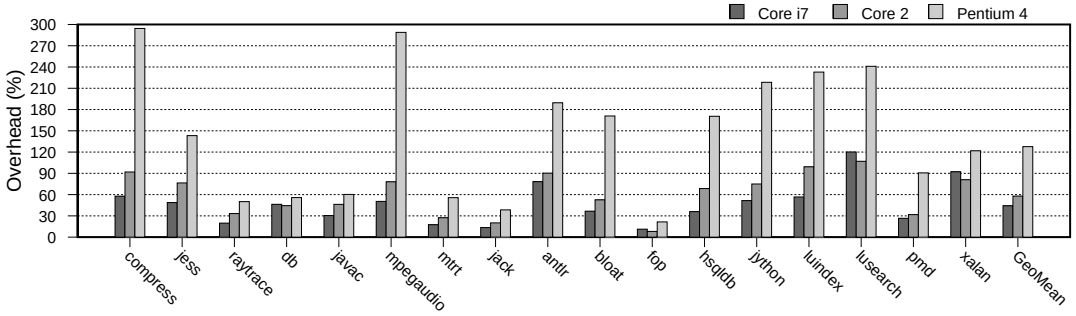
(a) Method Counting Overhead Percentage



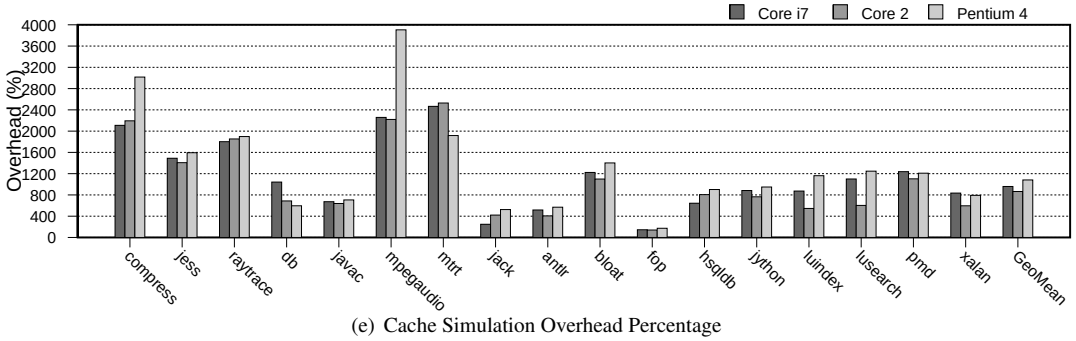
(b) Call Graph Profiling Overhead Percentage



(c) Call Tree Profiling Overhead Percentage



(d) Path Profiling Overhead Percentage



(e) Cache Simulation Overhead Percentage

Figure 14. Per-benchmark exhaustive mode overhead (native threading).

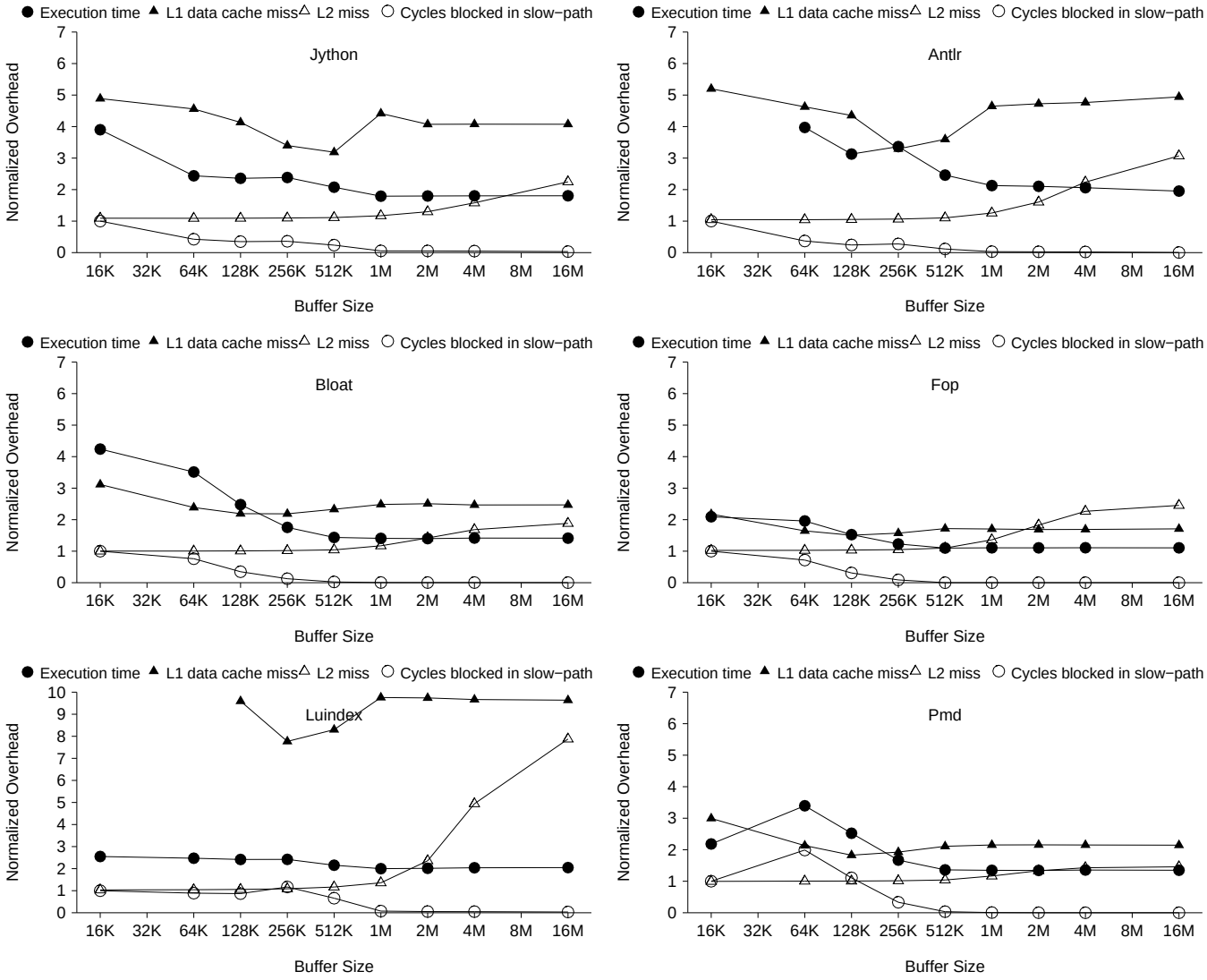


Figure 15. Performance as buffer size varies for each of the DaCapo benchmarks. Results are for path profiling using N:M threading on an Intel Core 2 Quad.

[24] N. Nethercote and J. Seward. Valgrind: a framework for heavy-weight dynamic binary instrumentation. In *ACM Conference on Programming Language Design and Implementation*, pages 89–100, San Diego, California, 2007.

[25] M. Paleczny, C. Vick, and C. Click. The Java HotSpot server compiler. In *Java Virtual Machine Research and Technology Symposium*, Monterey, CA, April 2001. Sun Microsystems.

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

[27] 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.

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

[29] S. Wallace and K. Hazelwood. SuperPin: Parallelizing dynamic instrumentation for real-time performance. In *International Symposium on Code Generation and Optimization*, pages 209–220, Washington, DC, 2007.

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

[31] C. Yuan, N. Lao, J.-R. Wen, J. Li, Z. Zhang, Y.-M. Wang, and W.-Y. Ma. Automated known problem diagnosis with event traces. In *ACM European Conference on Computer Systems*, pages 375–388, Leuven, Belgium, 2006.

[32] Q. Zhao, I. Cutcutache, and W.-F. Wong. PiPA: Pipelined profiling and analysis on multi-core systems. In *International Symposium on Code Generation and Optimization*, pages 185–194, Boston, MA, 2008.

[33] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas. iWatcher: Efficient Architectural Support for Software Debugging. In *ACM/IEEE International Symposium on Computer Architecture*, pages 224–235, München, Germany, June 2004.