Lazy Cache Invalidation for Self-Modifying Codes

Anthony Gutierrez Joseph Pusdesris Ronald G. Dreslinski Trevor Mudge
Advanced Computer Architecture Laboratory
University of Michigan
Ann Arbor, MI, USA
{atgutier, joemp, rdreslin, tnm}@umich.edu

ABSTRACT

Just-in-time compilation with dynamic code optimization is often used to help improve the performance of applications that utilize high-level languages and virtual run-time environments, such as those found in smartphones. Just-in-time compilation introduces additional overhead into the instruction fetch stage of a processor that is particularly problematic for user applications—instruction cache invalidation due to the use of self-modifying code. This software-assisted cache coherence serializes cache line invalidations, or causes a costly invalidation of the entire instruction cache, and prevents useful instructions from being fetched for the period during which the stale instructions are being invalidated. This overhead is not acceptable for user applications, which are expected to respond quickly.

In this work we introduce a new technique that can, using a single instruction, invalidate cache lines in page-sized chunks as opposed to invalidating only a single line at a time. Lazy cache invalidation reduces the amount of time spent stalling due to instruction cache invalidation by removing stale instructions on demand as they are accessed, as opposed to all at once. The key observation behind lazy cache invalidation is that stale instructions do not necessarily need to be removed from the instruction cache; as long as it is guaranteed that attempts to fetch stale instructions will not hit in the instruction cache, the program will behave as the developer had intended.

Categories and Subject Descriptors

B.3.2 [**Memory Structures**]: Design Styles—*Cache memories*; C.0 [**Computer Systems Organization**]: Hardware/software interfaces; C.1.3 [**Processor Architectures**]: Other Architecture Styles—*High-level language architectures*

General Terms

Design, Performance

CASES'12, October 7-12, 2012, Tampere, Finland.

Keywords

Architecture, Self-Modifying Code, Software-Assisted Coherence, Instruction Caching

1. INTRODUCTION

Instruction fetch is a critical component for achieving high performance in modern microprocessors, if there are no instructions to execute, progress cannot be made and the complex structures of out-of-order processors will not be efficiently utilized. Nearly two decades ago it was shown that real-world applications suffer from excessive instruction cache miss rates and fetch stalls [23]. More recent studies have shown that, for both smartphone and server workloads, instruction fetch performance has not improved, despite increasing instruction cache sizes and improved branch predictor accuracy [13, 18]. Real-world applications have become increasingly reliant on high-level languages, shared libraries, OS support, just-in-time (JIT) compilation and virtual machines. And, while the use of these modern programming constructs has made programmers more efficient and applications more portable, their use has led to increased code size and complexity, which stresses instruction fetch and memory resources.

Smartphone application developers in particular rely on portability and rapid development to ensure their applications are relevant and profitable. Similarly, smartphone manufacturers rely on the most popular applications being available on their platform to drive sales. This symbiotic relationship has led both application developers and smartphone manufacturers to rely on platforms that support high-level languages and virtual machines. Google's Android platform [12], which is currently the most popular smartphone OS on the market today [10], relies on the Dalvik virtual machine [9]. Android applications are written in Java and the Dalvik virtual machine now supports JIT compilation [5].

To overcome some of the performance loss incurred by the use of high-level programming constructs, JIT compilation with dynamic code optimization is often used [2]. JIT compilation uses dynamic code profile information to optimize and recompile code as it runs. This use of self-modifying code requires that the instruction cache be kept explicitly coherent. However, most systems, particularly mobile systems, do not support hardware coherence in the instruction cache. The cost of allowing the instruction cache to snoop the bus on every memory write is inefficient, because most memory writes are data, not instruction writes. To ensure that the instruction cache is kept coherent, and that stale

instructions are not fetched, JIT compilers use software-assisted coherence. In other words, the JIT compiler is responsible for managing the invalidation and writeback of the affected cache lines. There are two mechanisms by which stale instructions can be invalidated in the instruction cache:

- **Invalidate the entire instruction cache.** The benefit of this approach is that all stale instructions are invalidated with a single instruction, making this a very simple approach. The downside is that many useful instructions will be needlessly invalidated from the instruction cache, thus increasing instruction miss rates.

- **Invalidate a single line at a time.** This approach has the benefit of keeping most of the useful instructions in the instruction cache, which prevents excessive instruction cache miss rates. The downside of this approach is that the invalidations are serialized, which prevents useful work from being performed for the period during which invalidation occurs.

The pseudo code shown in listing 1 shows how software-assisted cache coherence is performed on most modern architectures that have Harvard caches and no hardware coherence for the instruction cache. Each instruction cache line in the range is invalidated in serial. Each invalidate also accesses the cache and performs tag lookups for each line in the range. No useful instructions can be fetched or executed during this period, primarily because of the possibility of fetching an instruction that is meant to be invalidated.

Listing 1: **Serial invalidation algorithm.**

```
invalidate_range(start, end) {
    addr = start

    while(addr < end) {
        dcache_clean(addr)
        addr += line_size
    }

    //ensure dcache_clean() completes
    barrier()

    addr = start
    while(addr < end) {
        icache_inv(addr)
        addr += line_size
    }

    //ensure icache_inv() completes
    barrier()

    //flush pipeline to remove any
    //stale instructions
    flush_pipeline()

    return
}
```

As noted above, both of these approaches have their strengths and weaknesses. Invalidating the entire instruction cache trades off invalidation precision for simplicity and very fast invalidation. Single line invalidation trades off simplicity and fast invalidation for very precise instruction invalidation. As shown in [14] evicting blocks from the code cache in medium sized chunks often leads to the best performance for the systems hardware caches. Thus, it is desirable to have a technique that can provide very fast invalidation, e.g., by using a single instruction, and that can do so without invalidating the entire instruction cache. However, it should be noted that we are not proposing to replace single line or full cache invalidation; we are proposing a new technique that gives more flexibility when choosing the granularity at which to invalidation cache lines.

In this work we make the following contributions:

- We analyze several benchmarks that make heavy use of JIT compiled code: the DaCapo benchmarks [4], a suite of open-source Java benchmarks, as well as BBench [13, 11], a web-page rendering benchmark. We show that these benchmarks invalidate a large number of cache lines and that they do so frequently.

- We develop a technique that provides the speed of entire instruction cache invalidation (it can be done with a single instruction), but gives much better precision (invalidations are on a per-page basis as opposed to invalidating the entire instruction cache), thus preventing long periods when useful work cannot be completed because cache maintenance operations are being performed.

Our analysis of BBench and the DaCapo benchmarks reveals that cache line invalidations frequently come in large bursts, always in multiples of an entire page, and that they happen frequently; a burst can happen as frequently as every 1,100 instructions for some benchmarks. We show that the fraction of time during which cache line invalidations occur can be decreased significantly ,which allows the CPU to more quickly resume doing useful work.

In section 2 we detail the design of our lazy cache invalidation hardware. In section 3 we discuss our experimental methodology. Section 4 outlines our experimental results. Finally, section 5 discusses related works and section 6 concludes and discusses future work.

2. LAZY CACHE INVALIDATION

In this section we outline our lazy cache invalidation technique. In particular, we describe a new instruction, which we call *pginv*, that can be used to invalidate an entire memory page instantaneously. We also describe the baseline hardware on which our design is implemented, as well as the additional hardware we introduce to support the *pginv* instruction.

2.1 Code Version Numbering

Lazy cache invalidation associates a version number with each piece of instruction data. Whenever new code is written its version is incremented. The version number is stored both per cache line and per TLB entry. The version number of a cache line is compared with the version number in its corresponding TLB entry in parallel with its tag lookup, and if the versions do not match, the access is considered a miss and the line will be fetched from memory.

The version number is maintained in the TLB; any time a new cache line is brought into the cache it is given the

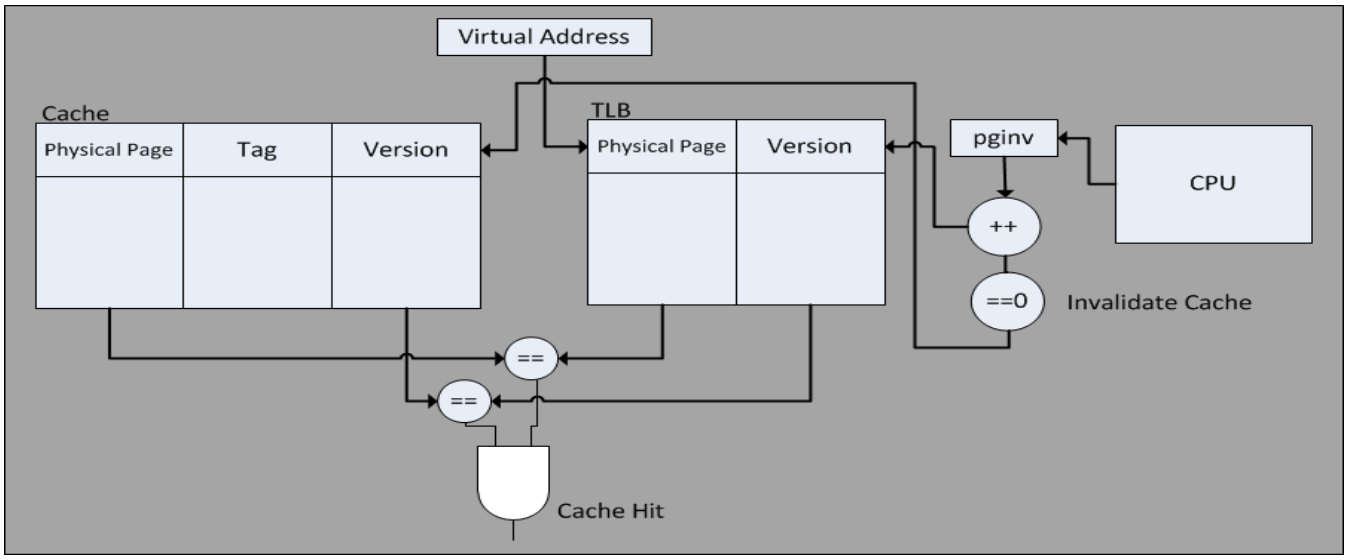


Figure 1: **Lazy cache invalidation hardware and implementation.** The CPU writes the virtual address to the *pginv* register. This increments the version in the TLB. If a page fault occurs the system waits until the new entry is loaded in the TLB before incrementing the version. If the version number overflows, the entire instruction cache will be invalidated.

version number of its corresponding TLB entry. If the version number rolls over the entire cache must be invalidated; this is to prevent the version number from rolling over to a version that matches a stale cache line. It is possible to avoid flushing the entire cache when the version number rolls over by using the existing page invalidate methodology, i.e., by invalidating each line serially. However, this option is not explored in this work because version number rollover is rare, even for small version number sizes as we will show in section 4.

The use of a TLB entry allows for more efficient instruction cache invalidation, primarily because each TLB entry is associated with an entire page of memory (typically 4kB). This allows a reasonable amount of code to be written without having to invalidate the entire cache or invalidate 4kB of data serially. Because JIT compilers often optimize at the granularity of basic blocks [2, 15], the instructions they write exhibit high spatial locality, making all lines written by self-modifying code likely to be on the same page.

Maintaining the version number in the TLB implies that the version number values do not need to be unique across TLB entries. Because physically tagged cache lines are assumed, each cache line maps to exactly one page thus, misreading stale data due to version aliasing is impossible. Storing the version number in the TLB has the added benefit of allowing the version number to be stored in the page table along with its corresponding TLB entry when it gets evicted; this prevents lazy cache invalidation from having to needlessly invalidate each line contained within the page pointed to by a new TLB entry, or in the worst case invalidate the entire cache.

2.2 Page Invalidate Instruction

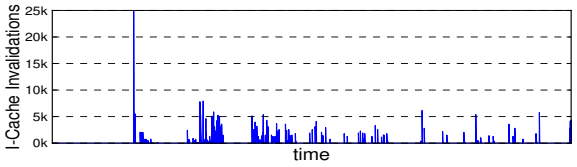
To implement the lazy cache invalidation technique we propose a simple ISA modification, the introduction of a new instruction we call *pginv*, to perform page invalidation. The *pginv* instruction operates in a similar fashion to an instruction that invalidates a single cache line, such as the ARM *mcr icimvau* [1] system control instruction, i.e., it takes as an argument the virtual address of a memory location to be invalidated. However, unlike the *mcr icimvau* instruction it is meant to invalidate the entire page that contains the line for the given address.

Because the *pginv* instruction is responsible for invalidating an entire page, the virtual address it receives must be aligned with a page boundary. It can be left up to the JIT compiler to ensure that the address it sends to the *pginv* instruction is page aligned but, for our implementation we make it the responsibility of the *pginv* instruction to align the address properly. This is typically done by masking off certain bits of the virtual address, making page alignment simple.

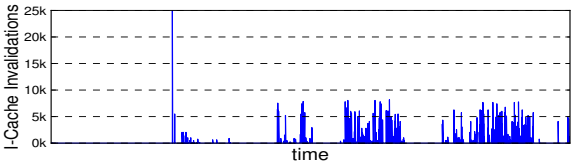
The instruction operates by accessing the TLB and, if the corresponding entry is present, it increments the version number in the TLB entry. If the TLB entry is not present a page fault is triggered, just as it would on any TLB miss, and once the fault is handled it attempts to increment the version number again. This effectively invalidates all cache lines contained within this page because their versions will no longer match. Note that existing invalidation techniques, such as using *mcr icimvau* also require a TLB lookup, and can also cause a page fault, because they also use the virtual address of a cache line.

2.3 Baseline Hardware

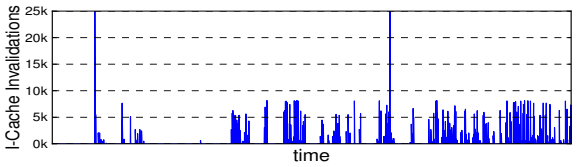
Lazy cache invalidation is most applicable when the base architecture uses Harvard style caches, separating data and instructions, with no hardware coherence for the instruction cache. In addition, the system should use virtual memory with physically tagged caches. TLBs must be used to cache virtual address translations. These features describe the vast majority of modern architectures. The JIT com-



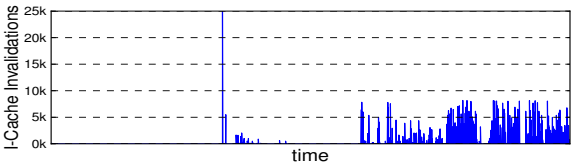
(a) avrora



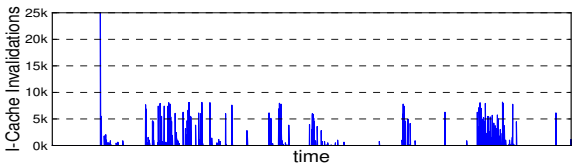
(b) batik



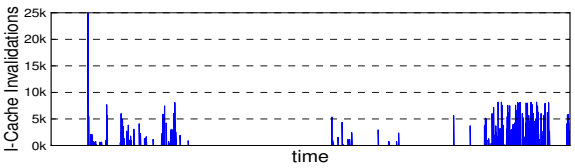
(c) eclipse



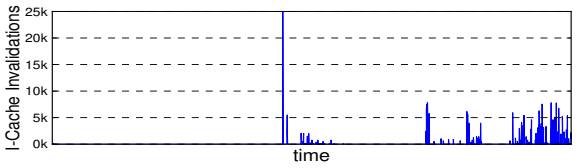
(d) fop



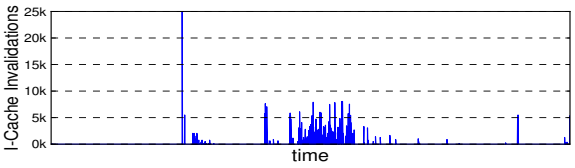
(e) h2



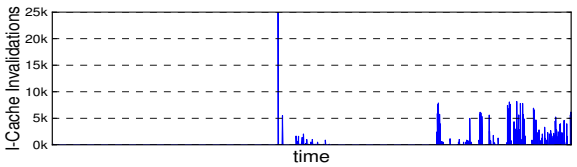
(f) jython



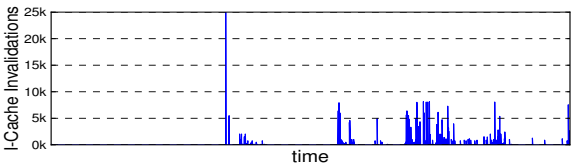
(g) luindex



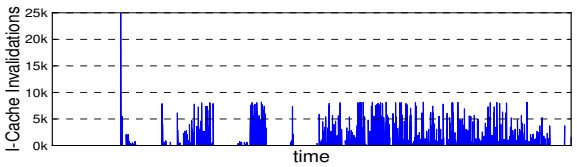
(h) lusearch



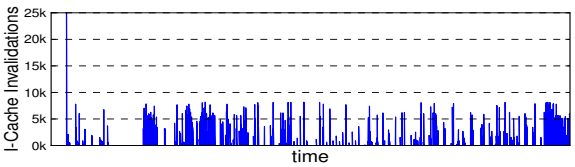
(i) pmd



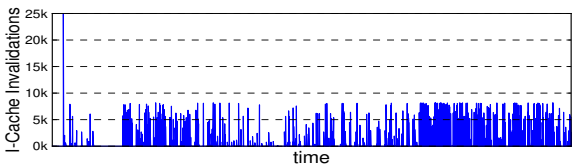
(j) sunflow



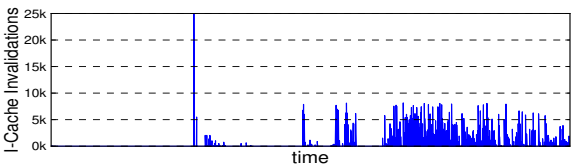
(k) tomcat



(l) tradebeans



(m) tradesoap



(n) xalan

Figure 2: **Instruction cache invalidations in the DaCapo benchmarks.** The number of i-cache invalidation instructions are sampled every 100,000 instructions. Most of the DaCapo benchmarks perform cache invalidations throughout the entire course of the execution, and they do so in bursts.

Frequency	1GHz
Cache Line Size	32 bytes
L1 Cache Size	32kB split I/D
L1 Associativity	4-way set associative
L2 Cache Size	1MB shared L2
L2 Associativity	8-way set associative

Table 1: **Hardware parameters.** The memory system parameters for our simulation framework. The memory system is modelled after an ARM Cortex-A9 processor, a current state-of-the-art smartphone core.

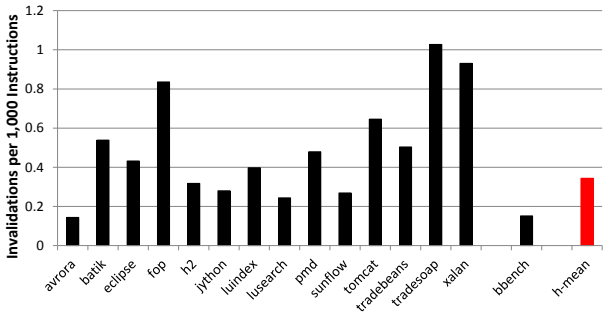


Figure 3: **Invalidation rates.** The rate of invalidate instructions per 1,000 instructions.

piler must issue invalidate instructions manually whenever it writes new code to the cache. This is typically how JIT compilers behave because most systems use Harvard caches with no hardware coherence in the instruction cache.

3. EXPERIMENTAL METHODOLOGY

In this section we describe our simulation framework and detail our experimental methodology. We first describe and analyze the benchmarks, in particular their use of cache invalidation instructions. Finally we describe the kernel modifications necessary to support lazy cache invalidation, as well as our implementation of lazy cache invalidation in the gem5 simulator.

3.1 Benchmarks

For our experiments we use several JIT compiled benchmarks. The DaCapo benchmarks [4] and BBench [13, 11], which we describe in further detail in the following sections. These benchmarks represent realistic and diverse user applications. They are primarily written in Java, make heavy use of shared libraries, OS support, JIT compilation, and state-of-the-art virtual machines.

3.1.1 BBench

BBench [13, 11] is a new web-page rendering benchmark that is designed to automate web browsing in such a way that makes a browser a useful interactive workload. It comprises several of the most popular web-pages, all of which utilize modern web technology such as CSS, HTML, flash, and multi-media. We run BBench on the native Android browser using Android version 2.3, Gingerbread [12], using version 2.6.35.8 of the Android kernel from Linux-ARM [17]. The version of Android we use has only simple modifications

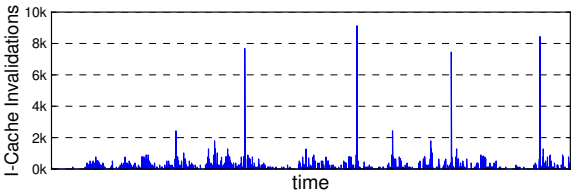


Figure 4: **Instruction cache invalidations in BBench.** The number of i-cache invalidation instructions are sampled every 100,000 instructions. BBench continuously performs cache invalidate operations and cache invalidations appear in repetitive bursts.

made to it, thus it is interactive and contains a full graphical user interface, which is accurately rendered by gem5.

Because the Android browser is part of the Android filesystem large portions of it are written in native C code; however, it still exhibits a fairly large amount of self-modifying code as we will show. Android applications are typically written entirely in Java and so we expect that most Android applications will exhibit even larger amounts of self-modifying code than our BBench results show.

3.1.2 DaCapo Benchmarks

The DaCapo benchmarks [4] are an open-source collection of realistic Java applications. We run all of the DaCapo benchmarks on Ubuntu 11.04 using version 2.6.38.8 of the Linux kernel. The Ubuntu disk image we used for our experiments is a modified version of a disk image created using the RootStock utility [22]. The disk image is *headless*, meaning it does not utilize a graphical user interface.

We installed the client side embedded Java runtime environment version 1.6.0.30 from Oracle [19] on our Ubuntu disk image and all DaCapo benchmarks were run on this virtual machine. We attempted to run the DaCapo benchmarks on the OpenJDK version of the Java virtual machine however, because of a known bug in OpenJDK for ARM, most of them were not able to run.

3.2 Simulation Environment

We use the gem5 [3] full-system simulator for all of our experiments. Our system parameters are shown in table 1. All of our experiments are run in full-system mode using the ARM ISA. We use the *simple timing* CPU model to obtain traces of all invalidation instructions executed. We use the ARM ISA because it is the most popular architecture used in modern smartphone and tablet devices. gem5 supports a large portion of the ARMv7 ISA; however, it does not currently provide support for the ARM *mcr icimvau* instruction, as well as several of the associated cache maintenance operations and registers. We first needed to provide support for the cache maintenance portions of the ARMv7 ISA relevant to our study. This includes the *mcr icimvau* instruction and the *mcr icialluis* instruction (these instructions invalidate a single cache line by its virtual address and the entire instruction cache respectively) as well as the cache size identification register *CCSIDR* and the cache type register *CTR* [1]. These registers provide information about cache line sizes in the system, which is necessary for the proper execution of the cache invalidation instructions.

3.3 Analysis of JIT Compiled Codes

We collected instruction traces from both the DaCapo benchmarks and BBench to determine how frequently they perform cache maintenance instructions. Figure 3 reports the overall number of invalidation instructions per 1,000 instructions executed. As can be seen in this figure some of the benchmarks execute cache line invalidate instructions as often as 1 for every 1,000 instructions executed, and on average 0.48 instructions per 1,000 are invalidations. This may not seem like a large number, but when you consider that these instructions are not performing useful work related to program execution this number is significant. And as we will see, these instructions often come in large bursts, which stall instruction fetch for a significant period of time.

In figure 2 we show the rate of invalidations per instruction sampled over time. The samples are taken for every 100,000 instructions executed. This graph shows that for many of the DaCapo benchmarks invalidations come in bursts, and that invalidation happens continuously throughout the entire run of the benchmark. Figure 4 shows the same data for BBench. Similar to the DaCapo benchmarks, BBench performs invalidations continuously throughout the entire run. However, unlike the DaCapo benchmarks, it happens less frequently (note the different y-axis values for the BBench graph).

From our analysis we discover that invalidations always happen in multiples of 128, which is exactly the size of a page. It is also shown that over 99% of the time invalidations occur in bursts of exactly 128, or one page. There are very few bursts of 2, 3, or more pages occurring. This shows that JIT compilers typically invalidate an entire page at a time, making lazy cache invalidation an ideal solution to speed up the invalidation process. Even if JIT compilers invalidate at granularities that are different than a page our technique is still useful; we are proposing that our technique offers a useful complement;

Figure 5 reports the median number of instructions between bursts of invalidations. This figure shows that for the DaCapo benchmarks bursts of invalidations occur as often as every 1,300 instructions, and for BBench every 20,000 instructions. We report the median, as opposed to the average, because a few very long periods without invalidations skew the average.

To quantify the effect that cache invalidation has on performance we define a segment of program execution we call a *work segment*, which we further divide into two segments, a *useful work* segment, and a *cache maintenance* segment. Based on our analysis we discover that for the DaCapo benchmarks the median size of a *cache maintenance* segment is roughly 30% of the *work segment*, and for BBench it is around 2.5%. As we will see in section 4, lazy cache invalidation reduces the fraction of time the *work segment* spends in a *cache maintenance* segment to almost nothing for both the DaCapo benchmarks and for BBench. In effect, lazy cache invalidation makes the effect of cache maintenance negligible.

3.4 Implementation of Lazy Cache Invalidation

The implementation of lazy cache invalidation required some modification to the Linux kernel source as well as gem5. In the following sections we describe the changes we made to our simulation framework to support lazy cache invalidation.

3.4.1 Kernel Source Modifications

We had to add support for our *pginv* instruction into the Linux kernel source. We profiled our benchmarks and discovered that all of the cache invalidation instructions executed were called from the *v7_coherent_user_range()* function. This function is defined in the ARM specific kernel assembly source and operates identically to the code shown in listing 1.

Because this function always invalidates a page at a time, we only needed to remove the loop around the i-cache invalidate instruction and insert our *pginv* instruction. We use the start address given to the *v7_coherent_user_range()* function as the input to the *pginv* instruction.

Although we never observed anything other than page-sized chunks being invalidated in the benchmarks we ran, it is possible for some codes to invalidate at some other granularity. There are two possibilities for ranges that do not invalidate only a single page:

- The range is less than a page and all lines in the range are contained within the same page.
- The range is greater than a page, or spans multiple pages.

In both cases we invalidate every page (using our lazy cache invalidation technique) touched by the range of addresses given. We do this by aligning the addresses to a page and calculating how many pages are touched by using the size of the range. This may cause needless invalidations, thereby increasing cache miss rates slightly, but it keeps implementation overhead low. Misaligned ranges are unlikely however, as is evident from the fact that we never observed a single occurrence of a misaligned range invalidation.

3.4.2 ISA Modification in gem5

To add support for lazy cache invalidation we needed to add a new instruction called *pginv* to the ISA. To do this we modified gem5 to add the functionality for the *pginv* instruction. We mimicked our implemenation of the *mcr icimvau* instruction by making *pginv* an ARM *mcr* system control register instruction [1].

When the instruction is encountered it triggers a write to the *pginv* control register. The value written to the register is the virtual address of the page being invalidated. Once the value is written the system control mechanism takes control and is responsible for aligning the virtual address to a page, looking up the corresponding TLB entry, and incrementing the version number of the page. If a page fault is encountered the instruction waits until it is handled and, once the proper entry is brought into the TLB, the version number is incremented. The *pginv* instruction automatically detects a version number overflow and is responsible for invalidating the entire instruction cache.

4. RESULTS

In this section we discuss the results of our lazy cache invalidation technique. In particular we discuss how lazy cache invalidation reduces the time spent performing *cache maintenance* segments. We also examine how frequently the entire instruction cache needs to be invalidated due to

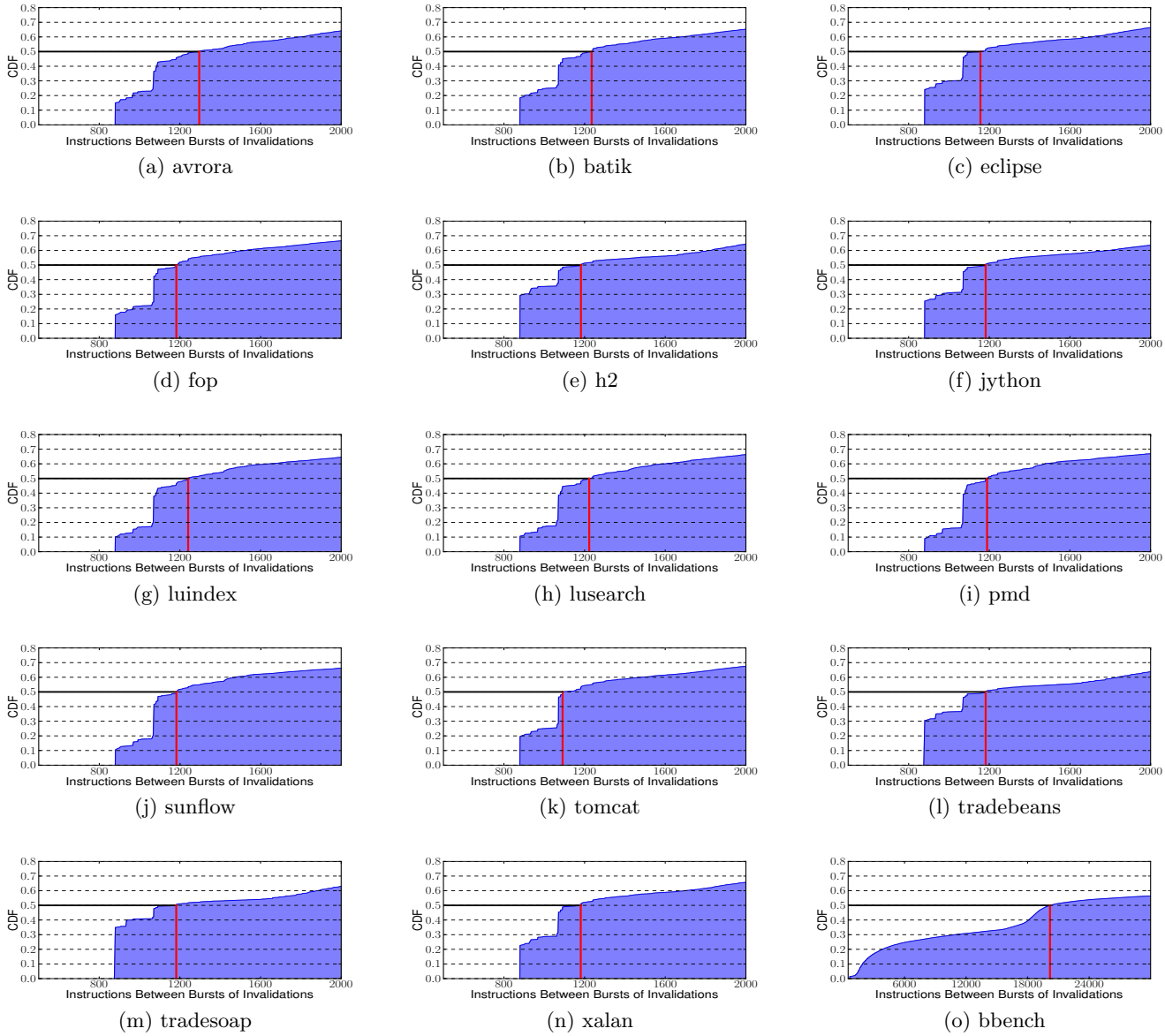
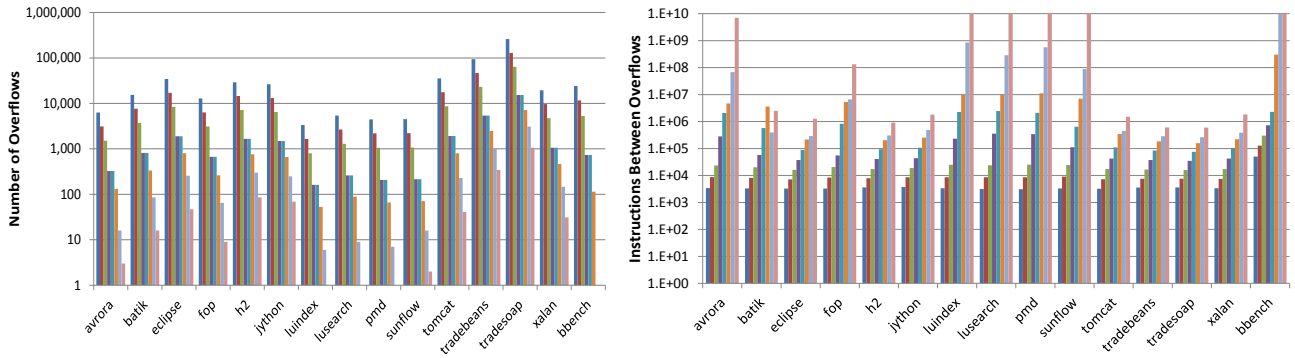


Figure 5: **Number of instructions between invalidation periods.** The CDF of the number of useful instructions executed between each invalidation period. The number of instructions between invalidation periods is small. In most cases fewer than 1,300 instructions are executed between *cache maintenance* segments for the DaCapo benchmarks. For BBench around 20,000 instructions are executed between *cache maintenance* segments.



(a) Number of Overflows

(b) Number of Instructions Between Overflows

Figure 7: **Sensitivity of overflows to counter size.** The bars represent, from left to right, version counter sizes of one through eight bits respectively. Note that the y-axis is logarithmic. For BBench, overflows never occur for a version counter size of seven or eight bits, therefore the distance between overflows is essentially infinite. This figure shows that overflows do not occur often, even for modest sized version counters. When overflows occur they happen far apart.

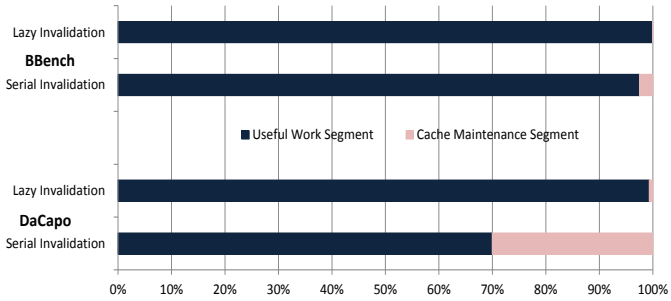


Figure 6: **Ratio of useful instructions to cache maintenance instructions.** The median size of a *cache maintenance* segment is reduced by around 30% for the DaCapo benchmarks, and around 3% for BBench.

version number overflow, as well as how close overflows occur in time.

4.1 Cache Invalidation Speedup

Figure 6 shows the fraction of time a *work segment* spends executing a *cache maintenance* segment. Again, we report the median segment sizes because a few very large data points skew the average. As can be seen from this figure the common case is to spend a significant fraction of time executing cache invalidate instructions. Because we are replacing segments of 128 instructions with a single instruction *cache maintenance* segments practically vanish.

During the *cache maintenance* segment the CPI for application instructions is essentially zero. Because the *cache maintenance* segments are always multiples of 128, and these periods are frequent, this can hinder the responsiveness of the application. As the authors in [20] have shown user satisfaction is highly correlated to application responsiveness. In [20] the authors used frequency as the metric by which to judge performance, but CPI and frequency are highly correlated.

4.2 Sensitivity of Overflows to Counter Size

One possible overhead of the lazy cache invalidation technique is forced invalidations caused by version number overflow, therefore the size of the version number counter is a critical design decision. Because of power and area constraints in smartphones, the version numbers cannot be too large. We measure how often each page is invalidated by a *pginv* instruction to determine how many times an overflow occurs. We also calculate how many instructions occur between version number overflows. If overflows occur too often, or if they occur in bursts, the benefits of lazy cache invalidation will be negated.

Figure 7a shows the median number of version number overflows for each benchmark and figure 7b reports the median number of instructions between version number overflows. For each benchmark the bars in the graph represent, from left to right, the results for counter values of one through eight bits respectively. As can be seen from these graphs, version number overflows are rare even for modest counter sizes. Forced cache invalidations are sparse, and infrequent enough that a version number size of five bits does not affect performance. Assuming 32kB caches with 32B lines, a five bit version counter would incur less than 2% overhead in the instruction cache.

5. RELATED WORK

This sections discusses previous work on software-assisted cache coherence. Techniques that propose efficient methods of cache coherence, as well as methods to speed up invalidation are discussed.

5.1 Software-Assisted Cache Coherence

Much of the previous work on software-assisted cache coherence focuses on using software as the primary means for maintaining cache coherence. This work was relevant in an era where hardware-based cache coherence was impractical and inefficient for large numbers of cores. However, modern technology scaling has led to multiple CPUs on a single chip and very fast and efficient busses, which makes snoopy-based hardware coherence the primary means for providing

cache coherence in modern microprocessors. However, because data are not written to the instruction cache in Harvard architectures, and because instruction writes are not as common as data writes, snoopy-based coherence for instructions is still not efficient for instruction caches. Due to energy constraints, mobile systems in particular cannot afford to allow the instruction cache to snoop the bus on every write. The lack of efficiency for snoopy-based coherence in instruction caches makes software-assisted coherence ideal for self-modifying codes.

5.1.1 Compiler Directed Techniques and Reference Marking

Knowing which lines to invalidate, and when to invalidate is of particular interest in software-assisted cache coherence techniques. Reference marking [16] is a compile-time method that segments program into chunks called *epochs*. Each memory reference in an epoch is marked as uncacheable if it is read by more than one processor and it is written by by at least one processor. To ensure that no stale data are read the cache is flushed at the end of every epoch. The work in [6, 8] improves on this approach by selectively invalidating marked references, as opposed to indiscriminately invalidating all marked references. Compile-time information is used to determine whether or not a reference is guaranteed to be up-to-date. These techniques reduce the overhead of cache invalidation by removing unnecessary cache invalidations; however, they do not speed up cache invalidations when they must occur (invalidations will still be carried out serially in these implementations).

5.1.2 Coherence Based on Data Versioning

Versioning has also been used to speed up the invalidation process and to dynamically identify which cache lines need to be invalidated [7, 21]. In [7] the authors propose associating each cache line with two counters. One is called a *birth version number (bvn)* and is stored on the cache line, the other is called a *Current Version Number (CVN)*, which is a global (per CPU) counter that represents the current version of the line. The *bvn* counter is checked against the *CVN* on each access, if the *bvn* is less than the *CVN* the access misses.

The *bvn* is set to the *CVN* when it is loaded from global memory and is incremented on each write. Compile-time analysis is used to determine when a *CVN* can change and it is updated based on this analysis and with minimal hardware and CPU communication. The *CVN* values are stored in global storage called the *version manager* and are obtained by indexing into the *version manager* with a unique ID number that is associated with each cache line. If any single *CVN* overflows all *CVNs* must be reset and the cache must be flushed; this ensures that a *CVN* doesn't rollover and match an invalid *bvn*.

In [21] the authors proposed using a version number, called a *one time identifier (OTI)*, for each line and its associated TLB entry. They have a global counter called the *OTI register* that contains the current value of the *OTI*. Each time a TLB entry is loaded into the TLB it reads from the *OTI register* and the *OTI register* is incremented. On every cache access to a page marked as shared the *OTI* value for the line is compared with the *OTI* value for the associated TLB entry. If the *OTI* values do not match the access is considered a miss and the line is fetched from memory.

It is assumed that the OS provides functionality to detect when a page is write-shared and when control is passed, e.g., by marking a page as shared when a lock within it is acquired, and marking it as not shared when the lock is released. After a page is released, the corresponding TLB entry must be marked as invalid, which implicitly changes its *OTI* value, to ensure that all subsequent accesses to this page are fetched from memory; this guarantees that no stale data are read. If the *OTI register* ever rolls over the entire cache must be flushed to prevent stale data from being misread as valid.

Our lazy cache flushing technique differs from these versioning approaches in both how we control updating of the version and how the version affects invalidation. In particular [7] increments the counters much more frequently, i.e., any time it is possible that a shared variable is modified. It also requires significantly more storage dedicated to version counters because it requires two counters per cache line, and they are typically larger than the counter values we use (to prevent frequent overflows), whereas we require only one counter per cache line and one per each TLB entry (which typically has far fewer entries than the cache). This overhead is not efficient for the instruction cache because it is not written to nearly as much as the data cache.

In [21] an entire page is invalidated each time a new TLB entry is brought into the TLB. This does not necessarily happen due to writes to a shared region, it could also happen when a TLB entry is evicted due to a conflict with another TLB entry. This increases needless invalidations and also accelerates counter rollover. Because the counter is global it also gets updated more frequently, which requires it to be larger. Because lazy cache invalidation targets self-modifying code, we only need to invalidate a page when instruction data are being written. Also, because lazy cache invalidation allows each TLB entry to maintain its own counter, the counters need not be as large. With lazy cache invalidation the counter for each TLB entry gets incremented only when that specific page is touched and not when any TLB entry is touched.

6. CONCLUSIONS

Software developers often make use of high-level languages and other advanced features when designing their software. The use of these high-level constructs improves the entire software development design flow. Smartphone sales are driven by the availability of applications that users care about, so the use of these high-level features will become more prominent going forward. Because of this fact, it is important for architects to design their CPUs with support for such high-level features.

In this work we have profiled several realistic user applications and we have demonstrated that these applications perform frequent, lengthy bursts of cache maintenance instructions. We have developed a new technique for fast cache invalidation, which we call lazy cache invalidation, that is shown to reduce the fraction of time a JIT compiler spends performing cache maintenance operations, and it does so with negligible overhead. By allowing a JIT compiler to invalidate an entire page with a single instruction we reach a good balance between the selective invalidation of the single line invalidation approach, and the speed of the entire cache invalidate instruction. We complement single line and whole cache invalidation and provide JIT compilier develop-

ers another option for cache invalidation that is often more efficient.

There are areas where further optimizations can be made to support self-modifying code. One effect of cache line invalidation is increased cache misses. The JIT compiler is responsible for writing back the newly written instructions out of the data cache into the second level cache—or memory if there is no second level cache. One of the consequences of this is that instruction fetch will miss on these instructions. However, because these instructions are being rewritten by the JIT compiler it is likely that they are frequently executed and will be fetched in the near future. Instruction pre-fetching of lines frequently written by the JIT compiler could reduce some of these instruction fetch misses.

The writeback of new instructions from the data cache is a serial, two-step process. The JIT compiler first writes the new instructions to the data cache, then it writes them back. Because the JIT compiler knows when it is writing instruction data, one possible optimization to this process is to require that instruction writes be write-through. This would allow the the JIT compiler to avoid the second step in the process, i.e., writing the instructions back.

Acknowledgments

This work was supported by a grant from ARM Ltd. In particular we would like to thank Ali Saidi and Stuart Biles. Special thanks to Korey Sewell for his advice on gem5. We would also like to thank the anonymous reviewers for their feedback.

References

[1] ARM. *ARM Architecture Reference Manual*, ARM v7-A and ARM v7-R edition.

[2] J. Aycock. A Brief History of Just-In-Time. *ACM Computing Surveys*, 35(2):97–113, 2003.

[3] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 Simulator. *SIGARCH Computer Architecture News*, 39(2):1–7, Aug. 2011.

[4] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, 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 *the Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 169–190, 2006.

[5] D. Bornstein. Dalvik JIT.

[6] H. Cheong and A. V. Vaidenbaum. A Cache Coherence Scheme With Fast Selective Invalidation. In *the Proc. of the 15th Annual International Symposium on Computer Architecture (ISCA)*, pages 299–307, 1988.

[7] H. Cheong and A. Veidenbaum. A version control approach to cache coherence. In *the Proc of the 3rd International Conference on Supercomputing (ICS)*, pages 322–330, 1989.

[8] H. Cheong and A. Veidenbaum. Compiler-Directed Cache Management in Multiprocessors. *IEEE Computer*, 23(6):39–47, June 1990.

[9] D. Ehringer. The Dalvik Virtual Machine Architecture.

[10] Gartner. Market Share: Mobile Communication Devcies by Region and Country, 3Q11.

[11] gem5. BBench Source.

[12] Google. Android Source.

[13] A. Gutierrez, R. Dreslinski, T. Wenisch, T. Mudge, A. Saidi, C. Emmons, and N. Paver. Full-System Analysis and Characterization of Interactive Smartphone Applications. In *the Proc. of the 2011 IEEE International Symposium on Workload Characterization (IISWC)*, pages 81–90, 2011.

[14] K. Hazelwood and J. E. Smith. Exploring Code Cache Eviction Granularities in Dynamic Optimization Systems. In *the Proc. of the International Symposium on Code Generation and Optimization (CGO)*, pages 89–99, 2004.

[15] K. Ishizaki, M. Kawahito, T. Yasue, M. Takeuchi, T. Ogasawara, T. Suganuma, T. Onodera, H. Komatsu, and T. Nakatani. Design, Implementation, and Evaluation of Optimizations in a Just-In-Time Compiler. In *the Proc. of the ACM 1999 Conference on Java Grande*, pages 119–128. ACM, 1999.

[16] R. L. Lee, P. C. Yew, and D. H. Lawrie. Multiprocessor Cache Design Considerations. In *the Proc. of the 14th Annual International Symposium on Computer Architecture*, pages 253–262, 1987.

[17] Linux-Arm.org. Armdroid Kernel.

[18] D. Meisner. *Architecting Efficient Data Centers*. PhD thesis, University of Michigan, 2012.

[19] Oracle. Embedded java runtime environment.

[20] A. Shye, Y. Pan, B. Scholbrock, J. S. Miller, G. Memik, P. A. Dinda, and R. P. Dick. Power to the People: Leveraging Human Physiological Traits to Control Microprocessor Frequency. In *the Proc. of the 41st annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 188–199, 2008.

[21] A. J. Smith. CPU Cache Consistency with Software Support and Using "One Time Identifiers". In *the Proc. of the Pacific Computer Communications Symposium*, pages 153–161, 1985.

[22] Ubuntu. Rootstock.

[23] R. Uhlig, D. Nagle, T. Mudge, S. Sechrest, and J. Emer. Instruction Fetching: Coping with Code Bloat. In *the Proc. of the 22nd annual International Symposium on Computer Architecture (ISCA)*, pages 345–356, 1995.