Locality Aware Zeroing: Exploiting Both Hardware and Software Semantics

Xi Yang

A thesis submitted for the degree of
Master of Philosophy at
The Australian National University

June 2011

Except where otherwise indicated, this thesis is my own original work.

Xi Yang
6 June 2011

to my wife, Liang

Acknowledgments

First I would like to thank my supervisor, Steve. Without his support and encouragement , it would not have been possible to start or finish this work. His great insights in computer system research inspired me to do excellent work. His financial support makes my life much easier.

I also would like to thank my supervisor, Peter, the freedom of research environment he provided encouraged me to explore different research areas in computer systems.

I would like to thank Daniel, who helped me to understand the design and implementation of Jikes RVM, and provided invaluable review feedback for this thesis. He makes me feel safe when he stands behind me as the goal keeper.

I would like to thank Kathryn. Her encouragement gave me confidence. I also like to thank Jenn, her write-back transaction saving work motivates the problem discussed in this thesis.

I also would like to thank folks who helped me to review previous versions of this thesis: Pete, Vivek, Rifat, Elton and Jie.

I also would like to thank my shifu, Huailin, who has been teaching me the architecture of computer systems, and sharing his system-design experiences in the last few years.

Finally, I would like to thank my parents, Qiudong and Yangai. Without their support, I would not have the chance to study overseas.

Abstract

Both managed and native languages use memory safety techniques to ensure program correctness and as a security defense. A critical element of memory safety is to initialize newly allocated memory to zero before making it available to the program. In this thesis I explore the performance impact of zero initialization and show that it comes with a substantial overhead. I also show that this overhead can be largely removed with new designs that exploit *both* the language semantics of zero initialization and the hardware semantics of memory hierarchies

Programmers increasingly choose managed languages to develop large scale applications. One of the reasons is that managed languages offer automatic memory management (garbage collection), which protects against memory leaks and dangling pointers. Garbage collection encourages programs to allocate large numbers of small and medium size objects, leading to significant overheads of zero initializing objects — on average the direct cost of zeroing is 4% to 6% and up to 50% of total application time on a variety of modern processors. Zeroing incurs indirect costs as well, which include memory bandwidth consumption and cache displacement. Existing virtual machines (VMs) either: a) minimize direct costs by zeroing in large blocks, or b) minimize indirect costs by integrating zeroing into the allocation sequence to reduce cache displacement.

This thesis first describes and evaluates zero initialization costs and the two existing designs. The microarchitectural analysis of prior designs inspires two better designs that exploit concurrency and non-temporal cache-bypassing store instructions to reduce the direct and indirect costs simultaneously. The best strategy is to adaptively choose between the two new designs based on CPU utilization. This approach improves over widely used hot-path zeroing by 3% on average and up to 15% on the newest Intel i7-2600 processor, without slowing down any of the benchmarks.

These results indicate that zero initialization is a surprisingly important source of overhead in existing VMs and that our new software strategies are effective at reducing this overhead. These findings also invite other optimizations, including software elision of zeroing and microarchitectural support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Much of the work presented in this thesis has been submitted for publication as the paper "Why Nothing Matters: The Impact of Zeroing" [Yang et al., 2011].

1.1 Thesis Statement

The overhead due to zero initialization is significant and will grow as the pressure on shared memory subsystems is increasing on chip multi-processors (CMPs). I believe this overhead can be substantially reduced by a) *avoiding* locality in favor of zeroing with instructions that bypass the cache, and b) exploiting available parallelism.

1.2 Introduction

Memory safety is an increasingly important tool for the correctness and security of modern language implementations. A key element of memory safety is initializing memory before giving it to the program. In managed languages, such as Java, C#, and PHP, the language specifications stipulate zero initialization. For the same reason, unmanaged native languages, such as C and C++, have begun to adopt zero initialization to improve memory safety [Novark et al., 2007]. We show that existing approaches of zero initialization are surprisingly expensive. On three modern IA32 architectures, the direct cost is 4% to 6% on average and up to 50% of all cycles in a high-performance Java virtual machine, without accounting for indirect costs due to cache displacement and memory bandwidth consumption.

Hardware trends towards chip multiprocessors (CMPs) are exacerbating these expenses because of their increasing demands on memory bandwidth [Burger et al., 1996; Liu et al., 2004; Hsu et al., 2006; Rogers et al., 2009; Yu and Petrov, 2010; Inoue et al., 2009; Zhao et al., 2009] and pressures on shared memory subsystems, such as shared on-chip caches and memory controllers. For example, Zhao et al. and Inoue et al. show that the memory bandwidth needs of both managed and unmanaged languages are a large performance bottleneck on CMPs [Zhao et al., 2009; Inoue et al., 2009]. If architects add processor cores without commensurate provision of shared memory resources (memory bandwidth and shared caches), the overhead of existing zero initialization techniques is likely to grow. Although hardware parallelism increases pressure on the memory system, it offers an optimization opportunity as well. In this case, it provides the opportunity to offload critical system services that must be done in a timely manner.

Existing zero initialization strategies face two problems: the direct cost of executing the requisite zeroing instructions and the indirect cost of memory bandwidth consumption and cache pollution. The two standard designs today are bulk zeroing and hot-path zeroing [Grcevski et al., 2004]. Bulk zeroing attacks the direct cost of zero-initialization by zeroing memory in large chunks and exploiting hardware prefetching, loop optimizations, and zeroing a cache line or more at a time. The drawback of this approach is that it introduces a significant reuse distance between when the VM zeroes a cache line and when the application first uses the cache line. This distance results in increased memory traffic and cache pollution. Another popular approach is hot-path zeroing, which injects zeroing instructions into the allocation sequence, attacking indirect costs by minimizing reuse distance. The drawback of this approach is that it expands and complicates the performance-critical allocation sequence and removes opportunities for hardware and software optimization of the instructions that perform the zeroing. The two designs are thus at poles, addressing either, but not both, of the direct and indirect costs of zeroing.

We introduce two better design points to reduce overheads by exploiting *both* the language semantics and the hardware semantics. The particular semantics of zero-initialization provide the flexibility to zero the memory at different times with different instructions. By understanding the CPU architecture, the run-time system is able to choose better designs dynamically. Our first design targets the indirect costs of zero initialization with *non-temporal* instructions. The second targets the direct costs by using *concurrency* to offload zeroing from the application's critical path. By dynamically choosing between these two solutions based on CPU utilization, we further reduce the cost of zeroing.

Non-temporal stores go directly to memory, bypassing the cache hierarchy. Designed for streaming applications, they avoid the cache-displacing effect of streaming writes. Because a non-temporal write does not miss in the cache, it does not generate a memory fetch, and therefore reduces memory traffic when there is no reuse. However, non-temporal writes have weak memory order and force the eviction of any cache line that is targeted by the write. If not used carefully, non-temporal writes are counterproductive. Concurrent zeroing is attractive when there is a surplus of hardware parallelism, as is becoming more common. However, to perform well, concurrent zeroing depends on synchronization, load balancing, and scheduling between the zeroing thread(s) and the application thread(s). When hardware parallelism is scarce concurrent zeroing is counterproductive.

We perform a detailed quantitative study of zero initialization. We study two existing and two new design points. We evaluate our work in the context of a high performance Java Virtual Machine (JVM) using 20 benchmarks drawn from DaCapo [Blackburn et al., 2006], SPECjvm98 [SPEC corporation, 1999], and pjbb2005 executing on three mainstream CMPs: an Intel Core2 Quad Q6600, an AMD Phenom II X6 1055T, and an Intel Core i7-2600. We measure the allocation rates of both real benchmarks and micro-benchmarks to establish the performance impact and limits of the various microarchitectures. For typical Java benchmarks on the i7-2600, existing zero initialization consumes 4% of CPU cycles, in addition to indirect costs. Compared with widely used hot-path zeroing, an adaptive policy built with our two new designs improves performance by 2.7% on average and by up to 15% on the i7-2600. Our technique is most effective on highly allocating memory intensive benchmarks, which stress the memory system the most.

Despite existing designs having overheads in the same ballpark as the cost of garbage collection, the other key element of memory safety, very little research has explored the cost of zeroing or alternative solutions. Our analysis sheds new light on the problem and the tradeoffs inherent to any solution. For example, hot-path zeroing has a lower reuse distance than bulk zeroing, but the entanglement of zeroing and allocation adds direct and indirect costs that negate the advantage of better data locality, especially on new CMPs. We demonstrate that non-temporal stores mitigate much of the cost of bulk zeroing, and make most efficient use of the available memory bandwidth. We show that using hardware parallelism to perform concurrent zero initialization hides the direct cost of zeroing for low CPU utilization applications. Because exploiting hardware parallelism when it is available reduces the overhead of zeroing, the best strategy adaptively chooses between concurrent zeroing and non-temporal bulk zeroing. Nonetheless, the total number of cycles devoted to the task of zero-initialization is often substantial, which suggests that further optimization of zeroing would be useful.

The contributions of this thesis are thus: (1) the detailed study of the cost of zero initialization, (2) identifying that zero initialization is often expensive on modern processors, (3) identifying and evaluating two new designs that together eliminate most of the overhead of zero initialization, and (4) pointing to the potential of future hardware and software approaches to further reduce the cost of zero initialization.

1.3 Thesis Outline

Chapter 2 introduces the motivation of zero initialization, provides an overview of hardware memory systems and the memory management mechanism in Jikes RVM. It also discusses related work. Chapter 3 describes the detail of four designs and our adaptive policy which utilizes the two best designs. Chapter 4 explains the experimental environment and the configuration of the systems we use. Chapter 5 analyses results of designs on three mainstream x86 CMPs. Finally, Chapter 6 concludes the thesis and describes the future work.

Chapter 2

Background and Related Work

Memory subsystems on modern processors are designed to sustain intense memory activity when accesses exhibit either: a) a high degree of temporal locality, or b) no temporal locality whatsoever. A cache hierarchy ensures that accesses with good temporal locality have low latency, while *non-temporal* streaming instructions go directly to memory with higher memory throughput, ensuring that they do not displace useful data in the cache. Unlike existing zero initialization designs using temporal instructions to zero, I propose better designs that exploit concurrency and non-temporal instructions. This chapter introduces: a) the non-temporal store instructions, b) the motivation of zero initialization, and c) memory management in Jikes RVM,

Section 2.1 discusses the motivation of zero initialization. Section 2.2 describes the memory allocation mechanism used in Jikes RVM. Section 2.3 explains the background of memory subsystems on modern processors. Section 2.4 discusses the prior published and unpublished (known to us through open-source implementations) related work.

2.1 Zero Initialization

Managed languages such as Java and C# have long touted memory safety as a software engineering and security benefit, and native languages, such as C and C++, are now embracing memory safety using compiler and library support [Novark et al., 2007]. Data initialization and pointer disciplines are the principle techniques for ensuring memory safety. Pointer safety disciplines protect against unintended or malicious access to memory by ensuring that the program accesses only valid references to reachable objects. Pointer safety is achieved through a combination of language specification and implementation techniques that enforce pointer declarations in static or dynamic type systems. The language implementation forbids reference forging, checks array indices, and forbids dangling references. It also ensures that all data is initialized before the program reads it. The language runtime typically zeros all memory systematically before making it available to the program. This approach is conservative—a program will often explicitly initialize the data before use as well, rendering the runtime's zeroing redundant. Both pointer safety and data initializa-

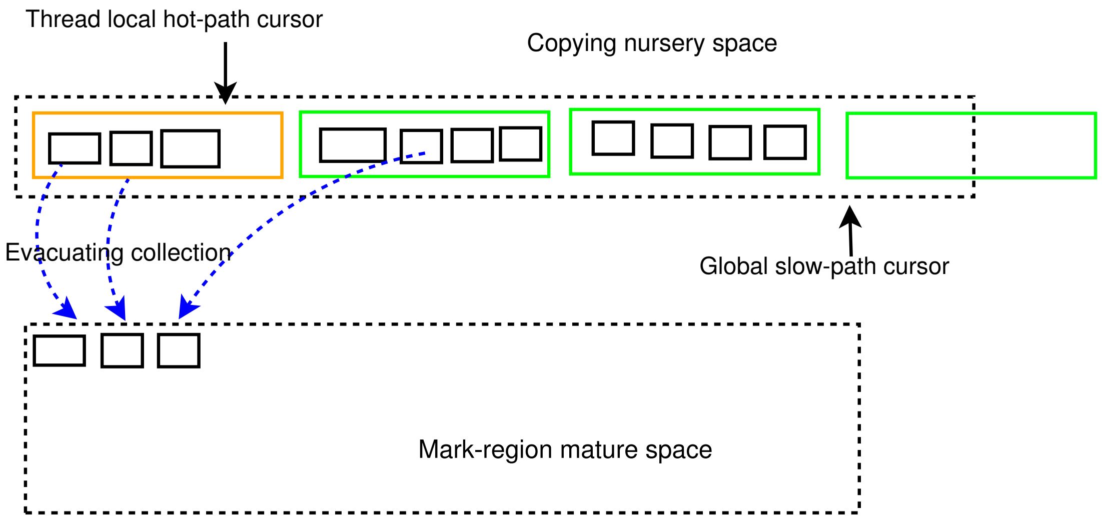


Figure 2.1: Generational Immix collector.

tion offer software engineering and security benefits, but they increase the number of memory operations.

2.2 Jikes RVM/MMTk

This section briefly describes the Jikes RVM [Alpern et al., 2005] memory management [Blackburn et al., 2004b] (MMTk) mechanism. Jikes RVM is a highly tuned research JVM. It uses the generational Immix collector by default [Blackburn and McKinley, 2008]. Figure 2.1 shows the two spaces of a generational immix collector: the copying nursery space and the mark-region mature space. Generational Immix is a stop-the-world collector. It allocates objects into a *nursery* using *bump-pointer* allocation. When the nursery fills, it copies all live objects into a mature mark-region space. Memory allocated in the nursery space must be initialized as zero. However, it is not necessary to zero memory allocated in the mature space since objects are explicitly initialized by the act of copying them from the nursery into the mature space.

The nursery space allocator design consists of a thread-local, unsynchronized hot path, and a global, synchronized slow path [Blackburn et al., 2004b]. Each thread thus allocates into a local buffer without any synchronization. This thread-local allocator design is very common and is used in virtually all commercial garbage collectors and explicit memory managers. When a thread exhausts a block, it executes a synchronized slow path, which either provides a free block or throws an out of memory exception. Figure 2.1 shows a snapshot of the spaces. Threads acquire blocks in the slow-path by updating a *global slow-path cursor*. Objects are created inside blocks by updating a *thread local hot-path cursor* without any synchronization. In Figure 2.1, there are two threads. The first one allocated the first block in the slow-path and created three objects in it. Another thread has already consumed two blocks and

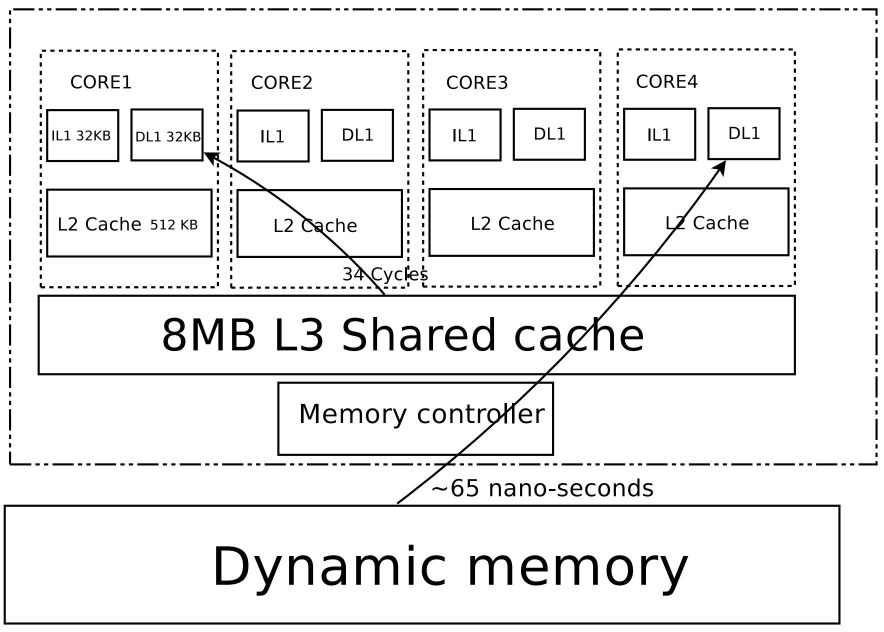


Figure 2.2: On-chip memory hierarchy of Nehalem.

intended to allocate the third one. However, this allocation triggered the nursery collection since there was not enough memory. The copying collector then evacuated survivors from the nursery space into the mark-region space, as shown in Figure 2.1. By default, Jikes RVM zero-initializes space by bulk zeroing 32KB blocks of memory on the allocation slow path.

2.3 Memory Hierarchy

Memory systems must make a trade-off between size, speed and price. Unlimited fast memory is desirable but prohibitively expensive. Based on the observation that most programs tend to exhibit temporal and spatial locality, memory resources are organized as layers to mimic a fast, cheap, and large memory system. Fast, expensive and small caches reside at the top of the memory hierarchy to cache recently referenced data. Slow, cheap and large dynamic memory holds programs' working sets and provides data when references miss higher levels.

Figure 2.2 shows the memory hierarchy of the Nehalem micro-architecture, a mainstream CMP designed by Intel. On Nehalem, each core has a private 32 KB L1 instruction cache, a 32 KB L1 data cache, and an exclusive 512 KB level 2 cache. Four cores on the chip share an inclusive 8 MB last level cache. Compared with on-chip caches, dynamic memory is much larger but slower. The latency to reference L1, L2, and L3 caches is 4, 10, and 38 cycles [Molka et al., 2009] respectively. Fetching data from dynamic memory to caches takes about 65 nano-seconds (189 cycles on 2.8 GHz CPU).

To further maximize the performance of programs that exhibit good temporal locality, modern caches use write-back, write-allocate semantics. On a write hit, the hardware writes the new data and marks the cache line as dirty. On a write miss, the hardware first fetches the cache line, and then writes the new data and marks it as dirty. When cache lines are evicted or synchronized with lower-level caches, dirty lines are written back to the next lower level of the hierarchy. For memory references that exhibit good temporal locality, write-back caches work well by reducing write transactions and speeding up memory references.

To maximize the performance of programs that exhibit good spatial locality, modern processors prefetch memory aggressively. After the hardware prefetching unit detects sequential access patterns by monitoring memory references, it issues prefetching requests to fetch data from the memory which is likely to be referenced later.

Non-temporal store instructions Programs sometimes exhibit poor temporal locality, such as when copying, or initializing large memory blocks. Due to the performance gap between the dynamic memory and caches, the performance of such streaming references is dominated by memory bandwidth. Write allocation limits memory throughput and pollutes the cache since in the worst case every write in addition to generating a store to memory, first requires a cache line load from memory, which is useless when the line will not be read.

To improve the performance of streaming stores and avoid cache pollution, x86 processors provide non-temporal store instructions. These instructions bypass the cache to avoid cache pollution, and directly write to memory. Because non-temporal instructions avoid fetching data from dynamic memory to cache, streaming stores with these instructions fully utilize memory bandwidth, leading to higher memory throughput.

Non-temporal store instructions also have some downsides. Clearly, they are not suitable if the program will access the data again soon after the write, since the data will be uncached. Moreover, to maintain cache coherency, the semantics of non-temporal store instructions usually include explicitly evicting any copy of the line from all caches. Non-temporal stores also typically come with weak memory ordering that requires the addition of explicit memory fences to enforce strong memory ordering. The x86 processors do not reorder temporal stores. However, non-temporal stores are allowed to be reordered with other temporal and non-temporal stores. If programs want to maintain the order between non-temporal stores or the memory order between non-temporal stores and other temporal stores, memory fences or memory store fences are required to be inserted to make non-temporal stores globally visible by other cache agents.

2.4 Related work

ISA Support. The x86 [Intel] instruction set architecture (ISA) includes *non-temporal* instructions that programs can use for reads and writes that have no temporal locality. Non-temporal store instructions, e.g., `movnti` and `movntdq`, bypass the cache hierarchy. They send writes directly to memory via a write combining buffer, bypassing the cache. When used effectively, they have two benefits: a) they do not displace other data in the cache, and b) they maximize memory bandwidth efficiency because, unlike normal stores that can generate one fetch and one write-back transaction, non-temporal stores only generate one write transaction.

On some processors, including the i7-2600 we use in our experiments, these benefits achieve significantly higher write bandwidth than regular writes.

The PowerPC ISA includes a data cache block zero (`dcbz`) instruction that zeros a cache-line directly without fetching it from memory [Sikha et al., 1994]. The processors designed by Azul [Click, 2009] have a similar instruction, (`CLZ`), that directly zeros a cache line without fetching old memory, but uses a more relaxed memory consistency model.

Efficient Zeroing. Programming language and OS implementations highly optimize zeroing, memory copying, and memory initialization. For example, the standard C library provides the `memset()` function to initialize memory. Since `memset()` has no semantic knowledge of the reuse distance between the initialized memory and its next use, it resorts to a simple heuristic to switch to non-temporal instructions. For x86 processors, GNU's C library (glibc) [GNU] uses non-temporal stores when the region being zeroed is larger than the processor's last level cache. Otherwise it uses standard (temporal) writes. The open64 compiler [AMD] provides a `-CG:movnti=N` flag. When it writes a memory block larger than N KB, the compiler generates non-temporal store instructions.

Zero Initialization Strategies. We examined the details of zero initialization in the open source versions of Oracle HotSpot [B.Kessler, 2007] VM. We extracted further details of the Azul [Click, 2009] and IBM J9 [Grcevski et al., 2004] JVMs from talks and publications. Each of these VMs zero initializes memory on the allocation hot path, minimizing reuse distance between initialization and first use. Where practical they also selectively zero only those parts of the objects that are not explicitly initialized when they are constructed. To save memory bandwidth, the J9 and Azul VMs use `dcbz` and `CLZ` instructions when targeting PPC and Azul hardware, respectively.

Jikes RVM [Blackburn et al., 2004a] and optionally HotSpot both bulk zero memory before providing it to the allocator. This approach forgoes temporal locality between initialization and first use, but minimizes the direct cost of zeroing by using a tight loop that can use coarse-grained zeroing instructions and that activates the hardware prefetcher. We found that the HotSpot implementation of bulk zeroing is extremely naive. We were able to substantially improve its performance by using `memset()` to perform the zeroing.

2.5 Summary

This chapter introduced non-temporal store instructions, explained the motivation for zero initialization, and discussed approaches different runtime systems have adopted. In the next chapter, I will discuss zero initialization designs in detail.

Chapter 3

Design and Implementation

In this chapter, I explore four design points for zero initialization. The first two, allocation-time (*hot-path*) zeroing and bulk zeroing, are widely deployed today. The other two new design points perform non-temporal bulk zeroing, and concurrent non-temporal bulk zeroing. Both of these new strategies perform better than deployed strategies. The best policy, however, adaptively chooses among the non-temporal bulk and concurrent non-temporal bulk based on available hardware parallelism.

Section 3.1 discusses *hot-path* zeroing that minimize indirect costs of zero initialization by integrating zeroing into the allocation sequence to reduce cache displacement. Section 3.2 describes bulk zeroing, which reduces the direct cost of zero initialization by zeroing memory in large chunks with temporal store instructions. Section 3.3 introduces non-temporal bulk zeroing. It zeros memory in large chunks with non-temporal store instructions to reduce the direct cost by improving memory throughput and indirect cost by bypassing caches. Section 3.4 introduces concurrent bulk non-temporal zeroing which offloads zero initialization to idle cores to further reduce the direct cost. Section 3.5 introduces the best policy, adaptive zeroing, which chooses among two new designs I proposed.

3.1 Hot-path Zeroing

Hot-path zeroing zeroes memory for each object at allocation time, immediately prior to its first use. By default IBM J9, Oracle HotSpot, and Azul HotSpot use this design. Hot-path zeroing trades better data locality against a diminished opportunity to optimize zeroing. It requires more instructions on the allocation hot-path and therefore degrades the program's instruction locality.

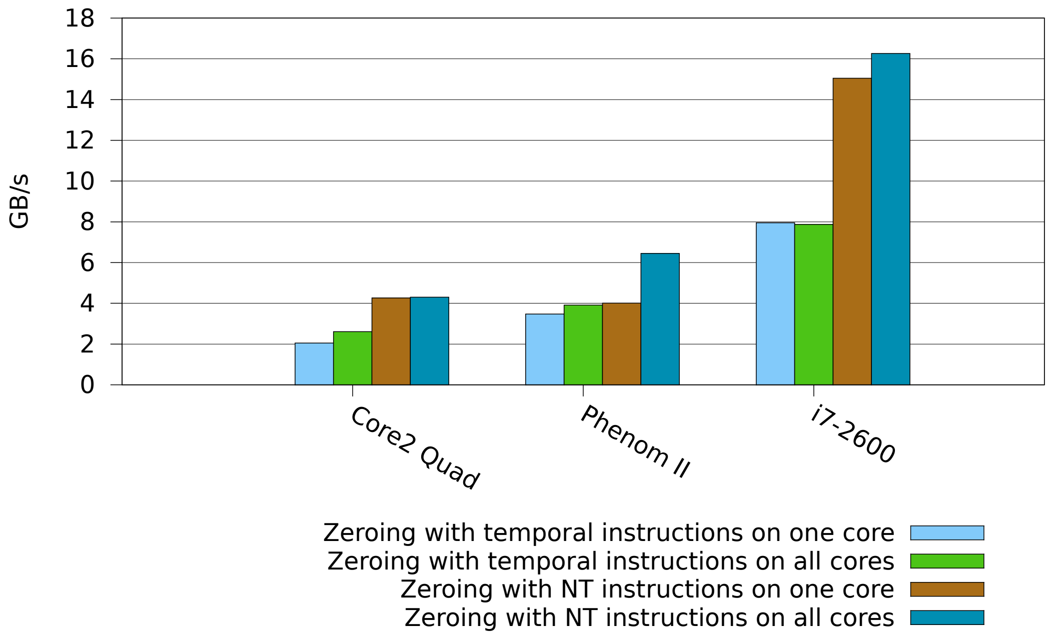
Hot-path allocation is performance critical in a modern JVM. To avoid the overhead of function calls and to enlarge the optimization scope, compilers usually inline hot-path allocation. Zeroing in the hot-path increases the size and complexity of generated code, which leads to more instruction cache pressure. On the other hand, hot-path allocation is friendly to modern memory systems since it provides good temporal locality with the use of the data. Modern processors aggressively prefetch the memory for sequential reference streams to hide latency. Hot-path allocation does provide a sequential access pattern, but since allocation is interspersed with other data accesses, spatial locality is reduced and thus hot-path allocation is less amenable to hardware prefetching compared to bulk zeroing.

Our hot-path implementation inserts instructions to initialize objects just prior to their creation. It zeroes objects sixteen bytes at time using an unrolled loop of four-byte `mov` instructions. We found that this version performed significantly better than two eight-byte (`movq`) instructions or one sixteen byte (`movdq`) instruction. We concluded that this was for two reasons: 1) The larger width instructions required the use of a register as the source, while the four byte instruction could use a zero immediate; and 2) only four-byte alignment is guaranteed in the allocation sequence, while the wider instructions are known to require aligned memory access for optimal performance. When objects are smaller than sixteen bytes, our approach will redundantly zero some trailing memory. Since the minimum object size is 8 bytes (the size of a header) and the average object size is around 28 bytes, redundant-zeroing is not a significant concern. Section 5.2 evaluates the performance overhead of hot-path zeroing.

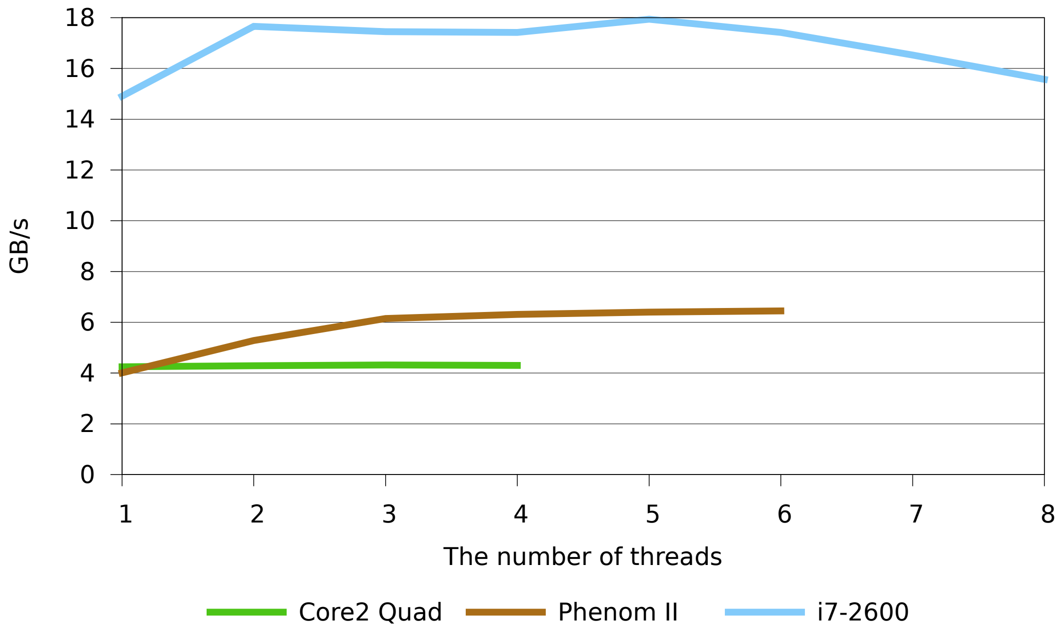
Opportunity for further optimization. Java's semantics require that a *constructor* be executed immediately after each object is allocated. A constructor includes arbitrary user code and may include the explicit initialization of all or part of the object, resulting in a duplication of effort. If the implicit zeroing and explicit initialization are both statically visible to an optimizing compiler, the compiler can remove redundant hot-path zeroing. The opportunities for performance improvement are modest because hardware will efficiently handle redundant writes with such good temporal locality. Such an optimization is also difficult to make correct in general, because it requires an analysis to guarantee that all object fields are initialized before they are seen by the program *or* the garbage collector, otherwise, the constructor could publish uninitialized objects or trigger the garbage collection which could collect live objects. The Oracle HotSpot VM implements such an optimization, but when we measured it, we found that it provides no benefit except for two older benchmarks (jess and pjbb2005) and only when they are running on old hardware. Due to this negative result, and the complexity involved in implementing the optimization, we do not consider it further.

3.2 Bulk Zeroing

Both Jikes RVM (by default) and HotSpot (with a command line option) provide bulk zeroing. Bulk zeroing initializes blocks of free memory to zero prior to returning them to the memory allocator. The default implementation in Jikes RVM performs zero initialization of 32KB blocks using glibc's `memset()` function. Because the size of the block (in this case 32KB) is much smaller than the size of the last level cache on the machines we evaluated, the `memset()` function uses normal store instructions to perform zeroing. The rationale for bulk zeroing design is improving the efficiency



(a) Performance of non-temporal vs. caching writes



(b) Memory bandwidth scalability of non-temporal writes as a function of thread count

Figure 3.1: Performance potential of non-temporal instructions

of zeroing. It performs highly optimized zeroing in a tight loop with a very small instruction cache footprint. For example, the zeroing routine straightforwardly utilizes instructions that zero at a coarse grain—zeroing a double word, quad word, or even a cache line at a time. Because of the high spatial locality of these instructions, hardware prefetching further reduces the latency of zeroing. The disadvantage of this design is that as the block size grows, the average reuse distance between allocation and data use also grows. Section 5.3 evaluates bulk zeroing.

3.3 Non-temporal bulk Zeroing

This section and the next present our new designs. Non-temporal instructions (also known as streaming instructions) provide a key addition to the ISA that locality-aware applications can use to override the default cache policies. Non-temporal instructions bypass the cache altogether with a weaker memory order. This control is particularly advantageous when an application stream writes to memory — an access pattern which would otherwise have the effect of systematically displacing the potentially useful contents of the cache. A non-temporal store eliminates fetches associated with a regular store and does not displace cache lines at any level of the hierarchy.

Non-temporal instructions also have some downsides as we discussed in Section 2.3. On the upside, these restrictions bring further benefits when non-temporal instructions are used correctly because they improve write bandwidth by a factor of two or more since no fetches are required and stores have week memory order.

Figure 3.1(a) shows the potential throughput benefits of non-temporal instructions, when used correctly on the three architectures used for our evaluation (see Section 4.2 for the architectural details). This simple limit study evaluates the throughput performance of non-temporal and caching write instructions in a tight zeroing loop with one thread and N threads, where N is the number of available hardware contexts (hardware threads). On the Phenom II, the throughput increase of non-temporal instructions is only 15% for a single core, and 64% when using all cores. Both the Core2 Quad and i7-2600 hardware perform non-temporal write instructions at *twice* the rate as caching write instructions.

Figure 3.1(b) shows the memory bandwidth scalability of non-temporal instructions in the same tight loop with respect to the number of hardware threads. The memory bandwidth on the Core2 Quad does not scale with the number of threads. The memory bandwidth on the i7-2600 scales by 18% with 2 threads, and a little more to 20% with 5. 5 parallel threads performing non-temporal stores thus achieving a 20% higher store rate than a single thread.

Non-temporal instructions therefore provide an excellent opportunity to mitigate the poor locality of the bulk-zeroing design point. Our *non-temporal bulk-zero* implementation replaces `memset()` — which uses regular store instructions for regions smaller than the last-level cache size — with a loop that uses the `movntdq` quad-word non-temporal store instruction. Note that unlike hot-path zeroing, for bulk zeroing we can guarantee aligned memory accesses and amortize the cost of zeroing the source register, avoiding the pitfalls that made wide instructions perform poorly in that case. Section 5.4 assesses the performance of this modest change and shows it has a significant performance impact.

3.4 Concurrent Zeroing

Non-temporal bulk zeroing mitigates the direct and indirect costs of zeroing. Nonetheless, the zeroing remains on the application's critical path, and as we show in Section 5.4, even with these optimizations, the direct cost of non-temporal bulk zeroing still adds around 5% on average to total execution time. Hardware parallelism creates the opportunity to perform zeroing concurrently, moving this overhead off the application's critical path. The mechanics of concurrent zeroing require some synchronization between the zeroing thread and the application threads consuming zeroed memory. The primary challenge for this design point is ensuring that this synchronization does not dominate performance.

We utilize a single initializing thread that zeroes memory. The JVM wakes up this thread up at the end of each nursery garbage collection, and it zeroes all blocks freed by the nursery collection. We use non-temporal instructions here as well, for the same reasons we described above. The initializing thread maintains a synchronized global cursor that consumer threads monitor to determine the progress of zeroing. The application's allocation slow path acquires memory from a global pool one block at a time as usual. However, it no longer zeros the newly acquired block, but instead busy-waits on the global zero cursor until the initialing thread has zeroed the block it will consume. Because the zeroing thread is typically well ahead of the application threads, the allocating consumer threads rarely wait.

Alternative design options. A natural extension of this design is to use multiple, parallel zeroing threads to further utilize parallel hardware and to improve the zeroing throughput. This design is particularly appealing when the application is multithreaded and allocates at a sufficiently high rate that outpaces the zeroing thread. The DaCapo benchmark lusearch is an example of such an application. However, this design requires coordination among the zeroing threads. There are two classic alternatives. First, the space can be statically partitioned, with each thread zeroing pre-determined blocks. This design requires no synchronization among the zeroing threads, but complicates the implementation of the global cursor that the application threads must check. Worse yet is that the throughput will be bounded by the slowest thread, which is problematic when a thread goes to sleep or is otherwise disrupted. The second alternative is for threads to race to zero blocks without a pre-determined order, which requires synchronization among the zeroing threads. This approach remains susceptible to one of the zeroing threads being interrupted whilst zeroing a block. Since application progress may be blocked whenever any thread is interrupted, parallel zeroing is more susceptible to slow allocation than single-threaded concurrent zeroing because there are more opportunities for threads to be blocked. We evaluated this design and found it to be substantially less effective than single-threaded concurrent zeroing, so we do not consider it further.

Scheduling priorities. One pitfall of concurrent zeroing is that the zeroing thread may be preempted by the application, particularly since we use a busy-wait loop for the application threads. To prevent the application from being starved of zeroed memory, we experimented with adjusting the operating system scheduler priority for the zeroing thread. We found that the most effective design was to give the initializing thread a real-time priority, which requires root privileges under linux. Although this alternative does improve performance by 1% on average, the gain is never substantial, and because it requires root privileges, we do not consider this design further.

3.5 Adaptive Zeroing

The lack of an effective design for parallel zeroing threads means that high allocation rate multi-threaded workloads, such as lusearch, overwhelm concurrent zeroing and incur high overheads (see Section 5.5). Because of this problem, we develop a simple adaptive strategy for use on hardware with good memory scalability, which conditionally uses either non-temporal bulk zeroing or concurrent non-temporal bulk zeroing. Our strategy simply checks at the end of each nursery collection whether the number of active application threads is less than the number of available hardware contexts. If there is a surplus of hardware contexts that the application is not using, the allocator uses concurrent non-temporal bulk zeroing until the next garbage collection. If the application is using all the hardware contexts, the allocator uses non-temporal bulk zeroing. We see in Section 5.6 that adaptive zeroing is the most effective technique for reducing the overhead of zero initialization across all benchmarks.

3.6 Summary

This Chapter introduced the design and implementation of four different zeroing initialization designs. Each of them makes a different trade-off between the throughput of zero initialization and temporal locality. The next chapter describes the methodology I will use to evaluate these designs on mainstream x86 CMPs in Chapter 5.

Chapter 4

Experimental Methodology

This chapter describes the experimental environment used in the remainder of the thesis. Section 4.1 explains the benchmarks, virtual machines, and operating systems used . Section 4.2 describes the machines on which we evaluate designs.

4.1 Software platform

Benchmarks We use 20 benchmarks drawn from the DaCapo [Blackburn et al., 2006] and SPECjvm98 [SPEC corporation, 1999] suites, and a modified version of SPECjbb2005 [SPEC corporation, 2005], pjbb2005, modified to run a fixed workload (8 warehouses with 10,0000 transactions per warehouse) rather than for a fixed duration. Table 4.1 lists key characteristics of each benchmark. Of the 20 benchmarks, pjbb2005, hsqldb, lusearch, xalan, avrora, and sunflow are multi-threaded benchmarks. We use the set of benchmarks from both the 2006-10-MR2 and 9.12 Bach releases of the DaCapo suite, excluding a small number of 9.12 benchmarks that we could not run on Jikes RVM and those 2006-10-MR2 benchmarks that are directly superseded in 9.12.

Java Virtual Machine We explore the performance of zeroing using OpenJDK 1.6.0 Oracle HotSpot Server JVM and the Jikes RVM 3.1.1 release [Alpern et al., 2005]. We perform most experiments and implement all the approaches in Jikes RVM, a highly tuned research-oriented VM. We show that the cost of the two traditional zeroing approaches in HotSpot have similar overheads and show the same trends as in Jikes RVM to demonstrate that these results have broad applicability.

In Jikes RVM, we use the default *production* generational Immix garbage collector [Blackburn and McKinley, 2008].

We use a 32MB fixed size nursery divided into 32KB blocks. We execute with a generous heap size: 6× the minimum required for each individual benchmark, as reported in the Heap column of Table 4.1. For each invocation of each benchmark, we run four warm-up iterations and then measure the fifth iteration. We run each benchmark 20 times (20 invocations) and report the average. The Total Allocation column of Table 4.1 shows the average volume of nursery objects allocated over 20

Benchmark	Suite	Heap MB	Total Allocation MB	Allocation Rate GB/s	Multi-threaded
antlr	DaCapo MR2	144	269	0.31	No
bloat	DaCapo MR2	198	1218	0.42	No
eclipse	DaCapo MR2	480	2889	0.20	No
fop	DaCapo MR2	240	78	0.09	No
hsqldb	DaCapo MR2	762	130	0.14	Yes
avrora	DaCapo Bach	300	73	0.02	Yes
jython	DaCapo Bach	240	1476	0.50	No
luindex	DaCapo Bach	132	54	0.08	No
lusearch	DaCapo Bach	204	8170	5.02	Yes
pmd	DaCapo Bach	294	435	0.47	No
sunflow	DaCapo Bach	324	1878	0.87	Yes
xalan	DaCapo Bach	324	1157	1.11	Yes
compress	SPECjvm98	114	105	0.05	No
db	SPECjvm98	114	53	0.04	No
jack	SPECjvm98	102	241	0.39	No
javac	SPECjvm98	198	218	0.22	No
jess	SPECjvm98	114	266	0.65	No
mpegaudio	SPECjvm98	78	7	0.01	No
mtrt	SPECjvm98	120	143	0.42	No
pjbb2005	SPECjbb2005	1200	2045	0.55	Yes

Table 4.1: Benchmark characteristics

invocations under the default bulk zeroing strategy. The Allocation Rate column shows the allocation rate of benchmarks on the i7-2600 in terms of execution time.

Operating System We use the Ubuntu 10.04.01 LTS server distribution running with a 64-bit (x86_64) 2.6.32-24 Linux kernel. To maximize performance for multi-threaded benchmarks, we turn off the SD_WAKE_AFFINE flag and turn on the SD_WAKE_IDLE flag to wake tasks up on idle CPUs—rather than the CPU on which they slept—to improve load balancing.

4.2 Hardware platform

We use three hardware platforms to explore the performance of zeroing on real hardware with different technologies, memory systems, and memory bandwidth provisioning. We choose three modern x86 processors to explore recent trends on contemporary hardware. We choose (1) the Core2 Quad which has a classic front-side bus, which eases memory system analysis, (2) the most recent Intel processor (i7-2600) we could buy, and (3) the recent six core AMD Phenom II.

Core2 Quad Q6600 We use a 65nm 2.4GHz Intel Core2 Quad Q6600 that includes two dies in a single package. On each die, there are two cores running at 2.4GHz, a 32KB L1 instruction cache and 32KB L1 data cache for each core, and a shared 4MB L2 inclusive cache (making a total of 8MB L2 across the two dies). The motherboard uses the G965 chip-set, which has a dual-channel memory controller. It has 2GB of DDR2-800 memory installed.

The characteristics of the Core2 Quad processor eases a detailed analysis of memory traffic. The two dies are connected by a 1066MHz front side bus (FSB). For normal memory references, the FSB transfers data between caches and memory in cache-line sized units (64 bytes), which means that we can measure the size of data transferred across the FSB by counting the number of full cache-line (burst) transactions. Two types of memory references generate fetch transactions to retrieve data from memory to cache: program cache misses and prefetching misses that are generated by the hardware automatic prefetching unit. We use performance counters to count the number of last level program cache misses, and the number of last level cache prefetching misses to help to understand the memory traffic on the bus.

i7-2600 We use a 32nm Core i7-2600 Sandy Bridge with 4 cores and 2-way SMT running at 3.4GHz. The two hardware threads on each core share a 32KB L1 instruction cache, 32KB L1 data cache, and 256KB L2 cache. All four cores share a single 8MB last level cache. A dual-channel memory controller is integrated into the CPU. It has 4GB of DDR3-1066 memory installed.

Phenom II X6 1055T We use a 45nm AMD Phenom II X6 1055T which has 6 cores and runs at 2.8GHz. Each core has a private 64KB L1 instruction cache, 64KB L1 data cache, and 512KB L2 cache. The six cores share a single 6MB L3 cache. A dual-channel memory controller is integrated into the CPU. 4GB of DDR3-2000 memory is installed.

Chapter 5

Results

This chapter quantitatively evaluates the cost of zero initialization, including the four individual design points outlined in Chapter 3 and the adaptive policy which selects between the two best designs. I start by showing the direct cost of zero initialization in Section 5.1. I then quantitatively explore the performance and tradeoffs made by hot-path zeroing and bulk zeroing in Section 5.2 and Section 5.3. I finish by exploring the new design points and the combined adaptive system in Section 5.4, Section 5.5, and Section 5.6.

5.1 Direct Cost of Zeroing Initialization

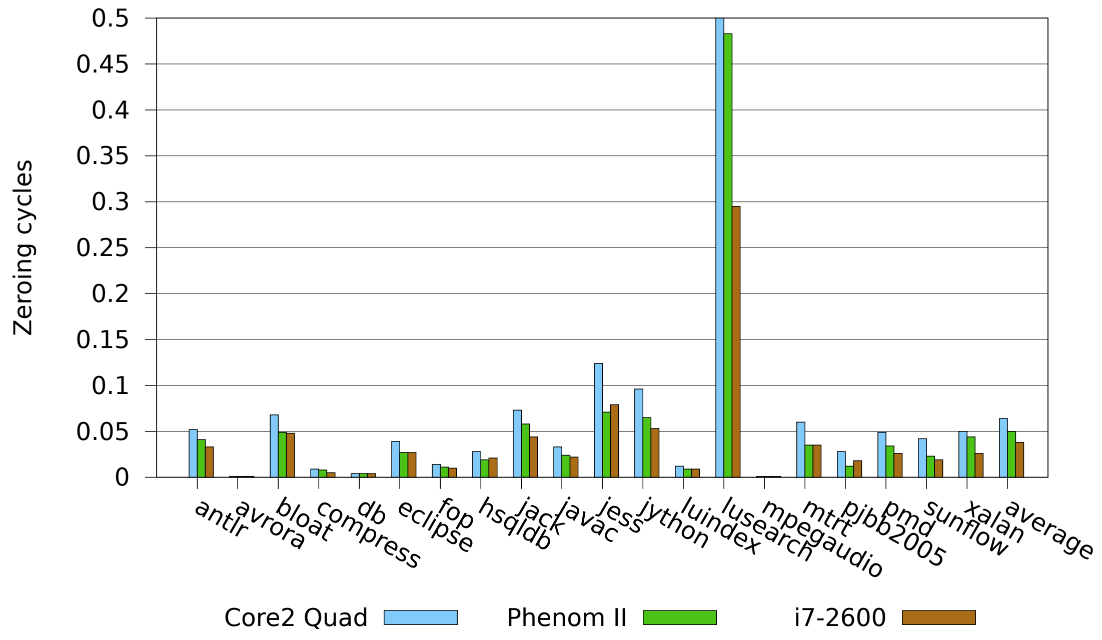
This section evaluates the direct costs of zeroing. Because bulk zeroing occurs at a coarse grain, it is easier to analyze than hot-path zeroing, where zeroing instructions are enmeshed at a fine grain within the allocation sequence. For the same reason, we use bulk zeroing as our baseline for comparison throughout the rest of the thesis unless otherwise mentioned. Here we use performance counters to report the number of cycles spent performing bulk zeroing.

The *direct* cost of zero initialization is the CPU time spent performing zero initialization computed as a fraction of the total CPU user time (*User Time*) and system time (*System Time*). *System Time* includes CPU cycles used by the OS on behalf of the process.

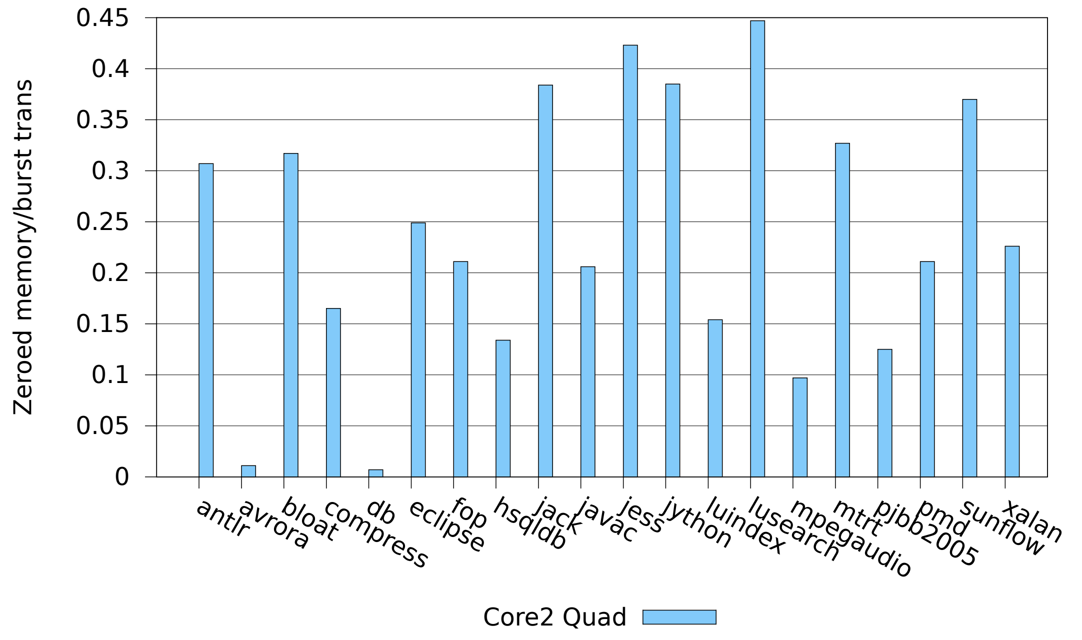
$$DirectZeroingCost = \frac{ZeroingCycles}{UserTime + SystemTime}$$

Note that this metric does not include the *indirect* costs such as reduced application locality due to cache displacement.

Figure 5.1(a) shows the results for all three architectures. On the i7-2600, zeroing consumes an average of 3.8% of total time, with almost a 30% overhead for lusearch! Just under half of the benchmarks spend 5% or more of their CPU cycles performing zero initialization. On the Phenom II, whose memory bandwidth is between Core2 Quad and i7-2600, the average direct cost is between those two architectures, 5%. On the Core2 Quad, which represents more memory bandwidth constrained platforms,



(a) Fraction of cycles spent on zeroing



(b) BytesZeroed / BytesBurstTransactionsTransferred

Figure 5.1: The cost of zero initialization

the cost of zero initialization increases to 6.4% on average, and up to as much as 50% on lusearch. More than half of the benchmarks spend more than 5% of all CPU cycles on zero initialization. We found the high direct cost of zeroing surprising.

Figure 5.1(b) shows the overall importance of zeroing by benchmark in terms of the amount of zero initialized memory as a fraction of the amount of memory transferred by burst bus transactions. This fraction ranges from 10 to 45% for most benchmarks. Only for avrora and db does zeroing account for negligible traffic, and as we will see, these benchmarks unsurprisingly show no sensitivity to the choice of zeroing strategy. The trend towards high allocation rates in managed languages such as Java, C#, PHP, and JavaScript, suggests that these low allocation rate programs may be outliers.

Lusearch incurs the highest direct cost of zero initialization among the benchmarks we used. This is because the multithreaded lusearch benchmark allocates at more than nine times the average rate and four times the rate of the next highest benchmark, as shown in Table 4.1. Although lusearch is an outlier among our benchmarks, it is important to note that it is a realistic benchmark based on Apache Lucene which is highly tuned and very widely deployed.

5.2 Hot-path Zeroing

This section evaluates hot-path zeroing, which is the most widely used strategy. Hot-path zeroing trades decreased zeroing performance for greater temporal locality due to a short reuse distance between storing zero and the first time the program touches the objects allocated on each cache line. We start by comparing overall performance against bulk zeroing, reporting results for all three microarchitectures.

Overall performance. Figure 5.2(a) shows the relative performance of hot path zeroing compared to bulk zeroing on all three architectures. These results show that hot-path zeroing offers a small advantage over bulk zeroing, but that advantage is steadily shrinking as we move to newer technology. Compared with bulk zeroing, hot-path zeroing performs on average 2.2% faster on the low-bandwidth Core2 Quad. On the higher bandwidth Phenom II and i7-2600, the advantage of hot-path zeroing is halved to just 1.0% and 0.8% respectively. Four benchmarks benefit a lot from hot-path zeroing: antlr, jack, jess, and lusearch. Table 4.1 shows that these correspond to several of the more highly allocating benchmarks. However, most benchmarks are neutral to this choice, including eclipse and jython, which are highly allocating as well.

We confirmed that these results hold on the HotSpot JVM. We added our optimized bulk zeroing to HotSpot and found similar results to those reported here for Jikes RVM, with hot-path zeroing providing a negligible advantage on modern machines (2.7% and just 0.1% speedup on the Core2 Quad and i7-2600 respectively). The remainder of this section explores hot-path zeroing in greater detail. We use the Core2 Quad for this analysis because it uses a front side bus which allows us to measure memory traffic via hardware performance counters. The two newer machines have integrated memory controllers and do not expose such detailed memory traffic information via performance counters.

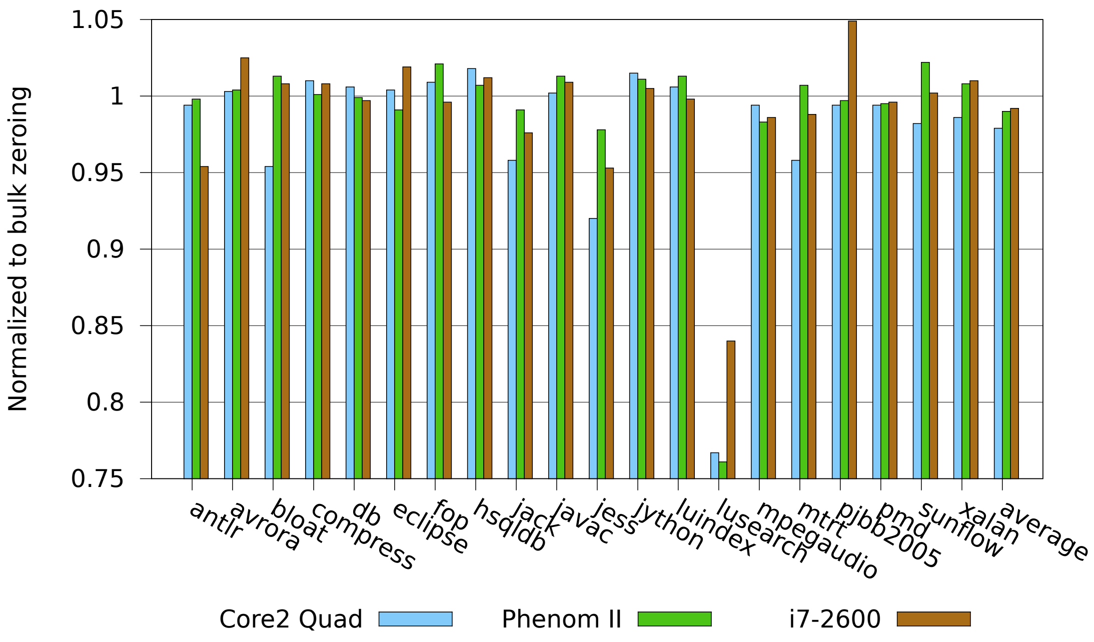
Although hot-path and bulk zeroing perform similarly on average, in Section 5.3 we show that they achieve this performance by making very different tradeoffs.

Instruction footprint. As mentioned in Section 3.1, the compiler often inlines the allocation sequence to improve performance. The compiler thus sprinkles additional hot-path zeroing instructions all over the program, generating a lot of instructions compared to a single out-of-line loop for bulk zeroing. As shown in Figure 5.2(b), this increase degrades instruction locality, increasing the number of instruction cache misses by 1.3% for hot-path zeroing compared to bulk zeroing. For several benchmarks (antlr, db, javac and jython), the instruction cache misses increase by more than 5%. This overhead is often significant. For example, the decrease in program cache misses in jython of 70% suggests the potential for a performance win, however, poor instruction cache locality counteracts this potential resulting in a net performance degradation.

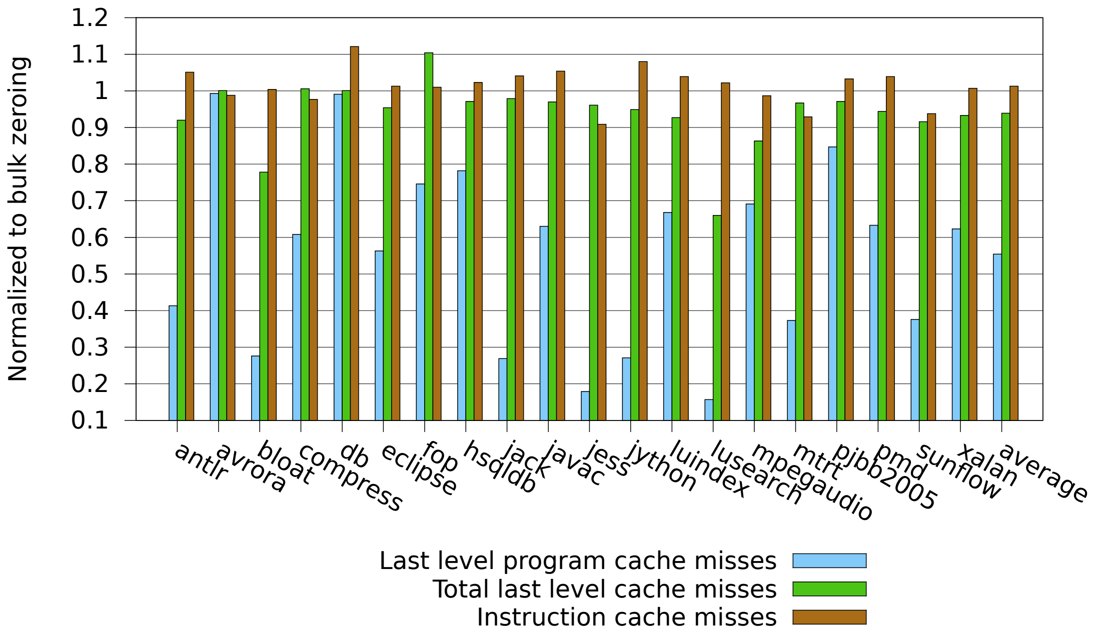
Data locality. We divide cache misses into those generated directly by the program (*program cache misses*), those generated by automatic hardware prefetching of memory (*prefetching cache misses*), and sum their total. Figure 5.2(b) shows that hot-path zeroing reduces total last-level cache misses by 6% on average on the Core2 Quad compared to bulk zeroing. Although the graph shows a large reduction in program misses, this is a little misleading because the reduction is mostly offset by an increase in prefetch misses. This is explained because hot-path zeroing zeros more slowly, providing more opportunity for memory requests to be satisfied by hardware prefetches. The penalty of slower zeroing throughput for hot-path zeroing counteracts its improved temporal locality. In a few cases, hot-path zeroing reduces more than 70% of the last-level program cache misses, e.g., bloat, jack, jess, and lusearch, which leads to 4.6%, 4.8%, 8% and 24% speedups respectively.

Compared with hot-path zeroing, bulk zeroing zeros larger continuous chunks sequentially in a tight loop, and consequently places higher demands on the bus, leading to less aggressive automatic prefetching, but more program misses. On the other hand, hot-path zeroing intertwines zeroing instructions with the allocation sequence and its surrounding context, and thus spreads the stores out, placing less pressure on the bus, as shown in Figure 5.3(a). Compared with the default bulk zeroing, hot-path zeroing results in more prefetch requests. Because the program does not uses the zeroed lines quickly, these results show that prefetching is not very effective at reducing the latency of the misses.

Memory performance. Figure 5.3(a) compares burst bus transactions, using performance counters to quantify the number and type of transactions. *Burst* transactions correspond to full cache line transfers, which form the vast majority of bus activity

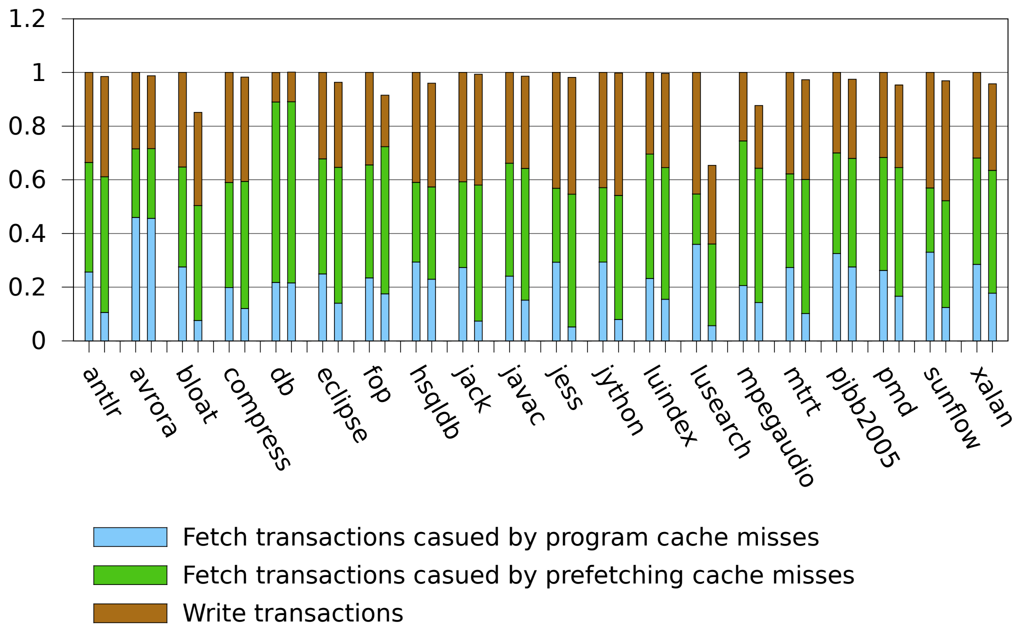


(a) Performance on Core2 Quad, Phenom II, and i7-2600

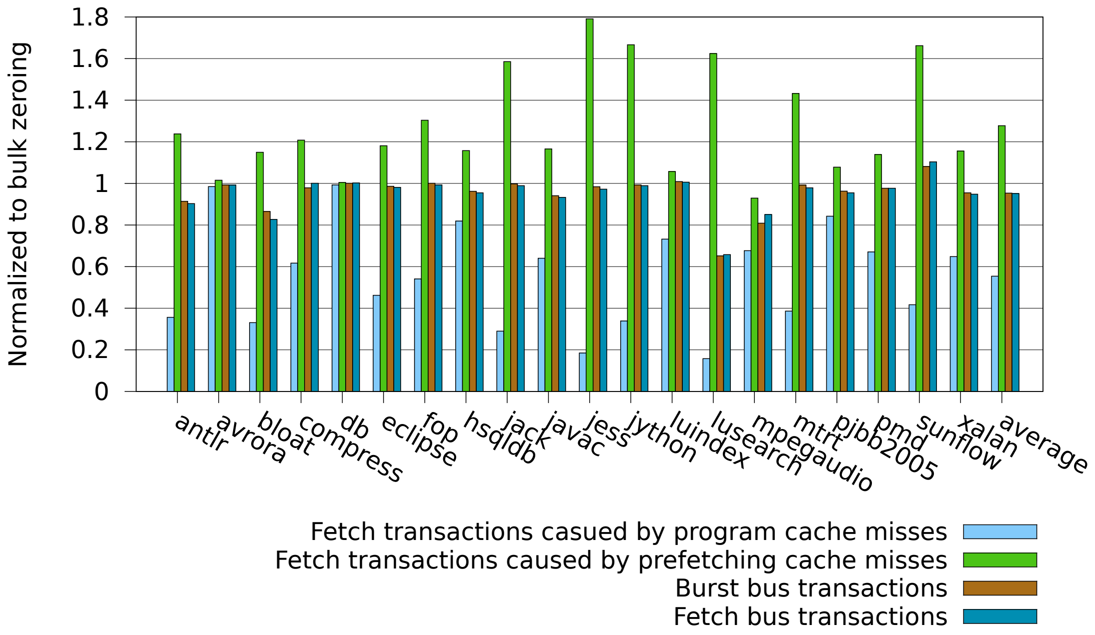


(b) Cache misses on Core2 Quad

Figure 5.2: Overall performance: Hot-path zeroing



(a) Bus burst transaction breakdown on Core2 Quad



(b) Bus burst transactions relative to bulk zeroing on Core2 Quad

Figure 5.3: Memory bus utilization: Hot-path zeroing

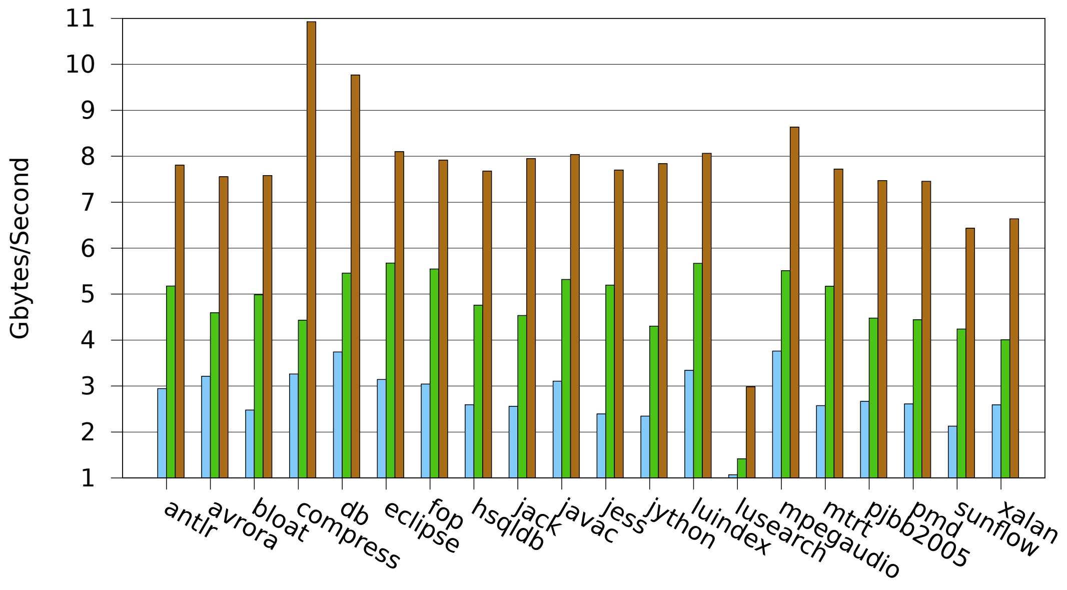


Figure 5.4: Zeroing performance: Bulk zeroing

for our workloads (the number of partial bus transactions is near zero). For each benchmark, the left bar shows bulk zeroing while the right bar shows hot-path zeroing. Each bar is broken down into *write*, *program cache misses*, and *prefetching cache misses* normalized to bulk zeroing. The first bar in each pair thus sums to 1.0, while the height of the second bar shows that hot-path path zeroing reduces total bus transactions by reducing program cache misses. Figure 5.3(b) normalizes *each metric* to the corresponding metric for bulk zeroing. Figure 5.3(b) shows that on average hot-path zeroing reduces burst transactions by only 5.2% on average, despite reducing last level program cache misses by 45% because it increases transactions caused by prefetching cache misses by 27%.

5.3 Bulk Zeroing

This section explores the behavior of bulk zeroing across two dimensions: a) raw *performance*, and b) the relationship between a benchmark's *CPU utilization* and its zeroing performance.

Zeroing performance. On modern CPUs, zeroing performance is primarily determined by two factors: the cache hit-rate, and the degree of contention on the shared memory subsystems. We measure the average number of cycles taken to zero a block of memory for each benchmark, and express the result in GB/sec.

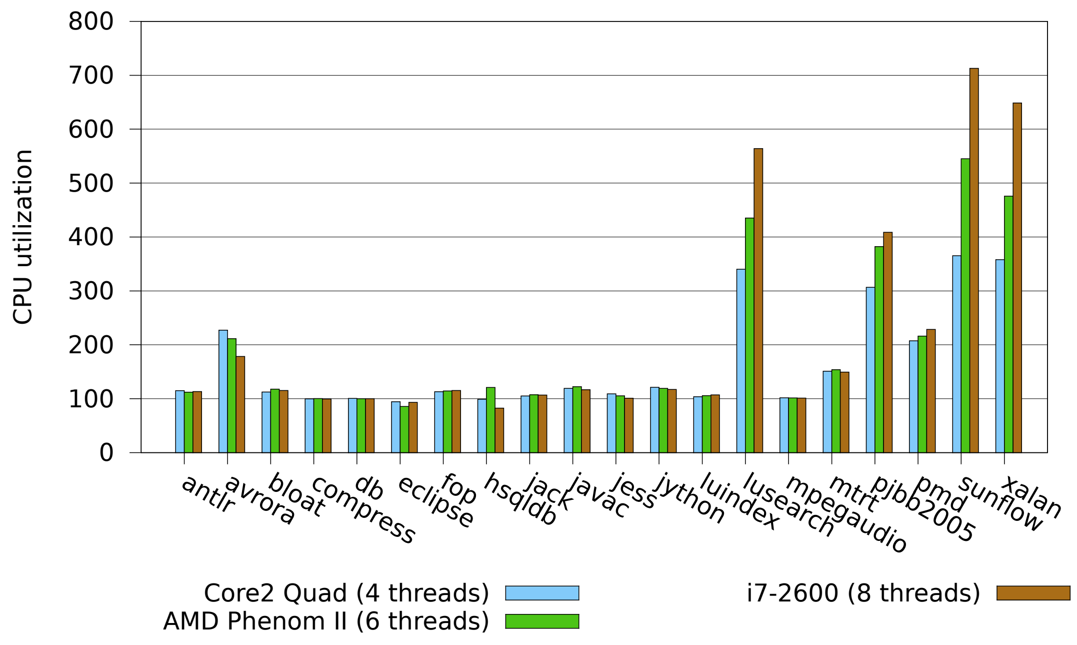


Figure 5.5: CPU utilization: Bulk zeroing

$$ZeroingPerformance = \frac{BytesZeroed}{ZeroingCycles}$$

Figure 5.4 shows the zeroing performance in the context of each benchmark. Note that the figure does *not* present the total memory zeroed by a benchmark divided by its execution time. The Core2 Quad, Phenom II, and i7-2600 bulk zero at around 3, 5, and 8 GB/sec respectively. Figure 5.4 shows that the zeroing performance is quite uniform among these benchmarks. The notable exception is lusearch, which zeroes at the lowest rate and uses the highest percentage of CPU cycles zeroing, as we explain next.

CPU utilization. CPU utilization provides an indication of the degree of contention to shared memory subsystems — if all CPUs are fully utilized, pressure on the memory subsystem is likely to be high. We derive CPU utilization based on user time, system time, and total execution time.

$$CPUUtilization = \frac{UserTime + SystemTime}{ExecutionTime}$$

Because *UserTime* and *SystemTime* are aggregated across threads, *CPUUtilization* is bounded by N ($N \times 100\%$), where N is the number of available hardware contexts. For example, the Core2 Quad, Phenom II, and i7-2600 have 4, 6, and 8 hardware

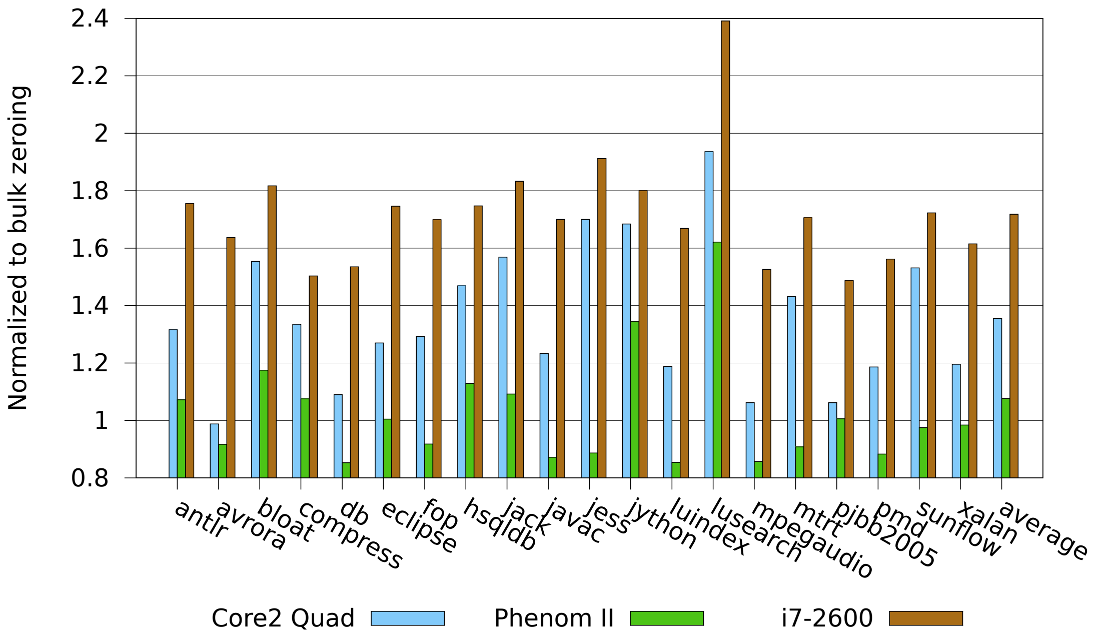


Figure 5.6: Zeroing performance: non-temporal bulk zeroing

contexts, and thus maximum *CPUUtilization* is 400%, 600%, and 800% respectively. Another approach to show CPU utilization is normalizing user time and system by dividing by the number of cores, which expresses the same information as ours.

Figure 5.5 shows CPU utilization. Four benchmarks — lusearch, pjbb2005, sunflow, and xalan — have relatively high CPU utilization, which suggests that contention to shared memory subsystems is high when they are executing. High CPU utilization depresses the average zeroing performance of pjbb2005, sunflow, and xalan somewhat, but less significantly than lusearch. The zeroing performance of lusearch is the worst because it has the highest allocation rate, high CPU utilization, and incurs significant contention. Sunflow and xalan have lower zeroing performance, especially on i7-2600, in part due to higher contention on shared memory subsystems.

Tradeoffs and trends. Together the bulk and hot-path zeroing results show that the two designs impose a significant overhead and that there is a tradeoff between the direct and indirect costs of zeroing. While hot-path zeroing has better data cache hit rates, it degrades code quality and does not hide memory latencies well. On the other hand, the performance of bulk zeroing is a function of memory bandwidth. Modern machines have increased bandwidth to a point where bulk zeroing essentially matches hot-path performance. However as future CMPs increase hardware parallelism, that bandwidth will be increasingly contended for. In summary, we see a stark tension between: a) bulk zeroing which has low direct costs but suffers significant cache pollution, and b) hot-path allocation which reduces data cache pollution, but imposes a significant direct cost on the program. The next three sections show how we break this tradeoff and produce a system that performs better than either of the prior approaches.

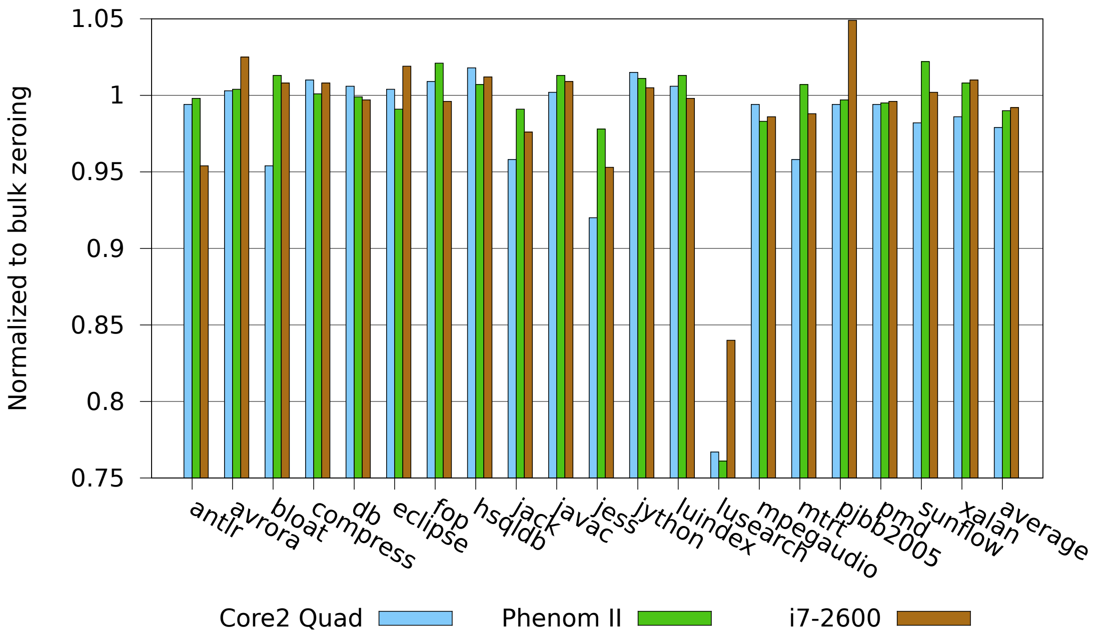
5.4 Non-temporal Bulk Zeroing

This section evaluates non-temporal bulk zeroing. The non-temporal writes of zeroes bypass the cache hierarchy and thus reduce the direct cost of bulk zeroing and cache pollution. Such a design should offer the benefits of bulk zeroing whilst minimizing its drawbacks.

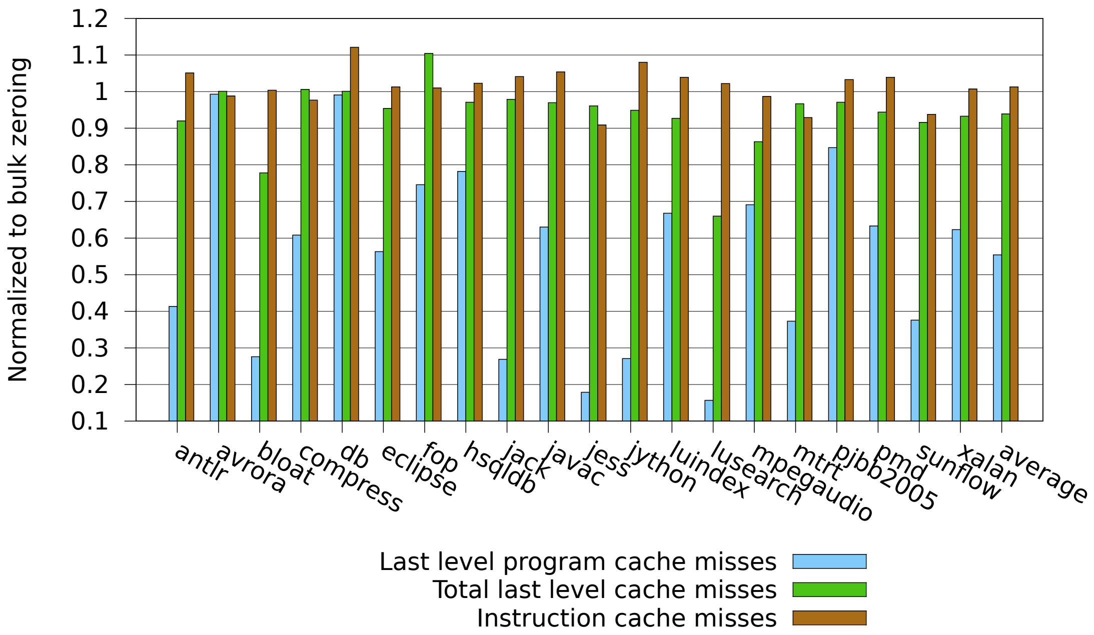
Zeroing performance. Figure 5.6 shows zeroing performance (not overall performance) for non-temporal bulk zeroing, normalized to bulk zeroing. We use the same metric and methodology as in Figure 5.4: how fast, on average, does the system zero a 32KB block in the setting of a given benchmark? Remember that Section 3.3 showed non-temporal instructions have much higher bandwidth than temporal instructions, so we expect an improvement. On the Core2 Quad, Phenom II, and i7-2600, zeroing performance for non-temporal bulk zeroing improves by 35%, 7%, and 71% on average. Overall, non-temporal instructions significantly improve the average rate at which these benchmarks zero a 32KB block on Core2 Quad and i7-2600.

Overall performance. Figure 5.7(a) shows the effect of non-temporal bulk zeroing on overall performance on all three architectures, normalized to bulk zeroing. On the Core2 Quad, non-temporal bulk zeroing improves execution time by 2.7% on average and by as much as 27% for lusearch. These improvements reflect reductions in the last-level program cache misses shown in Figure 5.7(b) of 43% on average and up to 92% for lusearch. Execution times on the Phenom II and i7-2600, also improve, but by less. The benefits of non-temporal zeroing come from reducing the *direct* cost and the *indirect* cost by avoiding cache pollution. For example, on the Core2 Quad, bloat spends 5% of CPU cycles zeroing memory. Non-temporal instructions improve the zeroing performance on bloat by 50%, suggesting that at best bloat's execution time could be reduced by $(1 - 1/1.5) \times 5\% = 1.7\%$ with non-temporal zeroing. However, in practice the overhead is reduced by 4.4%, almost entirely eliminating any overhead due to zeroing! This result indicates that the *indirect* benefit due to hugely reduced cache pollution is substantial. Similar results hold for benchmarks such as lusearch and jess.

Memory performance. Although non-temporal zeroing reduces cache displacement, it increases the number of bus transactions. Because non-temporal zeroing explicitly invalidates all resident cache lines that it writes to, each zero is always accompanied by a subsequent memory read so long as that line is eventually used. By contrast, regular bulk zeroing will leave the zeroed line in cache, attaining cheap cache line reuse if the program reuses it promptly.

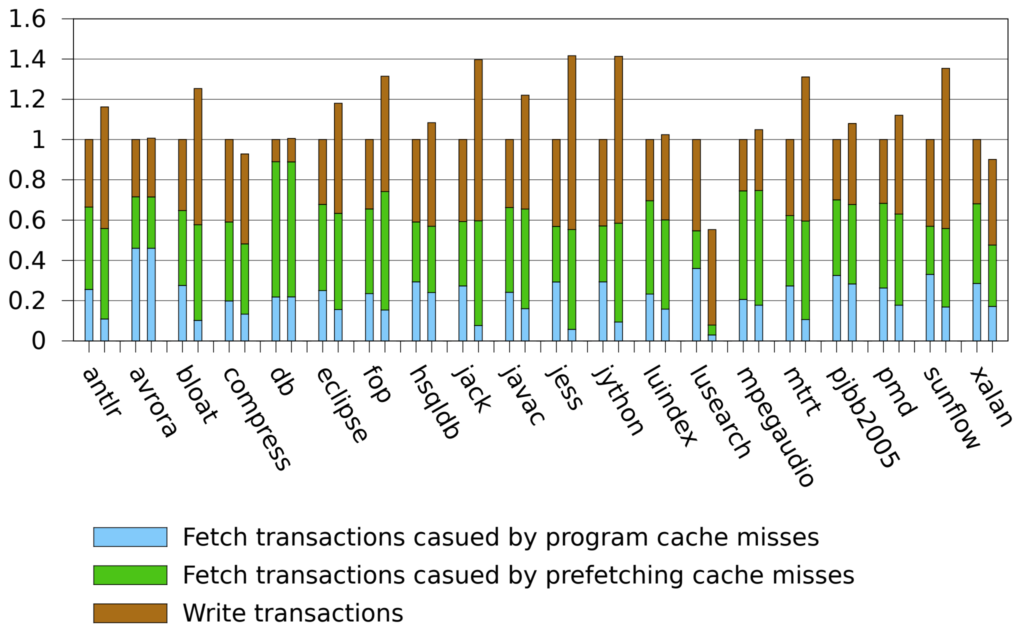


(a) Overall performance of non-temporal bulk zeroing

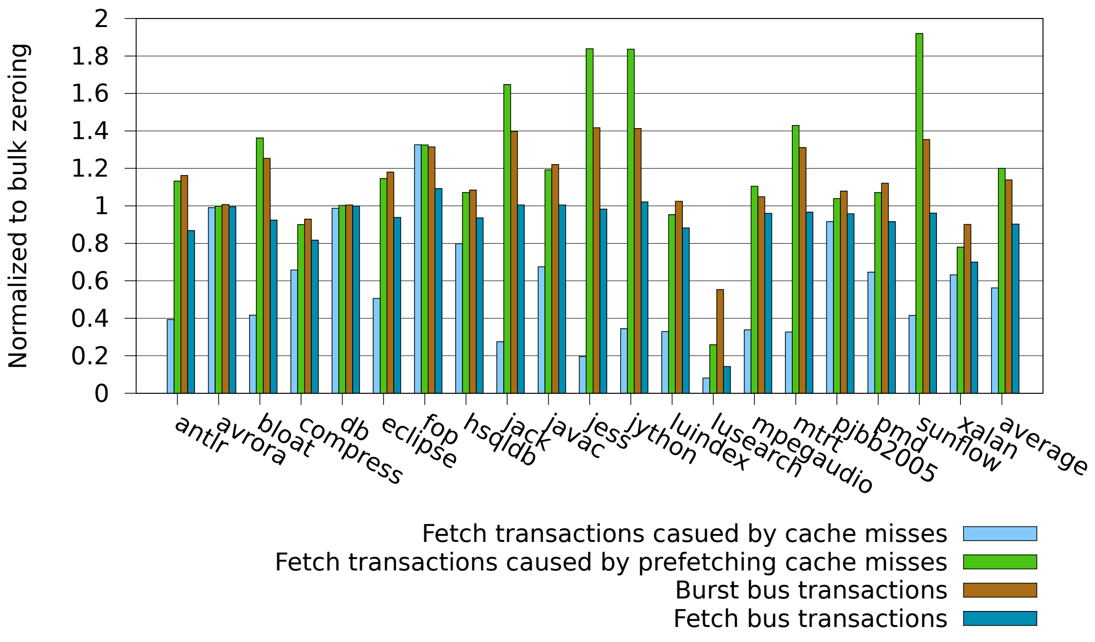


(b) Cache misses on Core2 Quad

Figure 5.7: Overall performance: Non-temporal bulk zeroing



(a) Bus burst transaction breakdown on the Core2 Quad.



(b) Bus burst transactions relative to bulk zeroing on the Core2 Quad.

Figure 5.8: Memory bus utilization: Non-temporal bulk zeroing

Benchmark	Direct cost of zeroing %	Concurrent zeroing improvement %
antlr	5.0	4.6
bloat	6.8	8.3
jack	7.3	6.3
jess	12.0	14.0
jython	9.6	5.7
mtrt	6.0	6.8

Table 5.1: Zeroing cost and concurrent zeroing improvements for high cost, low CPU utilization benchmarks on the Core2 Quad.

Figure 5.8(a) shows the number of burst bus transactions, broken down in the same way as in Section 5.2. In Figure 5.8(a), write transactions increase, except for db and avrora. They increase because unlike the temporal zero instruction which generates a fetch, the non-temporal zeroing instruction generates a write and invalidates the resident cache line, then the program later references the line, instantiating it in the cache. The program typically writes to the newly allocated object, which dirties the cache line, and thus requires another write.

Figure 5.8(b) shows each type of bus transactions normalized to default bulk zeroing. On average, burst transactions are increased by 13%. For benchmarks with higher zeroing cost (jython, jess, and jack), burst bus transactions are increased by 40%. As with hot-path zeroing, the burst transactions for db and avrora are unaffected by non-temporal zeroing.

Figure 5.8(b) shows that non-temporal bulk zeroing reduces fetches by 15% on average and up to 86% for lusearch. However, the fetches due to prefetching cache misses increase by 20% on average, and by as much as 80% or more for jess, jython and sunflow. Because non-temporal zeroing explicitly evicts each line from the cache, the first touch to an allocated object forces the processor to fetch it into the cache. Sequentially touching new objects appears to activate the processor's prefetching logic resulting in a larger number of prefetch requests being issued. However, this 20% increase in prefetching cache misses is lower than the 27% average increase seen by hot-path zeroing.

5.5 Concurrent Zeroing

Although non-temporal bulk zeroing improves zeroing performance, the benchmarks still spend on average 2 to 4.6% and as much as 26% of total time performing zero initialization on the Core2 Quad. This section evaluates the concurrent zero initialization design that exploits hardware parallelism to hide the cost of zero initialization.

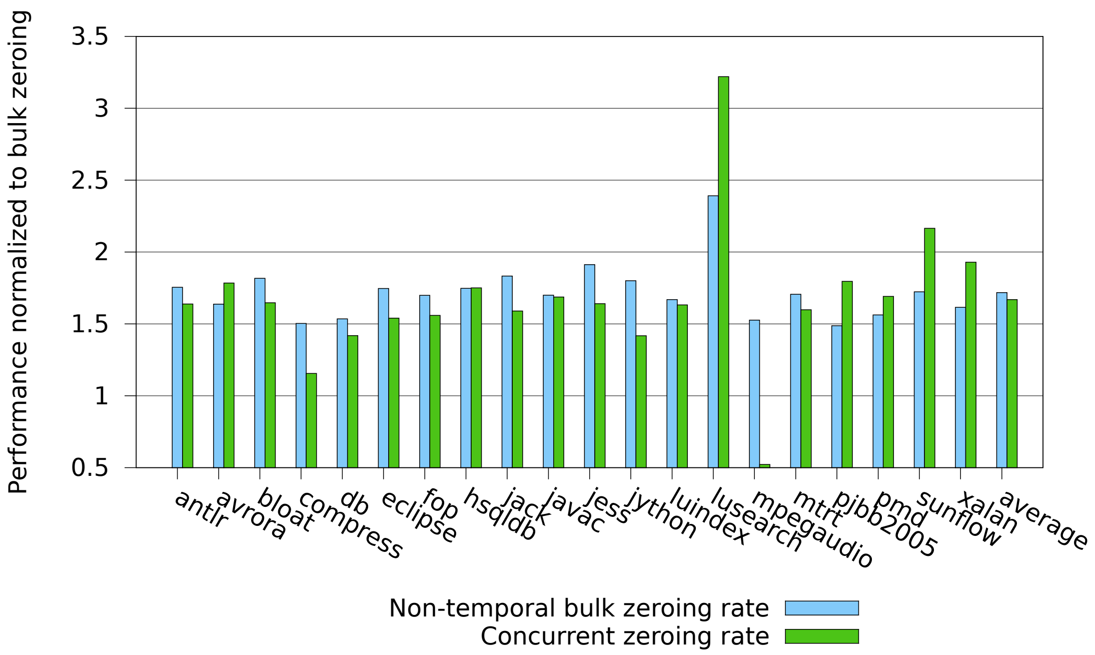


Figure 5.9: Zeroing performance: Non-temporal v concurrent on the i7-2600

Zeroing performance. Figure 5.9 compares non-temporal bulk zeroing to concurrent non-temporal bulk zeroing. These results are normalized to default bulk zeroing, and higher is better. For low CPU utilization benchmarks, the zeroing performance of a single concurrent thread is slightly worse than non-temporal bulk zeroing. Interestingly, this poor zeroing performance is due to good locality. When the application has low CPU utilization, the zeroing thread will execute unfettered, rapidly zeroing the nursery soon after the completion of the prior garbage collection. By contrast, bulk zeroing uses a large number of 32KB blocks interspersed with application activity, which displaces the cache. Concurrent zeroing for low CPU utilization threads therefore achieves good temporal and spatial locality. Perversely, this good locality leads to lower memory bandwidth since the data is typically in cache and must therefore be invalidated as it is written. Because these low CPU utilization benchmarks do not exhibit high allocation rates and there are underutilized cores, concurrent zeroing hides the cost of zeroing very well, even though zeroing performance is relatively low on these benchmarks (as shown in Figure 5.10). In Table 5.1, we single out the low CPU utilization benchmarks whose zeroing cost is higher than 5% and show their speedup on the Core2 Quad. Speedup is gained both from hiding zeroing latency and from improved temporal locality.

Overall performance. Figure 5.11 shows the performance of concurrent non temporal bulk zeroing normalized to hot-path zeroing. Concurrent zeroing improves performance over hot-path zeroing by 2.1% on average and up to 8.2% on Core2 Quad. Remember that hot-path zeroing and bulk zeroing perform about the same

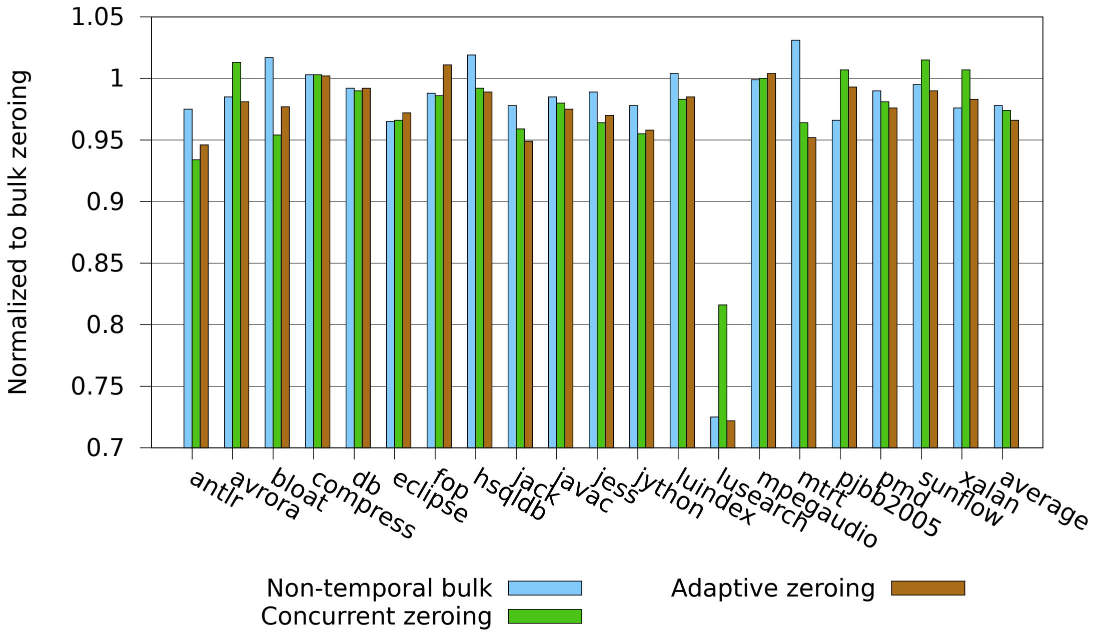


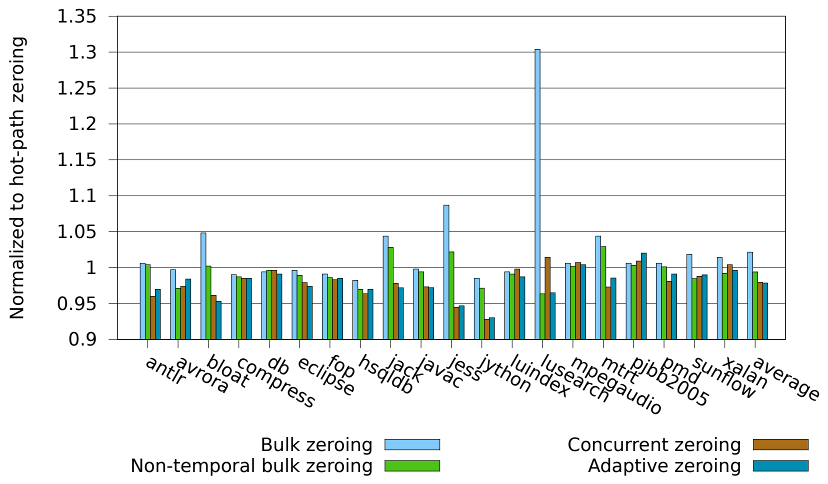
Figure 5.10: Overall performance: non-temporal bulk, concurrent non-temporal bulk, and adaptive zeroing relative to bulk zeroing on the i7-2600.

on the Phenom II and i7-2600, and therefore concurrent zeroing is the best design so far. Concurrent zeroing results in about 2.6% speedup on average and up to 19% on lusearch on the i7-2600 compared to hot-path zeroing. However, on the Phenom II, because the throughput of the zeroing thread is too low, concurrent zeroing only improves average performance 1.2%.

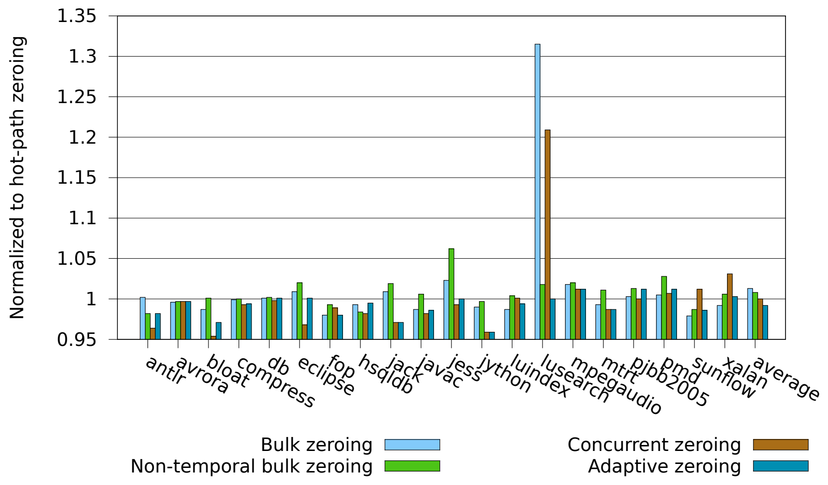
Concurrent zeroing however is not effective on xalan, sunflow, pjbb2005, lusearch, and avrora, especially on the i7-2600. In these highly multithreaded and high CPU utilization applications, the concurrent zeroing thread interferes with the application threads. Another problem is that a single zeroing thread is insufficient to provide allocation intensive benchmarks, such as lusearch, with zeroed blocks quickly enough. For example, we found that lusearch performs busy waits for a zeroed block 510 times on the i7-2600 and 15540 times on the Core2 Quad. However, the benchmarks that do not benefit from concurrent non-temporal zeroing, do well with straight non-temporal zeroing, which motivates our adaptive design.

5.6 Adaptive Zeroing

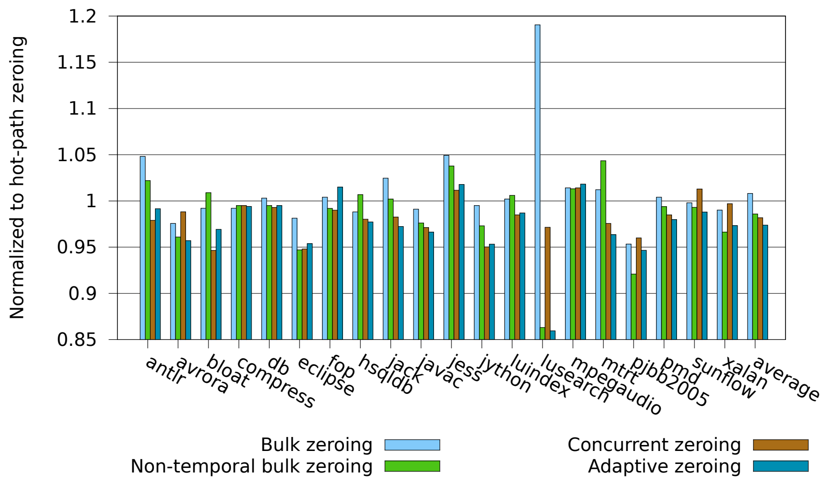
Adaptive zeroing adaptively chooses between non-temporal bulk zeroing and concurrent non-temporal bulk zeroing. At the end of each garbage collection, the system checks the number of active threads and then chooses between the two policies. This design tries to pick the best zeroing initialization choice for each benchmark. It



(a) Core2 Quad



(b) Phenom II



(c) i7-2600

Figure 5.11: Overall performance *relative to hot path zeroing*: Execution time for non-temporal bulk, concurrent non-temporal bulk, and adaptive zeroing

chooses concurrent zeroing only when the number of threads is less than or equal to the number of hardware contexts after each garbage collection. For most benchmarks, Figure 5.11 shows that adaptive zeroing selects the optimal zeroing approach. The performance of adaptive zeroing is typically the best and if not it lies between the two new strategies and is always closest to the better zeroing approach.

Overall performance. Finally, we show overall performance of each design, compared to the widely used hot-path design. Adaptive zeroing improves performance by 2.2% on average and up to 7% on the Core2 Quad, 0.8% on average and up to 4.1% on the Phenom II, and 2.7% on average and up to 15% for lusearch on the i7-2600.

5.7 Summary

In this chapter, I evaluated four zero initialization designs and adaptive policy on three mainstream x86 CMPs. These results showed that zeroing overheads are surprisingly expensive and existing designs either addressed the direct zeroing cost or the indirect cost. Two new designs I proposed reduced both direct and indirect costs simultaneously by using non-temporal store instructions. By taking advantage of these two new designs, an adaptive zeroing approach that switches among them was described and shown to achieve the best performance.

Chapter 6

Conclusion

This thesis shows that zero initialization incurs a significant overhead on modern processors and provides the first detailed analysis of those overheads. I quantitatively analyze the direct and indirect overheads of existing zero initialization designs on mainstream CMPs, and propose new designs. Unlike prior designs, these new designs exploit *both* the concurrency and non-temporal cache-bypassing instructions of modern architectures. I also propose a simple adaptive policy that dynamically chooses between the two new designs. The result is a substantial reduction in the overhead due to zero initialization, which leads to an average improvement of 2.7% over the best-performing prior technique, hot-path zeroing, across a wide range of benchmarks on the i7-2600.

The results highlight the importance of counter-intuitively sacrificing temporal locality to achieve high memory throughput and minimize cache pollution. I also point in the direction of other optimizations that could further lower the overhead of providing memory safety. These results also show an advantage of automatic memory management—the opportunity to understand the memory usage of applications accurately and dynamically adapt policies.

In the multicore era, the performance of the system depends more than ever on how the software system and hardware system cooperatively work together to improve the utilization of resources that multicore processors provide. The key to the best designs presented in this thesis lies in exploiting a detailed understanding of both hardware and software architectures. This is a very promising model for future research on the performance of managed languages on modern architectures.

6.1 Future Work

The best zero initialization policy, adaptive zeroing, chooses among new designs based on a simple heuristic which would not be reliable in a highly loaded system. Making optimal decisions requires understanding resource usage across the whole system. One approach for future work is to make the operating system scheduler more JVM aware. In the concurrent zeroing design, zeroed memory consumer threads monitor zero initialization progress by busy-waiting on the global zero cursor. In the future, we could investigate approaches to avoid overheads of busy-waiting, such as using synchronization (lock or mutex) services provided by operating system and allowing consumer threads to steal zero initialization work instead of waiting for zeroing thread. Also, bulk zeroing has the potential to take both advantages of non-temporal and temporal stores by mixing them to achieve good temporal locality and high memory throughput. In this thesis, we only evaluated zero initialization designs on single socket CMPs. In the future, we could investigate the ccNUMA effect, especially on modern interconnect network connected small scale (2-4 sockets) machines.

Bibliography

Alpern, B.; ; Augart, S.; Blackburn, S. M.; Butrico, M.; Cocchi, A.; Cheng, P.; Dolby, J.; Fink, S. J.; Grove, D.; Hind, M.; McKinley, K. S.; Mergen, M.; Moss, J. E. B.; Ngo, T.; Sarkar, V.; and Trapp, M., 2005. The Jikes RVM Project: Building an open source research community. *IBM System Journal*, 44, 2 (2005), 399–418. (cited on pages 6 and 17)

AMD. Using the x86 open64 compiler suite. http://developer.amd.com/assets/x86_open64_user_guide.pdf. (cited on page 9)

B.Kessler, P., 2007. Java HotSpot virtual machine. Talk at FOSDEM-2007. (cited on page 9)

Blackburn, S. M.; Cheng, P.; and McKinley, K. S., 2004a. Myths and realities: The performance impact of garbage collection. In *ACM Measurement and Modeling of Computer Systems (SIGMETRICS)*, 25–36. (cited on page 9)

Blackburn, S. M.; Cheng, P.; and McKinley, K. S., 2004b. Oil and water? High performance garbage collection in Java with MMTk. In *International Conference on Software Engineering (ICSE)*, 137–146. Scotland, UK. (cited on page 6)

Blackburn, S. M.; Garner, R.; Hoffman, C.; Khan, A. M.; McKinley, K. S.; Bentzur, R.; Diwan, A.; Feinberg, D.; Frampton, D.; Guyer, S. Z.; Hirzel, M.; Hosking, A.; Jump, M.; Lee, H.; Moss, J. E. B.; Phansalkar, A.; Stefanović, D.; VanDrunen, T.; von Dincklage, D.; and Wiedermann, B., 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (Portland, OR, USA, Oct. 2006), 169–190. (cited on pages 2 and 17)

Blackburn, S. M. and McKinley, K. S., 2008. Immix: A mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *ACM Programming Language Design and Implementation (PLDI)*, 22–32. (cited on pages 6 and 17)

Burger, D.; Goodman, J. R.; and Kägi, A., 1996. Memory bandwidth limitations of future microprocessors. In *Proceedings of the 23rd annual international symposium on Computer architecture*, ISCA '96 (Philadelphia, Pennsylvania, United States, 1996), 78–89. ACM, New York, NY, USA. doi:http://doi.acm.org/10.1145/232973.232983. (cited on page 1)

Click, C., 2009. Azul's experiences with hardware/software co-design. Keynote at VEE, (July 2009). (cited on page 9)

GNU. Gnu c library. http://www.gnu.org/s/libc/. (cited on page 9)

GRCEVSKI, N.; KIELSTRA, A.; STOODLEY, K.; STOODLEY, M.; AND SUNDARESAN, V., 2004. Java just-in-time compiler and virtual machine improvements for server and middleware applications. In *Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium - Volume 3* (San Jose, California, 2004), 12–12. USENIX Association. doi:http://portal.acm.org/citation.cfm?id=1267242.1267254. (cited on pages 2 and 9)

HSU, L. R.; REINHARDT, S. K.; IYER, R.; AND MAKINENI, S., 2006. Communist, utilitarian, and capitalist cache policies on CMPs: caches as a shared resource. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, PACT '06 (Seattle, Washington, USA, 2006), 13–22. ACM, New York, NY, USA. doi:http://doi.acm.org/10.1145/1152154.1152161. (cited on page 1)

INOUE, H.; KOMATSU, H.; AND NAKATANI, T., 2009. A study of memory management for web-based applications on multicore processors. In *ACM Programming Language Design and Implementation (PLDI)*, 386–396. (cited on page 1)

INTEL. Intel 64 and ia-32 architectures software developer's manuals. http://www.intel.com/products/processor/manuals/. (cited on page 8)

LIU, C.; SIVASUBRAMANIAM, A.; AND KANDEMIR, M., 2004. Organizing the last line of defense before hitting the memory wall for CMPs. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, HPCA '04, 176–. IEEE Computer Society, Washington, DC, USA. doi:http://dx.doi.org/10.1109/HPCA.2004.10017. (cited on page 1)

MOLKA, D.; HACKENBERG, D.; SCHONE, R.; AND MULLER, M., 2009. Memory performance and cache coherency effects on an Intel Nehalem multiprocessor system. In *Parallel Architectures and Compilation Techniques (PACT)*, 261 –270. doi:10.1109/PACT.2009.22. (cited on page 7)

NOVARK, G.; BERGER, E. D.; AND ZORN, B. G., 2007. Exterminator: automatically correcting memory errors with high probability. In *ACM Programming Language Design and Implementation (PLDI)*, 1–11. (cited on pages 1 and 5)

ROGERS, B.; KRISHNA, A.; BELL, G.; VU, K.; JIANG, X.; AND SOLIHIN, Y., 2009. Scaling the bandwidth wall: Challenges in and avenues for cmp scaling. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 371–382. (cited on page 1)

SIKHA, E.; SIMPSON, R.; MAY, C.; AND WARREN, H., 1994. *The PowerPC Architecture: A Specification for a New Family of RISC Processors*. Morgan Kaufmann Publishers. (cited on page 9)

SPEC CORPORATION, 1999. *SPECjvm98 Documentation*. Standard Performance Evaluation Corporation, release 1.03 edn. (cited on pages 2 and 17)

SPEC CORPORATION, 2005. SPECjbb2005 Java server benchmark. ftp://ftp.spec.org/jbb2005. (cited on page 17)

Yang, X.; Blackburn, S. M.; Frampton, D.; Sartor, J. B.; and McKinley, K. S., 2011. Why nothing matters: The impact of zeroing. In *OOPSLA 2011 (under review)*. (cited on page 1)

Yu, C. and Petrov, P., 2010. Off-chip memory bandwidth minimization through cache partitioning for multi-core platforms. In *Design Automation Conference*, 132–137. (cited on page 1)

Zhao, Y.; Shi, J.; Zheng, K.; Wang, H.; Lin, H.; and Shao, L., 2009. Allocation wall: A limiting factor of Java applications on emerging multi-core platforms. In *ACM Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 361–376. (cited on page 1)