Why Nothing Matters: The Impact of Zeroing *

Xi Yang‡, Stephen M. Blackburn‡, Daniel Frampton‡, Jennifer B. Sartor†, Kathryn S. McKinley*

‡Australian National University †EPFL *Microsoft Research *University of Texas at Austin

Abstract

Memory safety defends against inadvertent and malicious misuse of memory that may compromise program correctness and security. A critical element of memory safety is zero initialization. The direct cost of zero initialization is surprisingly high: up to 12.7%, with average costs ranging from 2.7 to 4.5% on a high performance virtual machine on IA32 architectures. Zero initialization also incurs indirect costs due to its memory bandwidth demands and cache displacement effects. Existing virtual machines either: a) minimize direct costs by zeroing in large blocks, or b) minimize indirect costs by zeroing in the allocation sequence, which reduces cache displacement and bandwidth. This paper evaluates the two widely used zero initialization designs, showing that they make different tradeoffs to achieve very similar performance.

Our analysis inspires three better designs: (1) bulk zeroing with cache-bypassing (*non-temporal*) instructions to reduce the direct and indirect zeroing costs simultaneously, (2) concurrent non-temporal bulk zeroing that exploits parallel hardware to move work off the application's critical path, and (3) adaptive zeroing, which dynamically chooses between (1) and (2) based on available hardware parallelism. The new software strategies offer speedups sometimes greater than the direct overhead, improving total performance by 3% on average. Our findings invite additional optimizations and microarchitectural support.

Categories and Subject Descriptors D3.4 [*Programming Languages*]: Processors—Memory management (garbage collection); Optimization; Run-time environments

General Terms Performance, Measurement

Keywords Memory safety, Zero initialization

* This work is supported by ARC DP0666059, NSF SHF0910818, NSF CSR0917191, NSF CCF0811524, NSF CNS0719966, Intel, Google and Microsoft Research. Any opinions, findings and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

1. 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 [26]. We show that existing approaches of zero initialization are surprisingly expensive. On three modern IA32 architectures, the direct cost is around 2.7-4.5% on average and as much as 12.7% of all cycles, in a high-performance Java Virtual Machine (JVM), 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 [9, 15, 16, 24, 28, 33, 34] 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 [16, 34]. Furthermore, energy is now constraining memory bandwidth [8]. If architects add processor cores without adding commensurate 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, such as offloading critical system services that must be done in a timely manner. To our knowledge, this paper is the first to explore the zero initialization design space and show that zero initialization is costly.

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 in Java Virtual Machines today are bulk zeroing (Jikes RVM and optionally HotSpot) and hot-path zeroing (Azul, HotSpot default, and J9 [14]). *Bulk* zeroing attacks the direct cost by zeroing memory in large chunks and exploiting instruction level parallelism, loop optimizations, and zeroing a cache line or more at a time. Bulk zeroing, however, introduces a significant reuse distance between when the VM zeroes a

cache line and when the application first uses it. This distance increases cache pollution. *Hot-path* zeroing injects zeroing instructions into the allocation sequence, attacking indirect costs by minimizing reuse distance and exploiting the hardware prefetcher to avoid stalls in modern fetch-on-write caches. Hot-path zeroing, however, expands and complicates the performance-critical allocation sequence and reduces opportunities for software optimization of the zeroing instructions. The two designs are thus at poles, addressing either, but not both, of the direct and indirect costs of zeroing.

Although this cost is significant, very little research explores zeroing costs or optimizations. We perform a detailed study in Jikes RVM and confirm the results via a preliminary implementation in the HotSpot JVM. We use 19 benchmarks from DaCapo [6], SPECjvm98 [32], and pjbb2005 [3], executing on three mainstream CMPs: an Intel Core2 Quad Q6600, an AMD Phenom II X6 1055T, and an Intel Sandy Bridge Core i7-2600. We measure the allocation rates of real and microbenchmarks to explore performance limits and costs. Our analysis reveals opportunities and tradeoffs in zeroing strategies. We show that an effective hardware prefetcher is critical to the performance of hot-path zeroing.

We introduce three better solutions. (1) Non-temporal bulk zeroing targets both direct and indirect costs using cache-bypassing instructions. (2) Concurrent non-temporal bulk zeroing targets direct costs by using parallelism to move zeroing off the application's critical path. (3) Adaptive zeroing chooses between the first two designs based on available hardware parallelism.

Our zeroing designs take advantage of non-temporal instructions and unutilized hardware parallelism to minimize zeroing costs. We demonstrate that non-temporal stores improve memory throughput and mitigate cache pollution due to bulk zeroing. Taking advantage of available hardware parallelism to move zeroing off the application's critical path further reduces the direct cost of zeroing. The best strategy adaptively chooses between concurrent and synchronous non-temporal bulk zeroing, adjusting based on the availability of unused hardware parallelism. The adaptive approach improves performance by 3.2% on average and up to 9.2% on the i7-2600. It is most effective on highly allocating, memory intensive benchmarks, which stress the memory system the most. Nonetheless, the total number of cycles devoted to zero-initialization is often substantial, which suggests that further optimization of zeroing will be useful.

The contributions of this paper are: (1) the first detailed study of the cost of zero initialization which shows zero initialization is often expensive on modern processors, (2) a detailed microarchitectural analysis of existing designs which shows they make different tradeoffs but have very similar performance, and (3) identification and evaluation of three new designs. The adaptive design uses non-temporal instructions and concurrency to provide speedups that sometimes exceed the direct cost of zero initialization.

2. Background and Related Work

Our work sits at the boundary of programming language implementation and microarchitecture. This section presents key background ideas and related work in hardware and software.

Language design. 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 [26]. Data initialization and pointer disciplines are the principal 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 specification forbids reference forging, and the implementation checks array indices, and uses garbage collection rather than manual freeing to avoid dangling references. The runtime also zero initializes all data before the program reads it. 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 initialization offer software engineering and security benefits, but they increase the number of memory operations.

Memory system design. Meanwhile, the era of chip multiprocessors is increasing pressure on memory performance [9, 15, 16, 24, 28, 33, 34]. Adding hardware parallelism increases computational power, but scaling memory performance to keep pace is challenging. Zhao et al. show that allocation-intensive Java programs create an *allocation wall* on modern chip multiprocessors (CMPs) that limits application scaling and performance [34]. Studying "partially scalable" benchmarks, they found a strong correlation between object allocation rates and memory bus write traffic, which is quickly saturated and limits scalability. Inoue et al. show that bandwidth problems are common to more than just managed languages—highly allocating web server applications written in native languages also have extremely high memory bandwidth demands that compromise performance on CMPs [16]. More generally, all shared elements of the memory subsystem, including shared caches, are increasingly subject to contention as hardware parallelism increases, both due to CMPs and simultaneous multithreading.

Memory subsystems on modern processors support intense memory activity when accesses exhibit either: a) a high degree of locality, or b) no locality whatsoever. A cache hierarchy ensures accesses that exhibit good temporal locality within a cache block have low latency. Modern hardware prefetchers hide latency when programs exhibit predictable spatial locality. For accesses lacking temporal locality, *non-*

temporal streaming instructions go directly to memory with higher memory throughput and do not displace useful data in the cache.

Jouppi investigated various cache policies and their effect on performance [21]. In particular, *write-validate* offered the best performance — it combines no-fetch-on-write and write-allocate. This policy requires per-byte valid bits to partially instantiate cache lines. This policy further motivates zeroing cache lines without reading them from memory and would improve zeroing performance; however, no modern caches use it.

Modern caches use a write-back with fetch-on-write semantics [11, 20, 22, 25]. On a write hit, the hardware writes to and marks the cache line dirty. On a write miss, the hardware first fetches the cache line, and then writes and marks it as dirty. When the cache line is evicted or is 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. When temporal locality is poor, this design limits the memory throughput since, in the worst case, every write generates a store to memory and a cache line load from memory, which is useless in the case when the line will not be read.

ISA support. Some instruction set architectures (ISAs) include special instructions that initialize the cache without fetching data from. The PowerPC ISA includes a data cache block zero (`dcbz`) instruction that zeros a cache-line directly without fetching it from memory [30]. The processors designed by Azul [10] have a similar instruction, (`CLZ`), that directly zeros a cache line without fetching old memory. The x86 [18] ISA includes *non-temporal* cache bypass instructions for reads and writes that have no temporal locality.

Non-temporal store instruction such as `movnti` bypass the cache hierarchy. They send writes directly to memory via a write combining buffer without a cache access. When used effectively, they have two benefits: a) they do not displace other data in the cache, and b) they maximize memory bandwidth utilization because, unlike normal stores that can generate one fetch and one write-back transaction, non-temporal stores only generate one write transaction.

However, non-temporal stores are expensive when incorrectly applied to temporal data. If the target of the write is currently cached, the hardware must invalidate all cache resident copies of the line, which is costly. Furthermore, if the program reuses the data soon after the non-temporal write, an additional bus transaction is required to fetch the data. Non-temporal stores are weakly ordered, which requires that the programmer use explicit fences when the semantics require consistency of writes. Because fences are expensive, they make non-temporal stores unsuitable for writes that require fine-grained consistency.

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) [13] uses non-temporal stores when the region being zeroed is larger than the processor's last level cache. Otherwise it uses standard writes. The open64 compiler [1] provides a `-CG:movnti=N` flag. When it writes to 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 [23] VM. We extracted further details of the Azul [10] and IBM J9 [14] 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, J9 and Azul VMs use `dcbz` and `CLZ` instructions when targeting PPC and Azul hardware, respectively.

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 efficiently elides redundant writes with good temporal locality. Correctly implementing this optimization is difficult because it requires an analysis to guarantee that all object fields are initialized before either the program *or* the garbage collector observes them. The Oracle HotSpot VM implements such an optimization, but when we measured it, we found that it provides limited benefit, on average only 0.4% compared with hot-path zeroing across our benchmarks. Due to this weak result and the complexity involved in implementing the optimization, we do not consider it further.

Jikes RVM [4] 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 to utilize available memory bandwidth. 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.

Benchmark	Suite	Heap Size MB (6× min)	Total Allocation MB	Allocation Rate normalized to mean	CPU Utilization maximum is 8.0	Multi-threaded
compress	SPECjvm98	114	105	0.01	1.00	No
jess	SPECjvm98	114	265	1.03	0.99	No
db	SPECjvm98	114	74	0.09	1.01	No
javac	SPECjvm98	198	175	0.29	1.03	No
mpegaudio	SPECjvm98	78	0.21	0.00	1.00	No
mtrt	SPECjvm98	120	75	0.43	1.39	Yes
jack	SPECjvm98	102	254	0.68	1.00	No
antlr	DaCapo MR2	144	217	0.46	1.01	No
avrora	DaCapo Bach	300	54	0.03	2.88	Yes
bloat	DaCapo MR2	198	1096	0.59	1.02	No
eclipse	DaCapo MR2	480	2752	0.31	0.92	Yes
fop	DaCapo MR2	240	48	0.10	1.04	No
hsqldb	DaCapo MR2	762	118	0.22	0.94	Yes
jython	DaCapo Bach	240	1395	0.70	1.05	Yes
luindex	DaCapo Bach	132	34	0.09	0.99	Yes
lusearch	DaCapo Bach	204	8152	8.24	6.34	Yes
lusearch-fix	DaCapo Bach	204	1071	2.57	7.22	Yes
pmd	DaCapo Bach	294	385	0.79	2.31	Yes
sunflow	DaCapo Bach	324	1832	1.47	7.33	Yes
xalan	DaCapo Bach	324	1104	1.92	7.06	Yes
pjbb2005	SPECjbb2005	1200	1930	0.92	4.77	Yes

Table 1. Benchmark characteristics

```
1 static int[] fresh;
2 public static void initnone() {
3   for (int i=0; i < 1<<26; i++) {  // 64 million
4     fresh = new int[8];
5   }
6 }
```

(a) initnone

```
1 static int[] fresh;
2 public static void initfresh() {
3   for (int i=0; i < 1<<26; i++) {  // 64 million
4     fresh = new int[8];
5
6     // initialize the fresh array
7     for (int j=0; j < 8; j++)
8       fresh[j] = j;
9   }
10 }
```

(b) initfresh

```
1 static int[] fresh;
2 static int[] stale;
3 public static void initstale() {
4   stale = new int[8];
5   for (int i=0; i < 1<<26; i++) {  // 64 million
6     fresh = new int[8];
7
8     // (re)initialize the stale array
9     for (int j=0; j < 8; j++)
10      stale[j] = j;
11  }
12 }
```

(c) initstale

Figure 1. Zero initialization locality microbenchmarks

3. Methodology

Empirical evaluation is used throughout the remainder of this paper, first to provide motivating analyses, then as part of a detailed analysis of existing design points, and finally to analyze our three new designs. So we now present the software, hardware, and measurement methodologies that we use.

3.1 Software platform

Benchmarks. Table 1 shows the benchmarks we use, the heap size we use, the total allocation and allocation rates of the benchmarks, their CPU utilization and whether the benchmarks are multi-threaded. The zeroing workload and CPU utilization of these benchmarks is discussed in Section 4. We draw the benchmarks from DaCapo [6] suite, the SPECjvm98 [32] suite, and pjbb2005 [3]. (A fixed workload version of SPECjbb2005 [31] with 8 warehouses that executes 10,000 transactions per warehouse.) We use benchmarks from both 2006-10-MR2 and 9.12 Bach releases of DaCapo to enlarge our suite and because a few 9.12 benchmarks do not execute on Jikes RVM.

We omit two outliers — mpegaudio and lusearch— from our figures and averages, but include them grayed-out in tables, for completeness. The mpegaudio benchmark is a very small benchmark that performs almost zero allocation while lusearch allocates at three times the rate of any other. The lusearch benchmark derives from the 2.4.1 stable release of Apache Lucene. Investigating the source of its high allocation rate, we found a performance bug in the method `QueryParser.getFieldQuery()`, which revision r803664 of Lucene fixes [29]. The heavily executed `getFieldQuery()` method unconditionally allocated a large data structure. In the fixed version the code only allocates the large data structure if it is unable to reuse an existing one. This fix cuts total allocation by a factor of eight, speeds the benchmark up considerably and cuts the allocation rate by over a factor of three. We patched the DaCapo lusearch benchmark with just this fix and we call the fixed benchmark lusearch-fix. The presence of this anomaly for over a year in public releases of a widely used package suggests that the behavior of lusearch is of interest and we occasionally call out lusearch as an example of a highly allocating workload. Our zeroing approaches improve the performance of lusearch by up to 30% on i7-2600, but we use lusearch-fix in our results.

Microbenchmarks. To better understand the behavior of zeroing, we use three simple microbenchmarks, illustrated in Figure 1. The initnone benchmark allocates 64 million arrays, each of eight integers. In our VM, this array consumes 44 bytes (8×4 bytes plus 12 bytes of header). The initfresh benchmark does the same, and then explicitly initializes each 44 byte array immediately after allocation. This benchmark has good temporal locality and we use it to explore the locality effects of the zeroing strategies. The third microbenchmark, initstale, allocates a single array, `stale` before executing the tight allocation loop and explicitly (re)initializes `stale` after each array is allocated. Explicit initialization of `stale` generates very little additional memory traffic, but it adds computation to the hot loop, which throttles the allocation rate.

Architecture	Core2 Quad	Phenom II	i7-2600
Model	Core2 Quad Q6600	Phenom II X6 1055T	Core i7-2600
Technology	65nm	45nm	32nm
Clock	2.4GHz	2.8GHz	3.4GHz
Cores × SMT	4 × 1	6 × 1	4 × 2
L1 D/I Caches	32KB × 4	64KB × 6	32KB × 4
L2 Cache	4MB × 2	512KB × 6	256KB × 4
L3 Cache	none	6MB	8MB
Memory	2GB DDR2-800	4GB DDR3-2000	4GB DDR3-1066

Table 2. Hardware platform characteristics

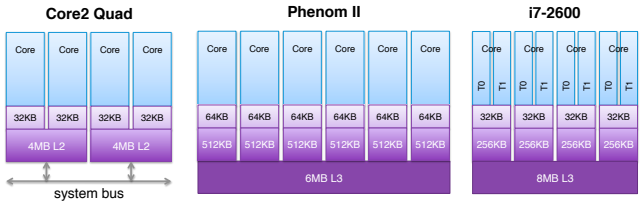


Figure 2. Processor memory system organization

Jikes RVM & MMTk. We use Jikes RVM release 3.1.1+hg r10392. We also confirm our results via a preliminary implementation in the OpenJDK 1.6.0 Oracle HotSpot Server JVM (see Sections 5.3 and 6.1). In Jikes RVM, we use the default *production* generational immix garbage collector [2]. Generational immix is a stop-the-world collector. It allocates objects into a *nursery* using *bump-pointer* allocation. When the nursery fills, it copies live objects into a mature mark-region space. The allocator design consists of a thread-local, unsynchronized *hot-path*, and a global, synchronized *slow path* [5]. The slow path replenishes each thread's local buffer with blocks from a global pool, and when necessary, triggers garbage collection. Each thread allocates into its local buffer without any synchronization. By default, Jikes RVM initializes space for fresh allocation by bulk zeroing 32KB blocks of memory as they are requested by the allocation slow path. No zero initialization is necessary for mature space allocation, since copying an object explicitly initializes it.

We use a 32MB fixed size nursery, which performs well for our benchmarks. We execute with a generous heap size: 6× the minimum required for each individual benchmark, as reported in the Heap column of Table 1. This heap arrangement produces a regular pattern of nursery collections and virtually eliminates full heap collections. We repeated our experiments with a more modest 3× heap and established that our findings are robust. In the limit, with a very small heap, garbage collection costs will dominate any mutator optimizations. The Total Allocation column of Table 1 shows the average volume of objects allocated in the nursery space using default bulk zeroing. The Allocation Rate column shows the allocation rate (bytes/execution time) normalized to the mean across all benchmarks on the i7-2600.

To reduce perturbation due to dynamic optimization and to maximize the performance of the underlying system that we improve, we use a *warmup replay* methodology. Before executing any experiments, we gathered compiler optimization profiles from the 10th iteration of each benchmark. When we perform an experiment, we execute one complete iteration of each benchmark without any compiler optimizations, which loads all the classes and resolves methods. We next apply the benchmark-specific optimization profile and perform no subsequent compilation. We then measure and report the subsequent iteration. This methodology greatly reduces non-determinism due to the adaptive optimizing compiler and improves underlying performance by about 5% compared the prior replay methodology [7]. We run each benchmark 20 times (20 invocations) and report the average. We also report 95% confidence intervals for the average using Student's t-distribution.

Operating System. We use Ubuntu 10.04.01 LTS server distribution running with a 64-bit (x86_64) 2.6.32-24 Linux kernel.

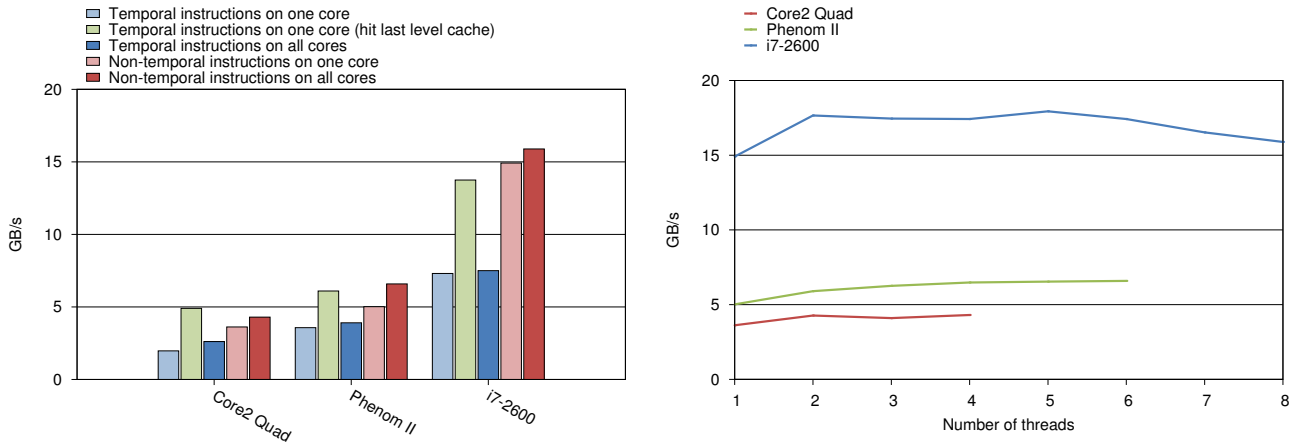
3.2 Hardware platforms

We use three contemporary IA32 architectures to explore the performance of zeroing with different technologies, memory systems, and memory bandwidth provisioning: (1) the Intel Core2 Quad Q6600 with a classic front-side bus that facilitates memory system analysis, (2) a six core AMD Phenom II X6 1055T, and (3) the recent Intel Sandy Bridge Core i7-2600 processor. Table 2 summarizes their key characteristics and Figure 2 illustrates their memory system and CPU organizations.

The 1066MHz front-side bus (FSB) on the Core2 Quad processor allows us to conduct a detailed analysis of memory traffic. Both the i7-2600 and the Phenom II processors have on-chip memory controllers and make it difficult to measure the individual contributions of different sources of memory traffic. 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 by counting full cache-line (burst) transactions using performance counters. Two types of memory references generate fetch transactions: program cache misses and prefetching misses generated by the hardware automatic prefetching unit. The Core2 Quad also provides two control bits in the `IA32_MISC_ENABLE` machine state register (MSR) that disable the hardware prefetcher and adjacent cache line prefetcher [19], enabling us to analyze the performance impact of hardware prefetching.

4. Motivating Analysis

To motivate our approach, we analyze benchmark characteristics, the effect of non-temporal instructions on bandwidth utilization, and the direct cost of zero initialization. Our analysis shows that some benchmarks have high memory bandwidth needs, that non-temporal instructions can use bandwidth more effectively, and that the cost of zero initialization can be high.



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

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

Figure 3. Performance potential of non-temporal instructions

4.1 CPU Utilization

CPU utilization gives an indication of the potential for contention of shared memory subsystems — if all hardware contexts 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, 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 contexts, and thus maximum *CPUUtilization* is 4.0, 6.0 and 8.0 respectively. The CPU Util. column in Table 1 shows CPU utilization for each of the benchmarks on the i7-2600 (where the maximum is 8.0). Four benchmarks — lusearch-fix,sunflow, xalan, and pjbb2005 — have relatively high CPU utilization, which suggests contention for the shared memory subsystem may be a problem for these.

4.2 Potential benefits of non-temporal instructions.

We perform a limit study to quantify the potential throughput benefits of non-temporal instructions, when used correctly on our evaluation machines. We compare non-temporal and regular write instructions in a tight zeroing loop with one thread and N threads, where N is the number of available hardware contexts. By default we use a 32MB buffer per thread, which is large enough to make store instructions miss the last level cache. To compare the memory bandwidth of non-temporal instructions with the memory bandwidth provided by the last level cache, we also evaluate performance when using a buffer sized to fit within the shared last level cache; 4MB, 6MB, and 8MB on Core2 Quad, Phenom II, and i7-2600 respectively.

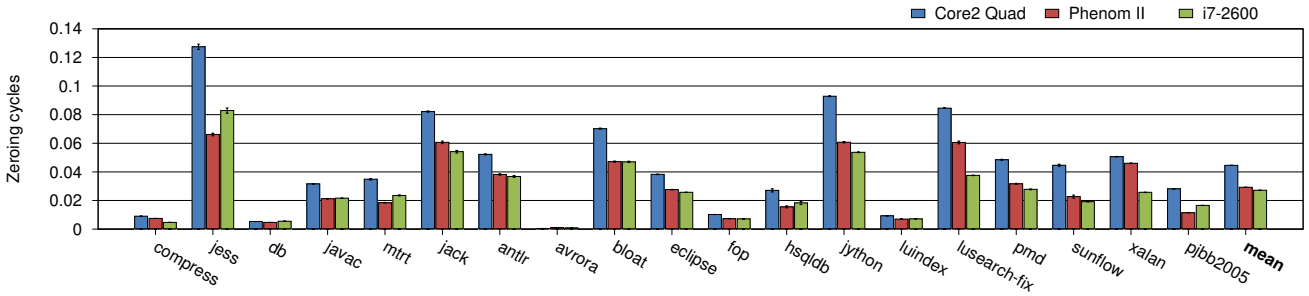
Figure 3(a) illustrates that on the Core2 Quad and Phenom II, the throughput increase of non-temporal instructions (red) over temporal instructions (blue) is 80% and 40% for a single core, 64% and 68% when using all cores. i7-2600 hardware performs non-temporal write instructions at *twice* the rate as temporal store instructions. On the Core2 Quad and Phenom II, when a regular (temporal) store instruction hits the last level cache (green bars in Figure 3(a)), the memory bandwidth provided by the last level cache is higher than that achieved by non-temporal instructions.

Figure 3(b) shows the memory bandwidth scalability of non-temporal instructions in the same tight loop, as a function of the number of hardware threads. None of the machines exhibit good memory bandwidth scalability in this setting. The memory bandwidth on the i7-2600 scales by 18% with 2 threads, up to just 20% with 5. The results for the Phenom II and Core2 Quad are similar to the i7-2600, although with substantially lower bandwidth.

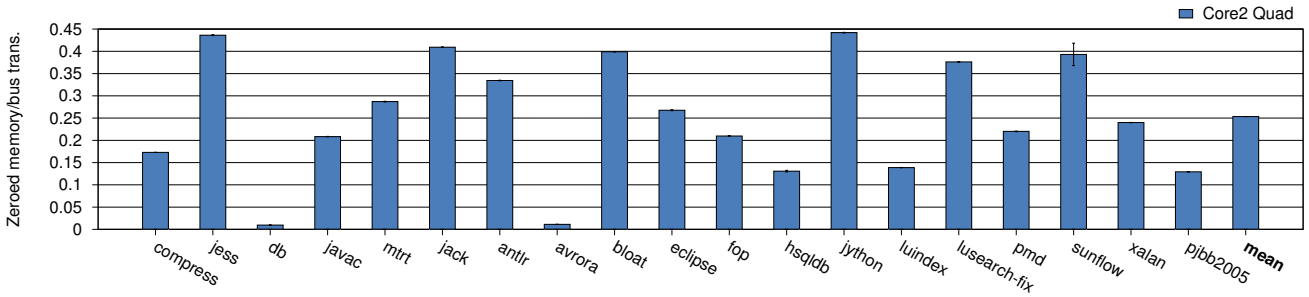
These results show that: a) non-temporal store instructions achieve much higher bandwidth than regular stores that miss the cache, and b) that on each of our processors, non-temporal stores can saturate the available bandwidth with less than half the available hardware contexts.

4.3 Direct Cost of Zero Initialization

We measure the direct cost of zero initialization — the number of cycles consumed by the application while performing zero initialization. We use bulk zeroing here because hot-path zero initialization enmeshes zeroing instructions with the allocation sequence at a fine granularity, which makes measuring its cost difficult, and the two approaches perform very similarly. Hot-path zeroing is on average just 0.3% faster than bulk zeroing on the i7-2600. For the same reason we use bulk zeroing as the default comparison point throughout the rest of the paper.



(a) Fraction of total application cycles due to zero initialization



(b) Zero initialization and memory traffic: BytesZeroed / BytesMemoryBusTransferred

Figure 4. The direct cost of zero initialization, in terms of cycles and memory traffic

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. This metric does not include *indirect* costs such as reduced application locality due to cache displacement. We sum this metric across all hardware contexts used by the process. The total includes CPU cycles due to both the mutator and garbage collection.

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

Figure 4(a) reports the fraction of total time spent performing zero initialization on all three architectures. The same data is presented numerically in the left half of Table 3. On the i7-2600, zeroing consumes an average of 2.7% of total time, and as much as 8.3% for jess. The Phenom II performs similarly. On the Core2 Quad, which represents more memory bandwidth constrained platforms, zero initialization cost increases to 4.5% on average, up to 12.7% for jess and a surprising 51% on the original lusearch (not included in the figure or averages, but shown in Table 3).

Figure 4(b) presents the bytes zero initialized as a fraction of all memory transferred on the bus, measured by performance counters of bus transactions. These results show that a large fraction of memory traffic is due to zeroing. The zero initialization fraction ranges from 10 to 45% for most benchmarks, with an average of 25%. Only avrora and db incur negligible traffic due to zeroing, and as we will see, 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 programs may be outliers.

5. Existing Zero Initialization Designs

Having presented motivating analysis for zero initialization design, we now conduct a detailed analysis of the two existing zero initialization designs before presenting our new designs in Section 6.

5.1 Hot-path Zeroing

Hot-path zeroing initializes memory for each object upon allocation, immediately prior to the first use. The allocation instruction sequence executes frequently, and is often called the fast or hot path. By default IBM J9, Oracle HotSpot, and Azul HotSpot use hot-path zeroing. Hot-path zeroing trades better data locality against a diminished optimization opportunity. It requires more instructions on the allocation path and degrades instruction locality.

Allocation is performance-critical in a modern JVM. To avoid function call overhead and enlarge the optimization scope, compilers often inline the allocation hot-path. Performing zeroing in the hot-path increases the size and complexity of generated code, which leads to more instruction cache pressure. We measured a slight increase in icache misses of around 6.3% compared to bulk zeroing, and around 5.3% increase in compilation time on the i7-2600. On the other hand, hot-path allocation should be friendly to modern memory systems since it minimizes the reuse distance between zeroing and first use of an object, and because

Benchmark	Zeroing Direct Cost (%) of Total Time C2Q	PII	i7	Hot-path vs. Bulk Improvement (%) C2Q	PII	i7
compress	0.91±0.00	0.75±0.01	**0.47**±0.00	-0.25±0.08	-0.37±0.07	**-0.47**±0.61
jess	12.74±0.18	6.61±0.09	**8.29**±0.19	8.39±0.26	0.01±0.41	**3.54**±0.96
db	0.53±0.00	0.47±0.00	**0.56**±0.00	0.05±0.18	0.15±0.29	**0.37**±0.84
javac	3.16±0.02	2.12±0.02	**2.16**±0.02	-0.01±0.27	-1.73±0.34	**-0.39**±0.79
mpegaudio	0.00±0.00	0.00±0.00	**0.00**±0.00	-0.06±0.14	-0.76±0.10	**-0.03**±0.91
mtrt	3.49±0.05	1.85±0.03	**2.36**±0.04	1.90±0.88	0.13±0.85	**-0.25**±1.42
jack	8.22±0.04	6.06±0.09	**5.41**±0.09	2.53±0.18	0.50±0.22	**-1.07**±1.07
min	0.53	0.47	**0.47**	-0.25	-1.73	**-1.07**
max	12.74	6.61	**8.29**	8.39	0.50	**3.54**
mean	4.84	2.98	**3.21**	2.10	-0.22	**0.29**
geomean				2.15	-0.22	**0.30**
antlr	5.22±0.04	3.82±0.05	**3.68**±0.06	0.43±0.11	-0.18±0.13	**0.26**±2.01
avrora	0.04±0.00	0.11±0.00	**0.09**±0.00	-8.69±5.03	1.21±0.37	**-0.65**±0.79
bloat	7.02±0.04	4.72±0.04	**4.70**±0.05	2.49±0.39	0.93±0.45	**2.39**±0.90
eclipse	3.83±0.02	2.76±0.02	**2.57**±0.01	-0.09±1.14	-1.03±6.58	**1.87**±2.25
fop	1.02±0.01	0.73±0.01	**0.71**±0.01	-0.58±0.62	1.08±0.53	**0.47**±1.33
hsqldb	2.71±0.12	1.55±0.07	**1.83**±0.11	-4.78±1.15	-3.62±1.45	**-2.84**±3.84
jython	9.29±0.03	6.07±0.05	**5.37**±0.03	1.65±0.35	0.79±0.56	**0.03**±0.66
luindex	0.93±0.01	0.70±0.01	**0.72**±0.01	0.25±0.76	0.34±0.18	**-0.13**±0.81
lusearch	51.40±0.14	50.59±0.43	**31.67**±0.13	25.22±0.24	27.73±0.52	**20.50**±0.72
lusearch-fix	8.46±0.03	6.06±0.10	**3.76**±0.02	2.76±0.30	0.94±1.08	**0.97**±0.95
pmd	4.85±0.03	3.17±0.04	**2.78**±0.03	0.58±0.31	0.88±0.47	**0.40**±1.07
sunflow	4.47±0.07	2.28±0.11	**1.92**±0.02	1.18±4.17	1.32±5.54	**-0.12**±1.49
xalan	5.06±0.02	4.60±0.03	**2.57**±0.01	1.42±0.26	0.90±3.21	**1.79**±0.67
min	0.04	0.11	**0.09**	-8.69	-3.62	**-2.84**
max	9.29	6.07	**5.37**	2.76	1.32	**2.39**
mean	4.41	3.05	**2.56**	-0.28	0.30	**0.37**
geomean				-0.24	0.30	**0.38**
pjbb2005	2.82±0.01	1.15±0.02	**1.66**±0.01	-0.30±2.85	-0.54±1.28	**-1.16**±0.60
min	**0.04**	**0.11**	**0.09**	**-8.69**	**-3.62**	**-2.84**
max	**12.74**	**6.61**	**8.29**	**8.39**	**1.32**	**3.54**
mean	**4.46**	**2.93**	**2.72**	**0.47**	**0.09**	**0.26**
geomean				**0.52**	**0.10**	**0.27**

Table 3. The direct cost of bulk zeroing (left) and total execution time for hot-path relative to bulk (right)

the access pattern is regular, it should be amenable to hardware prefetching.

In Jikes RVM we implement a hot-path instruction sequence that zeroes objects sixteen bytes at time using an unrolled loop of four-byte `mov` instructions. This version performs significantly better than two eight-byte (`movq`) instructions or one sixteen-byte (`movdq`) instruction. We identified two reasons that the eight-byte instruction performs worse: 1) it requires the use of a register as the source, whereas the four-byte instruction uses an immediate; and 2) the allocation sequence only guarantees four-byte alignment, so eight-byte or sixteen-byte stores may be unaligned, which on the IA32 is not incorrect, but incurs a performance penalty [17]. When objects are smaller than sixteen bytes, our approach redundantly zeroes 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.

5.2 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. Seeking to improve the efficiency of

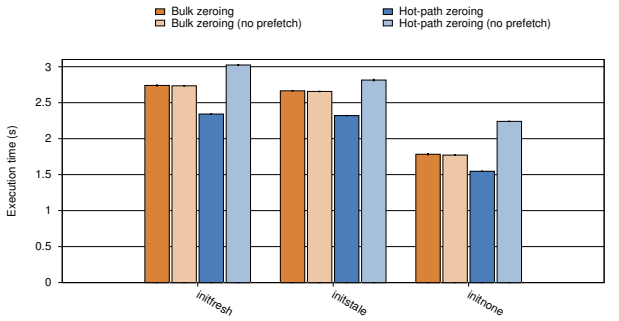


Figure 5. Execution time of microbenchmarks on Core2 Quad with and without hardware prefetch enabled

zeroing, bulk zeroing 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 aligned double word, quad word, or even a cache line at a time. Using our microbenchmark below, we dispel the expectation that the high spatial locality and regular access pattern of bulk zeroing would lend itself to hardware prefetching. The principal disadvantage of this design is that as the block size grows, the average reuse distance between allocation and data use also grows.

The default implementation in Jikes RVM performs zero initialization of 32KB blocks using glibc's `memset()` function. As the size of the block is much smaller than the size of the last level cache on the evaluation machines, the `memset()` function uses normal store instructions to perform zeroing.

5.3 Comparison: Hot-Path and Bulk Zeroing

This section presents a detail analysis of the memory system behavior and total performance of the two existing zero initialization techniques executing both our microbenchmarks and real workloads.

Microbenchmark performance. Figure 5 shows the execution time of the microbenchmarks on Core2 Quad with and without hardware prefetching. With hardware prefetching, hot-path zeroing outperforms bulk zeroing in all scenarios — by 14%, 13% and 14% for initfresh, initstale, and initnone respectively. We noted above that one of the downsides of hot-path zeroing is the fine-grained enmeshing of zero instructions that significantly reduces optimization opportunities compared to bulk zeroing. In this regard, the microbenchmark presents a best-case scenario for hot-path zeroing. When all allocation and initialization occurs in one very tight loop, the compiler can optimize it and make it similar to the bulk zeroing loop. Both approaches must write the header, even for initnone and initstale. Since there is no code advantage and a memory disadvantage for bulk zeroing on the microbenchmarks, the hot-path zeroing's 15% performance advantage is unsurprising.

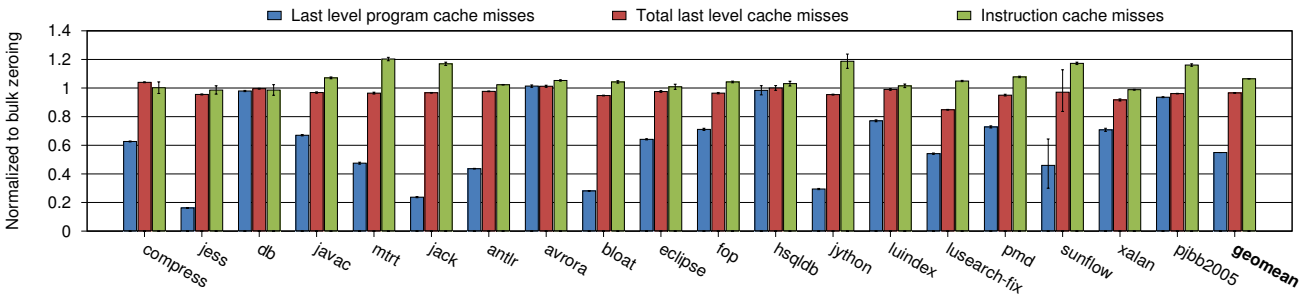


Figure 6. Last level cache misses on the Core2 Quad: hot-path/bulk zeroing

Without prefetching, bulk zeroing has a substantial advantage over hot-path zeroing: 11%, 6% and 26% for initfresh, initstale, and initnone respectively. Furthermore, the raw performance of bulk zeroing is *unaffected* whether the prefetcher is enabled or not, while the performance of hot-path is degraded significantly.

Since bulk zeroing writes large continuous chunks sequentially in a tight loop, it places high demands on the bus, throttling the automatic prefetcher. 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, which allows the hardware prefetcher to be more aggressive and gives the hardware more time to tolerate the latency. Thus, as Figure 5 shows, hot-path zeroing relies on prefetching to reduce the direct cost of zero initialization, and without it, performance suffers noticeably.

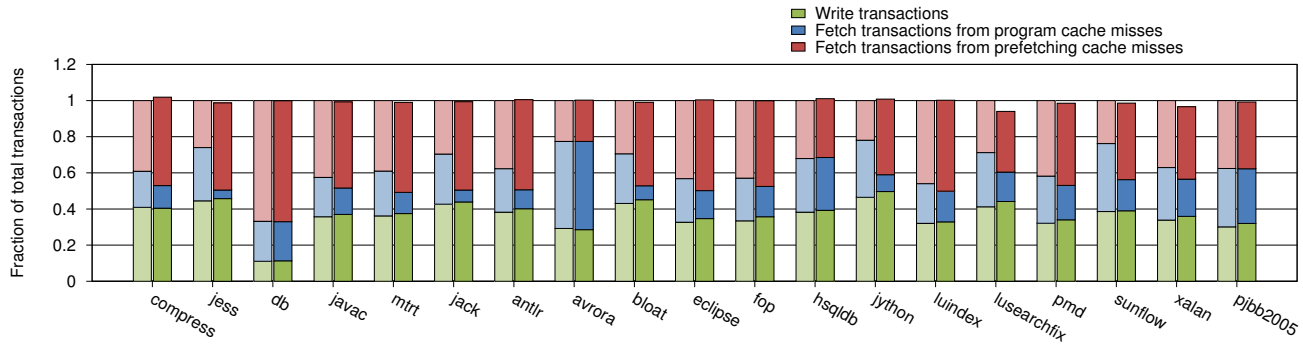
Overall performance. The right side of Table 3 shows the relative performance of hot path zeroing compared to bulk zeroing on the three architectures. These results show that hot-path zeroing offers a slim advantage over bulk zeroing (0.1% to 0.5%).The benchmark that benefits most from hot-path zeroing is the fast-allocating jess, with a 3.5% advantage over bulk-zeroing on the i7-2600. However, most benchmarks are neutral to this choice. These results contrast sharply with the microbenchmark results, highlighting the fact that the microbenchmarks is a best-case scenario for hot-path zero initialization.

We optimized HotSpot's bulk zeroing implementation and found similar results — hot-path zeroing provides only 0.7% advantage on the i7-2600, rising to 1.1% when combined with HotSpot's redundant initialization optimization, which is enabled by default. Summarizing, although hot-path and bulk zeroing perform remarkably similarly, our microbenchmark results show that they achieve this performance by making very different tradeoffs.

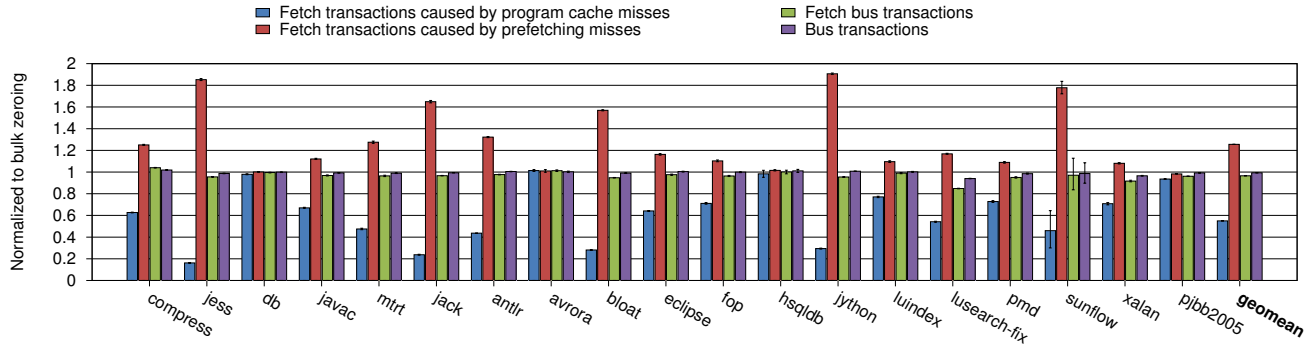
Instruction footprint. When the compiler inlines the allocation sequence to improve performance, it sprinkles additional hot-path zeroing instructions all over the program, generating a lot of instructions compared to a single out-of-line loop for bulk zeroing. The Instruction cache misses bar in Figure 6 shows that this increase degrades instruction locality. On the Core2 Quad, instruction cache misses increase by 6.4% for hot-path zeroing compared to bulk zeroing. For mtrt, jack, pjbb2005, sunflow and jython, the instruction cache misses increase by more than 15%. The decrease in program cache misses in jython of 70% suggests the potential for a performance win, but poor instruction cache locality counteracts this potential. The result is a net performance degradation due to hot-path zeroing. To understand the impact of hot-path zeroing on the compiler, we measured the compilation cost for both schemes. These measurements reveal a 5.3% increase in compilation time on the i7-2600 for hot-path zeroing. The rest of our experiments eliminate compilation time to focus on steady-state application performance.

Data locality. We divide cache misses into those generated directly by the program (*program cache misses*), those generated by automatic hardware prefetching (*prefetching cache misses*). We present *total last-level cache misses* which include both program and prefetching misses. The first bar in Figure 6 shows that hot-path zeroing nearly halves the number of program misses on the Core2 Quad compared to bulk zeroing. However, this effect is almost entirely offset by an increase in prefetch cache misses (not shown), leading to a modest 3.3% average reduction in total last-level cache misses. Because hot-path zeroing zeros more slowly, hardware prefetching has an opportunity to satisfy memory requests. The penalty of slower zeroing throughput for hot-path zeroing counteracts its improved temporal locality. Hot-path zeroing reduces last-level cache misses on the Core2 Quad by 45% on average and by 70% or more for bloat, jack, jess, and jython, which lead to 2.5%, 2.5%, 8.4% and 1.6% net speedups, respectively.

Memory performance. We evaluate the memory performance of the two existing zero initialization designs in Figure 7. Figure 7(a) depicts memory bus utilization on the Core2 Quad, with alternate bars showing bulk zeroing (left, light) and hot-path zeroing (right, dark). The total number of transactions is normalized to bulk zeroing, so the left bar in each pair is always exactly 1.0. When the right bar is lower, hot-path performs fewer bus transactions. Each bar is broken



(a) Bus transaction breakdowns on Core2 Quad: bulk (left, light) and hot-path (right, dark)



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

Figure 7. Memory bus utilization: hot-path/bulk zeroing

down into *write*, *program cache misses*, and *prefetch misses*, all normalized to the total bus transactions for bulk zeroing. The number of program cache misses (blue) is consistently lower for hot-path zeroing. On the other hand, the number of misses due to the hardware prefetcher (red) is consistently higher for hot-path zeroing, which is consistent with our microbenchmark results.

Figure 7(b) expresses the same data differently, normalizing *each metric* to the corresponding metric for bulk zeroing. Numbers higher than one signify that hot-path zeroing increases these memory transactions. The blue and red bars show fetch (read) transactions due to program and prefetch misses respectively. The green bar shows all fetch (read) transactions. The purple bar shows total bus transactions. Figure 7(b) shows that hot-path zeroing and bulk zeroing have similar bus transactions: on average a 0.7% difference (purple). The significant reduction in last level program cache misses by hot-path of 45% (blue) is offset by an increase in prefetch misses of 25% (red). Since on the whole prefetch misses dominate program misses, the 25% increase offsets the 45% decrease, resulting in a very small reduction in fetch transactions (green).

5.4 Zeroing performance in the application context

Having compared the two zero initialization designs in the previous section, we now look more closely at the *performance of the zeroing loop*, as used by bulk zeroing. 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 use bulk zeroing to measure the average number of cycles taken to zero a block of memory *as it executes as part of each benchmark*. We express the result in GB/sec.

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

Figure 8 shows the zeroing performance in the context of each benchmark by starting and stopping a timer at the beginning and end of zeroing. The Core2 Quad, Phenom II, and i7-2600 bulk zero at around 3, 5, and 8 GB/sec, respectively. While there exist a few outliers — compress and db on the high side, and lusearch-fix, sunflow, and xalan on the low side — Figure 8 shows that on the whole, bulk zeroing performance is similar in the context of each of these benchmarks.

Section 4 shows high CPU utilization often corresponds with high contention of the shared memory subsystem. Relating the CPU Util. column from Table 1 to the zeroing performance data in Figure 8 shows that the high CPU utilization benchmarks have the lowest zeroing performance. The zeroing performance of lusearch-fix is the worst because it has both a high CPU utilization with a very high allocation rate.

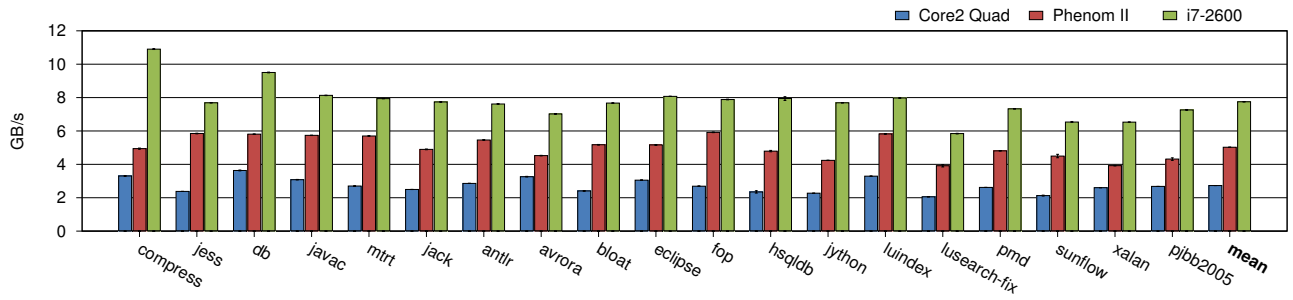


Figure 8. Zeroing performance in the application context: Bulk zeroing

sunflow and xalan have lower zeroing performance, especially on i7-2600, in part due to higher contention on shared memory subsystems.

5.5 Tradeoffs and trends

In summary, we see a stark tension between: a) hot-path allocation which reduces data cache pollution, but imposes a significant direct cost, and b) bulk zeroing which has low direct costs, but suffers significant cache pollution. While hot-path zeroing has better data cache hit rates, it degrades code quality and incurs higher direct costs. 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 and software generates more parallelism, programs are likely to contend more for shared memory bandwidth and caches. The next three sections show how we break this tradeoff and produce a system that performs better than these prior approaches.

6. New Zero Initialization Designs

We introduce three new design points: (1) non-temporal bulk zeroing, (2) concurrent non-temporal bulk zeroing, and (3) a design which adaptively switches between (1) and (2). To the best of our knowledge, these approaches are not discussed in the literature, nor deployed in existing virtual machines.[1] Each of these new designs perform better than deployed designs.

6.1 Non-temporal bulk zeroing

Figure 3(a) and Section 4 showed that *non-temporal* store instructions offer higher effective bandwidth and the potential to mitigate the direct cost of bulk-zeroing. They can eliminate cache displacement side effects, but they are weakly ordered and must be used carefully—if any extant copies of the written-to cache line exist, they will be evicted, which will significantly degrade performance.

[1] The G1 garbage collector of HotSpot [12] includes code for a concurrent zeroing thread (which is described as important for responsiveness). However this thread is disabled in HotSpot where G1 now exists only as the mature space in a generational collector [27], not as a full heap collector.

Our *non-temporal bulk-zero* implementation replaces `memset()` with a loop that uses the `movntdq` quad-word non-temporal store instruction. The original `memset()` uses regular store instructions for regions smaller than the last-level cache size. This modest change has a significant performance impact. Wide non-temporal instructions speed up the bulk zeroing rate by guaranteeing aligned memory accesses and amortizing the cost of zeroing the source register. This design avoids the alignment pitfalls that make wide instructions perform poorly in the hot-path and it avoids cache pollution that plagues bulk zeroing. This design relies on effective hardware prefetching to hide miss latencies that occurs at allocation time when the program first writes or reads each new object.

Microbenchmark performance. Figure 9 shows the execution time of the microbenchmarks on Core2 Quad with and without hardware prefetching. When hardware prefetching is enabled, non-temporal bulk zeroing outperforms bulk zeroing by 16% for initfresh and initstale and degrades performance for initnone. When prefetching is disabled, non-temporal bulk zeroing performs 0.7%. 7.5% and 41% worse than bulk zeroing for the three microbenchmarks, respectively. These results are very similar to the results for hot-path zeroing and show a heavy reliance on the hardware prefetcher. Because non-temporal stores bypass the cache, the cache miss for a new object will always occur in the allocation sequence, just as for hot-path zeroing. The significant divergence between non-temporal and hot-path zeroing

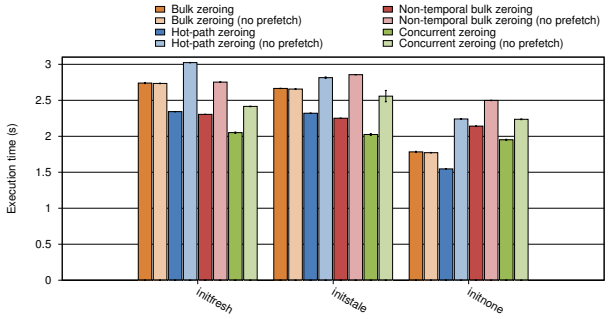


Figure 9. Execution time of microbenchmarks on Core2 Quad with and without hardware prefetch enabled

Benchmark	Zeroing Direct Cost (%) of Total Time C2Q	PII	i7	Hot-path vs. Bulk Improvement (%) C2Q	PII	i7	Non-temporal vs. Bulk Improvement (%) C2Q	PII	i7	Concurrent vs. Bulk Improvement (%) C2Q	PII	i7	Adaptive vs. Bulk Improvement (%) C2Q	PII	i7
compress	0.91±0.00	0.75±0.01	**0.47**±0.00	-0.25±0.08	-0.37±0.07	**-0.47**±0.61	0.44±0.08	0.05±0.08	**0.27**±0.60	0.79±0.08	0.44±0.07	**0.41**±0.57	0.81±0.07	0.44±0.08	**0.19**±0.59
jess	12.74±0.18	6.61±0.09	**8.29**±0.19	8.39±0.26	0.01±0.41	**3.54**±0.96	6.18±0.33	-0.04±0.29	**4.19**±1.00	13.92±0.31	7.23±0.24	**8.73**±0.97	13.95±0.36	7.06±0.29	**9.25**±1.02
db	0.53±0.00	0.47±0.00	**0.56**±0.00	0.05±0.18	0.15±0.29	**0.37**±0.84	0.01±0.17	-0.13±0.24	**-0.11**±0.82	0.65±0.20	0.45±0.27	**0.61**±0.89	0.66±0.20	0.41±0.24	**0.53**±0.76
javac	3.16±0.02	2.12±0.02	**2.16**±0.02	-0.01±0.27	-1.73±0.34	**-0.39**±0.79	0.38±0.24	-0.37±0.34	**1.07**±0.95	3.89±0.48	2.38±0.65	**3.21**±0.87	3.23±0.50	2.21±0.50	**3.23**±1.00
mpegaudio	0.00±0.00	0.00±0.00	**0.00**±0.00	-0.06±0.14	-0.76±0.10	**-0.03**±0.91	0.03±0.09	0.00±0.06	**-0.01**±0.76	0.03±0.08	-0.05±0.08	**-0.47**±0.86	0.04±0.10	-0.03±0.06	**-0.68**±0.91
mtrt	3.49±0.05	1.85±0.03	**2.36**±0.04	1.90±0.88	0.13±0.85	**-0.25**±1.42	1.80±0.96	-0.45±0.94	**0.84**±1.09	3.80±1.07	1.45±1.00	**1.57**±2.13	3.57±1.13	1.30±0.96	**3.33**±1.00
jack	8.22±0.04	6.06±0.09	**5.41**±0.09	2.53±0.18	0.50±0.22	**-1.07**±1.07	3.40±0.17	0.13±0.35	**3.38**±0.98	8.72±0.17	5.64±0.26	**6.92**±1.04	8.68±0.16	5.47±0.20	**6.79**±1.02
min	0.53	0.47	**0.47**	-0.25	-1.73	**-1.07**	0.01	-0.45	**-0.11**	0.65	0.44	**0.41**	0.66	0.41	**0.19**
max	12.74	6.61	**8.29**	8.39	0.50	**3.54**	6.18	0.13	**4.19**	13.92	7.23	**8.73**	13.95	7.06	**9.25**
mean	4.84	2.98	**3.21**	2.10	-0.22	**0.29**	2.03	-0.14	**1.61**	5.29	2.93	**3.57**	5.15	2.82	**3.89**
geomean				2.15	-0.22	**0.30**	2.06	-0.13	**1.62**	5.41	2.97	**3.63**	5.28	2.85	**3.94**
antlr	5.22±0.04	3.82±0.05	**3.68**±0.06	0.43±0.11	-0.18±0.13	**0.26**±2.01	1.29±0.12	-0.37±0.15	**2.24**±1.90	5.09±0.13	3.47±0.14	**4.51**±1.91	3.56±1.49	3.40±0.16	**5.31**±1.93
avrora	0.04±0.00	0.11±0.00	**0.09**±0.00	-8.69±5.03	1.21±0.37	**-0.65**±0.79	-0.14±4.72	-0.04±0.35	**0.47**±0.63	-3.43±4.55	-0.62±0.34	**-0.38**±0.81	-2.53±5.16	-0.44±0.38	**0.09**±0.56
bloat	7.02±0.04	4.72±0.04	**4.70**±0.05	2.49±0.39	0.93±0.45	**2.39**±0.90	2.57±0.47	0.06±0.42	**2.66**±0.83	6.83±0.43	4.75±0.43	**6.00**±0.94	6.67±0.46	4.46±0.41	**6.17**±0.81
eclipse	3.83±0.02	2.76±0.02	**2.57**±0.01	-0.09±1.14	-1.03±6.58	**1.87**±2.25	0.15±1.09	2.20±6.60	**3.11**±2.26	2.31±1.18	3.43±6.00	**3.15**±2.85	1.33±1.16	1.73±6.14	**1.74**±3.33
fop	1.02±0.01	0.73±0.01	**0.71**±0.01	-0.58±0.62	1.08±0.53	**0.47**±1.33	0.04±0.60	-0.17±0.53	**0.54**±1.41	0.71±0.56	0.41±0.59	**1.64**±1.39	0.82±0.53	0.35±0.57	**1.27**±1.41
hsqldb	2.71±0.12	1.55±0.07	**1.83**±0.11	-4.78±1.15	-3.62±1.45	**-2.84**±3.84	-0.65±1.29	-0.29±1.27	**4.53**±3.62	2.37±0.81	1.32±2.35	**3.52**±3.82	1.70±1.12	0.42±2.00	**5.86**±3.66
jython	9.29±0.03	6.07±0.05	**5.37**±0.03	1.65±0.35	0.79±0.56	**0.03**±0.66	2.05±0.39	0.40±0.39	**2.65**±0.67	6.48±0.31	4.76±0.51	**4.11**±2.17	6.38±0.28	4.71±0.42	**5.80**±0.82
luindex	0.93±0.01	0.70±0.01	**0.72**±0.01	0.25±0.76	0.34±0.18	**-0.13**±0.81	0.72±0.74	0.42±0.20	**-0.20**±0.82	1.84±0.87	0.99±0.20	**0.49**±1.01	2.00±0.80	0.84±0.18	**-0.12**±0.97
lusearch	51.40±0.14	50.59±0.43	**31.67**±0.13	25.22±0.24	27.73±0.52	**20.50**±0.72	28.35±0.13	25.85±0.66	**28.97**±0.82	20.34±6.59	18.26±2.83	**19.67**±3.95	28.13±0.16	26.65±0.61	**30.17**±0.35
lusearch-fix	8.46±0.03	6.06±0.10	**3.76**±0.02	2.76±0.30	0.94±1.08	**0.97**±0.95	3.80±0.20	0.42±1.09	**3.27**±1.14	-3.53±1.22	-1.78±1.77	**0.16**±1.29	2.24±0.35	0.06±1.39	**3.07**±1.09
pmd	4.85±0.03	3.17±0.04	**2.78**±0.03	0.58±0.31	0.88±0.47	**0.40**±1.07	0.41±0.34	-0.13±0.46	**1.76**±1.09	2.07±0.55	1.29±0.59	**3.21**±1.03	0.80±0.39	1.13±0.52	**2.81**±0.94
sunflow	4.47±0.07	2.28±0.11	**1.92**±0.02	1.18±4.17	1.32±5.54	**-0.12**±1.49	0.40±4.52	1.29±5.16	**1.79**±1.04	1.47±4.48	-2.22±6.08	**1.15**±1.25	2.49±3.93	2.72±5.44	**2.03**±1.17
xalan	5.06±0.02	4.60±0.03	**2.57**±0.01	1.42±0.26	0.90±3.21	**1.79**±0.67	3.10±0.30	0.69±2.57	**1.97**±2.43	1.75±0.31	-1.68±2.81	**0.78**±1.95	2.81±0.31	0.70±2.85	**2.64**±0.66
min	0.04	0.11	**0.09**	-8.69	-3.62	**-2.84**	-0.65	-0.37	**-0.20**	-3.53	-2.22	**-0.38**	-2.53	-0.44	**-0.12**
max	9.29	6.07	**5.37**	2.76	1.32	**2.39**	3.80	2.20	**4.53**	6.83	4.76	**6.00**	6.67	4.71	**6.17**
mean	4.41	3.05	**2.56**	-0.28	0.30	**0.37**	1.15	0.37	**2.07**	2.00	1.18	**2.36**	2.36	1.67	**3.06**
geomean				-0.24	0.30	**0.38**	1.16	0.38	**2.07**	2.05	1.21	**2.38**	2.38	1.69	**3.08**
pjbb2005	2.82±0.01	1.15±0.02	**1.66**±0.01	-0.30±2.85	-0.54±1.28	**-1.16**±0.60	-2.17±3.20	-1.16±0.88	**1.21**±0.52	0.92±2.57	0.89±0.89	**2.65**±0.53	-0.79±3.09	-1.87±2.28	**0.46**±3.14
min	**0.04**	**0.11**	**0.09**	**-8.69**	**-3.62**	**-2.84**	**-0.65**	**-0.45**	**-0.20**	**-3.53**	**-2.22**	**-0.38**	**-2.53**	**-0.44**	**-0.12**
max	**12.74**	**6.61**	**8.29**	**8.39**	**1.32**	**3.54**	**6.18**	**2.20**	**4.53**	**13.92**	**7.23**	**8.73**	**13.95**	**7.06**	**9.25**
mean	**4.46**	**2.93**	**2.72**	**0.47**	**0.09**	**0.26**	**1.25**	**0.13**	**1.88**	**2.98**	**1.72**	**2.76**	**3.07**	**1.85**	**3.18**
geomean				**0.52**	**0.10**	**0.27**	**1.27**	**0.13**	**1.89**	**3.06**	**1.75**	**2.79**	**3.14**	**1.87**	**3.22**

Table 4. The direct cost of bulk zeroing (left), and total execution time for hot-path (middle) and new zeroing designs (right) relative to bulk zeroing

is visible for initnone when the hardware prefetcher is enabled: hot-path outperforms bulk zeroing, but non-temporal bulk zeroing performs much worse than bulk. For initnone, hot-path zeros the array in the fast allocation path, then writes 12 bytes to the array header. These extra instructions give the hardware prefetcher time to prefetch the next cache line, which helps hot-path zeroing to reduce the direct cost of zero initialization. However, in the allocation path with non-temporal bulk zeroing, there are too few instructions to tolerate the latency, even with hardware prefetching.

Zeroing performance in the application context. Figure 11 shows zeroing performance for non-temporal bulk zeroing, normalized to bulk zeroing in the context of each application's workload. We use the same metric and methodology as in Figure 8: how fast, on average, does the system zero a block of memory in the setting of each benchmark? On the Core2 Quad, Phenom II, and i7-2600, non-temporal bulk zeroing improves zeroing performance by 41%, 3%, and 74% on average. Figure 3(a) shows that non-temporal instructions double the effective bandwidth on the i7-2600, which is reflected in the 74% improvement. However, on Core2 Quad and Phenom II, non-temporal bulk zeroing is not as effective as on i7-2600. The reason is evident in Figure 3(a) which shows that on these machines, but not the i7-2600, a temporal instruction that hits the last level cache (green bar, Figure 3(a)) will outperform a non-temporal instruction. As a result, non-temporal bulk zeroing will not perform well on benchmarks with low cache miss rates on the Core2 Quad and Phenom II. For example, avrora, which has the lowest cache miss rate among our benchmarks performs poorly with non-temporal bulk zeroing on the Core2 Quad and Phenom II but performs well on the i7-2600, as shown in Figure 11.

Overall performance. Table 4 and Figure 14 show the effect of non-temporal bulk zeroing on total performance normalized to bulk zeroing. On the Core2 Quad, non-temporal bulk zeroing improves execution time by 1.3% on average and up to 6.2% for jess. Execution time improvements on i7-2600 are similar. The average improvement on the Phenom II is a somewhat smaller 0.1%, and consistent with the modest 3% improvement in zeroing performance.The benefits of non-temporal zeroing come from reducing the *direct* cost and the *indirect* cost by avoiding cache pollution. Figure 10 shows that total last level cache misses are reduced by 10% on average, and by as much as 50% on lusearch-fix on the Core2 Quad.

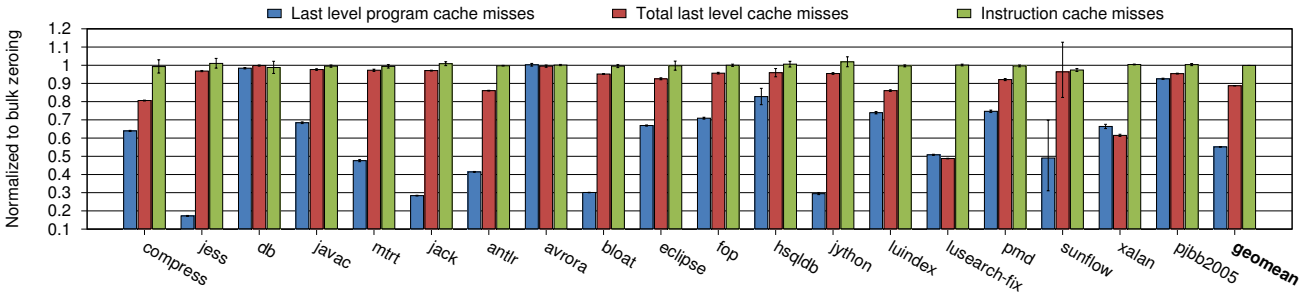


Figure 10. Cache misses for non-temporal bulk zeroing relative to bulk zeroing on the Core2 Quad.

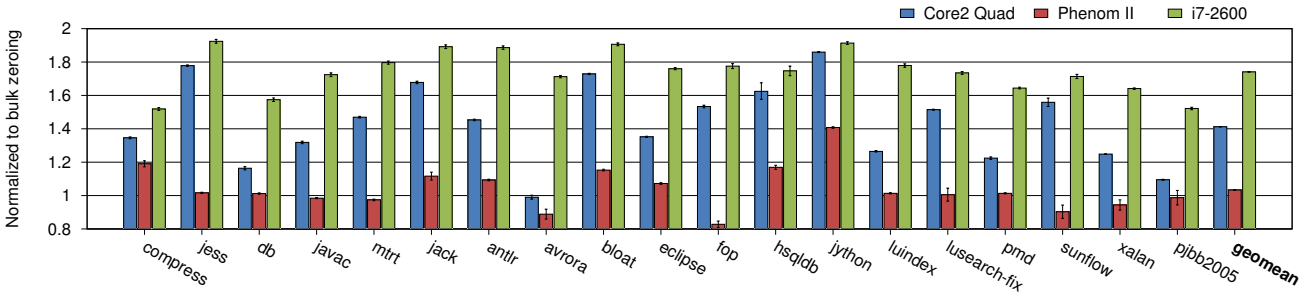


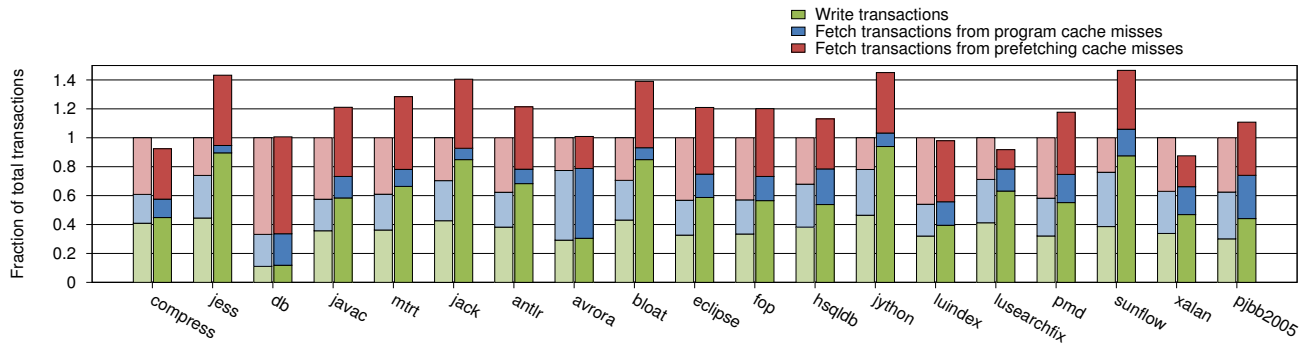
Figure 11. Zeroing performance in application context: Non-temporal bulk zeroing vs. bulk zeroing

Memory performance. Figure 12(a) plots the number of bus transactions, broken down in the same way as in Figure 7(a). The left (light) column in each pair shows the composition of bus transactions for bulk zeroing, and the right (dark) column shows the same for non-temporal bulk zeroing, normalized to bulk zeroing. Non-temporal writes cause additional write traffic and increase total memory traffic in about half of the benchmarks. In Figure 12(a), write transactions increase for non-temporal zeroing, but by negligible amounts for db and avrora. When the program references a line in memory, it must instantiate it in the cache. The program typically writes to the newly allocated object, which dirties the cache line, and thus requires a write. Because non-temporal zeroing explicitly invalidates all resident cache lines that it writes to, if the program would have reused the line given a standard write, non-temporal zeroing will result in a total traffic increase due to this read traffic. Although non-temporal zeroing reduces cache displacement, it increases the total number of bus transactions in about half the cases and has the same or fewer bus transactions in the other half.

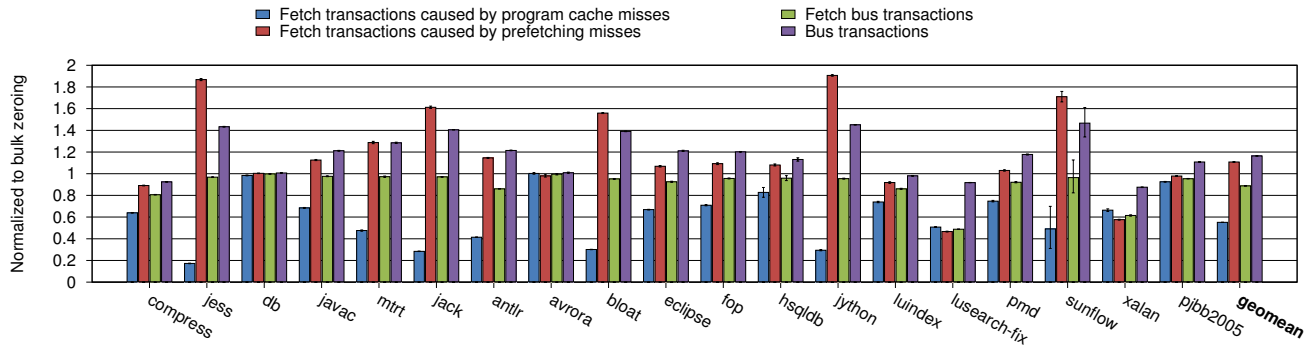
Figure 12(b) shows each type of bus transaction normalized to default bulk zeroing. On average, bus transactions are increased by 16%. For benchmarks with higher zeroing cost (jython, sunflow, jess, and jack), bus transactions increase by 40%. As with hot-path zeroing, the transactions for db and avrora are unaffected by non-temporal zeroing.

Figure 12(b) shows that non-temporal bulk zeroing reduces fetches by 11% on average and up to 51% for lusearch-fix. This reduction occurs because the non-temporal instructions only generate a write transaction, as opposed to the temporal stores, which generate a fetch and a write. 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. Aggressive memory prefetching can reduce the latency of the first touch. As shown in Figure 12(b), the fetches due to prefetching misses increase by 11% on average, and by as much 70% or more for jess, jython and sunflow.

HotSpot results. We also experimented with the OpenJDK 1.6.0 Oracle HotSpot Server JVM. Its default is hot-path zeroing optimized with software elision of redundant zeroing. It has the option of switching to bulk zeroing or unoptimized hot-path zeroing. Examination of their implementation of bulk zeroing revealed it was naive. We sped it up by replacing the zeroing loop with a simple call to `memset()`.We evaluated HotSpot's hot-path optimization, which avoids redundant initialization, and found that it improves over the regular hot-path by 0.4%. We then implemented a preliminary version of non-temporal bulk zeroing. We did not implement concurrent zeroing because software engineering decisions in HotSpot made it difficult. Our preliminary implementation of non-temporal bulk zeroing on average improves performance by a modest 0.7% over hot-path zeroing and we believe a more complete and mature implementation could do better. Our preliminary implementation of non-temporal bulk zeroing offers almost twice the advantage of HotSpot's existing hotpath optimization (0.7% v 0.4%), and is substantially simpler. This result suggests that new VMs should be implementing non-temporal bulk zeroing rather than hotpath



(a) Bus transaction breakdown on the Core2 Quad.



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

Figure 12. Memory bus utilization: Non-temporal bulk zeroing/bulk zeroing

zeroing, and that concurrent and adaptive zeroing will offer greater benefits. Non-temporal bulk zeroing may even worth implementing in existing VMs because it combines a small performance advantage with significantly simpler code in the allocation sequence, which appears all over the program.

6.2 Concurrent Zeroing

Bulk zeroing attacks the direct cost of zeroing. Non-temporal bulk zeroing further mitigates the direct cost with significantly better zeroing performance and also attacks the indirect cost of zeroing by reducing cache displacement. However, our benchmarks still spend 3% on average and up to 7.6% (38% if we include lusearch!) of total time performing zero initialization on the Core2 Quad.

Hardware parallelism creates the opportunity to perform zeroing concurrently, moving its overhead off of the application's critical path. The mechanics of concurrent zeroing require some synchronization between the zeroing thread and the application threads that consume the zeroed memory. The primary challenge is to ensure that synchronization does not dominate performance.

Our implementation uses a single zeroing thread. The JVM wakes up this thread up at the end of each nursery garbage collection, and it zeroes all of the blocks freed by the nursery collection with non-temporal instructions. The zeroing 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.

Microbenchmark performance. Figure 9 shows the execution time of our microbenchmarks on Core2 Quad with and without hardware prefetching for our new zeroing designs. When hardware prefetching is enabled, concurrent zeroing outperforms bulk zeroing by 25% for initfresh, 24% for initstale and degrades initnone by 9%. When prefetching is disabled, concurrent zeroing outperforms bulk zeroing by 12% for initfresh, 4%, for initstale, and degrades initnone by 26%. By taking work off the critical path, concurrent non-temporal zeroing outperforms all other designs for both initfresh and initstale, regardless of whether prefetching is enabled. Similar to non-temporal bulk zeroing, concurrent zeroing cannot improve upon bulk or hot-path designs for the initnone microbenchmark because the high allocation rate with no other computation does not give enough time for prefetching to be effective. These results show that concurrent zeroing is very effective at hiding the direct cost of zeroing, and prefetching plays an important role in hiding fetch latency after using non-temporal instructions.

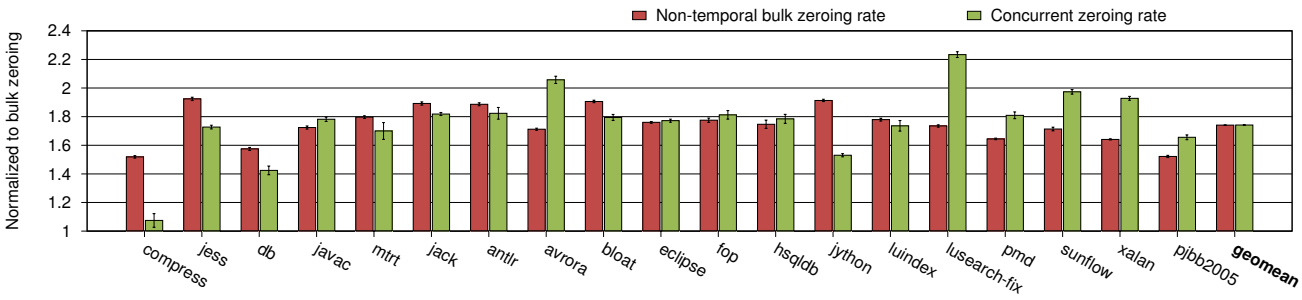


Figure 13. Zeroing performance in the application context: Non-temporal v concurrent on the i7-2600

Zeroing performance in the application context. Figure 13 compares the zeroing performance of concurrent and synchronous non-temporal bulk zeroing in the context of each application. The results are normalized to default bulk zeroing, so larger is better. For benchmarks with low CPU utilization (see Table 1), zeroing performance for a single concurrent thread is slightly worse than synchronous zeroing in the application thread. This is because the concurrent zeroing thread must contend for memory bandwidth with the application, unlike synchronous zeroing where the (single) active application thread is blocked waiting on the zeroing operation. While zeroing performance for concurrent zeroing is slightly worse, the availability of idle cores and generally lower allocation rates mean that the concurrent zeroing thread can hide zeroing costs, leading to better performance overall.

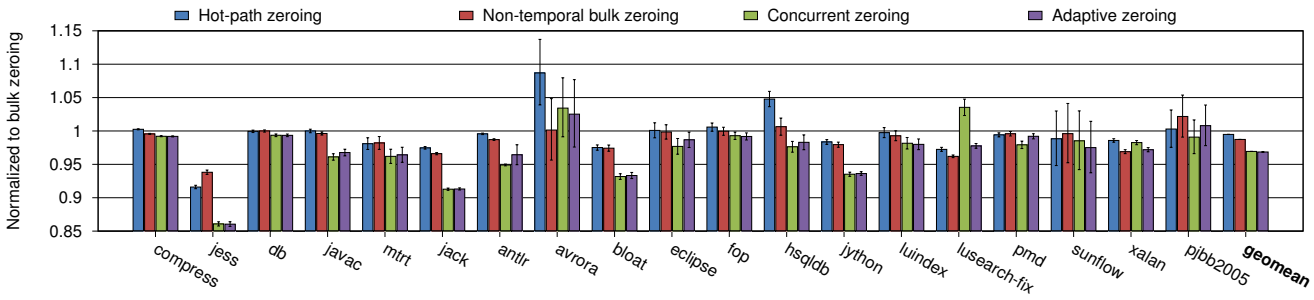
In high CPU utilization benchmarks, however, contention for memory bandwidth is different. With concurrent zeroing, the single zeroing thread will only contend with application threads for memory bandwidth in much the same way as for low CPU utilization benchmarks. When using synchronous bulk zeroing, however, in addition to contending with application threads, zeroing operations from multiple threads can also contend with each other, resulting in a dramatic reduction in overall zeroing performance, as predicted by the bandwidth scalability results in Figure 3(b). This additional contention means that zeroing performance for these benchmarks is higher when using concurrent zeroing.

Overall performance. Table 4 and Figure 14 show the performance of concurrent non-temporal bulk zeroing. It improves performance over bulk zeroing by 3.1%, 1.7%, and 2.8% on average and up to 13.9%, 7.2%, and 8.7% on the Core2 Quad, Phenom II and i7-2600, respectively. On the i7-2600, concurrent zeroing improves on 18 out of 19 benchmarks, with the only degradation being 0.4% on avrora. On the Core2 Quad 17 out of 19 are improved, while on the Phenom II, 15 out of 19 are improved. These results show that concurrent zeroing is a very robust and effective design choice.

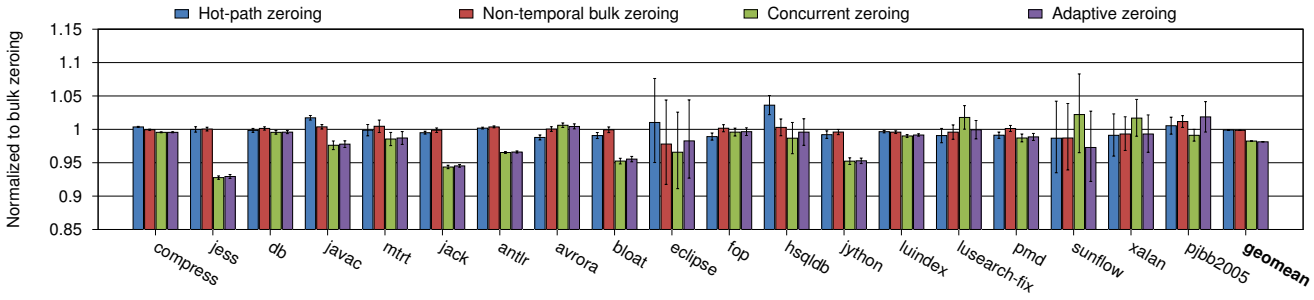
Alternative design options. A natural extension of concurrent zeroing is to use multiple, parallel zeroing threads to further utilize parallel hardware and to improve the zeroing throughput. This design is particularly appealing when the application is multithreaded and allocates at a sufficiently high rate that outpaces the zeroing thread (such as the DaCapo benchmark lusearch). However, due to the limited scalability of memory bandwidth, parallel zeroing is unlikely to provide a significant performance improvement, and may even degrade performance due to a reduction in per-thread zeroing performance. Parallel concurrent zeroing also requires coordination among 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, 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 while 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 the single-threaded concurrent zeroing that we recommend here.

6.3 Adaptive Zeroing

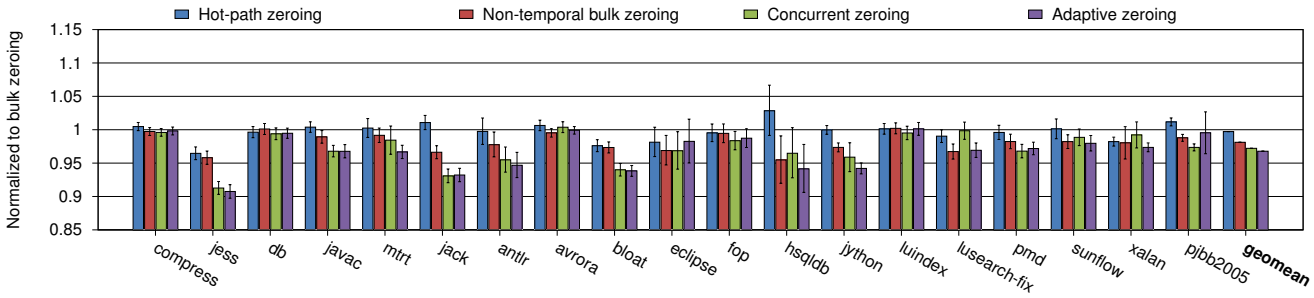
Although concurrent is an effective design choice, Figure 14 shows that the high CPU utilization benchmarks, such as lusearch-fix and avrora, have worse performance with a concurrent zeroing thread than with non-temporal bulk zeroing. The lack of an effective design for parallel zeroing threads means that these high CPU utilization multi-threaded workloads overwhelm concurrent zeroing by competing for parallel resources. 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 it is less, adap-



(a) Core2 Quad



(b) Phenom II



(c) i7-2600

Figure 14. Overall performance *relative to bulk zeroing*: Execution time for non-temporal bulk, concurrent non-temporal bulk, and adaptive zeroing

tive zeroing performs concurrent non-temporal bulk zeroing until the next garbage collection. Otherwise, adaptive zeroing performs non-temporal bulk zeroing. This design customizes the zeroing policy to the needs of each phase in a benchmark.

Overall performance. Table 4 and Figure 14 show that adaptive zeroing is the most effective technique for reducing the overhead of zero initialization across all benchmarks and all platforms. For most benchmarks, 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 best zeroing approach. Compared to bulk zeroing, adaptive zeroing improves performance by 3.1% on average and up to 14% on the Core2 Quad, 1.9% on average and up to 7.1% on the Phenom II, and 3.2% on average and up to 9.3% on the i7-2600.

7. Conclusions

This paper shows that zero initialization incurs a significant overhead on modern processors and provides the first detailed analysis of those overheads. We 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 language semantics of zero initialization and the hardware semantics of modern memory systems to reduce both direct and indirect costs. We also propose an adaptive policy that dynamically chooses between two new designs to take advantage of both cache-bypassing instructions and available hardware parallelism. The result is a substantial reduction in the overhead due to zero initialization, which leads to an average improvement of 3.2% over the prior bulk zeroing technique, across a wide range of benchmarks on the i7-2600.

Our results highlight the importance of counter-intuitively sacrificing temporal locality to achieve high memory throughput and minimal cache pollution. We 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 applications memory usage 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 chip multi-processors provide. The key to the best designs presented in this paper lies in exploiting a detailed understand of both hardware and software semantics. Our simple adaptive policy shows that customizing software parallelism to hardware parallelism is useful, and suggests the investigation of more general mechanisms for a wider class of VM services and application needs.

References

[1] AMD. *Using the x86 Open64 Compiler Suite*. Advanced Micro Devices, 2011. URL `http://developer.amd.com/assets/x86_open64_user_guide.pdf`.

[2] S. M. Blackburn and K. S. McKinley. Immix: A mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Languages Design and Implementation, Tucson, AZ*, PLDI '08, pages 22–32, 2008.

[3] S. M. Blackburn, M. Hirzel, R. Garner, and D. Stefanović. pjbb2005: The pseudojbb benchmark. URL `http://users.cecs.anu.edu.au/~steveb/research/research-infrastructure/pjbb2005`.

[4] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and realities: The performance impact of garbage collection. In *Proceedings of the 2004 ACM SIGMETRICS Conference on Measurement & Modeling Computer Systems, New York, NY*, SIGMETRICS-Performance '04, pages 25–36, 2004.

[5] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and water? High performance garbage collection in Java with MMTk. In *Proceedings of the International Conference on Software Engineering, Edinburgh, UK*, ICSE '04, pages 137–146, May 2004.

[6] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, Portland, OR*, OOPSLA '06, pages 169–190, Oct. 2006.

[7] S. M. Blackburn, K. S. McKinley, R. Garner, C. Hoffman, A. M. Khan, 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. Wake Up and Smell the Coffee: Evaluation Methodology for the 21st Century. *Communications of the ACM*, 51:83–89, Aug. 2008.

[8] S. Borkar and A. A. Chien. The future of microprocessors. *Communications of the ACM*, 54(5):67–77, May 2011.

[9] D. Burger, J. R. Goodman, and A. Kägi. Memory bandwidth limitations of future microprocessors. In *Proceedings of the 23rd Annual International Symposium on Computer architecture, Philadelphia, PA*, ISCA '96, pages 78–89, 1996.

[10] C. Click. Azul's experiences with hardware/software co-design. Keynote at ECOOP '09, (July 2009), 2009.

[11] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes. Cache hierarchy and memory subsystem of the AMD Opteron processor. *IEEE Micro*, 30(2):16 –29, March–April 2010. ISSN 0272-1732.

[12] D. Detlefs, C. Flood, S. Heller, and T. Printezis. Garbage-first garbage collection. In *Proceedings of the 4th International Symposium on Memory Management, Vancouver, BC*, ISMM '04, pages 37–48, 2004.

[13] GNU. *GNU C Library*. Free Software Foundation, 2011. URL `http://www.gnu.org/software/libc/manual/`.

[14] N. Grcevski, A. Kielstra, K. Stoodley, M. Stoodley, and V. Sundaresan. Java just-in-time compiler and virtual machine improvements for server and middleware applications. In *Proceedings of the 3rd Virtual Machine Research and Technology, San Jose, CA*, VM'04, pages 12–12, 2004.

[15] L. R. Hsu, S. K. Reinhardt, R. Iyer, and S. Makineni. 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, Seattle, WA*, PACT '06, pages 13–22, 2006.

[16] H. Inoue, H. Komatsu, and T. Nakatani. A study of memory management for web-based applications on multicore processors. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Languages Design and Implementation, Dublin, Ireland*, PLDI '09, pages 386–396, 2009.

[17] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Intel Corporation, Apr. 2011. Order Number 248966-024.

[18] Intel. *Intel 64 and IA-32 Architectures, Software Developer's Manual, Volume 2: Instruction Set Reference, A-Z*. Intel Corporation, May 2011. Order Number 325383-039US.

[19] Intel. *Intel 64 and IA-32 Architectures, Software Developer's Manual, Volume 3: Systems Programming Guide*. Intel Corporation, May 2011. Order Number 325384-039US.

[20] Intel. *MMX Technology Developer's Guide*. Intel Corporation, Mar. 1996. URL `ftp://download.intel.com/ids/mmx/MMX_Manual_Tech_Developers_Guide.pdf`.

[21] N. P. Jouppi. Cache write policies and performance. In *Proceedings of the 20th Annual International Symposium on Computer architecture, San Diego, CA*, ISCA '93, pages 191–201, 1993.

[22] R. Kalla, B. Sinharoy, W. Starke, and M. Floyd. Power7: IBM's next-generation server processor. *IEEE Micro*, 30(2): 7–15, March–April 2010. ISSN 0272-1732.

[23] P. B. Kessler. Java HotSpot virtual machine. Talk at FOSDEM-2007, Feb. 2007.

[24] C. Liu, A. Sivasubramaniam, and M. Kandemir. 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-10, pages 176–185, 2004.

[25] D. Molka, D. Hackenberg, R. Schone, and M. S. Muller. Memory performance and cache coherency effects on an Intel Nehalem multiprocessor system. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques, Raleigh, NC*, PACT '09, pages 261–270, 2009.

[26] G. Novark, E. D. Berger, and B. G. Zorn. Exterminator: automatically correcting memory errors with high probability. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Languages Design and Implementation, San Diego, CA*, PLDI '07, pages 1–11, 2007.

[27] Oracle Corporation. Java bug 6977804: G1:remove the zero-filling thread. URL `http://bugs.sun.com/view_bug.do?bug_id=6977804`.

[28] B. Rogers, A. Krishna, G. Bell, K. Vu, X. Jiang, and Y. Solihin. Scaling the bandwidth wall: Challenges in and avenues for cmp scaling. In *Proceedings of the 36th Annual International Symposium on Computer architecture, Austin, TX*, ISCA '09, pages 371–382, 2009.

[29] Y. Seeley. JIRA issue LUCENE-1800: QueryParser should use reusable token streams. URL `https://issues.apache.org/jira/browse/LUCENE-1800`.

[30] E. Sikha, R. Simpson, C. May, and H. Warren. *The PowerPC Architecture: A Specification for a New Family of RISC Processors*. Morgan Kaufmann Publishers, 1994.

[31] SPEC. *SPECjbb2005 (Java Server Benchmark), Release 1.07*. Standard Performance Evaluation Corporation, 2006. URL `http://www.spec.org/jbb2005`.

[32] SPEC. *SPECjvm98, Release 1.03*. Standard Performance Evaluation Corporation, Mar. 1999. URL `http://www.spec.org/jvm98`.

[33] C. Yu and P. Petrov. Off-chip memory bandwidth minimization through cache partitioning for multi-core platforms. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 132–137, 2010.

[34] Y. Zhao, J. Shi, K. Zheng, H. Wang, H. Lin, and L. Shao. Allocation wall: A limiting factor of Java applications on emerging multi-core platforms. In *Proceedings of the 21st ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, Orlando, FL*, OOPSLA '09, pages 361–376, 2009.