Cooperative Cache Scrubbing

Jennifer B. Sartor
Ghent University
Belgium

Wim Heirman
Intel ExaScience Lab*
Belgium

Stephen M. Blackburn
Australian National University
Australia

Lieven Eeckhout
Ghent University
Belgium

Kathryn S. McKinley
Microsoft Research
Washington, USA

ABSTRACT

Managing the limited resources of power and memory bandwidth while improving performance on multicore hardware is challenging. In particular, more cores demand more memory bandwidth, and multi-threaded applications increasingly stress memory systems, leading to more energy consumption. However, we demonstrate that not all memory traffic is necessary. For modern Java programs, 10 to 60% of DRAM writes are *useless*, because the data on these lines are *dead* - the program is guaranteed to never read them again. Furthermore, reading memory only to immediately zero initialize it wastes bandwidth. We propose a software/hardware cooperative solution: the memory manager communicates dead and zero lines with cache *scrubbing* instructions. We show how scrubbing instructions satisfy MESI cache coherence protocol invariants and demonstrate them in a Java Virtual Machine and multicore simulator. Scrubbing reduces average DRAM traffic by 59%, total DRAM energy by 14%, and dynamic DRAM energy by 57% on a range of configurations. Cooperative software/hardware cache scrubbing reduces memory bandwidth and improves energy efficiency, two critical problems in modern systems.

Categories and Subject Descriptors

C.0 [**Computer Systems Organization**]: General—*Hardware/software interfaces, instruction set design*; D.3.4 [**Programming Languages**]: Processors—*Memory management (garbage collection), optimization, runtimes*

1. INTRODUCTION

Historically, improvements in processor speeds have outpaced improvements in memory speeds and current trends exacerbate this memory wall. For example, as multicore processors add cores, they rarely include commensurate memory or bandwidth [43]. Today, the memory wall translates into a power wall. More cores need bigger memories to store all of their data, which requires more power. The limits on total power due to packaging and temperature constrain system design and consequently, the fraction of total system power consumed by main memory is expected to grow. For some large commercial servers, memory power already dominates total system power [31].

Software trends exacerbate memory bandwidth problems. Evolving applications have seemingly insatiable memory needs with increasingly larger working set sizes and irregular memory access patterns. Recent studies of managed and native programs on multicore processors show that memory bandwidth limits performance and scaling [21, 43, 55]. Two factors in managed languages pose both opportunities and challenges for memory bandwidth optimization: (1) rapid object allocation and (2) zero initialization.

(1) Many applications rapidly allocate short-lived objects, relying on garbage collection or region memory managers to clean up. This programming style creates an object stream with excellent temporal locality, which the allocator exploits to create spatial locality [5, 6, 7]. Unfortunately, short-lived streams displace useful long-lived objects, leave dead objects on dirty cache lines, and increase bandwidth pressure. The memory system must write dirty lines back to memory as they are evicted, but many writes are *useless* because they contain dead objects that the program is guaranteed to never read again. Our measurements of Java benchmarks show that 10 to 60% of write backs are useless!

(2) Managed languages and safe C variants require zero initialization of all fresh allocation. With the widely used fetch-on-write cache policy, writes of zeroes produce useless reads from memory that fetch cache lines, only to immediately over-write them with zeroes. Prior work shows memory traffic due to zeroing ranges between 10 and 45% [54].

While prodigious object allocation and zero initialization present challenges to the memory system, they also offer an opportunity. Because the memory manager identifies dead objects and zero initializes memory, software can communicate this semantic information to hardware to better manage caches, avoid useless traffic to memory, and save energy.

These observations motivate software/hardware cooperative cache *scrubbing*. We leverage standard high-performance generational garbage collectors, which allocate new objects in a region and periodically copy out all live objects, at which point the collector guarantees that any remaining objects are *dead* – the program will never read from the region before writing to it. However, any memory manager that identifies dead cache-line sized blocks, including widely used region allocators in C programs [4], can

*Wim Heirman was a post-doctoral researcher at Ghent University while this work was being done.

PACT'14, August 24–27, 2014, Edmonton, AB, Canada.

http://dx.doi.org/10.1145/2628071.2628083.

use scrubbing to eliminate writes to DRAM. The memory manager communicates dead regions to hardware. We explore three cache scrubbing instructions: *clinvalidate* invalidates the cache-line, *clundirty* unsets the dirty bit, and *clclean* unsets the dirty bit and moves the cache line to the set's least-recently-used (LRU) position. While *clinvalidate* is similar to PowerPC's discontinued *dcbi* instruction [20], and ARM's privileged cache invalidation instruction [3], *clundirty* and *clclean* are new contributions. We also use *clzero*, which is based on existing cache zeroing instructions [20, 32, 47] to eliminate read traffic. We extend the widely used MESI cache coherence protocol to handle these instructions and show how they maintain MESI invariants [14, 20, 22, 24, 37, 41]. After each collection, the memory manager issues one of the three scrubbing instructions for each dead cache line. During application execution, the memory manager zero initializes memory regions with *clzero*.

To evaluate cache scrubbing, we modify Jikes RVM [2, 7], a Java Virtual Machine (JVM), and the Sniper multicore simulator [11] and execute 11 single and multi-threaded Java applications. Because memory systems hide write latencies relatively well, simulated performance improvements are modest, but consistent: 4.6% on average across a range of configurations. However, scrubbing dramatically improves the efficiency of the memory system. Scrubbing and zeroing together reduce DRAM traffic by 59%, total DRAM energy by 14%, and dynamic DRAM energy by 57%, on average. Across the three scrubbing instructions, we find that *clclean*, which suppresses the write back and puts the cache line in the LRU position, performs best.

While we focus on managed languages, prior work by Isen et al. identifies and reduces useless write backs in explicitly managed sequential C programs [23]. They require a map of the live/free status of blocks of allocated memory, which hardware stores after software communicates the information. Their approach is not practical since (1) it requires a 100 KB hardware lookup table and a mark bit on every cache line, (2) it exacerbates limitations of the C memory model on undefined references, and (3) the separate map breaks cache coherence on multicore processors.

Our paper proposes a practical approach to save memory bandwidth and energy on multicore processors. While we leverage region allocation semantics, any memory manager that identifies dead cache lines can save traffic with cache scrubbing instructions. Our instructions require only small changes to the widely used MESI coherence protocol [20, 22] for multicore hardware. In summary, our contributions are:

- We identify an enormous opportunity to eliminate memory traffic in managed languages and region allocators.
- We design a practical cooperative software/hardware approach with the necessary modifications to the MESI cache coherence protocol to scrub cache lines.
- We quantify how cooperative scrubbing, used together with zeroing instructions, significantly reduces memory traffic and DRAM energy.
- We develop a novel multicore architectural simulation technology to simulate multi-threaded Java workloads on multicore hardware in reasonable amounts of time.

By exploiting the semantics of programming languages, software and hardware can cooperate to tackle some of the hardest bottlenecks in modern computer systems related to energy, power, and memory bandwidth.

2. BACKGROUND AND MOTIVATION

We first present background on the bandwidth bottleneck and memory management. We then characterize the opportunity due to useless write backs. In our Java benchmarks, from 10 to 60% of write backs are useless, i.e., the garbage collector has determined that the data is dead before the cache writes it back. We then describe why existing cache hint instructions are insufficient to solve this problem.

2.1 Bandwidth and Allocation Wall

Much prior research shows that bandwidth is increasingly a performance bottleneck as chip-multiprocessor machines scale up the number of cores, both in hardware and software [10, 21, 35, 40, 43, 55]. Rogers et al. show that the *bandwidth wall* between off-chip memory and on-chip cores is a performance bottleneck and can severely limit core scaling [43]. Molka et al. perform a detailed study on the Nehalem microarchitecture [40], showing that on the quad-core Intel X5570 processor, memory read bandwidth does not scale beyond two cores, while write bandwidth does not scale beyond one core.

On the software side, Zhao et al. show that allocation-intensive Java programs pose an *allocation wall* which quickly saturates memory bandwidth, limiting application scaling and performance [55]. Languages with region allocators and garbage collection encourage rapid allocation of short-lived objects, which sometimes interact poorly with cache memory designs. Scrubbing addresses this issue by communicating language semantics down to hardware.

2.2 Memory Management

The best performing memory managers for explicit and managed languages have converged on a similar regime. Explicit memory managers use region allocators when possible [4, 21] and high-performance garbage collectors use generational collectors with a copying nursery [5, 55]. Both allocate contiguously into large regions of memory, putting objects allocated together in time adjacent in memory, creating spatial locality based on the program's temporal locality [5]. Region allocation is only applicable when all the objects in the region die together, but this case is relatively frequent [4, 21], especially in transactional workloads, such as web services.

A standard copying generational garbage collector puts all fresh allocation into the *nursery*, i.e., a large region of contiguous memory. When the region is full, the collector copies out the survivors into the old space. Most managed programs follow the weak generational hypothesis, i.e., most objects die young [33, 48]. Therefore, frequent nursery collections copy small amounts of live objects, yet yield a large free region for subsequent fresh allocation.

Safe C dialects and managed languages, such as Java, PHP, JavaScript, and C#, require the runtime to zero initialize all fresh allocation. Prior work shows that zeroing incurs time overheads of 3 to 5% on average and produces 10 to 45% of memory traffic [54].

Sizing nurseries. Memory management is a time-space tradeoff. Smaller nurseries and heap sizes induce more frequent collections and higher collection costs, but consume less memory. In particular, small nursery sizes repeatedly incur fixed per-collection costs (e.g., scanning the thread stacks and coordinating application threads). Applications that allocate a lot, or have many threads, or have high nursery survival rates, need large nursery sizes to minimize garbage collection time. Prior work established that large nurseries offer the best performance on average, with the advantage tapering off at nursery sizes twice the size of the last-level

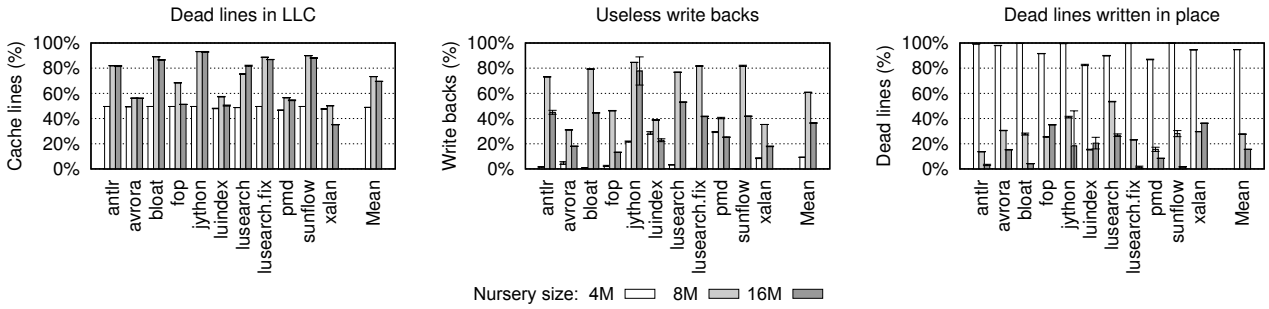

Figure 1: Dead data in last-level cache after nursery collections. (a) Percent of last-level cache lines that are dead. (b) Percent of write backs that are useless. (c) Percent of dead lines in last-level cache that are written in cache. ***Dirty cache lines are often dead and many dead lines are uselessly written to memory.***

cache [5, 55]. We confirmed this result on modern hardware and applications by running the DaCapo Java benchmarks [8], using the default generational Immix garbage collector in Jikes RVM [2, 7]. We found that a nursery size of 16 MB offered close to optimal performance on a Bloomfield processor with 8 MB of last-level cache, with 8 MB and 4 MB nurseries showing average overall slowdowns of 3 to 8%.

Since nursery sizes are adjustable and, in part, sized to coordinate with memory systems for responsiveness, in the remainder of this paper we explore a range of practical nursery sizes (4, 8, and 16 MB) that pivot around the last-level cache size (8 MB), which we keep constant.

2.3 Useless Write Backs

Allocation in both garbage collected and region allocation settings produce an *object stream*, a small irregular window of temporal and spatial reuse [6]. Object streams march linearly through the cache, displace other useful data, and have a short window of use. When the cache later evicts these lines, they are dirty and must be written to memory. Because these objects and their cache lines are mostly short-lived, the objects on the line are often dead; i.e., the program never reads them again.

A lot of this write traffic is useless, wasting bandwidth and energy. To explore this opportunity, we executed single and multithreaded DaCapo Java benchmarks [8], using the default generational Immix garbage collector in Jikes RVM [2, 7]. We execute them with the Sniper simulator [11], which models a multicore processor with four cores, inclusive caches, and a shared 8 MB L3 cache. (Section 4 has more methodological details.) After each nursery collection, the memory manager communicates the address range of the dead nursery to the simulator. The simulator marks resident cache lines in this range as dead. When the program subsequently writes a new object on a dead line, the simulator unmarks the line.

Figure 1 presents 'dead lines in last-level cache', the average fraction of the last-level cache that contains dead lines immediately after each nursery collection. 'Useless write backs' shows the fraction of all lines written to DRAM that exclusively contain dead objects and are thus useless. 'Dead lines written in place' shows the fraction of lines marked dead, but later written while in cache when the application allocates new objects on these lines.

Figure 1 shows that at the end of each collection, the last-level cache is dominated by dead lines (49 to 73% on average), and the fraction is a function of the nursery size.

A small nursery (4 MB) that fits in cache performs collections more often, degrading total time. However, because dead lines remain cache resident, this nursery size has very few useless write backs and around 95% of subsequent writes to these lines hit in the cache. Larger nurseries (8 and 16 MB) generate a substantial number of useless write backs (61% and 36% on average). Since the program takes longer to reuse a given line, the cache evicts more dead lines. Likewise, larger nurseries are less likely to write (28% and 16%) to the dead lines while they are still cache resident.

For 8 MB and 16 MB nursery sizes, over 70% of the last-level cache contains dead lines and the cache generates many useless write backs. For instance, over 75% of write backs are useless for bloat, jython, lusearch, lusearch.fix, and sunflow with an 8 MB nursery and can be eliminated. However, even in these cases, over 23% of the lines marked dead are written while still cache resident. These writes to dead lines in cache motivate using a scrubbing instruction that does not preemptively evict all dead cache lines, which will result in subsequent useless memory fetches, but instead motivates delaying the decision to evict based on program cache demand. In summary, there is a significant opportunity for saving traffic and improving memory efficiency.

2.4 Cache Hint Instructions

Scrubbing is, in part, inspired by prior instruction-set architectures (ISAs). For instance, the PowerPC ISA [20] contains a data cache block flush (*dcbf*) instruction, which writes a line back to memory. Prior to 2005, it contained a supervisor-level data cache block invalidate (*dcbi*), which invalidated a cache line and suppressed write backs. However, *dcbi* was never included in multicore processors. The ARM ISA also includes supervisor-level instructions to invalidate a cache line and clean a cache line, but the latter does not avoid the write back [3]. Section 3.1 describes how to implement user-level cache line invalidate instructions that avoid writing back data, including required changes to the cache coherence protocol for multicore processors.

The Intel Xeon Phi's ISA [22] introduces two instructions, *clevict0* and *clevict1*, which suggest cache lines to evict from the L1 and L2 cache, respectively. These instructions are local hints. The intent of these instructions is to remove lines from the cache that are no longer needed, but, like *dcbf*, they do not suppress write backs.

The current PowerPC ISA also includes a data cache block zero (*dcbz*) instruction that forgoes fetching from memory to zero a cache line directly, eliminating the default fetch on write [20, 47]. However, none of IBM's J9, Oracle HotSpot, or Jikes RVM uses it to perform zero initialization. Azul Systems, which built custom hundreds-of-core hardware for Java applications, used a cache line zero (CLZ) instruction to zero in place without fetching in order to save bandwidth. Azul states that CLZ avoided the fencing overhead of *dcbz* [13]. Our contribution is showing that zeroing works well and synergistically with scrubbing instructions.

Scrub Instructions	Description
clinvalidate(addr)	Set cache line state to invalid, no write back
clundirty(addr)	Unset cache line dirty bit
clclean(addr)	Unset dirty bit and move to LRU position
Zero Instruction	**Description**
clzeroX(addr)	Allocate and zero cache line in level X

Table 1: Each scrubbing and zeroing instruction takes an address and operates on one cache line.

3. COOPERATIVE SCRUBBING

This section describes cache scrubbing instructions that eliminate write backs to relieve off-chip memory bandwidth pressure, including the required changes to the cache coherence protocol and how we maintain its invariants. We then describe our software approach for inserting scrub instructions in any region allocator and, in particular, in a copying generational garbage collector.

3.1 ISA Modifications

Scrubbing instructions. We propose simple yet powerful instructions in which software conveys to hardware that the data in the specified cache line is dead, and that its contents can safely be discarded. Table 1 summarizes these instructions. The scrubbing variants differ in the expected distance until the program will next write the cache line to explore the space. The immediate or eventual eviction of the cache line will forgo the write back to main memory, saving off-chip *write* bandwidth.

The *clinvalidate* (cache line invalidate) instruction immediately invalidates the respective cache line from the cache. A subsequent access to the line will trigger a memory fetch, assuming the widely implemented fetch-on-write allocation policy. This instruction will be effective when the program does not write the line for a long time.

The *clundirty* (cache line unset dirty bit) instruction keeps the line in cache but clears its dirty bit. If the program writes to the line soon, it is still in cache and will not require a read and fetch from memory. If the line is evicted from the cache, *clundirty* eliminates the useless write to main memory.

The *clclean* instruction clears the dirty bit and moves the line into the LRU position, making the line the preferred candidate for replacement. This instruction is a flexible middle ground between *clinvalidate* and *clundirty*. If the program writes the line again soon, it will still be in cache. If instead the program needs capacity for other lines, the hardware will evict this line and forego a useless write back to main memory. Section 5.1 shows that this flexibility pays off: *clclean* is the most effective of the three at reducing main memory traffic.

Zeroing instructions. The semantics of many languages guarantee zero initialization of newly allocated memory, which can waste *read* memory bandwidth. Typical cache policies, fetch-on-write and write-allocate, cause the memory system to read data from main memory in large blocks, only to overwrite it with zeroes immediately, thus consuming unnecessary read bandwidth. As discussed in Section 2.4, the PowerPC ISA and others already include a data cache block zero instruction which forgoes fetching from memory and allocates a cache line filled with zeroes in the L1 cache. In this paper, we explore variants of existing zeroing instructions, and show that they are synergistic with scrubbing instructions. We evaluate *clzero1*, *clzero2* and *clzero3*, which allocate cache lines in the L1, L2 and L3 cache, respectively, to optimize for block size and expected reuse distance.

3.2 Cache Coherence

Each scrubbing or zeroing instruction operates on a single cache line and updates cache status bits and coherence states. With respect to memory consistency, both scrubbing and zeroing instructions potentially change memory contents. The processor core therefore treats them as writes with respect to ordering and serialization. Zeroing instructions target a specific level of cache, as defined by the instruction. We perform scrubbing instructions at the last level of cache, L3, and rely on existing mechanisms to propagate invalidations to the L1 and L2 caches sharing each L3. In our implementation, the software issues many scrubbing instructions in a row and thus the hardware can perform them in parallel.

Scrubbing instructions do not affect program correctness, and hardware may ignore them. The hardware is free to return undefined values if the software violates scrubbing instruction guarantees. In a memory-safe language, the garbage collector guarantees that the program will never read a dead line before writing it and communicates this fact to hardware. The correctness of memory-safe languages and region allocators depends on this guarantee.

Scrubbing instructions require minor changes to cache coherence protocols. We describe changes to a MESI protocol with inclusive last-level caches and then show how these changes do not violate key protocol invariants proved in prior work [14, 24, 37, 41]. Many multicore processors use MESI or close variants, such as MESIF (Intel [22]) or MERSI (IBMs PowerPC [20]).

Each letter in the MESI protocol name stands for a cache-line state: Modified, Exclusive, Shared, and Invalid. A finite state machine describes the transitions between states on bus and processor actions. Table 2 shows the transitions that occur in the MESI protocol at the last-level cache in response to each scrubbing or zeroing instruction. Each row shows the initial state and each column header shows the instruction. Given the current state (row) and the scrubbing instruction (column), the row-column intersection shows the resulting state and actions.

Scrubbing instructions. Scrubbing instructions operate at the last-level cache of the core that issues the scrubbing instruction. On systems with multiple last-level caches, such as multi-socket systems and cache-coherent multiprocessors, scrubbing instructions do not generate off-chip coherence traffic since all transitions happen locally. As shown in Table 2, starting from the M (modified) state, the *clinvalidate* instruction transitions cache lines in the last-level cache to I (invalid), as would a bus write request. Remember that a line in the M state in one last-level cache is not present in any other last-level cache. With inclusive caching, the protocol invalidates other cache levels that replicate this line, suppressing all write backs to memory. The *clclean* and *clundirty* instructions transition lines from M to E (exclusive), suppress the write, and invalidate other cache levels that contain this line. We choose to invalidate lines in the other cache levels (as opposed to undirtying them) because this uses existing hardware mechanisms. We did explore undirtying lines in the other cache levels and found no significant performance difference.

Starting from the E and S states, *clinvalidate* transitions the line to I (invalid) and invalidates copies of the line in the other levels of the cache hierarchy. The *clundirty* and *clclean* instructions do not change the state from E or S, but do trigger invalidations of the line in other cache levels. From all M, E, or S states, *clclean* modifies

State		*clinvalidate*	*clundirty*/*clclean*	*clzero*	BusInvalidate
M	modified	invalidate L1/L2 (no WB) → I	invalidate L1/L2 (no WB) → E (*clclean*: → LRU)	—	invalidate L1/L2 (no WB) → I
E	exclusive	invalidate L1/L2 → I	invalidate L1/L2 (*clclean*: → LRU)	→ M	invalidate L1/L2 → I
S	shared	invalidate L1/L2 → I	invalidate L1/L2 (*clclean*: → LRU)	BusInvalidate → M	invalidate L1/L2 → I
I	invalid	—	—	BusInvalidate → M	—

Table 2: Coherence state transitions for scrubbing instructions in the last-level cache.

the LRU position of the line, a cache-local property. *Clinvalidate*, *clundirty*, and *clclean* do nothing if the line is invalid.

Our simulator implements this protocol and scrubbing in the context of a multi-socket multicore system with snooping coherence and a single shared inclusive last-level cache per processor socket. However, scrubbing instructions will work with non-inclusive cache hierarchies by using their snoop filter, but fleshing out this design is left for future work.

Zeroing instructions. Zeroing instructions operate at the specific cache level encoded in the instruction. If the cache block is in a modified or exclusive state, the lines are instantiated with zeros, and an exclusive line transitions to M. Otherwise (from an S or I state), the access is propagated to the next-level caches, through to the last-level cache, using a special zeroing request. At the last-level cache, the protocol delivers a *BusInvalidate* message off-chip to the coherence network to obtain exclusive access, which is similar to the existing *BusRdX* request but elides the write back from other last-level caches if the line is in a modified state. This step prevents dirty but soon to be overwritten data from being sent across the off-chip coherence network. If the cache line started in the S or I states, it transitions to the M state upon a *clzero* instruction.

Maintaining protocol invariants. Proofs of the multiprocessor cache coherence protocols establish properties such as deadlock freedom, the fact that requests for cache lines are satisfied correctly, and consistency of cache line contents between caches and memory [14, 24, 37, 41]. We assume a proof of existing MESI invariants and show the effect of each of our instructions. We focus on the following key invariants for cache line states and contents.

1. Given a line in the M state in one cache, the line will be in the I state or not present in other caches.
2. Given a line in the E state in one cache, the line will be in I state or not present in other caches.
3. If a line is in the S state in one cache, the line will be in the S or I state or not present in other caches.
4. If a line is in the I state in one cache (or not present), the line may be in any other state in other caches.
5. When a cache requests a line, another cache will satisfy the request if it contains the line in the M state. Otherwise, memory will satisfy the request.

The simplest case is the *clzero* instruction. All five invariants are easily established, since *clzero* is the same as any other write. Invariants 2, 3, and 5 are trivially maintained. For invariants 1 and 4, when the protocol finds and invalidates copies of the line in other caches (taking invalidation and bus actions to re-establishing invariants 1 and 4), it suppresses fetches from both caches and memory. A subsequent request for the line will use this copy, since it is in the M state.

The *clinvalidate* instruction easily maintains these key five invariants. Transitioning any line in the M, E, and S states to I trivially maintains the first four invariants. Invariant 5 is maintained because in all cases, no cache contains a valid copy (M, E, or S state) of the line. For the purposes of validation, it is as if the system writes the original data to memory.

The *clundirty/clclean* instructions are the most subtle because they transition a line from M to E. Since invariant 1 holds before this transition, invariant 2 must hold afterward. Invariants 1, 3, and 4 are unaffected in this case. In the case where the line starts in the E, S, or I state, the line stays in the same state and, as such, maintains all five invariants.

Optimizations of invariant 5 where caches may satisfy read requests for data they have in the E or S state are affected because the cache and main memory may contain different data values due to scrubbing. However, the correctness of the programming language implementation *guarantees* that the application cannot generate any such subsequent read for the line after a scrubbing instruction. However for the purposes of completeness, if such a read were to occur, the response is undefined.

3.3 Security Implications of Scrubbing

Our premise for the correctness of the application is that the programming language implementation only uses scrubbing instructions on data that is in fact never subsequently read. If the application were somehow to violate this guarantee, it will read an undefined value. Such misuse does not constitute a security violation because the application is never made privy to another process' data. However, without correct support from the OS, scrubbing can open up an attack. When process *A* gives up a physical page, the OS may subsequently make it available to process *B*. Before making the page available to *B*, the OS will initialize the page, either zeroing it, when *B* requests a fresh page, or populating the page when *B* pages in from a swap. In either case, unless the OS ensures that the writes that it performs are propagated to main memory, *B* may use *clinvalidate*, discarding the data written by the OS, before reading from the page to reveal *A*'s data. The OS can avoid this problem by using a non-temporal store instruction, which invalidates all copies of the relevant cache line throughout the memory hierarchy, and writes the data straight into main memory — effectively destroying all remaining copies of the previous owner's data. To ensure that scrubbing instructions are only used in the presence of safe OS support, the instructions are disabled by default, and the OS only enables them by issuing a privileged instruction or setting a bit in a model-specific register. If disabled, the system safely ignores scrubbing instructions, requiring no change to existing operating systems.

3.4 Software Responsibilities

This section describes modifications to the memory manager to enable it to eliminate useless reads and writes to main memory using scrubbing instructions. A region allocator or a garbage collector may only reclaim memory if it determines that the region is dead and the program will never read from the region again. We implement scrubbing in a stop-the-world high-performance generational Immix collector [7]. At the end of each nursery collection, the collector has copied all reachable objects from the nursery region to the mature space. The collector thus guarantees that the entire nursery is dead. We modify the collector to iterate over the entire nursery's contiguous address range, calling a scrubbing instruction (*clinvalidate*, *clundirty*, or *clclean*) for each cache-line-sized block. We also modify the memory manager to issue our zeroing instructions to zero initialize regions during regular program execution, replacing the store instructions normally used. Our memory manager manages memory, and in particular the nursery, in 32 KB regions, zeroing a region on demand when the program first allocates into it. We modify the allocator to iterate over each cache-line-sized block in the 32 KB region, calling *clzero2* to initialize each line without fetching from memory. We evaluated zeroing at all three cache levels, but found L2 to be most effective because of the granularity of zeroing.

We also explored instructions that perform invalidations at larger granularities with similar results [44], but choose cache-line granularity instructions because they better correspond to existing memory instructions, easing their hardware implementation. Cache-line granularity is also more suitable for other memory managers, such as free-lists and Immix's line and block organization [7]. We leave exploration of larger granularities to future work.

4. METHODOLOGY

This section describes our software and simulator methodology, including the processor and memory architecture model, simulator extensions, JVM, and benchmarks.

4.1 Simulation Methodology

Sniper simulator. We modify Sniper [11], a parallel, high-speed, and cycle-level x86 simulator for multicore systems. We are unaware of any other publicly available cycle-level simulator that executes Java workloads on multicore hardware, which is required to evaluate scrubbing. Sniper uses a higher abstraction level than most cycle-level simulators and exploits parallelism on modern multicore hardware to achieve simulation speeds of up to 2 MIPS. Sniper can transition between a fast functional-only simulation model and a fully detailed simulation mode with timing and microarchitectural state changes. Sniper is validated against Intel Core2 and Nehalem class hardware with average absolute errors of around 20% for performance compared to real hardware, which is in line with other academic architectural simulators. In addition to performance, Sniper models and predicts power consumption using the McPAT framework [17].

Processor architecture. Sniper models a multicore processor with four superscalar out-of-order cores. Table 3 describes the basic architecture characteristics, including the private L1-I, L1-D and L2 caches, and shared last-level cache, which is similar to Intel's Nehalem machine. We model a 32-entry store queue which tracks architecturally committed stores while they are issued to the memory hierarchy; stores do not block commit until this store buffer fills up. Memory writes use a write-fetch write-allocate write-back caching policy. We model write buffers in front of DRAM and simulate a DDR3 DRAM main memory based on specifications from Micron [39].

Component	Parameters
Processor	1 socket, 4 cores
Core	2.66 GHz, 4-way issue, 128-entry ROB
Branch predictor	hybrid local/global predictor
Max. outstanding	48 loads, 32 stores, 10 L1-D misses
Cache line size	64 B, LRU replacement
L1-I	32 KB, 4 way, 4 cycle access time
L1-D	32 KB, 8 way, 4 cycle access time
L2 cache	256 KB per core, 8 way, 8 cycle
L3 cache	shared 8 MB, 16 way, 30 cycle
Coherence protocol	MESI
Memory controller	FR-FCFS scheduling, line-interleaved mapping, open-page policy
DRAM organization	32 GB, 2 channels, 4 ranks/channel, 8 banks/rank, 64 K rows/bank, 8 KB rows
DRAM device	Micron DDR3 [39]

Table 3: Architecture model.

Extensions for Java. We use Jikes RVM [2, 7] and modified Sniper to work with its Just-In-Time (JIT) compilers that generate code during execution. To faithfully simulate multicore hardware running Java workloads, we extended Sniper in a number of ways that are detailed in the following paragraphs.

System call emulation. Because Sniper is a user-level simulator, it may emulate or natively run system calls made by either the application or Jikes RVM. In an initial study, we measured system time with the `strace` and `time` utilities, and found that the benchmarks spend between 1 to 93% of time in system calls on an Intel Nehalem machine. We measured these numbers on the first iteration with replay compilation [9, 16] to reflect system calls during JVM start-up. We found that only two system calls contribute to almost all system time (all others are less than 0.5% of system time), those relating to synchronization: `futex` and `nanosleep`.

Consequently in all the experiments in this paper, we emulate these two system calls in Sniper and run other system calls natively. Sniper emulates futex-based synchronization calls by making threads wait and wake each other up at the correct simulated times. Emulation thus ensures that Sniper accounts for 99.5% of the total execution time. Since Java virtual machines use separate threads for GC, JIT, profiling, and application threads, even single-threaded applications are implicitly multi-threaded and the number of threads almost always exceeds the four cores we model. Because Sniper is a user-level simulator, we add a simple round-robin thread scheduler to Sniper. Sniper selects an available thread for a core once the running thread's time quantum expires (after 1 millisecond), or when the running thread blocks on a synchronization event.

JVM-simulator communication. Simulating cooperative hardware/software mechanisms requires communication between the JVM and hardware simulator. Sniper defines the *magic* instruction *xchg %bx, %bx* for this purpose, which is an x86 instruction that in normal execution has no effect. The simulator intercepts the magic instruction and reads the *%eax* and *%ebx* registers, which specify the type of scrubbing or zeroing instruction and the address.

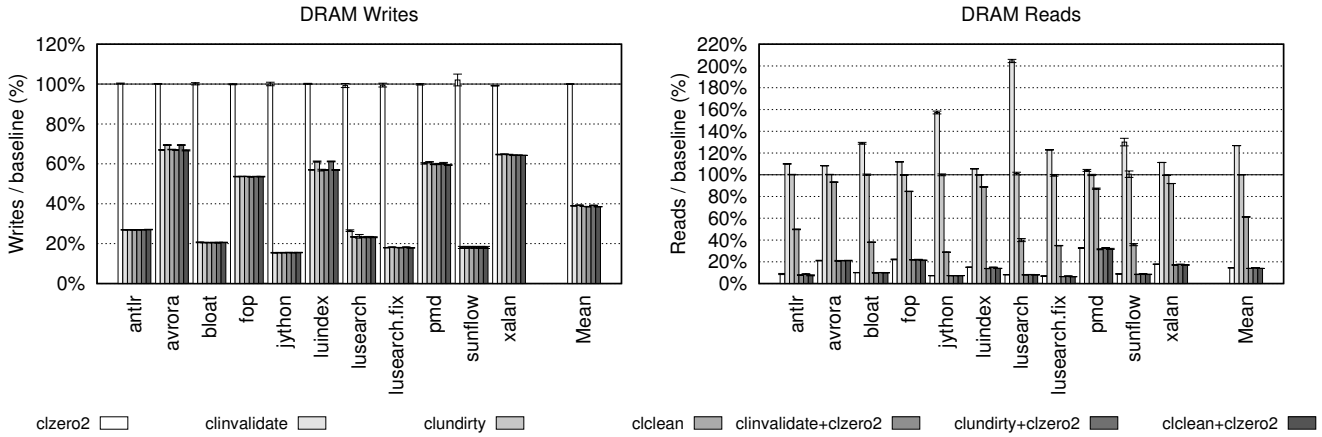

Figure 2: Relative number of DRAM reads and writes for an 8 MB nursery. ***All scrubbing instructions eliminate a large fraction of useless writes and* clclean *also eliminates reads. Zeroing instructions eliminate most DRAM reads. Together they effectively eliminate most useless memory traffic.***

We use Jikes RVM's low-level, low-overhead *syscall* mechanism to invoke Sniper's magic instructions. At the appropriate time, Jikes RVM invokes a scrubbing or zeroing instruction for each of the appropriate cache lines. Sniper models each instruction's effect in hardware, fully accounting for all overheads (including coherence and propagation events to other cache levels).

4.2 Software

Java virtual machine. We modify version 3.1.2 of Jikes RVM [2] to communicate with Sniper when it zeroes and scrubs memory. We specify four garbage collection threads, and for configurable multi-threaded workloads, we specify four application threads. All application and JVM service threads migrate between the four cores (on one socket) using Sniper's scheduler. This configuration works well, as measured by prior work [45].

We use the default *production* Jikes RVM configuration which uses a generational collector with an Immix mature space [7]. We use a practical heap size of 2× the minimum required for each benchmark. This configuration reflects moderate heap pressure. We use the default *boundedNursery*, which initializes the nursery to the bound size, but when the mature space is too limited to accommodate all nursery survivors, it reduces the nursery size accordingly, down to a minimum size of 1 MB. We eliminate the non-determinism of the adaptive compiler by performing replay compilation [9, 16]. The replay JIT compiler applies a fixed optimization plan when it first compiles each method [9, 19]. The plan includes a mix of optimization levels, which is calculated offline from recorded profiling information from numerous executions. We select the plan that gives the best performance, producing a highly optimized base configuration. We report the second invocation of the application, further limiting non-determinism and excluding JIT time to measure application steady-state behavior. We repeat this process three times and report averages and standard deviations in all graphs.

Java benchmarks. We execute 11 benchmarks from the DaCapo suite, taken from versions 2006-10-MR2 and 9.12-bach [8]. We use seven benchmarks from the 9.12-bach DaCapo release: those that work on our version of Jikes RVM and with our simulator. We use two versions of lusearch: the original lusearch and lusearch.fix which eliminates needless allocation (identified by [54]). We use three benchmarks from the 2006-10-MR2 DaCapo suite: antlr, bloat, and fop. Five benchmarks (antlr, bloat, fop, jython, luindex) are single-threaded and six (avrora, lusearch, lusearch.fix, pmd, sunflow, xalan) are multi-threaded.

5. EVALUATION

This section first evaluates the DRAM bandwidth, energy, and performance savings of scrubbing with the 8 MB nursery in detail. We then study the relationship of the nursery size to the last-level cache size, showing that scrubbing is effective at all ratios. The best performing scrubbing instruction alone is *clclean* because it saves both write and read traffic. Combining *clclean* with *clzero2* is the most effective at saving critical off-chip bandwidth and DRAM energy.

5.1 Reduction in DRAM Bandwidth Demand

Figure 2 plots the percentage of DRAM writes and reads saved with *clzero2*, *clinvalidate*, *clundirty*, and *clclean*, individually, and then each scrubbing instruction plus *clzero2*. The results for an 8 MB nursery are normalized to unoptimized runs for each benchmark.

On average, scrubbing alone saves about 60% of DRAM writes as compared with an unoptimized run, as predicted in Section 2.3. In particular, bloat, jython, lusearch, lusearch.fix, and sunflow substantially benefit from scrubbing, saving around 80% of off-chip writes. As expected, *clzero2* has a small effect on DRAM writes, but saves about 86% of DRAM reads.

All three scrubbing instructions (second, third, and fourth bars in Figure 2) are similarly highly effective at reducing DRAM writes. However, differences emerge when we analyze their impact on DRAM reads. The *clinvalidate* instruction increases the number of reads because it evicts all dead lines from cache. Using the common fetch and write-allocate cache policy, a subsequent write (to zero initialize the address upon fresh allocation) must fetch the data again from memory. The *clundirty* instruction does not change the number of reads from memory, since it has no effect on cache replacement.

The best performing scrubbing instruction is *clclean*. It eliminates both useless writes and reads, reducing DRAM reads by about 40% on average. The *clclean* instruction unsets the dirty bit *and*

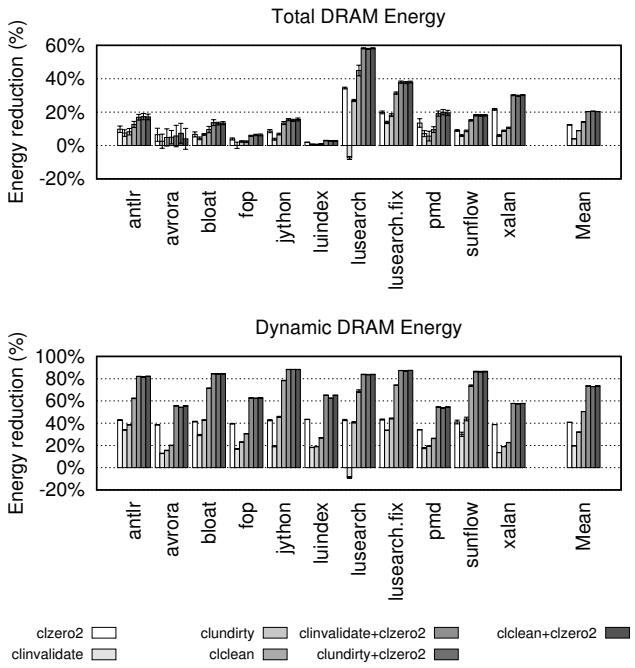

Figure 3: Reduction in DRAM total energy and DRAM dynamic energy for an 8 MB nursery. *Cooperative cache scrubbing improves total DRAM energy and dramatically improves dynamic DRAM energy.*

moves the line to the LRU position, such that if this set requires a subsequent eviction, the cache will evict the line. The dead and unreachable data is evicted lazily if necessary, and the cache retains more likely useful, live data. Furthermore, *clclean* moves nursery data to the LRU position in a *sequential order*, from the beginning to the end of the nursery address range, which puts addresses from the end of nursery to be evicted next and keeps early nursery addresses, which will be used again soon for allocation. In contrast, *clundirty* evicts nursery data based on last-touch, which is less likely to retain early nursery addresses. The *clclean* instruction thus hits a sweet spot, improving beyond the overly aggressive eviction policy with *clinvalidate* and no change to the eviction policy with *clundirty*.

We next evaluate the impact of each scrubbing instruction when combined with cache-line zeroing and we show that they perform synergistically together. The right three bars of each grouping in Figure 2 show that adding zeroing to scrubbing has little additional effect on writes to DRAM. However, DRAM reads are significantly reduced by adding *clzero2* to scrubbing. Combining zeroing with *clinvalidate* is especially advantageous as it results in a significant decrease in DRAM reads instead of an increase. *Clinvalidate* alone kicks dead data out of cache, and thus requires a fetch on subsequent access. Because the zeroing instruction, however, avoids this fetch by instantiating the data directly in cache, *clinvalidate+clzero2* saves around 86% of DRAM reads. Adding *clzero2* to either *clundirty* or *clclean* similarly saves the DRAM fetch for zeroing nursery data, equalizing the scrubbing instructions that keep nursery data in cache but evict it in different orders. All benchmarks save at least 65% of reads from DRAM. Using scrubbing and zeroing together, we save on average 86% of reads, and overall 75% of DRAM traffic (see Table 4).

5.2 DRAM Energy Reduction

The graphs in Figure 3 plot total and dynamic DRAM energy. The average reduction in dynamic DRAM energy is substantial at 73%. Because the majority of DRAM energy is currently static [39], the average total energy reduction is 20%. Energy reduction is larger than, but mirrors, the performance improvements in Figure 4. Because modern memory systems hide the latency of writes and some reads, reducing memory traffic does not always translate directly into performance. Optimizing with *clclean+clzero2* leads to an overall 20% reduction in total DRAM energy, larger than the average 12% for *clzero2* alone or 14% for *clclean* alone. For lusearch, *clclean* is particularly effective: it saves 45% of total DRAM energy by itself and together with *clzero2*, it saves 58%.

The bottom graph of Figure 3 shows that scrubbing dramatically saves dynamic DRAM energy. Alone, *clzero2* saves 41% of dynamic energy on average. *Clinvalidate* saves 20% on average, while *clundirty* and *clclean* save 32% and 50% of dynamic energy, respectively. Adding *clzero2* to scrubbing improves dynamic energy savings even more: 73% on average. The large savings of both DRAM reads and writes correspond to large reductions in dynamic and total DRAM energy, which will only become a more precious commodity in future systems.

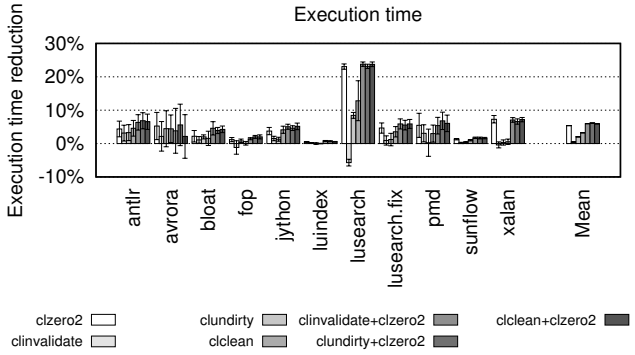

Figure 4: Reduction in execution time for an 8 MB nursery. *Cooperative cache scrubbing improves performance.*

5.3 Performance

Figure 4 shows the reduction in total execution time for each benchmark when using zeroing, scrubbing, and both optimizations together. Table 4 shows that scrubbing not only reduces memory traffic, it also reduces the average last-level cache (LLC) misses substantially: 86% for the 8 MB nursery. Reductions in memory traffic and decreases in last-level cache misses explain execution time improvements. Because the popular write-back policy coalesces writes and writes generally do not stall the instruction pipeline, performance improvements are modest.

By itself, *clzero2* improves performance by 5%, because of its large reduction in DRAM reads and in last-level cache misses.Even though non-blocking writes should achieve some of the benefits of *clzero2*, we found that the bursts of initializing writes for 32 KB regions overwhelmed the 32 entry store queues on occasion. Avoiding these DRAM-related stalls with *clzero2* removes stalls due to full store queues that occur with normal zeroing store instructions.

Scrubbing with *clinvalidate*, *clundirty*, and *clclean* each alone improves performance on average by 0.5%, 2%, and 3.3%, respectively. *Clclean* reduces execution time by saving reads and writes to memory, as well as reducing last-level cache misses by 39%, proving itself to be the most effective scrubbing instruction. In par-

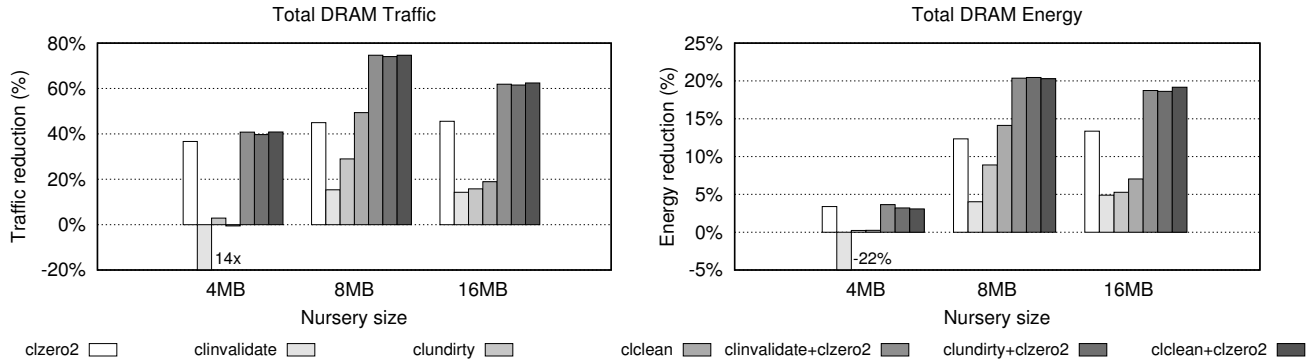

Figure 5: Reduction in DRAM traffic and energy for 4, 8, and 16 MB nurseries, normalized to unoptimized for each nursery size. *Scrubbing saves DRAM traffic and energy at all nursery sizes, and is particularly effective at nursery sizes equal to or larger than the last-level cache.*

Nursery size	4 MB	8 MB	16 MB
DRAM Reads	-57.54%	-86.10%	-86.35%
DRAM Writes	-12.54%	-61.50%	-35.52%
Total DRAM Traffic	-40.86%	-74.65%	-62.46%
LLC misses	-57.54%	-86.10%	-86.35%
Execution time	-0.84%	-5.92%	-6.93%
Dynamic DRAM Energy	-38.80%	-73.45%	-59.98%
Total DRAM Energy	-3.09%	-20.30%	-19.17%

Table 4: Improvements for all metrics normalized to unoptimized at the same nursery size due to *clclean+clzero2*. *Cooperative cache scrubbing is very effective at reducing DRAM accesses and improving energy efficiency.*

ticular, *clclean* improves the performance of lusearch by 13% because lusearch stresses DRAM bandwidth more than other benchmarks. We recommend *clclean* since it performs best by itself and all scrubbing instructions perform similarly with *clzero2*, saving 6% of execution time by saving substantial DRAM traffic.

5.4 Varying the Nursery Size

The effectiveness of scrubbing is influenced by the ratio of the last-level cache and the nursery size, particularly because we scrub at the last level of cache (which propagates to all levels in an inclusive cache hierarchy). This section presents results for three nursery sizes, and shows that scrubbing is effective on all of them. We expect these results to hold for other similar ratios of nursery and last-level cache sizes. Figure 5 compares both the overall reduction in DRAM traffic and total DRAM energy averaged over all benchmarks on 4 MB, 8 MB, and 16 MB nurseries normalized to no scrubbing for each nursery size. Dynamic DRAM energy savings follow the same trends as traffic, and execution time reductions follow the same trends as total DRAM energy (results omitted due to space limitations). Scrubbing is less effective on very small nursery sizes such as 4 MB, half the size of our 8 MB last-level cache.

The left side of Figure 5 shows that *clinvalidate* saves the least DRAM traffic. In fact, for a 4 MB nursery, which fits in the 8 MB last-level cache, kicking dead cache lines out of the last-level cache after collection increases DRAM traffic by 14× over the unoptimized version due to extra reads from memory. This phenomenon is especially a problem for bloat, jython, lusearch.fix and sunflow, as they incur 22×, 90×, 50×, and 49× increases, respectively, in DRAM reads. (We do not show individual program results due to space limitations.) This increase is large because the unoptimized baseline number of reads is very low. Figure 1 confirms these results by showing that for a 4 MB nursery, almost 100% of dead lines are written to while in cache. With a 16 MB nursery, *clinvalidate* slightly increases DRAM reads (by 5.6%), and lowers DRAM writes (by 36%), leading to an overall 14% reduction in traffic. Results show, however, that when combined with *clzero2*, all scrubbing instructions perform well.

All scrubbing configurations improve DRAM traffic over the unoptimized version (besides *clinvalidate* with the small nursery). With a 4 MB nursery, DRAM traffic is reduced slightly or kept the same for scrubbing, improves by 37% for zeroing, and 40% for both together. With the more realistic 16 MB nursery, scrubbing instructions alone reduce traffic by 14 to 19% from the unoptimized 16 MB amount. While zeroing the 16 MB nursery saves 45% of unoptimized traffic, together scrubbing and zeroing save 62%. Scrubbing and zeroing work together synergistically to save significantly more traffic than either alone, at all nursery sizes.

The right side of Figure 5 shows that all scrubbing configurations improve DRAM energy, except for the degradation for *clinvalidate* with a 4 MB nursery, which reflects the increase in traffic experienced by that configuration. The 8 MB and 16 MB nurseries obtain similar energy improvements with our optimizations, while the smaller nursery sees smaller reductions. With a 4 MB nursery, *clzero* with scrubbing sees an energy reduction of 3 to 3.7%. While *clclean* is most effective at reducing energy with an 8 MB nursery (14%), with the largest nursery it reduces energy by 7%, while speeding up execution time by 2% (not shown). Zeroing alone on a 16 MB nursery reduces energy by 13%, which corresponds to a 6% execution time savings (not shown). Together, scrubbing and zeroing save on average 19% of DRAM energy with a 16 MB nursery, compared to 20% with 8 MB (see Table 4). The larger nursery size with both optimizations achieves the smallest overall execution time across our benchmarks, and a 7% execution time savings over unoptimized. Scrubbing and zeroing together can save significant energy, especially at larger nursery sizes, which are preferred by many JVMs.

Results summary. Table 4 presents savings averaged over the benchmarks for the best configuration, *clclean+clzero2*, for all three nursery sizes, normalized to each nursery size's unoptimized run. The dynamic energy savings have similar trends as DRAM

traffic savings. Last-level cache miss savings are the same as DRAM read savings. The results for execution time are smaller than, but follow similar trends as, the total DRAM energy results in Figure 5. While using a small 4 MB nursery saves less traffic and less energy, we still see benefits due to scrubbing. We see the most improvements on larger nursery sizes. While the 16 MB nursery saves less total DRAM traffic than with 8 MB, 62% versus 75%, and less total DRAM energy, 19% versus 20%, the larger nursery saves slightly more last-level cache misses, and leads to a larger performance improvement of 6.9%. Averaging savings across the three nursery sizes, we get improvements of 59% for traffic and 14% and 57% for total and dynamic DRAM energy, respectively. Scrubbing is very effective and robust across nursery sizes, and will likely work well on nurseries larger than the last-level cache in future machines. Furthermore, in multiprogrammed workloads where programs must share the last-level cache, we believe that scrubbing will also improve cache management and decrease competition for bandwidth to main memory, while reducing energy.

6. RELATED WORK

This section surveys work that optimizes cache behavior by using software-hardware cooperation, prefetching, by designing a new architecture, by identifying dead cache lines, and by changing cache set replacement policies.

Cooperative cache management. The ESKIMO system is most closely related to our work [23]. They identify useless write backs in C programs. They propose changes to hardware including a 100 KB map in memory, cache line tags, and instructions for communication between software and hardware. Software communicates allocated and freed data blocks, which may be smaller or larger than the cache line granularity, to hardware to reduce energy and power requirements for DRAM. To optimize cache performance as well as memory performance, they reimplement previous work by Lewis et al. [32] to avoid fetching from memory for writes to addresses in this map. The large hardware map, fine-grained changes to the allocation and free software interfaces, and lack of a memory coherence approach for multicore processors make this work impractical. Whereas ESKIMO targets sequential C programs, our approach works for parallel programs on multicore hardware with small changes to cache coherence mechanisms. On the software side, we exploit contiguous region allocators that identify a large region of memory that is dead at one time, and present an opportunity for en-masse scrubbing of those cache lines. Reducing memory traffic is more important for managed languages, as we consider in this paper, than for explicitly-managed C programs because of the higher allocation rates typically observed in these applications [8]. In summary, our scrubbing instructions are simple, practical to implement, and work well with multicore cache coherence policies.

Prior work extensively analyzed the cost of zero initialization [54], which is required by Java and essentially all managed language specifications. They report that zeroing consumes 2.7-4.5% of execution time on average across several architectures, causes cache pollution, and is responsible for 25% of memory traffic. They propose non-temporal writes of zeroes that go directly to memory, bypassing the cache entirely. This software-only technique and another that spawns a concurrent thread to perform zeroing reduce both the performance overhead due to zeroing instructions and the perturbation in the cache. However, without the hardware support of an in-cache zeroing instruction, their non-temporal zeroing increases traffic to memory and bandwidth usage significantly, sometimes doubling the bandwidth over normal temporal stores [54]. We instead zero cache lines directly without fetching them from memory to save the critical resource of off-chip bandwidth, while minimizing cache displacement.

Prefetching. Prior work proposes a cooperative software-hardware technique to improve cache replacement decisions and prefetching based on program data usage [46, 49, 50]. Static analysis identifies data that both will and will not be reused again and passes this information to hardware as hints on memory instructions. The hardware has one extra bit per cache line to optimize the choice of what to evict from the cache for C and Fortran programs. While this research focuses on software-hardware cooperation, the hints are based on static program information and do not guarantee that the data that should be evicted is dead or will not be used again by the program. Other work has used software-only prefetching to improve garbage collection performance [12, 15].

Co-designed architecture. Prior works propose new memory architectures that are co-designed with the JVM, directly supporting objects and/or garbage collection [38, 51, 53]. While such an architecture can improve memory performance, it requires a radical re-design, whereas our technique works with minimal changes to existing hardware and is language-agnostic.

Write policies. Jouppi investigated cache policies and their effect on performance [26]. The best *write-validate* policy combines no-fetch-on-write and write-allocate for good performance. This prior work motivates zeroing cache lines without reading from memory and limiting write-back traffic to memory, both of which we target in this paper.

Cache replacement and dead cache line prediction. A large body of work focuses on changing the order of replacement for cache lines within a cache set to improve performance, see for example [6, 25, 27, 42, 52]. Hardware approaches also predict which blocks in cache can be evicted based on usage patterns, mainly for more agressive prefetching [1, 18, 28, 29, 30, 34, 36]. This work uses the cache hierarchy more efficiently by replacing cache lines the program will not reference again soon. Any hardware-only approach suffers from mispredictions, and because they lack semantic information about future program accesses, they *must* always write modified data to other levels of the memory hierarchy.

7. CONCLUSION

This paper introduces cache scrubbing, a software/hardware cooperative technique to reduce memory bandwidth demand. We identify useless write backs in modern Java benchmarks running on a modern JVM. We find that 10 to 60% of all write backs to memory, depending on the nursery size, contain dead data in the highly-mutated, short-lived nursery address range. We propose cache scrubbing instructions to remove these useless write backs. Scrubbing requires modest changes to both software and hardware. The Java virtual machine calls scrubbing and zeroing instructions, for regions that are dead or being zeroed, to communicate program semantics to hardware. In order for hardware to scrub and zero lines in cache, we present minor changes to the popular MESI cache coherence protocol, showing how our instructions maintain its invariants.

We explore three simple cache line scrubbing instructions, and evaluate them with an existing zeroing instruction. Our scrubbing instructions may be used by any programming language implemen-

tation to reduce memory traffic at the point the runtime identifies cache-line-sized blocks of dead data and this paper shows they are very effective when used with region allocators. We also show that in-cache zeroing is particularly effective for languages that require zero initialization; many current architectures fetch lines that will be completely overwritten with zeroes, wasting a lot of bandwidth and computing resources.

To evaluate these instructions, we built a new simulation methodology and followed existing best practices to simulate multi-threaded Java workloads relatively quickly on multicore hardware. We reveal that while the best scrubbing instruction is *clclean*, which undirties cache lines and moves them to the LRU position, adding *clzero2* to *clclean* overall is the most effective at saving substantial amounts of DRAM traffic, DRAM energy, and execution time across a range of nursery sizes.

In summary, scrubbing effectively diminishes the use of two of the most critical system resources in multicore systems: bandwidth and power. Cooperating between software and hardware will only become more important as future multiprocessors demand increasing levels of efficiency.

Acknowledgements

This work was supported in part by the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013) / ERC Grant agreement no. 259295 and by NSF grant SHF-0910818. We would like to thank Karin Strauss for her help on the cache coherence protocol invariants and how scrubbing can maintain them. Also, thank you to Jeff Stuecheli and John Carter for their help on understanding the PowerPC instructions.

8. REFERENCES

[1] J. Abella, A. Gonzalez, X. Vera, and M. O'Boyle. IATAC: A smart predictor to turn-off L2 cache lines. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2(1):55–77, 2005.

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

[3] ARM. ARM946E-S Revision: r1p1 Technical Reference Manual, 2007.

[4] E. D. Berger, B. G. Zorn, and K. S. McKinley. Reconsidering custom memory allocation. In *ACM Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 1–12, Nov. 2002.

[5] S. Blackburn, P. Cheng, and K. S. McKinley. Myths and realities: The performance impact of garbage collection. In *Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 25–36, 2004.

[6] S. Blackburn and K. S. McKinley. Transient caches and object streams. Technical Report TR-CS-06-03, Australian National University, Department of Computer Science, October 2006.

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

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

[9] 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(8):83–89, Aug. 2008.

[10] D. Burger, J. R. Goodman, and A. Kagi. Memory bandwidth limitations of future microprocessors. In *International Symposium on Computer Architecture (ISCA)*, pages 78–89, 1996.

[11] T. E. Carlson, W. Heirman, and L. Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Conference on High Performance Computing Networking, Storage and Analysis (Supercomputing – SC)*, number 52, 2011.

[12] C.-Y. Cher, A. L. Hosking, and T. N. Vijaykumar. Software prefetching for mark-sweep garbage collection: Hardware analysis and software redesign. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 199–210, 2004.

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

[14] D. I. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *IEEE International Conference on Computer Design*, pages 522–525, 1992.

[15] R. Garner, S. M. Blackburn, and D. Frampton. Effective prefetch for mark-sweep garbage collection. In *The 2007 International Symposium on Memory Management (ISMM)*, pages 43–54, Oct 2007.

[16] J. Ha, M. Gustafsson, S. Blackburn, and K. S. McKinley. Microarchitectural characterization of production JVMs and Java workloads. In *IBM CAS Workshop*, 2008.

[17] W. Heirman, S. Sarkar, T. E. Carlson, I. Hur, and L. Eeckhout. Power-aware multi-core simulation for early design stage hardware/software co-optimization. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 3–12, Sept. 2012.

[18] Z. Hu, S. Kaxiras, and M. Martonosi. Timekeeping in the memory system: Predicting and optimizing memory behavior. In *International Symposium on Computer Architecture (ISCA)*, pages 209–220, 2002.

[19] X. Huang, Z. Wang, S. Blackburn, K. S. McKinley, J. E. B. Moss, and P. Cheng. The garbage collection advantage: Improving mutator locality. In *ACM Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 69–80, 2004.

[20] IBM. Power ISA Version 2.06 Revision B, 2010.

[21] H. Inoue, H. Komatsu, and T. Nakatani. A study of memory management for web-based applications on multicore processors. In *ACM Programming Language Design and Implementation (PLDI)*, pages 386–396, 2009.

[22] Intel. Intel Xeon Phi coprocessor instruction set architecture reference manual, 2012.

[23] C. Isen and L. John. ESKIMO: Energy savings using semantic knowledge of inconsequential memory occupancy for DRAM subsystem. In *ACM/IEEE International Symposium on Microarchitecture*, pages 337–346, 2009.

[24] L. Ivanov and R. Nunna. Modeling and verification of cache coherence protocols. In *IEEE International Symposium on Circuits and Systems*, pages 129–132, 2001.

[25] A. Jaleel, K. B. Theobald, S. C. Steely Jr., and J. Emer. High performance cache replacement using re-reference interval prediction (RRIP). In *International Symposium on Computer Architecture (ISCA)*, pages 60–71, 2010.

[26] N. P. Jouppi. Cache write policies and performance. In *International Symposium on Computer Architecture (ISCA)*, pages 191–201, 1993.

[27] M. Kampe, P. Stenstrom, and M. Dubois. Self-correcting LRU replacement policies. In *Conference on Computing Frontiers*, pages 181–191, 2004.

[28] S. Kaxiras, Z. Hu, and M. Martonosi. Cache decay: Exploiting generational behavior to reduce cache leakage power. In *International Symposium on Computer Architecture (ISCA)*, pages 240–251, 2001.

[29] M. Kharbutli and Y. Solihin. Counter-based cache replacement and bypassing algorithms. *IEEE Transactions on Computers*, 57(4):433–447, 2008.

[30] A.-C. Lai, C. Fide, and B. Falsafi. Dead-block prediction & dead-block correlating prefetchers. In *International Symposium on Computer Architecture (ISCA)*, pages 144–154, 2001.

[31] C. Lefurgy, K. Rajamani, F. L. Rawson III, W. M. Felter, M. Kistler, and T. W. Keller. Energy management for commercial servers. *IEEE Computer*, 36(12):39–48, 2003.

[32] J. A. Lewis, B. Black, and M. H. Lipasti. Avoiding initialization misses to the heap. In *International Symposium on Computer Architecture (ISCA)*, pages 183–194, 2002.

[33] H. Lieberman and C. E. Hewitt. A real time garbage collector based on the lifetimes of objects. *Communications of the ACM (CACM)*, 26(6):419–429, 1983.

[34] W.-F. Lin, S. K. Reinhardt, and D. Burger. Designing a modern memory hierarchy with hardware prefetching. *IEEE Transactions on Computers*, 50(11):1202–1218, 2001.

[35] F. Liu, X. Jiang, and Y. Solihin. Understanding how off-chip memory bandwidth partitioning in chip multiprocessors affects system performance. In *International Symposium on High-Performance Computer Architecture (HPCA)*, pages 57–68, 2010.

[36] H. Liu, M. Ferdman, J. Huh, and D. Burger. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency. In *ACM/IEEE International Symposium on Microarchitecture*, pages 222–233, 2008.

[37] K. McMillan. Symbolic model checking. Technical Report CMU-CS-92-131, Carnegie Mellon University, 1992.

[38] M. Meyer. An on-chip garbage collection coprocessor for embedded real-time systems. In *Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 517–524, Washington, DC, USA, 2005.

[39] Micron. TN-41-01: Calculating memory system power for DDR3, 2007.

[40] D. Molka, D. Hackenberg, R. Schone, and M. Muller. Memory performance and cache coherency effects on an Intel Nehalem multiprocessor system. In *Parallel Architectures and Compilation Techniques (PACT)*, pages 261–270, 2009.

[41] M. S. Papamarcos and J. H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *International Symposium on Computer Architecture (ISCA)*, pages 348–354, 1984.

[42] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer. Adaptive insertion policies for high performance caching. In *International Symposium on Computer Architecture (ISCA)*, pages 381–391, 2007.

[43] 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 *International Symposium on Computer Architecture (ISCA)*, pages 371–382, 2009.

[44] J. B. Sartor. *Exploiting Language Abstraction to Optimize Memory Efficiency*. PhD thesis, The University of Texas at Austin, 2010.

[45] J. B. Sartor and L. Eeckhout. Exploring multi-threaded Java application performance on multicore hardware. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications (OOPSLA)*, pages 281–296, 2012.

[46] J. B. Sartor, S. Venkiteswaran, K. S. McKinley, and Z. Wang. Cooperative caching with keep-me and evict-me. In *Workshop on Interaction between Compilers and Computer Architectures*, pages 46–57, 2005.

[47] 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.

[48] D. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Software Engineering Symposium on Practical Software Development Environments (SESPSDE)*, pages 157–167, 1984.

[49] Z. Wang, D. Burger, K. S. McKinley, S. K. Reinhardt, and C. C. Weems. Guided region prefetching: A cooperative hardware/software approach. In *International Symposium on Computer Architecture (ISCA)*, pages 388–398, 2003.

[50] Z. Wang, K. S. McKinley, A. L. Rosenberg, and C. C. Weems. Using the compiler to improve cache replacement decisions. In *Parallel Architectures and Compilation Techniques (PACT)*, pages 199–210, 2002.

[51] M. Wolczko and I. Williams. An alternative architecture for objects: Lessons from the mushroom project. In *ACM OOPSLA Workshop on Memory Management and Garbage Collection*, 1993.

[52] W. A. Wong and J.-L. Baer. Modified LRU policies for improving second-level cache behavior. In *High-Performance Computer Architecture (HPCA)*, pages 49–60, 2000.

[53] G. Wright, M. L. Seidl, and M. Wolczko. An object-aware memory architecture. *Sci. Comput. Program.*, 62(2):145–163, Oct. 2006.

[54] X. Yang, S. M. Blackburn, D. Frampton, J. B. Sartor, and K. S. McKinley. Why nothing matters: The impact of zeroing. In *ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 307–324, 2011.

[55] 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 *ACM Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 361–376, 2009.