Avoiding Initialization Misses to the Heap

Jarrod A. Lewis[†], Bryan Black[‡], and Mikko H. Lipasti[†]

[†]*Electrical and Computer Engineering*
University of Wisconsin-Madison
{lewisj, mikko}@ece.wisc.edu

[‡]*Intel Labs*
Intel Corporation
bryan.black@intel.com

Abstract

*This paper investigates a class of main memory accesses (*invalid memory traffic*) that can be eliminated altogether. Invalid memory traffic is real data traffic that transfers invalid data. By tracking the initialization of dynamic memory allocations, it is possible to identify store instructions that miss the cache and would fetch uninitialized heap data. The data transfers associated with these initialization misses can be avoided without losing correctness. The memory system property crucial for achieving good performance under heap allocation is* cache installation *- the ability to allocate and initialize a new object into the cache without a penalty. Tracking heap initialization at a cache block granularity enables cache installation mechanisms to provide zero-latency prefetching into the cache. We propose a hardware mechanism, the* Allocation Range Cache*, that can efficiently identify initializing store misses to the heap and trigger cache installations to avoid invalid memory traffic.*

Results: *For a 2MB cache 23% of cache misses (35% of compulsory misses) to memory are initializing the heap in the SPEC CINT2000 benchmarks. By using a simple base-bounds range sweeping scheme to track the initialization of the 64 most recent dynamic memory allocations, nearly 100% of all initializing store misses can be identified and installed in cache without accessing memory. Smashing invalid memory traffic via cache installation at a cache block granularity removes 23% of all miss traffic and can provide up to 41% performance improvement.*

1. Introduction

Microprocessor performance has become extremely sensitive to memory latency as the gap between processor and main memory speed widens [17]. Consequently, main memory bus access has become a dominating performance penalty and machines will soon be penalized thousands of processor cycles for each data fetch. Substantial research has been devoted to reducing or burying these large memory access latencies. Latency hiding techniques include lockup-free caches, hardware and software prefetching, and multithreading. However, many of these techniques used to tolerate growing memory latency do so at the expense of increased bandwidth requirements [3]. It is apparent in our quest for performance that memory bandwidth will be a critical resource in future microprocessors.

This work investigates the reduction of bandwidth requirements by avoiding initialization misses to dynamically-allocated memory. The use of dynamic storage allocation in application programs has increased dramatically, largely due to the use of object-oriented programming [18]. Traditional caching techniques are generally ineffective at capturing reference locality in the heap due to its extremely large data footprint [7][18]. Dynamic memory allocation through the heap can cause invalid, uninitialized memory to be transferred from main memory to on-chip caches. *Invalid memory traffic* is real data traffic that transfers invalid data. This traffic can be avoided without affecting program correctness. We observe that a significant percentage of bus accesses transfer invalid data from main memory in the SPEC CINT2000 benchmarks. For a 2MB cache, 23% of all misses (35% of all compulsory misses) that access memory are transferring invalid heap data.

First, this paper discusses the program semantics that lead to invalid memory traffic in Section 2, then it quantifies its contribution to compulsory misses and total cache misses in Section 5. In Section 6, we propose an allocation range base-and-bounds tracking scheme for dynamically tracking and eliminating excess invalid memory traffic. Finally, we propose an implementation scheme and quantify potential performance gains in Section 7.

2. Invalid Memory Traffic

Invalid memory traffic is the transfer of data between caches and main memory that has either not been initialized by the program, or has been released by the program. Invalid memory traffic can only occur in the dynamically-allocated structures of the heap and stack, because instruction and static memory are always valid to the application. Hardware will transfer data, based on demand, regardless of memory state, but the operating system must maintain a strict distinction and track valid and invalid data in order to maintain program correctness. During program execution, all stack and heap memory is invalid until allocated and initialized for use. Figure 1 illustrates the memory states and transitions for dynamic heap space. Until heap space is allocated, it remains unallocated-invalid. After allocation the new memory location transitions from unallo-

Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA'02)
1063-6897/02 $17.00 © 2002 **IEEE**

cated-invalid to allocated-invalid. Memory transferred in allocated-invalid state is considered *invalid memory traffic*. It remains allocated-invalid until it is initialized by a write to that memory location. It will then transition to allocated-valid. Once a memory location is allocated-valid it is ready for program use. The application program can read and write this location numerous times until it is no longer needed. When the application is finished with the memory, it returns the memory back to the heap, and the memory location's state transitions back into unallocated-invalid. There are three memory states in Figure 1, of which only the allocated-valid state contains valid data. All memory transfers in the remaining two states transfer invalid data. There are two causes of invalid memory traffic: 1) An initializing store miss to allocated-invalid memory; 2) A writeback of allocated-invalid or unallocated-invalid memory. It is also possible to load from allocated-invalid memory, but reading uninitialized data is an undefined operation.

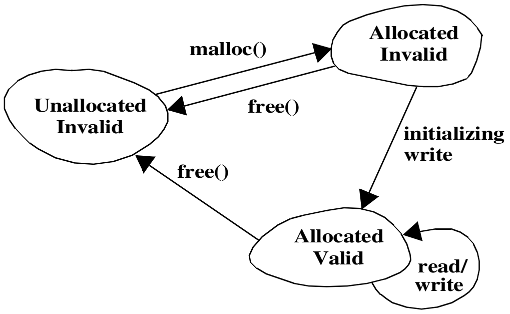


Figure 1. Dynamic memory states and transitions

Initializing stores may occur each time a program allocates new memory. A data writeback occurs when a dirty/modified cache line is evicted from a cache that is not write-through. If the evicted line was deallocated by the program before eviction, the writeback becomes invalid memory traffic. If an invalid writeback occurs or an initializing store misses all on-chip caches, an unnecessary and avoidable bus transfer of invalid data is created to access main memory.

This study focuses on invalid memory traffic that arises from initializing stores to the heap. All dynamic memory allocation activity is tracked in the SPEC CINT2000 benchmarks via the `malloc()` memory allocation routine. Using the memory states of Figure 1 (unallocated-invalid, allocated-invalid, and allocated-valid) heap data traffic can be tracked and identified as either valid or invalid memory traffic. Note that this discussion is specific to the semantics of C/C++ dynamic memory allocation; other languages have differing semantics and must be treated accordingly.

3. Related Work

Diwan et.al. [7] observe that heap allocation can have a significant memory system cost if new objects cannot be directly allocated into cache. They discover that by varying caching policies (sub-blocking) and increasing capacity, the allocation space of programs can be captured in cache, thus reducing initializing write misses. Similarly, in Jouppi's [12] investigation of cache write policies, he introduces the "write-validate" policy, which performs word-level sub-blocking [6]. With write-validate, the line containing the write is not fetched. The data is written into a cache line with valid bits turned off for all but the data that is being written. A write validate policy would effectively eliminate 100% of initializing write misses; however, the implementation overhead of this scheme is significant.

Wulf and McKee [20] explore the exponential advancement disparity between processor and memory system speeds. They conclude that system speed will be dominated by memory performance in future-generation microprocessors. To hurdle the imminent memory wall [20][7], they propose the idea of reducing compulsory misses, arising from dynamic memory initialization, by possibly having the compiler add a "first write" instruction that would bypass cache miss stalls. Such instructions now exist, for example `dcbz` in PowerPC [11]. These instructions allocate entries directly into cache and initialize them without incurring a miss penalty (cache installation). These installation instructions can be an extremely effective method for eliminating initializing write misses.

The compiler is statically limited to using cache installation immediately after new memory is allocated because it can not track memory use beyond the initial allocation. The operating system, on the contrary, could potentially make effective use of an installation instruction. Our work proposes eliminating initializing write misses at a cache block granularity, in contrast to the sub-blocking of write-validate and the software-controlled page-granular cache installation of uninitialized memory by an operating system. In Section 7 we show that both cache block- and page-granular cache installation can improve performance dramatically. Moreover we will demonstrate instances where block-granular installation performs significantly better than page-granular installation by avoiding cache pollution effects.

4. Methodology

This section outlines the full-system simulation environment used to gather all data for this study.

4.1. Simulation Environment

This work utilizes the PharmSim simulator, developed at the University of Wisconsin-Madison. PharmSim incorporates a version of SimOS adapted for the 64-bit PowerPC ISA that boots AIX 4.3.1. SimOS is the full-system simulator originally developed at Stanford University [15][16]. SimOS is a unique simulation environment that simulates both application and operating system code, and enables more accurate workload simulations by accounting for the interaction between the operating system and applications. PharmSim incorporates with SimOS a detailed, execution-driven out-of-order processor and memory subsystem model that precisely simulates all of the semantics of the entire PowerPC instruction set. This includes speculative execution of supervisor-mode instructions, memory barrier semantics, all aspects of address translation, including hardware page table walks, page faults, external interrupts, and so on. We have found that accurate modeling of all of these effects is vitally important, even when studying SPEC benchmarks. For example, we found that the AIX page fault handler already performs page-granular cache installation for newly-mapped uninitialized memory using the `dcbz` instruction. Had we employed a user-mode-only simulation environment like Simplescalar, this effect would have been hidden, and the performance results presented in Section 7 would have been overstated.

For the characterization data in Section 5 and Section 6, all memory references are fed through a one-level data cache model. Cache sizes of 512KB, 1MB, and 2MB are simulated for block sizes of 64, 128, and 256 bytes. To reduce the design space, a fixed associativity of 4 was chosen for each configuration. It is assumed that this single cache will represent total on-die cache capacity, thus all cache misses result in bus accesses. For the detailed timing simulations presented in Section 7, the baseline machine is configured as an 8-wide, 6-stage pipeline with an 8K combining predictor, 128 RUU entries, 64 LSQ entries, 64 write buffers, 256KB 4-way associative L1D cache, 64KB 2-way associative L1I, and a 2MB 4-way associative L2 unified cache. All cache blocks are 64 bytes. L2 latency is 10 cycles; memory latency is fixed at 70 cycles. We purposely chose an aggressive baseline machine to devaluate the impact of store misses.

The SPEC CINT2000 integer benchmark suite is used for all results presented in this paper. All benchmarks were compiled with the IBM xlc compiler, except for the C++ eon code which was compiled using g++ version 2.95.2. The first one billion instructions of each benchmark were simulated under PharmSim for all characterization and performance data. It is necessary to simulate from the very beginning of these applications in order to capture all dynamic memory allocation and initialization. The input set, memory instruction percentage, and miss rates for a 1MB 4-way set-associative cache with 64 byte blocks are summarized for all benchmarks in Table 4-1.

Table 4-1. Characteristics of benchmark programs

SPEC CINT2000	Input Sets	Memory Instr%	Misses per 1000 Instr
bzip2	lgred.graphic	37.9%	0.683
crafty	oneboard.in	39.3%	0.053
eon	cook	55.9%	0.015
gap	test.in	46.1%	0.335
gcc	lgred.cp-decl.i	42.7%	0.159
gzip	lgred.graphic	41.0%	0.156
mcf	lgred.in	37.2%	7.533
parser	lgred.in	39.5%	0.982
perlbmk	lgred.makerand	55.1%	0.346
twolf	lgred.in	42.8%	0.022
vortex	lgred.raw	48.1%	0.164
vpr	lgred.raw	34.2%	0.015

4.2. Dynamic Memory Allocation Tracking

In order to study initialization cache misses to the heap all dynamic memory allocation and initialization must be tracked. Tracking dynamic memory behavior allows the simulator to identify initializing stores that cause invalid memory traffic. Dynamic memory behavior is easily identified through the C standard library memory allocation function `malloc()`. The operating system maintains a free list of available heap memory. During memory allocation, the free list is searched for sufficient memory to handle the current request. If there is insufficient memory the heap space is extended. When available memory is found, a portion of the heap is removed from the free list and an allocation block is created. By identifying the calls to `malloc()` during simulation, the dynamic memory allocation activity can be precisely quantified and analyzed.

5. Heap Initialization Analysis

Before any memory traffic activity results are presented it is important to discuss dynamic memory allocation patterns. As discussed in Section 2, dynamic memory allocation is the source of the invalid memory traffic this work seeks to eliminate.

5.1. Dynamic Memory Allocation

All dynamic memory activity to the heap is tracked by monitoring both user- and kernel-level invocations of the `malloc()` memory allocation routine. Figure 2 itemizes the raw number of dynamic procedure calls to `malloc()` according to different allocation sizes. For example twolf

Dynamic memory allocation instances

	<64 B	<2 KB	<256 KB	<16 MB	≥16 MB
bzip2	320	47	9	9	0
crafty	319	78	12	2	0
eon	1,948	145	28	0	0
gap	325	46	11	0	1
gcc	665	258	1,594	4	0
gzip	2,492	636	95	3	0
mcf	354	52	16	0	1
pars	390	46	59	0	1
perl	804	87	12	2	0
twolf	28,438	841	38	0	0
vortex	319	29,279	1,006	0	0
vpr	1865	93	22	0	0

Total dynamic memory allocated (in KB)

	<64 B	<2 KB	<256 KB	<16 MB	≥16 MB
bzip2	6.3	19.1	295.8	13,198	0
crafty	6.4	22.1	631.8	512	0
eon	35.6	41.3	371.8	0	0
gap	6.3	18.1	362.8	0	100MB
gcc	13.9	63.7	7,199.8	1,654	0
gzip	65.9	212.4	640.5	3,372	0
mcf	6.9	21.5	639.4	0	92MB
pars	8.1	18.1	496.6	0	30MB
perl	17.9	32.1	311.7	8,192	0
twolf	742.6	234.5	420.3	0	0
vortex	6.4	3,798.5	8,157.4	0	0
vpr	23.9	35.8	416.5	0	0

Figure 2. Dynamic memory allocation activity for SPEC CINT2000 benchmarks.

has 28,438 calls to `malloc()` that request less than 64 bytes of space. The raw number of allocations varies significantly across the benchmarks and some benchmarks allocate very large single blocks of memory, e.g. gap, mcf, and parser.

Figure 2 also quantifies the total dynamic memory allocated according to allocation size as observed in each benchmark. The total allocated memory represents all memory space that is assigned from the heap through calls to the `malloc()` routine. For example gcc has 7,199.8KB of its dynamically-allocated memory allocated between 2KB and 256KB at a time. This data shows a drastic difference in memory allocation behavior across the SPEC CINT2000 benchmarks. Gap, mcf, and parser allocate the bulk of their dynamic memory through 1 very large allocation (100MB, 92MB, and 30MB respectively). Although small allocations dominate the call distribution, the larger less frequent allocations are responsible for the bulk of allocated memory simply because they are so large. In contrast, gcc, twolf, and vortex allocate most of their dynamic memory through a large number of `malloc()` calls that allocate less than 2KB of data at a time.

Even though these allocation patterns are significantly different, we will show in Section 6 that the initialization of these different allocation sizes demonstrate very similar locality. Most allocations are initialized soon after they are allocated, and they are often initialized by a sequential walk through the memory. Therefore a similar mechanism can be used to track small allocations just the same as very large allocations. This fundamental observation will be discussed more in Section 6.

5.2. Initialization of Allocated Memory

Since the cache block is the typical granularity of a bus transfer, memory initialization is tracked by cache block for all results. Once allocated, all blocks remain in the allocated-invalid state until they are initialized. A store is required to move the allocated-invalid blocks to the allocated-valid state. Figure 3 shows what percentage of dynamically allocated memory (at a cache block granularity) is initialized and if it is initialized by a store miss or a store hit. Eon, parser, twolf, and vpr use 40% or less of their allocated memory, while gap, mcf, perlbmk, and vortex initialize most allocated cache blocks. Interestingly, on average 88% of all blocks initialized (60% of all allocated blocks) are initialized by a store miss. As discussed in

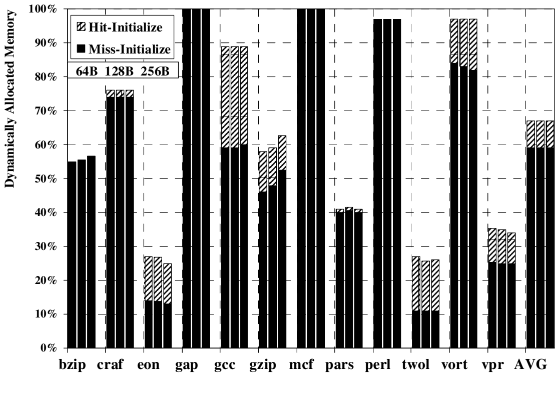


Figure 3. Initialization of dynamic memory
Initialization is shown for a 2MB 4-way set-associative cache with block sizes of 64, 128, and 256 bytes. On average, 60% of allocated cache blocks are initialized on a cache miss.

Section 2, these store misses are a source of invalid memory traffic. The miss rate of initializing stores gives insight into the reallocation of heap memory. If a memory block is initialized on a cache hit, and there is no prefetching, the block must have been brought into the cache on an earlier miss initialization from a previous allocation instance. The miss rates in Figure 3 are very high, so there is very little temporal reallocation of heap space. Section 5.3 will now discuss initialization misses and quantify how much of this cache miss traffic can be eliminated.

5.3. Invalid Cache Miss Traffic

Cache misses to the heap are references to memory allocated through `malloc()`, while non-heap misses are all other traffic, namely stack references and static variables. Store misses are distinguished as misses to either heap or non-heap memory space. Figure 4 illustrates all main memory accesses caused by stores initializing allocated-invalid memory (Initialize), stores that modify allocated-valid memory (Modify), and stores to non-heap memory (Non-Heap). Load misses represent the difference between the top of the accumulated store miss bars and 100% of cache misses. From Figure 4, 23% of all misses in a 2MB cache with 64 byte blocks initialize allocated-invalid memory space. All data fetches for these misses can be eliminated because they are invalid memory traffic that fetch invalid data just to initialize it when it eventually reaches the cache. Therefore nearly 1/4 of all incoming data traffic on the bus can be eliminated.

Figure 5 shows the sensitivity of the percentage of initializing stores to cache size and block size averaged across the SPEC CINT2000 benchmarks. One noticeable trend in this data is that the percentage of misses that initialize the heap (Initialize) increases with increasing cache capacity. However, initialization misses decrease with larger block sizes due to spatial locality prefetching from the larger blocks.

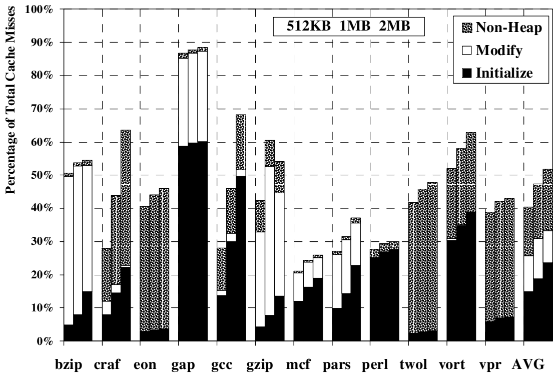


Figure 4. Cache miss breakdown
Misses are shown for cache sizes of 512KB, 1MB, and 2MB, all with associativity 4 and block size 64 bytes. Up to 60% and on average 23% of cache misses for 2MB of cache are initializing the heap.

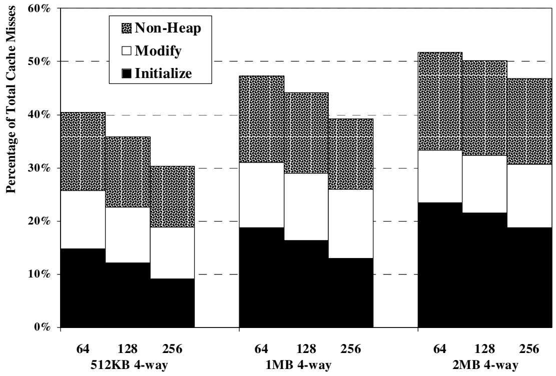


Figure 5. Initializing store miss percentage sensitivity to cache size and block size
The relative percentage of cache misses that initialize the heap (Initialize) increases with increasing cache capacity. However, initializing store ratios decrease as block size increases.

Reducing bus traffic by avoiding initialization misses can improve performance directly by reducing pressure on store queues and cache hierarchies. Indirectly, avoiding invalid memory traffic will decrease bus bandwidth requirements, enabling bandwidth-hungry performance optimizations such as prefetching and multi-threading to consume more bandwidth.

5.4. Compulsory Miss Initialization

Compulsory miss initializations occur when portions of the heap are initialized for the first time. Capacity miss initializations occur when data is evicted from cache and is subsequently re-allocated and re-initialized. Figure 6 demonstrates a semantic breakdown of all compulsory misses for a range of cache block sizes. Compulsory misses are categorized as initializing the heap (Initialize-Cold), non-heap stores (Non-Heap-Cold), or loads (Load-Cold). Note that compulsory misses, or *cold-start misses*, are caused by the first access to a block that has never been in the cache. Therefore the number of compulsory misses for any size cache is proportional only to block size. Figure 6 shows that for 2MB of cache, across all SPEC CINT2000 benchmarks approximately 50% of all cache misses are compulsory misses, and 35% of compulsory misses are initializing store misses. Thus 35% of compulsory misses are avoidable invalid memory traffic. Over 1/3 of all unique memory blocks cached are brought in as uninitialized heap data. As an extreme, mcf shows 95% of compulsory misses are initializing heap memory. The elimination of invalid compulsory miss traffic breaks the infinite cache

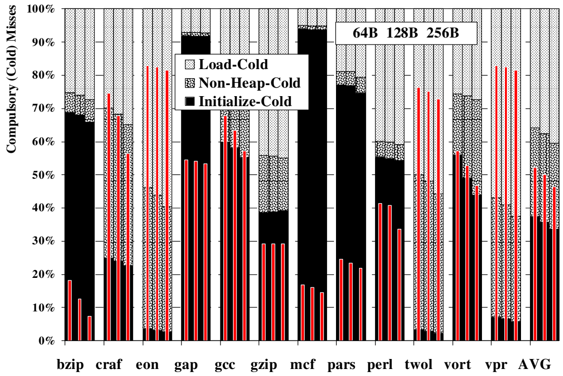


Figure 6. Cache compulsory miss breakdown
Compulsory misses are shown for a 2MB 4-way set-associative cache for 64, 128, and 256 byte blocks. The narrow bars inside each stacked bar represent the percentage of all cache misses that are compulsory for each program.

miss limit, where the number of compulsory misses of a finite-sized cache is equal and bound by that of an infinite-sized cache with the same block size [6]. Note that as block size increases, both the percentage of compulsory misses that initialize the heap (Initialize-Cold) and the percentage of all misses that are compulsory decrease. Larger block sizes perform spatial locality prefetches and reduce compulsory misses.

5.5. Initialization Throughout Execution

Figure 7 shows an accumulated distribution of all initializing stores identified in the first one billion instructions in the SPEC CINT2000 benchmarks. This data gives insight into the initialization of the heap throughout program execution. Here, largely as a design artifact of the SPEC benchmarks, most initializations of the heap occur in the first 500 million instructions. From Figure 2 gap, mcf, and parser are identified as having one very large dynamic memory allocation (100MB, 92MB, 30MB respectively). Figure 7 shows that these programs initialize their working set of dynamic memory rather quickly. Also from Figure 2 bzip2, gcc, gzip, twolf, and vortex are observed to allocate their memory in frequent, smaller chunks. Figure 7 shows these programs initialize their memory more steadily throughout the first one billion instructions of their execution. Note that although initializations are shown here for the first billion instructions (due to finite simulation time), dynamic memory allocation and initialization can occur steadily throughout program execution, depending on the application.

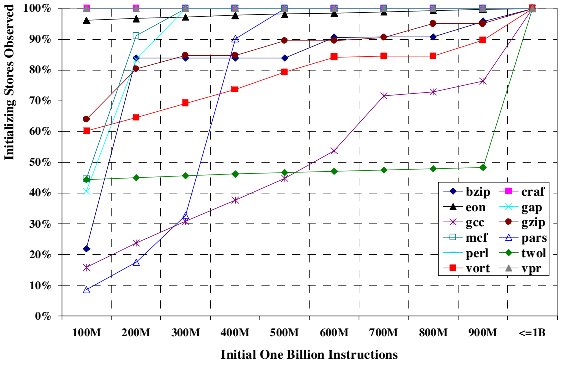


Figure 7. Initializing stores identified in the first one billion instructions

6. Identifying Initializing Stores

As discussed in Section 2, all initializing store misses in a write-allocate memory system cause invalid memory traffic (off-chip bus accesses) that can be eliminated. To eliminate this traffic we must be able to identify a cache miss as invalid before the cache miss handling procedure begins, i.e. before allocating entries in miss queues and arbitrating for the memory bus. A table structure that records allocation ranges used by the program can be used for this purpose. Each dynamic memory allocation creates a new range in the table. Table entries track the store (initialization) activity within the recorded allocation ranges using a base-bounds range summary technique. When a store miss to uninitialized heap memory is detected the cache block is automatically created in the cache hierarchy without a fetch to main memory (cache installation), effectively eliminating invalid memory traffic. In a cache coherent system, a processor can issue a cache installation (e.g. `dcbz`) as soon as write permission is granted for that block. Once granted, the block is installed in the cache with the value zero, thus realizing a zero-latency data prefetch for the uninitialized heap memory.

Before an implementation such as this can be feasible three main questions must be answered. (1) How can the hardware detect a dynamic memory allocation call? (2) Is the working set of allocation ranges small enough to cache in a finite table? (3) How can a single table entry track the behavior of potentially millions of cache blocks within a single allocation range?

6.1. Identifying Allocations in Hardware

Again, this study is limited to programs written in C and C++, but could easily extend to all programs that utilize dynamic memory allocation, regardless of programming language. Identifying memory allocation through `malloc()` or any other construct can be accomplished with a new special instruction. A simple instruction that writes the address and size of the allocation into the base-bounds tracking table can be added to the memory allocation routine. In PowerPC a move to/from special register [11] can be used to implement these new operations, making identification of memory allocation quite straightforward.

6.2. Allocation Working Set

Figure 2 shows there are anywhere between 300 and 30,000 dynamic memory allocations during the first one billion instructions of the SPEC CINT2000 benchmarks. However, the working set of uninitialized allocations is much smaller. Figure 8 presents the number of allocations (tracked with a first-in-first-out FIFO policy) required to identify all initializing store misses to all allocations. This data shows that the initialization of the heap is not separated far from its allocation. For all benchmarks (except parser) it is necessary to track only the eight most recent dynamic memory allocations to capture over 95% of all initializing stores. Parser requires knowledge of the past 64 allocations. Even at 64 entries, a hardware allocation tracking table could feasibly be implemented to track this small subset of all allocations.

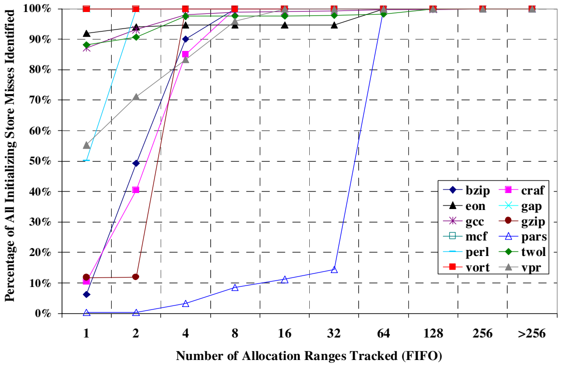


Figure 8. Memory allocation working set for FIFO initialization tracking table

6.3. Tracking Cache Block Initializations

The next question that must be addressed is how to efficiently represent large allocated memory spaces in a finite allocation cache. As discussed in Section 5.1., a cache block is the typical granularity of a bus transfer. Therefore memory initialization must be tracked by cache block or larger to identify invalid memory traffic. All cache blocks within an allocation range must be tracked in order to determine which pieces of the allocation space are valid and invalid. If all cache blocks can not be tracked then it is not possible to identify initializing stores at this granularity. The straightforward approach of maintaining a valid bit for each cache block in the allocated space is not feasible. The largest allocation in gap (100 MB) would require 1.56MB of valid bits in a single entry for 64 byte cache blocks. It turns out the spatial and temporal locality of initializing stores lends nicely to implementation.

6.3.1. Initialization Distance From Allocation

The temporal distance (the number of memory references encountered between the time of allocation and the dynamic memory initialization) and the spatial distance (the distance from the beginning address of the allocation space to the dynamic memory initialization address) of initializing store instructions is presented in Figure 9.

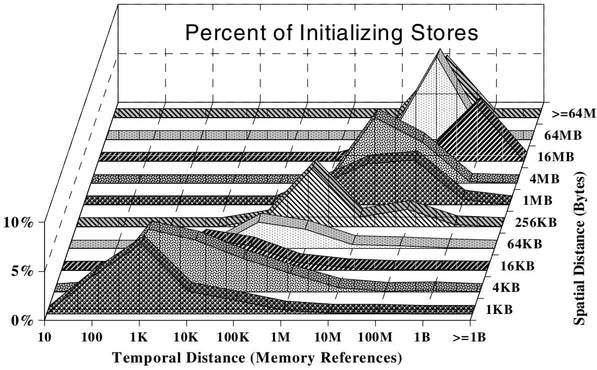


Figure 9. Average temporal and spatial distance of initializing stores from memory allocation

Dynamic instances of initializing stores are classified according to the distance away from the beginning of the allocation space (Spatial Distance) and the number of memory references after the allocation occurred (Temporal Distance).

This figure illuminates the locality pattern of initialization for all dynamic memory allocations, averaged across all SPEC CINT2000 benchmarks. A significant observation is that allocations tend to be initialized sequentially. Blocks at the beginning of an allocation range are initialized quickly and blocks toward the end of the range are initialized much later. This is shown by the diagonal bottom-left to top-right trend in Figure 9. The trend indicates that initializing stores that occur temporally early (to the left of the graph) also occur spatially near (towards the bottom of the graph) to the beginning of an allocation space. This observation coincides with Seidl and Zorn [18] who claim there may exist a sequential initialization bias of heap memory if large amounts of memory are allocated without subsequent deallocations. Figure 9 illustrates this sequential behavior is present across all allocation sizes.

6.3.2. Exploiting Initialization Patterns

Although an approximate sequential initialization pattern is shown in Figure 9, there are actually three main initialization patterns observed in the SPEC CINT2000 benchmarks: *sequential, alternating,* and *striding* as depicted in Figure 10. Three distinct heuristics for tracking these initialization patterns can be employed. *Forward*

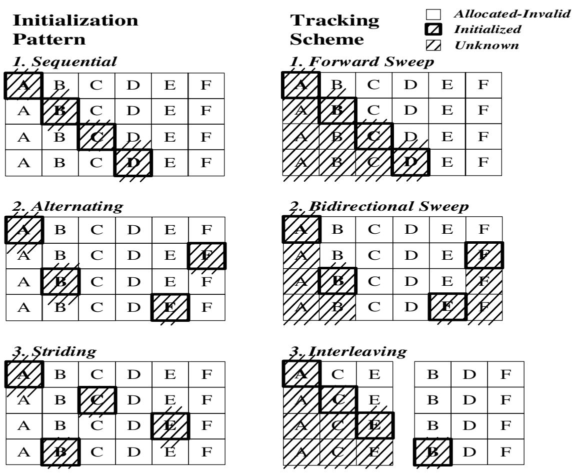


Figure 10. Tracking initialization patterns of dynamic memory allocations

Three main initialization patterns of dynamic memory ranges are observed in the SPEC CINT2000 benchmarks: *sequential, alternating,* and *striding.* Forward sweeping, bidirectional sweeping, and interleaving are effective range tracking schemes for capturing these unique initialization patterns.

sweep tracks the first and last address limits for each allocation, truncating the first address limit on initialization. *Bidirectional sweep* also tracks the two address limits per allocation, but truncates the first or last address limit depending on the location of the initialization. *Interleaving* maintains multiple address limit pairs for each allocation, splitting the range into multiple discontinuous segments. This scheme is extremely effective at capturing striding reference patterns. Writes are routed to an interleaved entry based on the write address, the interleaving granularity and the number of interleaves per range (address/granularity `modulo` interleaves). Forward or bidirectional sweeping is performed on each interleave entry. The idea is to route striding initializations to the *same* interleave entry so that each stride does not truncate the allocation range for all future store addresses; the range is only truncated for addresses that map to the same interleave entry. Thus future initializations to addresses *between* strides will route to a different interleave entry and can be correctly identified as initializing.

6.3.3. Allocation Range Cache

Figure 10 illustrates the tracking schemes that capture multiple initialization patterns in allocation ranges. The *base* and *bound* address limits representing the uninitialized portion of an allocation range are used to identify initialization activity into a single allocation. To identify writes to allocated-invalid memory in an allocation range, it is sufficient to determine if the write falls within the current address limits of the uninitialized portion of the range.

Figure 8 shows that the maximum working set of dynamic memory allocations for the SPEC CINT2000 benchmarks is typically 8 and at most 64 allocations. Tracking the 64 most recent allocations is sufficient to capture nearly all initializations. Therefore we propose a structure called the *Allocation Range Cache* to track the initialization of dynamic memory allocation ranges and identify initializing stores. Since the physical mapping for newly allocated space may not always exist, the Allocation Range Cache will track initializations by *virtual addresses.* To illustrate the operation of this structure we will walk through a simple allocation and initialization example. The example in Figure 11 shows an allocation of addresses *A* through *F* with initializing stores to addresses *A*, C, and *B*.

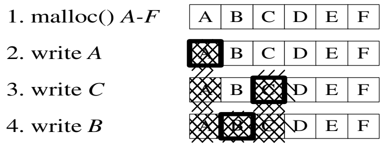


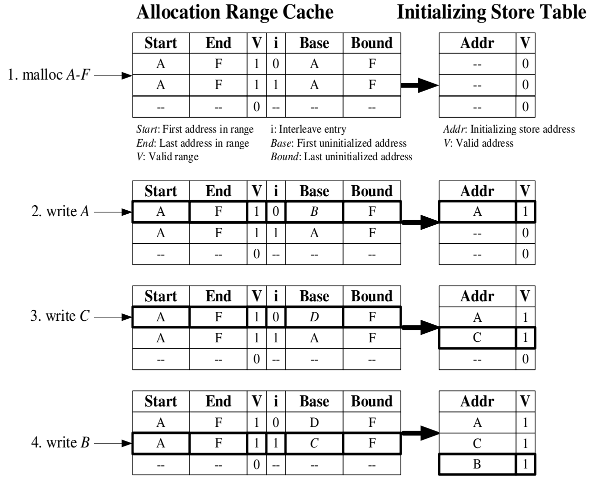
Figure 11. Initializing store example

We will now demonstrate how the Allocation Range Cache can track allocation *A-F* and identify the initializing stores to *A, C,* and *B*.

(1) To capture this activity the Allocation Range Cache represents the uninitialized allocation range *A-F* with two base-bound pairs as shown in Figure 12. This is two-way interleaving. The *Start-End* and *Base-Bound* values for both interleave entries are initialized to *A-F.*

(2) The write of address *A* occurs and a fully-associative search is performed on all *Start-End* pairs for a range that encompasses address *A*. When range *A-F* is found, address *A* is routed to interleave entry i=0 of this range. The *Base* and *Bound* values for this entry are referenced to determine if address *A* is to uninitialized memory. As this is the first write to this range, the *Base-Bound* pair still holds the initial value of *A-F*. Therefore, this write of address *A* is identified as an initializing store and the address is placed in the Initializing Store Table. The Initializing Store Table is simply a list of write addresses that have been identified as initializing stores by the Allocation Range Cache. To record this initialization, the Allocation Range Cache truncates the *Base* value of the referenced entry so that the *Base-Bound* values are now *B-F.* This is forward range sweeping.

(3) The write of address *C* is handled similarly to the previous write of address *A.* The write is identified as initializing by interleave entry i=0, address *C* is sent to the Initializing Store Table, and the *Base* value is truncated to address *D.*



1. malloc *A-F*

Start	End	V	i	Base	Bound
A	F	1	0	A	F
A	F	1	1	A	F
--	--	0	--	--	--

Addr	V
--	0
--	0
--	0

Start: First address in range
End: Last address in range
V: Valid range
i: Interleave entry
Base: First uninitialized address
Bound: Last uninitialized address
Addr: Initializing store address
V: Valid address

2. write *A*

Start	End	V	i	Base	Bound
A	F	1	0	*B*	F
A	F	1	1	A	F
--	--	0	--	--	--

Addr	V
A	1
--	0
--	0

3. write *C*

Start	End	V	i	Base	Bound
A	F	1	0	*D*	F
A	F	1	1	A	F
--	--	0	--	--	--

Addr	V
A	1
C	1
--	0

4. write *B*

Start	End	V	i	Base	Bound
A	F	1	0	D	F
A	F	1	1	*C*	F
--	--	0	--	--	--

Addr	V
A	1
C	1
B	1

Figure 12. Allocation Range Cache
The Allocation Range Cache represents address range *A-F* with 2 base-bound pairs (2-way interleaving) as shown above. Assume that we interleave with a granularity such that addresses *A,C,* and *E* will be routed to interleave entry i=0, and addresses *B,D,* and *F* will be routed to entry i=1. The Initializing Store Table holds store addresses that have been identified as initializing stores by the Allocation Range Cache.

(4) The write of address *B* is routed to interleave entry i=1 for range *A-F*. Since this is the first reference to interleave i=1, the *Base-Bound* pair has the initial value *A-F*. Therefore this write of address *B* is identified as initializing, sent to the Initializing Store Table, and the *Base* value is truncated to address *C*. Note that if address *B* had been routed to interleave i=0, it would **not** have been identified as initializing because the previous write of address *C* truncated the *Base* value to address *D*. There would have been a lost opportunity to correctly identify an initializing store. This is an example of how range interleaving can track striding initialization patterns effectively.

The effectiveness of identifying initializing store misses dynamically with simple forward sweep and bidirectional sweep tracking policies is presented in Figure 13. Simple range sweeping, with one base-bound pair per allocation, captures nearly 100% of all initializations for ten benchmarks. Most benchmarks adhere strictly to sequential initializations. Perl exhibits alternating initialization; therefore a bidirectional policy is more effective than forward sweep.

Initializations in bzip2 and gzip are not captured well with forward or bidirectional range sweeping. These programs often initialize memory in strides of 128, 256, and 1024 bytes. Range interleaving as shown in Figure 10 is required to effectively capture striding initializations. Figure 14 shows that maintaining multiple base-bound pairs for

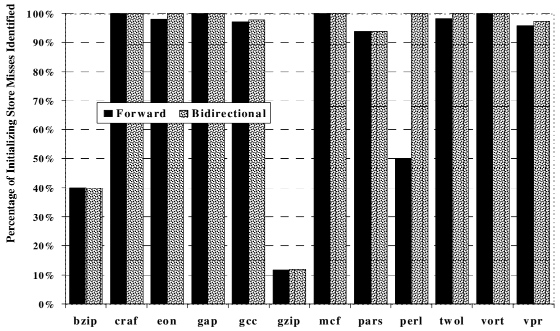


Figure 13. Identifying initializing stores with *forward* and *bidirectional* range sweeping
The percentage of all initializing stores that can be identified by range sweeping for a 1MB 4-way set-associative cache with 64 byte blocks is shown above.

each allocation can significantly improve the effectiveness of range sweeping at identifying initializing stores. Note that only 60% of all initializations in bzip2 can be captured by range sweeping. Bzip2 has one large allocation that is initialized at random locations at random times. Random initialization patterns are not captured with any range sweeping scheme proposed in Figure 10.

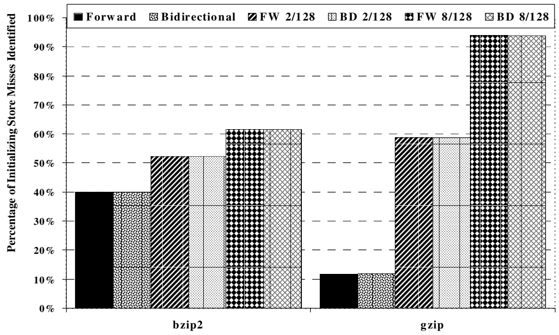


Figure 14. Improving identification of initializations with range sweeping by *interleaving* ranges
The percentage of all initializing stores that can be identified by range interleaving and sweeping, for a 1MB 4-way set-associative cache with 64 byte blocks, is shown above. Forward (FW) and bidirectional (BD) sweeping is performed at an interleave granularity of 128 bytes on two (2/128) and eight (8/128) interleaves per allocation.

7. Implementation and Performance

Initializing store misses cause *invalid memory traffic*, real data traffic between memory and caches that transfers invalid data from the heap. To avoid this traffic, store misses must be identified as invalid before the cache hierarchy initiates a bus request to fetch missed data from memory. The block written by the store can then be installed directly into the cache without fetching invalid data over the bus. This is *block-granular* cache installation. Initializing store miss identification can be done anytime after the store address is generated and before the store enters miss handling hardware. These relaxed timing constraints allow multiple cycles for an identification to resolve. Therefore the mechanism that identifies initializing stores, e.g. the Allocation Range Cache, is not latency sensitive and could be implemented as a small, fully-associative cache of base-bound pairs. This structure could effectively reduce bus bandwidth requirements at a minimal implementation cost. We now propose an integration of the Allocation Range Cache that can effectively identify and smash initializing store misses.

7.1. Smashing Invalid Memory Traffic

Figure 15 demonstrates a conceptual example of how an Allocation Range Cache and Initializing Store Table can be integrated into a typical cache hierarchy to smash invalid memory traffic. The identification of an initializing store in the Allocation Range Cache is accomplished using the *virtual address* of store instructions. When a store is presented to the cache hierarchy, the translation look-aside buffer (TLB) and Allocation Range Cache (ARC) are accessed in parallel. The TLB translates the store address tag from virtual to physical, sends the tag to the Level-1 cache for tag comparison, and also sends the physical tag to the ARC. Meanwhile the ARC uses the virtual store address to reference into its base-bound pairs to determine if the store is initializing, as described in Figure 12. If the store is identified as an initializing store to heap space, the Allocation Range Cache takes the physical tag (supplied by the TLB) and inserts the complete physical address of the store instruction into the Initializing Store Table.

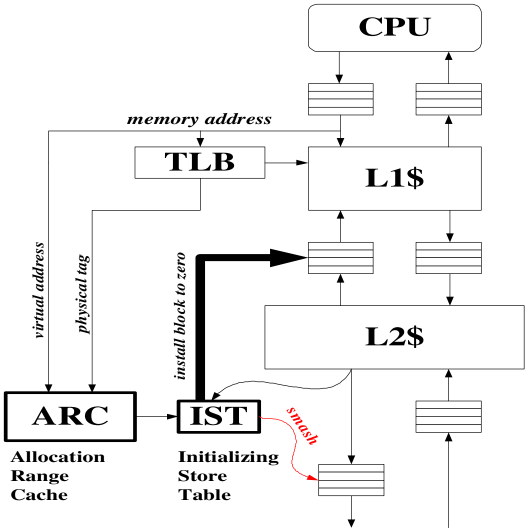


Figure 15. Integration of Allocation Range Cache

If a store address misses in the Level-1 and Level-2 caches, and at least one cache employs a write-allocate policy, a data fetch request is queued in the outgoing memory request queue. The address is also sent to the Initializing Store Table (IST). The IST performs a fully-associative search for a matching physical address. A match implies this store has been identified as an initializing store by the Allocation Range Cache. Since initializations are tracked on cache block granularity, we know that the entire cache block encompassing an initializing store address contains invalid data. Therefore we can install the entire block directly into cache and avoid fetching the data from memory. To accomplish this, the Initializing Store Table invalidates (smashes) the store address entry in the outgoing memory request queue and sends a response to the Level-1 cache queue, or whichever cache allocates on writes, to install the cache block with the value zero. Finally, the store address is removed from the Initializing Store Table. This demonstrates how the Allocation Range Cache can smash invalid memory traffic using cache installation.

7.2. Alternative Implementations

As discussed in Section 3, there are other methods for avoiding invalid memory traffic: sub-blocking and software-controlled cache installation. Sub-blocking has obvious limitations. First, sub-block valid bits cause significant storage overhead, especially in systems that allow un-aligned word writes or byte writes. In practice, fetch-on-write must be provided for un-aligned word writes. Second, sub-blocking requires that lower levels in the memory system support writes of partial cache lines. This can become a significant problem in a multi-processor environment with coherent caches, since the owner of a line may possess only a partially valid line, and cannot respond directly to the requestor.

Software-controlled cache installation (on a page granularity) can be accomplished by an operating system's page fault handler. When a mapping is created for a new page, the operating system can issue a cache installation (e.g. `dcbz`) for the entire page. This will install the entire page directly into cache, effectively prefetching all initialization misses to that page. However, this scheme can cause excessive cache pollution, e.g. given a 64 byte block size, 64 valid blocks could be evicted when a 4KB page is installed. This problem gets worse when the page size grows, as in the presence of superpages [13]. Given page sizes of 4MB or 16MB, directly installing an entire page into cache is not feasible. Page-granular installing is inefficient for large striding initialization patterns and this scheme cannot optimize capacity miss initializations to pages that have already been mapped. If heap space is reused, initializing store misses will occur if that heap space has fallen out of cache.

Tracking and eliminating initializing store miss data transfers at the cache block granularity can alleviate subblocking overhead and avoid excessive cache pollution from page-granular cache installation. We now evaluate the performance benefits of smashing invalid memory traffic via cache installation.

7.3. Performance Speedup via Cache Installation

Figure 16 presents performance results for smashing initializing store misses via cache installation by an Allocation Range Cache. This structure triggers cache block-granular installation instructions (`dcbz`) when an initializing store miss is identified. The entire cache block is installed directly into the Level-1 data cache, thus performing a zero-latency prefetch. The store instruction will now hit in cache. Note that coherence permission must be received before installing a cache block. The performance of a page-granular installation scheme (Page) as performed by the AIX page fault handler is compared against our block-granular scheme (Block). Results are reported relative to a baseline machine configuration (Base) as described in Section 4.1. The `dcbz` cache installation instruction is disabled in this baseline. For most programs, smashing invalid memory traffic results in a direct performance improvement. In bzip2, gap, mcf, parser, and perlbmk using the Allocation Range Cache to trigger block-granular cache installations outperforms the page-granular installation scheme. Figure 6 shows that mcf and gap have the largest percentage of compulsory misses that are initialization misses, 95% and 92% respectively. Figure 16 demonstrates that avoiding these compulsory misses can have significant performance benefits.

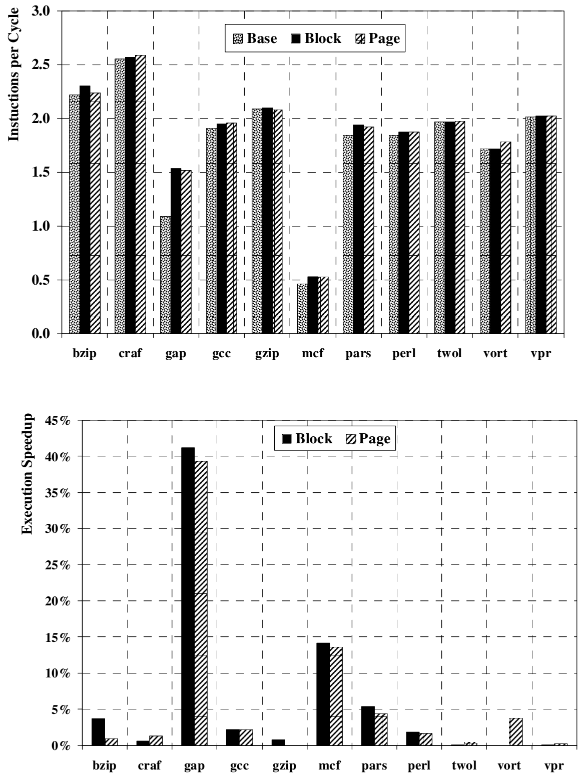


Figure 16. Performance speedup via block- and page-granular cache installation

Instructions per cycle (IPC) comparisons for page-granular (Page) and block-granular (Block) cache installation schemes using `dcbz` are shown on the top. Execution speedups are presented on the bottom graph. All programs were simulated for one billion instructions.

Bzip2 and gzip exhibits striding initialization patterns with observed strides of 1024 bytes as discussed in Section 6.3.2. With this large stride, a new 4KB page is encountered every fourth stride. From Figure 16, installing the entire 4KB page after the first initialization is causing significant cache pollution since block-granular installations provide larger performance gains. The Allocation Range Cache does not excessively pollute the cache with extraneous prefetching. Rather, blocks are installed on demand, eliminating cache pollution effects for striding initializations.

8. Conclusion

This paper introduces the concept of *invalid memory traffic* - real data traffic that transfers invalid data. Such traffic arises from fetching uninitialized heap data on cache misses. We find that initializing store misses are responsible for approximately 23% of all cache miss activity across the SPEC CINT2000 benchmarks for a 2MB cache. By smashing invalid memory traffic, 35% of compulsory misses and 23% of all cache miss data traffic on the bus can be avoided. This is an encouraging result, since compulsory misses, unlike capacity and conflict misses, cannot be eliminated by improvements in cache locality, replacement policy, size, or associativity. Eliminating invalid compulsory miss traffic breaks the infinite cache limit, where compulsory misses of a finite-sized cache are finite and bound by that of an infinite-sized cache [6].

We propose a hardware mechanism, the Allocation Range Cache, that tracks initialization of dynamic memory allocation regions on a cache block granularity. By maintaining multiple base-bound representations of an allocation range (interleaving), this structure can identify nearly 100% of all initializing store misses with minimal storage overhead. By directly allocating and initializing a block into cache (cache installing) when an initializing store miss is identified, it is possible to avoid transferring invalid memory over the bus. This is essentially a zero-latency prefetch of a cache miss. Reducing bus traffic via cache installation can directly improve performance by reducing pressure on store queues and cache hierarchies. We quantify a direct performance improvement from avoiding initialization misses to the heap. Speedups of up to 41% can be achieved by smashing invalid memory traffic with the Allocation Range Cache triggering cache block installations. Indirectly, smashing invalid memory traffic will decrease bus bandwidth requirements, enabling bandwidth-hungry performance optimizations such as prefetching and multi-threading to consume more bandwidth and improve performance even further.

9. Future Work

There are issues to be addressed for avoiding invalid memory traffic in a multi-processor environment, including coherence of the Allocation Range Cache. For correctness, all ARC entries must be coherent across multiple threads or processors. The ARC can be kept coherent among multiple threads in the same address space by architecting the cache entries as part of coherent physical memory. Thus updates to an ARC entry by one thread will be seen by other threads through the existing coherence mechanisms. Coherence is more challenging when virtual address aliasing to shared physical memory exists. These issues are subject of continued research.

10. References

[1] AIX Version 4.3 Base Operating System and Extensions Technical Reference, Volume 1, *http://www.unet.univie.ac.at/aix/libs/basetrf1/malloc.htm*

[2] Barrett David A., Zorn, Benjamin G. *Using lifetime predictors to improve memory allocation performance*. ACM SIGPLAN Notices, v.28 n.6, p.187-196, June 1993.

[3] Burger, D., Goodman, J.R., Kägi, A. *Memory Bandwidth Limitations of Future Microprocessors.* Proceeding of the 23rd Annual International Symposium on Computer Architecture, pages 78-89, PA, USA, May 1996.

[4] Chen, T.-J., Baer, J.-L. *Reducing Memory Latency via Non-blocking and Prefetching Caches*. Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 51-61, Boston, MA, October, 1992.

[5] Chen, T.-J., Baer, J.-L. *A Performance Study of Software and Hardware Data Prefetching Schemes.* Proceedings of the 21st annual International Symposium on Computer Architecture, pp. 223 - 232 Chicago, IL, 1994.

[6] Cragon, H.G. Memory Systems and Pipelined Processors. Jones and Bartlett Publishers, Inc., Sudbury, ME, 1996.

[7] Diwan, A., Tarditi, D., Moss, E. *Memory System Performance of Programs with Intensive Heap Allocation*. ACM Transactions on Computer Systems, Vol13, No 3, pp. 244-273, August 1995.

[8] Dubois, M., Skeppstedt, J., Ricciulli, L., Ramamurthy, K., Stenström, P. *The Detection and Elimination of Useless Misses in Multiprocessors.* Proceedings of the 20th Annual International Symposium on Computer Architecture, pp. 88-97, May 1993.

[9] Gonzalez, J., Gonzales, A. *Speculative execution via address prediction and data prefetching*. Proceed-ings of the 11th International Conference on Supercomputing, pp. 196-203, June 1997. [10]

[10] Grunwald, D., Zorn, B., Henderson, R. *Improving the Cache Locality of Memory Allocation*. ACM SIGPLAN PLDI'93, pp. 177-186, Albuquerque, N.M., June 1993.

[11] IBM Microelectronics, Motorola Corporation. PowerPC Microprocessor Family: The Programming Environments. Motorola, Inc., 1994.

[12] Jouppi, Norman P. *Cache write policies and performance*. ACM SIGARCH Computer Architecture News, v.21 n.2, p.191-201, May 1993.

[13] Talluri, M., Hill, Mark D. *Surpassing the TLB performance of superpages with less operating system support*. ACM SIGPLAN Notices, v.29 n.11, p.171-182, Nov. 1994.

[14] Peng, C.J., Sohi, G. *Cache memory design considerations to support languages with dynamic heap allocation*. Technical Report 860, University of Wisconsin-Madison, Dept. of Computer Science, July 1989.

[15] Rosenblum, M., Herrod, S., Witchel, E., Gupta, A. *Complete Computer Simulation: The SimOS Approach.* IEEE Parallel and Distributed Technology, Fall 1995.

[16] Rosenblum, M., Bugnion, E., Devine, S., Herrod, S. *Using the SimOS Machine Simulator to Study Complex Computer Systems.* ACM Transactions on Modeling and Computer Simulation, vol. 7, no. 1, pp.78-103, January 1997.

[17] Saulsbury, A., Pong, F., Nowatzyk, A. *Missing the Memory Wall: The Case for Processor/Memory Integration.* Proceedings of the 23rd Annual International Symposium on Computer Architecture, pages 90-101, PA, USA, May 1996.

[18] Seidl, Matthew L., Zorn, Benjamin G. *Segregating heap objects by reference behavior and lifetime*. ACM SIGPLAN Notices, v.33 n.11, p.12-23, Nov. 1998.

[19] Tullsen, D.M., Eggers, S.J. *Limitation of cache prefetching on a bus-based multiprocessor.* Proceedings of the 20th Annual International Symposium on Computer Architecture, 1993.

[20] Wulf, Wm.A. and McKee, S.A. *Hitting the Memory Wall: Implications of the Obvious.* ACM Computer Architecture News. Vol. 23, No.1 March 1995.