# Eager Writeback - a Technique for Improving Bandwidth Utilization

Hsien-Hsin S. Lee† Gary S. Tyson† Matthew K. Farrens‡

†ACAL, EECS Department
University of Michigan
Ann Arbor, MI 48109
{*linear, tyson*}*@eecs.umich.edu*

‡Department of Computer Science
University of California
Davis, CA 95616
*farrens@cs.ucdavis.edu*

**Abstract**

Modern high-performance processors utilize multi-level cache structures to help tolerate the increasing latency of main memory. Most of these caches employ either a *writeback* or a *write-through* strategy to deal with store operations. Write-through caches propagate data to more distant memory levels at the time each store occurs, which requires a very large bandwidth between the memory hierarchy levels. Writeback caches can significantly reduce the bandwidth requirements between caches and memory by marking cache lines as *dirty* when stores are processed and writing those lines to the memory system only when that dirty line is evicted. Unfortunately, for applications that experience significant numbers of cache misses due to streaming data, writeback cache designs can degrade overall system performance by clustering bus activity when dirty lines contend with data being fetched into the cache.

In this paper we present a new technique called *Eager Writeback*, which re-distributes and balances memory traffic by writing and "cleaning" dirty cache lines prior to their eviction. Eager Writeback can be viewed as a compromise between write-through and writeback policies, in which dirty lines are written later than write-through, but prior to writeback. We will show that this approach can reduce the large number of writes seen in a write-through design, while avoiding the performance degradation caused by clustering bus traffic in a writeback approach.

## I. INTRODUCTION

Caches are very effective at reducing memory bus traffic by intercepting and handling most of the read requests generated by the processor. However, caches must deal with both reads *and* writes to memory. Support for writes (stores) tends to be simple – on a store the data item is either written into both the cache and through the cache hierarchy to the memory (referred to as a *write-through* policy), or it is written into the cache exclusively and the data item is written out to memory only when the cache line is evicted (known as a *writeback* policy.)

Caches employing a write-through policy generate memory traffic every time a store occurs in the program. Since it would largely defeat the purpose of having a cache if the processor had to block on each store until the write completed, write-through caches use a structure known as a *store buffer* or *write buffer* [10] to buffer writes to memory. Whenever a write occurs, the data item is written into both the cache and this structure, allowing the processor to continue executing without blocking (until the store buffer becomes full). The store buffer will send its contents to memory as soon as the bus is idle.

Writeback caches, on the other hand, generate memory traffic much less frequently. When a store occurs in a writeback cache the data value is written into the corresponding line in the cache, which is then marked *dirty*. Writes to memory occur only when a line marked *dirty* is evicted from the cache (usually due to a cache miss) in order to make room for the incoming data item.

Whenever there are many consecutive misses (caused by context switches, or working set changes, or by certain graphics algorithms and applications, for example) the writeback cache can find itself blocked waiting for a dirty line to be written to memory. This is the same problem faced by the write-through cache, and can be dealt with in much the same manner by adding a writeback buffer. However, there are certain classes of programs which suffer from memory delay penalties that even a large writeback buffer cannot eliminate. For example, many newer applications (e.g. 3D graphics or multimedia) have enormous incoming data streams. In these programs, the stream of incoming data items can cause many conflict cache misses and trigger the eviction of many dirty lines. This dirty writeback traffic must compete for available memory bandwidth with the arriving data, and often impedes the delivery of the data to the processor. For programs where overall performance is bound by memory bandwidth, this competition for bandwidth can have a substantial negative impact.

In this paper we propose a modification to the writeback policy which spreads out memory activity by selectively writing some dirty lines to memory whenever the bus is free, instead of waiting until that line in the cache is replaced. This early writing of dirty lines to the memory system reduces the potential impact of bursty reference streams, and can effectively re-distribute and balance the memory bandwidth and thereby improve system performance.

## II. BACKGROUND

As discussed in the introduction, caches that employ a writeback policy reduce memory traffic by delaying the transfer of data to memory as long as possible. Many modern microprocessors using a writeback cache policy incorporate a writeback (or cast-out) buffer, which is used as temporary storage space for holding dirty cache lines while the data request that caused the eviction is serviced. Upon eviction, a dirty cache line is deposited into the writeback buffer, which usually has the highest bus scheduling priority among all types of non-read bus transactions. Once the writeback buffer fills up, subsequent dirty line replacements cannot take place. As a result, their corresponding data demand fetch operations cannot be committed into the cache, and the processor pipeline stalls waiting for the dependent data.

It is possible to alleviate this problem somewhat by using existing cache hardware. *Non-blocking caches* have been proposed by Kroft [7] which use a set of *miss status holding registers* (MSHRs) to manage several outstanding cache misses. When a cache miss occurs in a non-blocking cache, it is allocated an empty MSHR entry. Once the MSHR entry is allocated, processor execution can continue. If none of the MSHRs are available (i.e. a structural hazard [6] exists due to resource conflicts), the processor will have to block until an MSHR entry becomes free.

By adding data fields to the MSHRs, it would be possible to use them to temporarily store returning cache lines. This would allow fetched data to be immediately forwarded to the appropriate destination registers, and help overcome the situation where the cache cannot be written to because the writeback buffer is full. However, this scenario delays MSHR deallocation and can lead to processor stalls on a cache miss if there are no free MSHRs.

In addition, in a modern computer system memory bandwidth is not exclusively dedicated to the host processor. There are often multiple agents on the bus (such as graphics accelerators or multiple processors) issuing requests to memory over a short period of time. In a contemporary multimedia PC platform with an Accelerated Graphics Port (AGP) interface running a graphics-centric application, for example, the graphics accelerator shares system memory bandwidth with the host processor in order to retrieve graphics commands and texture maps from the system memory. A typical system architecture of a contemporary multimedia PC system is illustrated in Figure 1.

In a common 3D graphics application, the processor reads instructions and triangle vertices, performs the specified computations, and then stores them with rendering state commands back into AGP memory space. The graphics accelerator then reads these commands out of AGP memory for rasterization. In addition to the command traffic, the graphics accelerator also reads a large amount of texture data (which constitutes the major portion of AGP traffic on the bus). These textures are mapped onto polygon surfaces to increase the visual realism of computer-generated images.

Current cache designs have difficulty in efficiently managing the flow of data in and out of the cache hierarchy in these data intensive applications. Buffering techniques, including write buffers and MSHRs can help, but do not alleviate the problems of clustering bus traffic caused by writeback data. In the next section we introduce a new technique designed to distribute the writes of dirty blocks to times when the bus is idle.

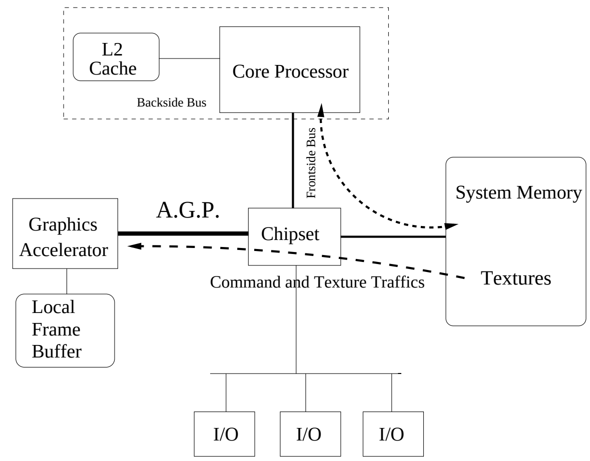


**Figure 1: A Multimedia PC Architecture.**

## III. EAGER WRITEBACK

### 1 Overview

To address the performance drawbacks of a conventional writeback policy, we are proposing a new technique called *Eager Writeback*. The fundamental idea behind Eager Writeback is to write dirty cache lines to the next level of the memory hierarchy and clear their dirty bits earlier than in a conventional writeback cache design, in order to better distribute bandwidth utilization and alleviate memory bus congestion. If dirty cache lines are written to memory when the bus is less congested, there will be fewer dirty lines that require eviction during peak memory activity.

In essence, we are speculating that certain dirty lines will not be re-written before eviction and thus there is no need to wait until eviction time to perform the cache line write. An Eager Writeback will never impact the correctness of the architectural state even if the operation that triggers it was wrongly speculated - if our speculation is incorrect and we write too often, we approach the limiting case of write-through cache behavior. If we do not speculate often enough, we approach writeback cache behavior. However, in either case we do not violate any correctness constraints.

This work is similar in spirit to that of Lai and Falsafi [8], in which they identify cache lines in a shared memory system that can be speculatively self-invalidated in order to hide the invalidation time and reduce the coherence overhead. However, we are applying the idea to uniprocessor caches instead of DSM machines, which enables us to use a far simpler mechanism to identify which lines should be speculatively written out.

In order to select the best "trigger" to cause an eager writeback, we examined the probability of rewriting a dirty line in a set-associative cache when it was in a given state (MRU through LRU) for the well-known SPEC95 benchmarks [5] and four applications from the lesser-known X benchmark suite [9]. The X benchmark suite consists of four applications representing different graphics algorithms based on X Windows. X-DOOM, a popular video game, uses a polygon-based rendering algorithm. POV-ray is a

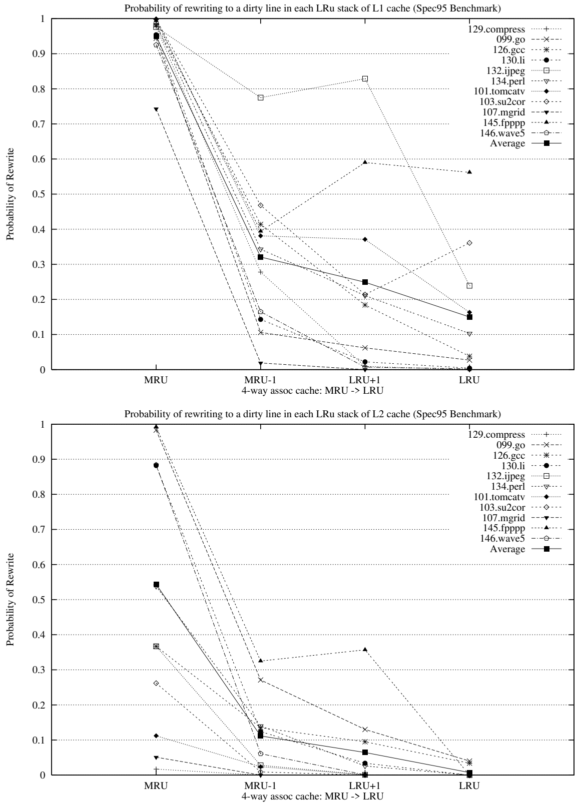


**Figure 2: Probability of writing to a dirty line in each LRU stack of L1 and L2 caches (SPEC95)**

public domain ray tracing package developed for generating photo-realistic images on a computer. xlock, a popular screen saver, renders a 3D polygonal object on the screen. The final application is an animation viewer which processes an MPEG-1 data stream to display an animated sequence.

Our results indicate that cache lines that have been marked dirty and reach the LRU (Least Recently Used) state in a 4-way set-associative data cache are rarely written to again before they are evicted. In Figure 2 and Figure 3, we show the probability of a line that was marked dirty being written to again as it moves from the MRU (Most Recently Used) state to the LRU state for both L1 and L2 caches. The cache configurations are described in Table 1. The graph on the top in Figure 2, for example, shows that in the L1 cache the average probability (the solid line) of a dirty line in the LRU state being re-written is 0.15, while the similar probability for a dirty line in the MRU state is 0.95. The probabilities of re-dirtying lines in the LRU state are even lower in the L2 cache - in fact, close to 0 as shown in the graphs on bottom of Figure 2 and Figure 3.

These figures indicate there are some programs (such as *fpppp* and *su2cor*) that have a fairly high probability of writing to dirty lines after they have entered the LRU state, however. In order to further evaluate these cases, we looked at the ratio of the number of times a dirty line in the LRU state is written to, normalized to the number of times a dirty line in the MRU state is written to. The results are presented in Figure 4, which shows that while

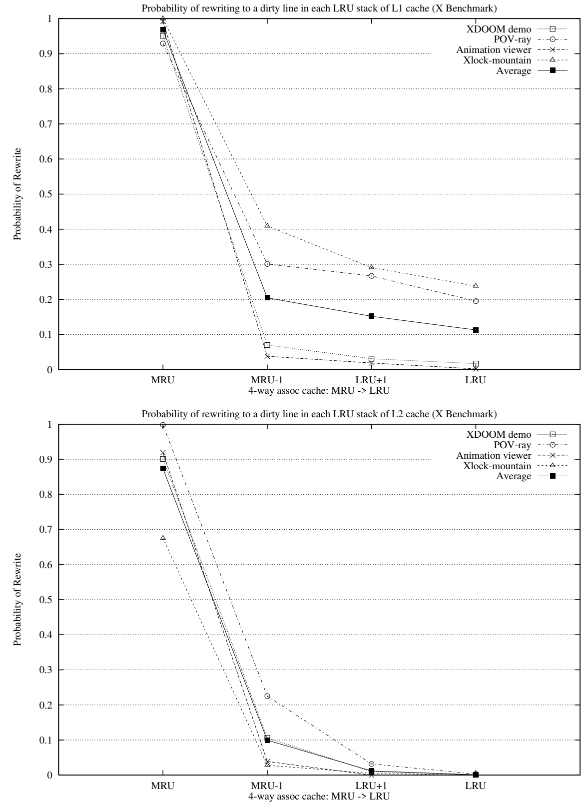


**Figure 3: Probability of writing to a dirty line in each LRU stack of L1 and L2 caches (X benchmark)**

the probabilities may be high, the actual number of these occurrences is negligible compared to the rewriting that occurs when a line is in other states (MRU, MRU-1, etc.). These trends held across a wide range of cache configurations, and imply that once a line enters the LRU state it becomes a prime candidate for Eager Writeback, since it has a very low occurrence of being written to again.

## 2 Design Issues in Eager Writeback Caches

There can be many different approaches to deciding when to trigger an Eager Writeback. As was shown in the previous section, one obvious candidate is to use the transition of a dirty line into the LRU state as a trigger point for an Eager Writeback. For example, when a cache set is being accessed and its corresponding LRU bit is being updated,

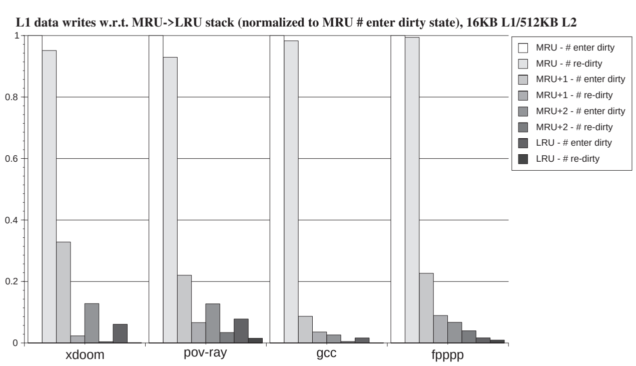


**Figure 4: Normalized number of writes and rewrites to a dirty line in each MRU-LRU state**

the line can be checked to see if it is marked dirty. If it is, then a dirty writeback can be scheduled, and the dirty bit can be reset.

If the writeback buffer is full at this point, two approaches can be considered; (a) simply abort the Eager Writeback - the actual dirty writeback will take place later when the line is evicted, or (b) perform the eager writeback when an entry in the writeback buffer becomes free. This provides the ability to perform eager writeback anytime between when a line is marked LRU and when it is evicted.

To provide this capability using a minimum of hardware, we chose to implement an *Eager Queue* which holds attempted eager writebacks which were unable to acquire writeback buffer entries. Whenever an entry in the writeback buffer becomes available, the Eager Queue checks the cache set on the top of the queue to see if the dirty bit of the LRU line in the indexed set is set. If it is, the line is moved into the writeback buffer.

An alternate implementation considered during this research was *Autonomous Eager Writeback*. This implementation used a small independent state machine which autonomously polled each cache set in round-robin fashion and checked the dirty bit of its LRU line, initiating eager writeback on those lines when the writeback buffer was not full. Whether eager queues or the autonomous state machine is more feasible is highly dependent on the processor and cache organization. For this study we present results for the more conservative approach which used eager queues.

## IV. SIMULATION FRAMEWORK

Our simulation environment was based on the SimpleScalar tool set [1], a re-targetable execution-driven simulator which models speculative and out-of-order execution. The machine employs a Register Update Unit (RUU), which combines the functions of the reservation stations and the reorder buffer necessary for supporting out-of-order execution [11]. Separate address and data buses were implemented and their contentions were all modeled appropriately. Writeback buffers were implemented between cache hierarchies. All the binaries used were compiled using the SimpleScalar GCC compiler that generates code in the portable ISA (PISA) format.

The microarchitectural parameters used in our baseline processor model are shown in Table 1. Table 2 lists the latencies of each functional unit modeled in the simulation. A non-blocking cache structure, writeback buffer and eager queue associated with each cache level were added to the simulator for this study. The number of entries in each buffer was re-configurable from 1 to 256, and varied from simulation to simulation.

A pseudo-Rambus DRAM model was used in the external memory system. This single-channel RDRAM with 64 dependent banks can address up to 2GB of system memory. In the model, 32 independent banks can be accessed simultaneously (contiguous banks share the same sense amplifier for driving data out of the RAM cells). Row control packets, column control packets and data packets can be pipelined and use separate busses. RDRAM address remapping [4] was modeled to reduce the rate of bank interference. The peak bandwidth that can be reached in our RDRAM model is 1.6GB/sec.

A simplified uncacheable write-combining (or write-coalescing) memory [2][3] was implemented as well for the purpose of correctly simulating our benchmark behavior. Whenever a data write to an uncacheable region results in an L1 cache miss, the write operation will immediately request access to the bus and drive data out to the system memory directly (skipping a next-level cache look-up). Only cache line writes are modeled - any partial cache line update will be treated as a full cache line write in the simulator.

For modeling multiple agents on the memory bus, a memory traffic injector was also implemented. This injector allowed us to imitate the extra bandwidth consumed by other bus agents by configurable periodic injections of data streams onto the memory bus.

### 1 Benchmarks

In order to evaluate the effectiveness of the Eager Writeback technique, we ran extensive simulations on the SPEC benchmark suite and two programs representative of future graphics and streaming applications. Concentrating the analysis on these small, representative kernels enables us to illustrate the potential benefits of our scheme in far greater detail than can be achieved running an entire application.

The first of these kernels is a 3D geometry processing pipeline (*mini-geometry*), which is present in most triangle-based rendering algorithms [13]. Two different geometric rendering configurations were simulated, one which was very simple (i.e. ambient light with no external light sources), and one which included multiple diffuse light sources. The ambient light configuration reduces the computational requirements of the algorithm in order to maximize frame rate at the expense of image realism[1]. The multiple light source configuration increases the computational demands, reducing the relative impact of bus utilization as the processor spends more time processing between each data element request.

The second kernel represents the class of streaming data algorithms which work with large data sets. This kernel

[1]This would be preferred in real-time 3D applications (e.g. Doom or Quake) when processor performance is lacking.

| Processor Parameters | Cycles in Processor Clocks |
|---|---|
| 1st Level I- and D-Cache | 3 clks, thruput = 1 clk |
| 2nd Level Cache | 18 clks, thruput = 10 clks |
| I- and D-TLBs | 2 clks, thruput = 1 clk |
| Backside bus arbitration | 4 clks |
| Frontside bus arbitration | 10 clks |
| RDRAM Trcd, RAS-to-CAS | 20 clks |
| RDRAM Tcac, CAS-to-data return | 20 clks |
| RDRAM Trp, Row Precharge | 20 clks |
| INT ALU latency/thruput | 1 / 1 |
| INT multiplier latency/thruput | 3 / 1 |
| INT divider latency/thruput | 20 / 19 |
| FP ALU latency/thruput | 2 / 1 |
| FP multiplier latency/thruput | 4 / 1 |
| FP divider latency/thruput | 12 / 12 |

**Table 2: Latency Table (in core clocks) of Functional Units in the Baseline Processor.**

| Processor Architectural Parameters | Specifications |
|---|---|
| **Core frequency** | 1 GHz |
| **1st Level I-Cache** | 2-way 16KB, virtual-index physical-tag |
| **1st Level D-Cache** | 4-way 16KB, virtual-index physical-tag |
| **2nd Level Cache** | Unified, 4-way 512KB, physical-index physical-tag |
| **Cache line size** | 32 bytes |
| **I- and D-TLBs** | 2-way 8KB |
| **Backside bus** | 500 MHz (half-speed), 8B wide |
| **Frontside bus** | 200 MHz, 8B wide |
| **Memory model** | Rambus DRAM (peak: 1.6 GB/s) |
| **Branch predictor** | 2-level adaptive, 10-bit gshare |
| **Instr. fetch/decode/issue/commit width** | 8 / 8 / 8 / 8 |
| **Load/Store Queue size** | 32 |
| **Register update unit size** | 64 |
| **Memory port size** | 2 |
| **INT/FP ALU size** | 4 / 4 |
| **INT/FP MULT/DIV size** | 1 / 1 |

**Table 1: Summary of the Baseline Processor Model.**

```
MINI-GEOMETRY()
 while ( frames )
 for ( objects in each frame )
     for ( every 4 vertices )
         /* Transformation */
         tx = m11 * InV[]_x + m21 * InV[]_y + m31 * InV[]_z + m41;
         ty = m12 * InV[]_x + m22 * InV[]_y + m32 * InV[]_z + m42;
         tz = m13 * InV[]_x + m23 * InV[]_y + m33 * InV[]_z + m43;
         w = m14 * InV[]_x + m24 * InV[]_y + m34 * InV[]_z + m44;
         OutV[]_rw = 1/w;
         OutV[]_tx = X_offset + tx * OutV[]_rw;
         OutV[]_ty = Y_offset + ty * OutV[]_rw;
         OutV[]_tz = tz * OutV[]_rw;
         /* Texture coordinates copying */
         OutV[]_tu = InV[]_u;
         OutV[]_tv = InV[]_v;
         /* Lighting Loop */
         ID_r = ID_g = ID_b = 0.0;
         for ( every light source )
             dot = LDir[]_x * InV[]_nx + LDir[]_y * InV[]_ny + LDir[]_z * InV[]_nz;
             ID_r = ID_r + Ambient_r + Diffuse_r * dot;
             ID_g = ID_g + Ambient_g + Diffuse_g * dot;
             ID_b = ID_b + Ambient_b + Diffuse_b * dot;
             OutV[]_cd = ((int)ID_r << 24)|((int)ID_g << 16)|((int)ID_b << 8|α);

     /* Device driver loop */
     for ( each transformed and lit vertex )
         /* Assume Tri-Strip triangles */
         /* Copy entire OutV records to graphics AGP memory */
         GfxCommand[vertex - 2] = OutV[vertex - 2];
         if (even - numberedvertex)
            GfxCommand[vertex] = OutV[vertex];
            GfxCommand[vertex - 1] = OutV[vertex - 1];
         else
            GfxCommand[vertex - 1] = OutV[vertex - 1];
            GfxCommand[vertex] = OutV[vertex];
```

**Figure 5: Algorithm of the mini-geometry pipeline**

processes a large array of data, performing both reads and writes, generating frequent cache misses as well as many dirty writebacks (behavior common to many current streaming applications). In a real program the data would have some computation performed upon it – however, in order to highlight the behavior of the memory system in this kernel no actual computations are performed.

### 1.1 Mini-Geometry Pipeline

The 3D geometry processing pipeline, shown in Figure 5, is representative of a very frequently used algorithm in most triangle-based rendering engines. Geometry processing, consisting of intensive floating-point operations on a large quantity of data (mostly vertices that describe the geometry of each 3D object), is mainly performed by the processor. It is one of the two key portions of a three-dimensional graphics rendering pipeline (the other portion being rasterization, which is typically performed by a dedicated graphics accelerator nowadays).

This kernel consists of three nested loops wrapped by two outer loops which iterate through frames and 3D objects in the world space. The first innermost loop processes vertices for each 3D object assuming the entire object is modeled by a single triangle strip. The basic functions performed inside this loop are *transformation*, *lighting*, and *rendering command output*.

The *transformation* function projects the new location of each vertex on screen through a 4x4 matrix multiplication and a viewport transformation. The *lighting* function calculates the interaction of each vertex with light sources and generates the color intensity for each vertex. This calculation involves a dot product between the light direction vector and the vertex normal vector using a Phong illumination model [12]. A single parallel light source with diffuse only components is assumed in the lighting model. For a parallel light source, per-vertex normal transformations can be replaced by an inverse transformation of the light source location on a per-scene basis, thus eliminating a large number of computations. A color packing conversion then packs four single-precision floating-point RGBA color intensities into a packed 4-byte integer. (We assume the machine ISA supports four wide SIMD computation).

After finishing with all the vertices in one object, a loop imitating the functionality of a device driver is invoked (the *command output* function). This driver loop breaks one triangle strip into individual triangles and copies these transformed and lit vertices to the uncacheable graphics memory.

### 1.2 Streaming Kernel

The *Streaming* kernel is presented in Figure 6, and consists of three inner loops that exercise the L2 cache. The first loop writes data into $array_A$. The second loop reads data from $array_A$, performs some floating-point computation and passes the results to inner loop invariant array elements. Finally the third loop accesses a new array ($array_B$), displacing elements of $array_A$ from the cache.

This program is designed to represent the typical behavior of many streaming applications. However, as pointed out previously, in order to highlight the interaction of Eager Writeback and the memory system in this uniprocessor, no actual computational work is performed per data read.

```
STREAMING()
  float array_A[MAX], array_B[MAX];
  for (m = 0; m < loop; m + +)
      for ( array_A[i] ∈ each set of L2 cache )
          write array_A[i] to way 0;
          write array_A[i + 1 * 8 * set_size] to way 1;
          write array_A[i + 2 * 8 * set_size] to way 2;
          write array_A[i + 3 * 8 * set_size] to way 3;
      for ( array_A[j] ∈ each cache line in L2 cache )
          read array_A[j];
          compute array_A[j];
          write array_A[m];
      for ( array_B[k] ∈ each set of L2 cache )
          read array_B[k] into way 0;
          read array_B[k + 1 * 8 * set_size] into way 1;
          read array_B[k + 2 * 8 * set_size] into way 2;
          read array_B[k + 3 * 8 * set_size] into way 3;
          write array_A a[m];
```

**Figure 6: Algorithm of the Streaming Kernel**

## V. SIMULATION RESULTS AND ANALYSIS

For each kernel studied, we present two different data sets, one with no memory contention from other potential bus agents, and one with artificially injected memory traffic.

### 1 Spec95 Benchmarks

Table 3 shows the simulation results for the SPEC95 benchmark suite using 3 configurations - Baseline, Eager and Free Writeback. The Baseline case uses a single entry writeback buffer, while Free Writeback models a system in which dirty writebacks do not generate any memory traffic on the bus (thus serving as an upper bound on performance.)

Looking at the table it is apparent that there is little performance gain possible for the programs in this suite, since the difference in the cycle count between the baseline case and the upper bound is negligible. This is not surprising, since it is well-known that the SPEC95 benchmark suite is not a good candidate for memory system performance studies primarily due to its small working set size. For the rest of this study we will focus on the benchmarks that more aggressively exercise the memory system, and are arguably more representative of future workloads.

### 2 Analysis of Mini-Geometry Pipeline

#### *2.1 Without Injected Memory Traffic*

Table 4 contains the number of mini-geometry pipeline execution cycles for a variety of memory configurations. In this table, each row represents a different combination of writeback buffer size and lighting conditions, while the columns contain different writeback strategy cycle counts. The first column, *Baseline*, shows the cycle count using a conventional writeback policy. The next 6 columns contain the results for 3 different variations of the *Eager Writeback* scheme and the speed-up of each scheme over the baseline case, with each scheme identified by the size of its *Eager Queue* (EQ). The simplest design choice is EQ=0, in which Eager Writebacks are dismissed if the writeback buffer is full. The other two cases can queue up attempted eager writebacks within Eager Queues of specified sizes. The rightmost column contains the Free Writeback case, which as stated earlier is the upper bound to available performance.

There are several things of interest to note in this table. Perhaps most significantly, it can be seen that increasing the depth of the writeback buffer has virtually no impact

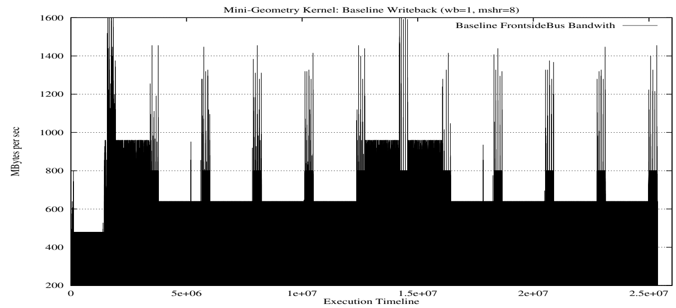


**Figure 7: Memory Bandwidth Profile by *Baseline Writeback* for Mini-Geometry Pipeline (No light)**

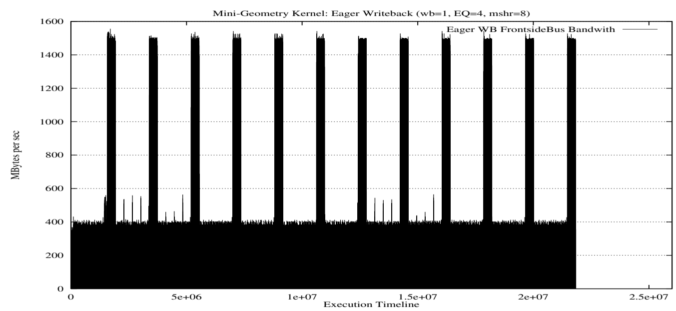


**Figure 8: Memory Bandwidth Profile by *Eager Writeback* for Mini-Geometry Pipeline (No light)**

on the performance of the Baseline case. In fact, going from 1 to 256 entries in the writeback buffer only improves performance by 0.17%. This is because a large number of dirty writebacks are competing for bandwidth with the demand fetches, and the bus congestion can not be alleviated by a deeper writeback buffer.

On the other hand, adding Eager Writeback scheme increases the performance of the system by 4.9% to 16.2% (depending on the light sources and the depth of the Eager Queue). For the simplest case of no Eager Queue and a single entry writeback buffer, the speedup ranges from 6.2% (for no light source) to 4.9% (with 3 light sources). This speedup is smaller than for the other cases, because many eager writebacks are dropped due to the lack of space in the writeback buffer. When the Eager Queue size is increased (or the number of writeback buffer entries is increased), the speedup achieved approaches the upper bound.

The "bandwidth shifting" effect is quite apparent in Figure 7 and Figure 8. These two figures present the utilization profile of memory bandwidth requested by the processor using the Baseline (Figure 7) and Eager Writeback (Figure 8) configurations, running the mini-geometry pipeline. The y-axis plots the instantaneous bandwidth versus the execution timeline on the x-axis, which was calculated by sampling the data phase on the memory bus every 2000 core clocks (e.g. if 1600 bytes are seen on the bus in 2000 core cycle period, its instantaneous bandwidth is 800MB/sec for a 1GHz processor).

The 12 broad spikes that saturate the peak RDRAM bandwidth in Figure 8 occur within the driver loop, where ren-

| *sim cycle* | *Baseline* | *Eager* | | *Free Writeback* | |
|---|---|---|---|---|---|
| benchmark | cycles | cycles | speedup | cycles | speedup |
| go | 4106741898 | 4106316586 | 1.000 | 4105050891 | 1.000 |
| gcc | 1425690611 | 1423578223 | 1.001 | 1419981686 | 1.004 |
| li | 401639628 | 401635232 | 1.000 | 401481752 | 1.000 |
| ijpeg | 2125521487 | 2123322070 | 1.001 | 2117908634 | 1.004 |
| perl | 3705579465 | 3701065936 | 1.001 | 3683430936 | 1.006 |
| tomcatv | 5436594306 | 5436670500 | 1.000 | 5436456381 | 1.000 |
| su2cor | 4625207540 | 4625248569 | 1.000 | 4625117247 | 1.000 |
| mgrid | 2138832527 | 2132120132 | 1.003 | 2061823555 | 1.037 |
| fpppp | 8404705112 | 8410760399 | 0.999 | 8404047239 | 1.000 |
| wave5 | 2221747518 | 2208702430 | 1.006 | 2179225372 | 1.020 |

**Table 3: Performance of SPEC95 Benchmarks. (WB buffer = 1, EQ = 4)**

| | *Baseline* | *Eager (EQ=0)* | | *Eager (EQ=4)* | | *Eager (EQ=256)* | | *Free Writeback* | |
|---|---|---|---|---|---|---|---|---|---|
| write buffer size | cycles | cycles | speedup | cycles | speedup | cycles | speedup | cycles | speedup |
| No light, WB Buf=1 | 25364637 | 23876911 | 1.062 | 21838002 | 1.162 | 21837952 | 1.162 | 21798206 | 1.164 |
| No light, WB Buf=4 | 25320139 | 21820627 | 1.160 | 21820566 | 1.160 | 21820566 | 1.160 | 21798206 | 1.162 |
| No light, WB Buf=256 | 25320139 | 21820566 | 1.160 | 21820566 | 1.160 | 21820566 | 1.160 | 21798206 | 1.162 |
| 3 diff. lights, WB Buf=1 | 30643341 | 29200004 | 1.049 | 27176616 | 1.128 | 27176333 | 1.128 | 27134147 | 1.129 |
| 3 diff. lights, WB Buf=4 | 30643153 | 27158044 | 1.128 | 27158049 | 1.128 | 27158044 | 1.128 | 27134147 | 1.129 |
| 3 diff. lights, WB Buf=256 | 30643153 | 27158044 | 1.128 | 27158044 | 1.128 | 27158044 | 1.128 | 27134147 | 1.129 |

**Table 4: Simulated cycles of Mini-Geometry Pipeline.**

dering command output is being written into the write-combining graphics memory while eager writebacks of dirty lines are concurrently taking place. Since within the driver loop there is still some computation occurring, the bandwidth is not fully utilized, and eager writeback writes can use the available idle slots and maximize bandwidth. Conversely, in the baseline case, the same writebacks occur within the geometry computation loop. This means these requests compete for the bus with the return of the data requested by vertex loads, and thus slow down the processing. This maximization of the utilization of the bandwidth during the driver loop leads to a lower and sparser average memory bandwidth in Eager Writeback than in the Baseline case outside the driver loop[2].

The overall performance improvement is obviously gained from the shifting of dirty writeback traffic to where this traffic does not impede the return of any data on the critical path of performance. This can be seen in Figure 9, which presents an execution profile of the benchmark. In this figure the sequence of vertex data load requests appears on the y-axis, and the cycle upon which the corresponding data item returns is plotted on the x-axis. As execution begins, the profiles of Baseline and Eager Writeback are completely overlapped, because data is returning at the same time for both schemes. Beginning at around 2.6 million cycles, these two curves start to deviate from one other, and continue to diverge as execution time increases. The first deviation indicates the location where data returns of the Baseline model were getting delayed because of dirty writeback contentions. The speedup due to Eager Writeback as measured is around 16%.

By looking carefully at this figure it is possible to distinguish the geometry computation loop from the device driver loop on each curve. The segments with shorter but steeper slopes are where the driver loop is executing. The steepness of the slope occurs because the requested data was returned faster (since the loop read the output vertices generated in the transformation and lighting stages from the L2 cache directly, rather than from memory).

[2]It should be emphasized that the total bandwidth required by a system using Eager Writeback is not reduced; rather, it is re-distributed by the early eviction of dirty cache lines.

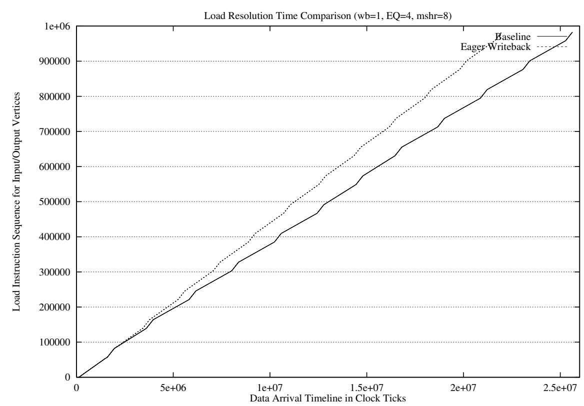


**Figure 9: Load Response Time for Input Vertex in Mini-Geometry Pipeline**

Table 5 shows how Eager Writeback affects the performance bottleneck in the Register Update Unit (RUU) of the processor. The layout of this table is similar to Table 4, and contains the number of cycles the processor is stalled due to the RUU being full. As the table shows, Eager Writeback is able to remove a substantial number of stall cycles due to a full RUU and keep the execution pipeline running smoother. These stalls are reduced because in conventional writeback schemes dirty writebacks are competing with demand fetches for available bandwidth, causing delays in data arrival and the filling of the reservation stations in the RUU. The eager writebacks shift the dirty writes to an earlier time, freeing up the bandwidth to handle just data reads and reducing the pressure on the RUU.

| | *baseline* | *Eager (EQ=0)* | | *Eager (EQ=4)* | | *Eager (EQ=256)* | | *Free Writeback* | |
|---|---|---|---|---|---|---|---|---|---|
| RUU Full cycles | cycles | cycles | improved | cycles | improved | cycles | improved | cycles | improved |
| No light, WB Buf = 1 | 8404023 | 6678659 | 20.5% | 4452469 | 47.0% | 4452265 | 47.0% | 4409553 | 47.5% |
| No light, WB Buf = 4 | 8375679 | 4439397 | 47.00% | 4439226 | 47.00% | 4439226 | 47.00% | 4409553 | 47.35% |
| 3 diffuse lights, WB Buf=1 | 8045791 | 6541028 | 18.7% | 4361651 | 45.8% | 4361259 | 45.8% | 4313710 | 46.4% |
| 3 diffuse lights, WB Buf=4 | 8033850 | 4344799 | 45.92% | 4344670 | 45.92% | 4344653 | 45.92% | 4313710 | 46.31% |

**Table 5: Resource Hazard Improvement of Mini-Geometry Pipeline.**

### 2.2 *With Injected Memory Traffic*

In order to evaluate the effectiveness of Eager Writeback in a real system, we implemented a memory traffic injector which we used to model other bus agents requesting the memory bus and consuming memory bandwidth. For this study, we injected three different external bandwidths using two different injection frequencies onto the bus during the simulations. The external bandwidths chosen were 400MB/sec, 800MB/sec and 1.2GB/sec. For each bandwidth configuration, data was injected at a high frequency (every 400 processor clock cycles) and a low frequency (every 3200 processor clock cycles). Data was injected onto the bus in blocks - for example, in the 800MB high frequency case, every 400 cycles the injector took over the bus and held it until it had completed transferring 320 bytes of data. The injections are uniformally distributed throughout the simulation.

The results for simulations of the mini-geometry pipeline using no light sources are shown in Table 6. The top line of the table is the base case with no injected memory traffic, while the other entries are for the different injected bandwidths at the different frequencies. In this table we can see that (as expected) memory traffic injection causes extra stall cycles in the RUU. In addition, as the amount of injected bus traffic increases, the opportunity to do Eager Writeback decreases and the RUU stalls climb dramatically.

The table also shows that Eager Writeback provides virtually no speedup when a bandwidth of 0.8GB/sec is injected at the higher frequency, while the same bandwidth injected at a lower frequency allows a speedup of 11%. By examining the dirty writeback bandwidth utilization profile of this scenario ( Figure 10 and Figure 11), one can see that many eager writebacks (i.e. the spikes) are prevented from occurring by the higher frequency injection. The advantages of Eager Writeback are lost and it performs almost on par with the baseline scenario, due to more frequent bus contention.

## 3 Streaming Kernel

The mini-geometry pipeline highlighted the problem of implicit dirty writebacks causing loss of performance due to delays in receiving data. Finite memory peak bandwidth is another serious performance issue, which is exposed by the Streaming kernel.

### 3.1 *Without Injected Memory Traffic*

Table 7 contains the results of simulation runs of the Streaming kernel, presented in the same format used in Table 4. For this benchmark, an eager queue of length 0 (EQ=0) is enough to approximate the optimal case of no dirty writeback traffic at all. Further size increases of the EQ provide only marginal performance gains.

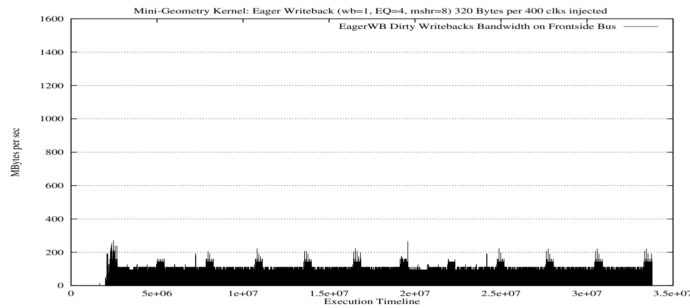


**Figure 10: Dirty WB L2-to-Mem Bandwidth with 320B/400clks Injection (*Eager*) for Geometry**

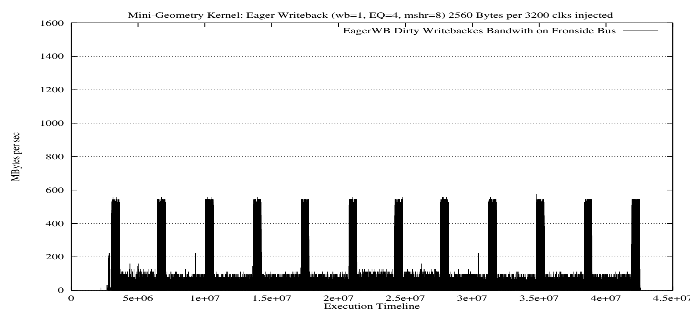


**Figure 11: Dirty WB L2-to-Mem Bandwidth with 2560B/3200clks Injection (*Eager*) for Geometry**

Looking at the memory bandwidth utilization profiles for this kernel (Figure 12 and Figure 13), we see three spikes that appear repeatedly in both writeback schemes (because the outer loop contains three iterations). The spikes are much wider in the Baseline case, however, indicating the program is spending more execution cycles in these phases. Examining the algorithm, it is clear these spikes are related to the time during the third inner loop where incoming $array_B$ data items collide and share memory bandwidth with the induced dirty writebacks of $array_A$. Because the finite memory bandwidth (1.6 GB/sec in this study) must be shared between both memory accesses[3], the rate of demand fetches for $array_B$ in the third inner loop is (theoretically) cut in half and thus the overall performance degrades.

Figure 12 also shows three bandwidth grooves where memory bus bandwidth has dropped to zero. These correspond to the second inner loops, where all data references hit in

[3]Read and write turnarounds between demand fetch and dirty writeback streams also prevent peak memory bandwidth from being achieved.

| *bandwidth* | *sim cycles* | | | *RUU Full cycles* | | |
|---|---|---|---|---|---|---|
| *injection (no light)* | Baseline | Eager | speed-up | Baseline | Eager | improved |
| 0 GB/sec | 25364637 | 21838002 | 1.16 | 8404023 | 4452469 | 47.0% |
| 0.4GB/sec (160B/400clks) | 27323771 | 25434535 | 1.07 | 10529817 | 8448695 | 19.76% |
| 0.8GB/sec (320B/400clks) | 33567580 | 33775835 | 0.99 | 16760998 | 17024045 | -1.6% |
| 1.2GB/sec (480B/400clks) | 60699573 | 59162773 | 1.03 | 44206642 | 42864369 | 3.0% |
| 0.4GB/sec (1280B/3200clks) | 32539684 | 28636072 | 1.14 | 15604083 | 11364679 | 27.2% |
| 0.8GB/sec (2560B/3200clks) | 47365936 | 42559653 | 1.11 | 30356564 | 25269290 | 16.8% |
| 1.2GB/sec (3840B/3200clks) | 87400980 | 83426435 | 1.05 | 70248220 | 66015191 | 6.0% |

**Table 6: Memory Traffic Injection to Mini-Geometry Pipeline. (EQ = 4)**

| *sim cycle* | *Baseline* | *Eager (EQ=0)* | | *Eager (EQ=4)* | | *Eager (EQ=256)* | | *Free Writeback* | |
|---|---|---|---|---|---|---|---|---|---|
| write buffer size | cycles | cycles | speedup | cycles | speedup | cycles | speedup | cycles | speedup |
| WB buf = 1 | 10230328 | 9054559 | 1.130 | 9053851 | 1.130 | 9053851 | 1.130 | 9045154 | 1.131 |
| WB buf = 4 | 10067331 | 9052957 | 1.112 | 9052957 | 1.112 | 9052957 | 1.112 | 9045154 | 1.113 |
| WB buf = 256 | 10223055 | 9052821 | 1.129 | 9052821 | 1.129 | 9052821 | 1.129 | 9045154 | 1.113 |

**Table 7: Simulated cycles of Streaming Kernel.**

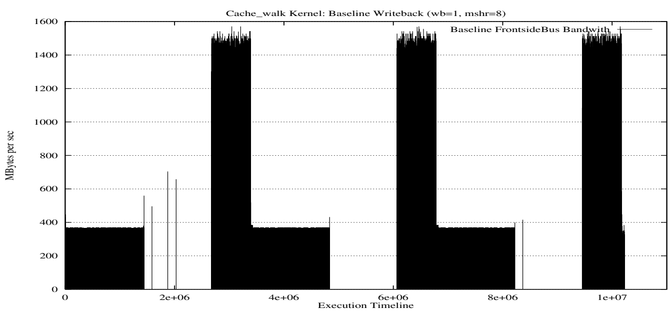


**Figure 12: Memory Bandwidth Distribution by *Baseline Writeback* for Streaming Kernel**

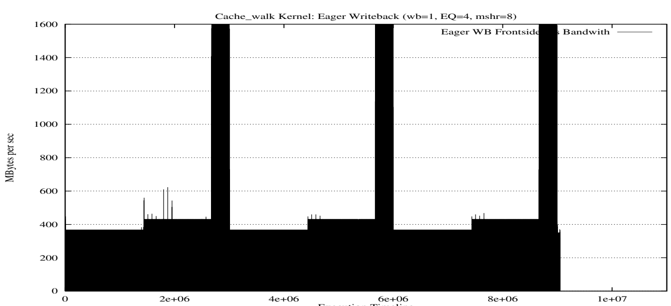


**Figure 13: Memory Bandwidth Distribution by *Eager Writeback* for Streaming Kernel**

the cache. To take advantage of this available resource, Eager Writeback fills these bus idle states with early evictions of dirty data cache lines (as shown in Figure 13). By shifting these bandwidth requests to idle cycles, the memory bandwidth during the course of the third inner loop can be fully dedicated to the demand fetches of $array_B$, speeding up the cache fill requests.

As was done for the mini-geometry pipeline, we examined how Eager Writeback interacted with internal processor resources when running this benchmark. Table 8 shows that the Load/Store Queue is used heavily by this benchmark, and that Eager Writeback can remove more than half of the stalls due to a full Load/Store Queue. As the LSQ is kept less full, instructions are able to leave the Instruction Fetching Queue (IFQ) faster and as a result cycles lost due to a full IFQ are reduced substantially.

### *3.2 With Injected Memory Traffic*

We also repeated the experiments involving injecting memory traffic onto the bus for this benchmark program. The results are shown in Table 9, and indicate that higher frequency injection seems to have a greater impact on the Baseline case than on the Eager Writeback case. The number of simulated cycles for the Baseline case using high frequency injection increases faster than for the Eager Writeback case, while the increase stays roughly the same for both schemes while injecting lower frequency traffic.

The reason the cycle count climbs faster for the Baseline case than for the Eager Writeback case can be understood by analyzing Figure 14. This figure contains an execution profile of the Streaming benchmark, plotting the arrival time for each load instruction. Each curve can be divided into 3 repeated patterns, which bear the following three piecewise line segments: flat (zero increment), steep rise, and slowdown knee. These 3 line segments correspond to the three inner loops in the benchmark.

The first loop contains only data stores, so the load instruction count stays flat as execution time continues. The steep vertical climb corresponds to the second inner loop, which has a high number of cache hits (a large number of loads completing in a short period of time). Finally, the third segment represents the behavior of the third loop, which loads another array that misses in both the caches.

This third segment, shown as a knee in the curve, reveals the reason for the performance deviation between Baseline and Eager Writeback. Figure 15 shows a close-up view of part of Figure 14, focusing on the knees of the curve. The slopes ($\tan\theta$) of these knees are the key - the flatter the slope, the longer it will take to complete. Comparing the slope changes between Baseline and Eager Writeback, it is obvious that the slope of the Baseline segment is much shallower than that of the Eager Writeback segment. This means that for the same number of loads in the third loop, the execution time of the Baseline case was more sensitive to and severely delayed by other transactions, which in this case are composed of the dirty writebacks induced by the

| Bottlenecks | baseline cycles | Eager (EQ=0) cycles | Eager (EQ=0) improved | Eager (EQ=4) cycles | Eager (EQ=4) improved | Eager (EQ=256) cycles | Eager (EQ=256) improved | Free Writeback cycles | Free Writeback improved |
|---|---|---|---|---|---|---|---|---|---|
| IFQ Full cycles | 5770175 | 4594401 | 20.38% | 4594631 | 20.37% | 4594631 | 20.37% | 4587638 | 20.49% |
| RUU Full cycles | 4274868 | 4260784 | 0.33% | 4260703 | 0.33% | 4260703 | 0.33% | 4258811 | 0.38% |
| LSQ Full cycles | 1978596 | 864867 | 56.29% | 866341 | 56.21% | 866341 | 56.21% | 862880 | 56.39% |

**Table 8: Resource Constraint Improvement of Streaming Kernel. (Writeback buffer = 1)**

| *bandwidth injection* | *simulated cycles* Baseline | Eager | speed-up | *IFQ Full cycles* Baseline | Eager | improved | *LSQ Full cycles* Baseline | Eager | improved |
|---|---|---|---|---|---|---|---|---|---|
| 0 MB/sec | 10230328 | 9053851 | 1.13 | 5770175 | 4594631 | 20.4% | 1978596 | 866341 | 56.2% |
| 0.4GB/sec (160B/400clks) | 11807448 | 10039848 | 1.18 | 7340618 | 5576536 | 24.0% | 2903145 | 1205358 | 58.5% |
| 0.8GB/sec (320B/400clks) | 15025957 | 12389159 | 1.21 | 10540877 | 7908077 | 25.0% | 4428473 | 1882587 | 57.5% |
| 1.2GB/sec (480B/400clks) | 24250335 | 21412735 | 1.13 | 19717746 | 16880309 | 14.4% | 8309036 | 5480188 | 34.05% |
| 0.4GB/sec (1280B/3200clks) | 12379290 | 10991058 | 1.13 | 7908538 | 6521201 | 17.5% | 2030932 | 1417595 | 30.2% |
| 0.8GB/sec (2560B/3200clks) | 16593748 | 15115348 | 1.10 | 12101456 | 10622058 | 12.2% | 4264295 | 2818313 | 33.9% |
| 1.2GB/sec (3840B/3200clks) | 29048835 | 27135235 | 1.07 | 24495295 | 22585042 | 7.8% | 8903039 | 7007451 | 21.3% |

**Table 9: Memory Traffic Injection to Streaming Kernel. (Eager Queue = 4)**

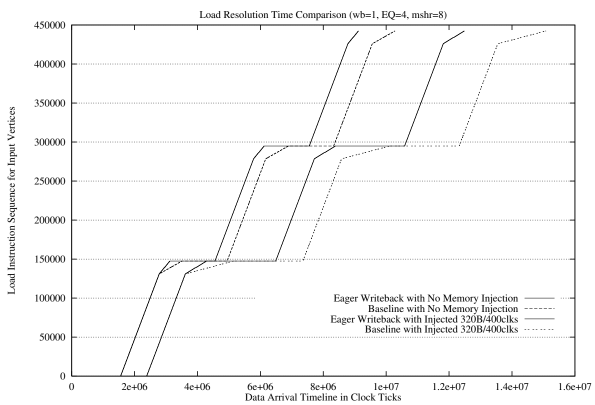


**Figure 14: Load Response Time for Data Reads in Streaming Kernel (Higher Frequency Injection)**

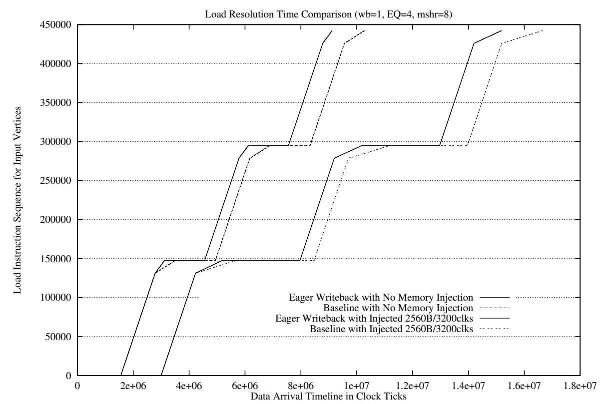


**Figure 16: Load Response Time for Data Reads in Streaming Kernel (Lower Frequency Injection)**

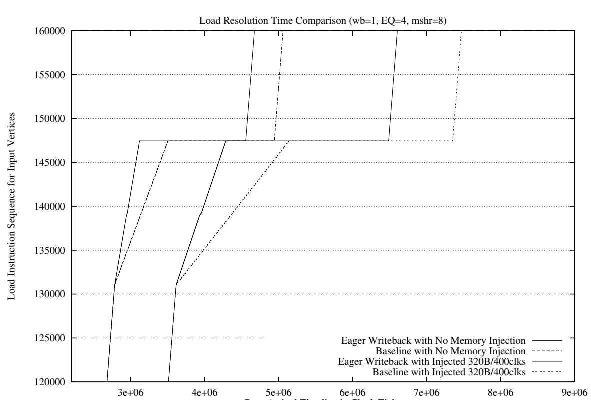


**Figure 15: Details of the Load Response Time for Data Reads (Higher Freq. Injection)**

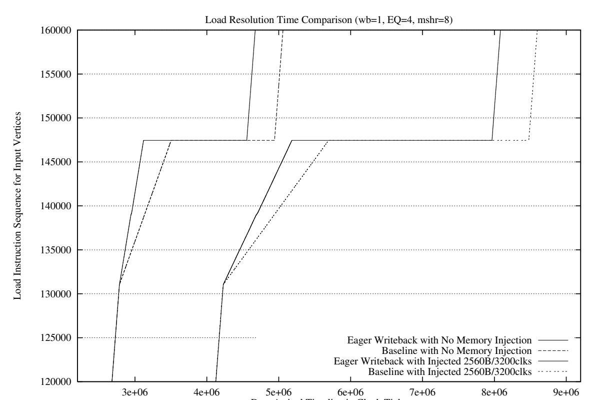


**Figure 17: Details of the Load Response Time for Data Reads (Lower Frequency Injection)**

loads and the periodic injection of memory traffic. For the Eager Writeback case, the dirty writebacks were mostly completed in the second loop, so the slope of the knee is steeper and the third loop can be completed more swiftly than its Baseline counterpart.

Repeating the same experiment using lower frequency injection (as plotted in Figure 16 and Figure 17) reveals that the slope of the knees of the curve are more similar to one another. As a result, roughly the same number of penalty cycles were added to both Baseline and Eager Writeback, and the speedups due to Eager Writebacks are smaller in Table 9. These results suggest higher frequency interference can deteriorate performance in the baseline case more in a bandwidth-limited code.

## VI. CONCLUSIONS

Systems employing write-back caches have to contend with the following two issues: (1) Dirty writebacks contend with demand fetches for bandwidth and can impede the delivery of data, and (2) Finite memory bandwidth shared between demand fetches and implicit dirty writebacks limit the performance of memory bound programs. These performance issues are important to a large and growing class of programs – those that consume large amounts of memory bandwidth and generate many data stores.

In this paper we have presented a new technique for dealing with these issues, called Eager Writeback, which can effectively improve overall system performance by shifting the writing of dirty cache lines from on-demand to times when the memory bus is idle. This time-shifting is accomplished by identifying and speculatively writing ("cleaning") dirty lines whenever the bus is free. We have shown that for a wide variety of programs, once a dirty cache line has entered the LRU state it is rarely written to again. We use this fact to identify the lines that should be speculatively written (although this information could be of interest to many other intelligent cache management techniques as well).

We have shown that applying this technique can alleviate bandwidth congestion and improve performance for two kernels that are representative of these classes of applications. We have shown that when conventional writebacks compete with memory loads and defer the delivery of data, the Eager Writeback technique is able to remove the competition by evicting dirty data earlier. We have also shown that when "finite" memory bandwidth limits overall performance, eager writeback can alleviate this situation by utilizing earlier idle bus cycles. Eager Writeback can be implemented in a number of was - for example, as an additional programmable memory type on top of the exisiting memory types provided by a processor to speed up bandwidth-hungry applications, e.g. 3D games or content-rich applications.

Further investigation of this Eager Writeback mechanism will include the effect this approach has on other system performance issues. For example, Eager writeback can be expected to reduce context switching time overhead by flushing dirty lines in advance of the context switch. In addition, Eager Writeback can push modified data closer to the globally observable memory level earlier to reduce coherence miss latency, and as a result, respond to other processors' requests faster. Similarly, the same analysis performed in this paper can be applied to write-update and write-invalidate protocols in a shared memory system to reduce coherence traffic in a way similar to (but perhaps simpler to implement than) that presented in [8].

## VII. ACKNOWLEDGMENTS

This research is sponsored by NSF grant CCR-9812415, NSF Career grant MP-9734023 and generous gifts from IBM and Intel corporations.

## VIII. REFERENCES

[1] Doug C. Burger and Todd M. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report 1342, Computer Science Department, University of Wisconsin-Madison, 1997.

[2] Intel Corporation. Pentium Pro Family Developer's Manual, volume 3: Operating System Writer's Manual. Intel Literature Centers, 1996.

[3] Intel Corporation. IA-64 Architeture Software Developer's Manual, Volume 2: IA-64 System Architecture. Intel Literature Centers, 2000.

[4] Rambus Corporation. Direct Rambus Memory Controller (RMC.dl) Data Sheet. http://www.rambus.com/docs/RMC.d1.0036.00.8.pdf, 1999.

[5] Standard Performance Evaluation Corporation. SPEC CPU95 Benchmarks. http://www.specbench.org/osg/cpu95/, 1995.

[6] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kauffmann Publishers, Inc., second edition, 1996.

[7] David Kroft. Lockup-Free Instruction Fetch/Prefetch Cache Organization. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, 1981.

[8] An-Chow Lai and Babak Falsafi. Selective, Accurate and Timely Self-Invalidation Using Last-Touch Prediction. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, 2000.

[9] Simplescalar Tool set. X benchmark suite. http://www.cs.wisc.edu/ austin /simple/xbenchmarks.tar.gz, 1998.

[10] Kevin Skadron and Douglas W. Clark. Design Issues and Tradeoffs for Write Buffers. In *Proceedings of the 3th International Symposium on High Performance Computer Architecture*, 1997.

[11] Guri Sohi and Sriram Vajapeyam. Instruction Issue Logic for High-Performance Interruptable Pipelined Processors. *Proceedings of the 14th Annual International Symposium on Computer Architecture*, 1987.

[12] Alan Watt. *3D Computer Graphics.* Addison-Wesley Publishers, 1993.

[13] Paul Zagacki, Deep Buch, Emile Hsieh, Daniel Melaku, Vladimir Pentkovski, and Hsien-Hsin Lee. Architecture of a 3D Software Stack for Peak Pentium III Processor Performance. *Intel Technology Journal*, Q2 1999. http://developer.intel.com/technology/itj/q21999/arcticles/art_4.htm.