

Memory System Support for Image Processing

Lixin Zhang, John B. Carter, Wilson C. Hsieh, Sally A. McKee
Department of Computer Science
50 South Central Campus Drive, Room 3190
University of Utah
Salt Lake City, UT 84112
{lizhang,retrac,wilson,sam}@cs.utah.edu

Abstract

Image processing applications tend to access their data non-sequentially and reuse that data infrequently. As a result, they tend to perform poorly on conventional memory systems due to high cache and TLB miss rates and are particularly sensitive to the growing latency of main memory. In this paper, we analyze the memory performance of three image processing algorithms (volume rendering, image warping, and image filtering) on both a conventional memory system and on the Impulse memory system. The Impulse memory system allows application software to control how, when, and where data are loaded into a conventional processor cache. It does this by letting software configure how the memory controller interprets the physical addresses exported by the processor, which enables an application to dynamically change how data are fetched. Sparse data can be accessed densely, which improves both cache and TLB utilization, and memory latency is hidden by prefetching data within the memory controller. We find that for these image processing codes, using an Impulse memory system yields speedups of 40% to 226% over an otherwise identical machine with a conventional memory system.

1. Introduction

Because of the growing processor-memory performance gap, only applications with very high degrees of locality are able to exploit the full performance potential of modern microprocessors. Applications that suffer frequent cache or

TLB misses find their performance limited by the speed of the memory system. Image processing applications are particularly sensitive to the memory bottleneck problem. These algorithms are characterized by very high data bandwidth needs, large cache footprints, lack of data reuse, and a tendency to access data along non-unit strides. These behaviors make traditional caching much less effective than for other types of programs. Fortunately, image processing algorithms have the advantage that their memory access patterns are often predictable. This predictability, combined with their poor cache behavior, makes the application domain a particularly interesting one for memory system architects.

To attack the memory bottleneck problem, architects have increased the number and size of processor caches and have added a variety of mechanisms to memory hierarchies. These mechanisms can be roughly split into three categories: (i) those that increase the hit rate of processor caches (e.g., skewed associative caches [22] and prefetching [5, 7, 12]), (ii) those that increase the processor's ability to tolerate memory latency (e.g., multithreaded processors [1, 8, 25]), and (iii) those that migrate computation to or integrate the processor with the memory system (e.g., IRAM [19] and RADRAM [15]). These approaches all have merit, but they also have limitations. Many trade bandwidth for latency, which places a heavier burden on the memory system by fetching unneeded data. In general, complex cache organizations cannot keep up with aggressive processor cycle times, and thus are poor design candidates for the top levels of the memory hierarchy. Finally, most of the proposed mechanisms require significant modifications to conventional processor or DRAM designs, and may interact with other parts of the microarchitecture in complex ways. The result is that few have found their way into commercial chips.

In contrast, we are developing an adaptable memory system that can decrease memory latency and increase the hit rate of processor caches and TLBs *without* requiring mod-

This effort was sponsored in part by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under agreement number F30602-98-1-0101 and DARPA Order Numbers F393/00-01 and F376/00. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of DARPA, AFRL, or the US Government.

ifications to conventional processor cores, caches, memory buses, or DRAMs. Instead, we concentrate on *increasing the effective bandwidth* of data supplied to the processor and *decreasing the observed latency* of memory accesses that miss in the processor cache(s). The Impulse adaptable memory system represents a combined hardware/software approach to bridging the processor-memory performance gap for applications with poor, but predictable, memory locality (e.g., streaming and vector-style applications).

Impulse enables several optimizations that let an application control how, when, and where its data are loaded into the on-chip caches via the following mechanisms [3, 24]: (i) gathering sparse data into dense cache lines, (ii) tiling without copying data, (iii) recoloring data structures without copying, and (iv) mapping non-contiguous physical pages to a single TLB entry. To decrease the observed latency of memory accesses, Impulse supports many in-flight DRAM accesses and aggressively prefetches data within the memory controller [3].

Previous work describes Impulse’s support for irregular applications and presents preliminary results for two scientific applications on the Impulse memory system [3]. Here we report the impact of Impulse optimizations on three image processing algorithms: *volume rendering*, *separable image warps*, and *image filtering*. For these codes, our detailed simulations indicate that an Impulse memory system improves execution time by up to 226% compared to a similar conventional memory system.

2. Impulse System Architecture

Impulse expands the traditional virtual memory hierarchy by adding address translation hardware to the memory controller. This optional, extra level of mapping is motivated by three observations:

- caches operate on contiguous “physical” memory at the granularity of a cache line,
- many levels of the modern cache hierarchies are physically indexed and tagged, and
- not all physical addresses in a traditional virtual memory system map to valid memory locations.

The Impulse system software configures the memory controller to reinterpret some of the physical addresses presented to it, “virtualizing” unused physical addresses to map data in ways that improve the efficiency of on-chip caches.

The unused physical addresses constitute a *shadow address space*. Data items whose physical DRAM addresses are not contiguous can be mapped to contiguous shadow addresses, so that sparse data can be compacted into dense cache lines before being transferred on-chip. To map elements in these compacted cache lines back to physical

memory, Impulse must recover their offsets within the original data structure. It does so by first translating shadow addresses to *pseudo-virtual addresses*, and then translating these to physical DRAM addresses. The *pseudo-virtual address space* page layout mirrors the virtual address space, which allows Impulse to remap data structures that lie on disjoint physical pages. The shadow→pseudo-virtual→physical mappings all take place within the memory controller. The operating system manages all the resources in the expanded memory hierarchy and provides an interface for the application to specify optimizations for particular data structures. The programmer (or the compiler) inserts directives into the application code to configure the memory controller.

In this section, we present an overview of the Impulse hardware and the way in which applications can exploit it; a more detailed description can be found elsewhere [3].

2.1 Hardware Organization

The organization of the Impulse controller architecture is depicted in Figure 1. The memory controller includes:

- a *Shadow Descriptor Unit* that contains a small number of shadow-space descriptors, SRAM buffers to hold prefetched shadow data, and logic to assemble sparse data retrieved from DRAM into dense cache lines mapped in shadow space;
- a *Page Table Unit* that contains a simple ALU and *Memory Controller TLB* (MTLB) that map addresses in dense shadow space to pseudo-virtual and then to DRAM physical addresses, along with a small number of buffers to hold prefetched page table entries; and
- a *Scheduling Unit* that contains circuitry that orders and issues accesses to the DRAMs, along with an *SRAM Memory Controller Cache* (Mcache) to buffer non-shadow data.

Since the extra level of address translation is optional, addresses appearing on the memory bus may be to physical (backed by DRAM) or shadow memory space. Valid physical addresses pass untranslated to the DRAM scheduler. The Page Table Unit uses the corresponding shadow descriptor to turn shadow addresses into physical DRAM addresses. Depending on how Impulse is used to access a particular data structure, this translation can be *direct*, *strided*, or *scatter/gather*.

Direct mapping translates a shadow address directly to a physical DRAM address. This mapping can be used to recolor physical pages without copying [3] or to construct superpages dynamically [24]. *Strided mapping* creates dense cache lines from array elements that are not contiguous. The mapping function maps an address *offset* in

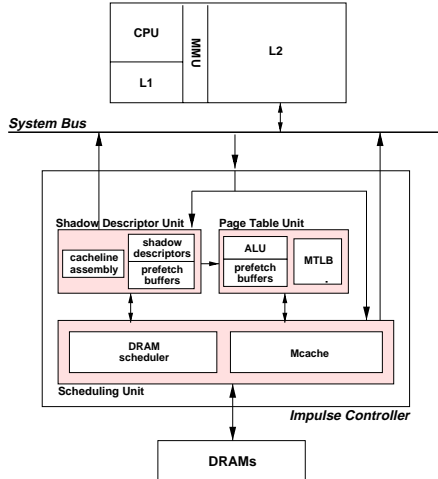


Figure 1. Impulse memory controller organization.

shadow space to pseudo-virtual address $pvaddr + stride \times soffset$, where $pvaddr$ is the starting address (assigned by the OS) of the data structure’s pseudo-virtual image. *Scatter/gather mapping* uses an indirection vector vec to translate an address $soffset$ in shadow space to pseudo-virtual address $pvaddr + stride \times vec[soffset]$. Investigating support for other mappings is part of ongoing work.

2.2 Software Interface and OS Support

To exploit Impulse, the application first instructs the operating system to allocate a range of contiguous virtual addresses large enough to map the necessary elements of a data structure. Subsequent accesses to the original data structure are made via this new virtual image. Next, the application requests that the new virtual data structure be mapped through shadow memory to the physical data elements. The OS responds by allocating a contiguous range of shadow addresses and downloading two pieces of information to the memory controller: (i) a function that the Page Table Unit uses to perform the mapping from shadow to pseudo-virtual space and (ii) a set of page table entries that the MTLB uses to translate pseudo-virtual addresses to physical DRAM addresses.

As an example, consider remapping the diagonal of an $n \times n$ matrix $A[]$. Figure 2 depicts the memory translations for both the matrix $A[]$ and the remapped image of its diagonal. The OS allocates shadow addresses from a pool of unused physical addresses. It also allocates pseudo-virtual pages so that no remapped data structures overlap. The OS interface allows alignment and offset characteristics of the remapped data structure to be specified, which gives the compiler or the application some control over L1 cache behavior. In the current Impulse design, coherence

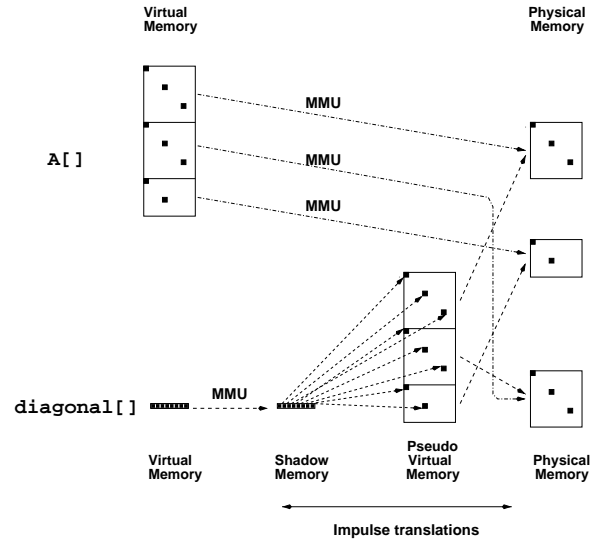


Figure 2. Accessing the (sparse) diagonal elements of an array via a dense `diagonal` variable in Impulse.

is maintained in software: the OS, the compiler, or the application programmer must keep aliased data consistent by explicitly flushing the cache.

3. Image Processing Algorithms

We examine three image processing algorithms that stress memory system performance:

- *Volume rendering* generates an image of a volume from a given point of view.
- *Separable image warping* algorithms perform successive, separate, one-dimensional image transformations. Our benchmark rotates an image by performing three, one-dimensional shears.
- *Image filtering* applies a numerical filter function that modifies an image’s appearance.

These operations differ from the benchmarks traditionally used to evaluate memory system behavior in that they often:

- need to be performed in real time, as part of interactive applications;
- “stream” through large amounts of data with little reuse, and thus have poor temporal locality and high bandwidth requirements;
- access their data along non-unit strides (i.e., walk along a spatial dimension that does not match the data layout); and

- touch many pages, putting enormous pressure on processor TLBs.

The code that implements these algorithms is sufficiently complex that we cannot usefully include it herein. Interested readers can find the complete source code for all three algorithms, including the modifications necessary to exploit Impulse, at <http://www.cs.utah.edu/impulse/releases/>.

3.1 Volume Rendering

We examine the volume rendering technique demonstrated by Parker et al. [18]. In contrast to other volume rendering methods, this method does not generate an explicit representation of the isosurface and render it with a z-buffer, but instead uses brute-force ray tracing to perform interactive isosurfacing. For each ray, the first isosurface intersected determines the value of the corresponding pixel. The approach has a high intrinsic computational cost, but its simplicity and scalability make it ideal for large data sets on current high-end systems.

Traditionally, ray tracing has not been used for volume rendering because it suffers from poor memory behavior. Each ray must be traced through a potentially large fraction of the volume, giving rise to two problems. First, many memory pages may need to be touched, which results in high TLB pressure. Second, a ray with a high angle of incidence may visit only one volume element (voxel) per cache line, in which case bus bandwidth will be wasted loading unnecessary data that pollutes the cache.

By carefully hand-optimizing their ray tracer’s memory access patterns, Parker et al. achieve acceptable speeds for interactive rendering (about 10 frames per second). They improve data locality by organizing the data set into a multi-level spatial hierarchy of tiles, each composed of smaller cells. The smaller cells provide good cache-line utilization. “Macro cells” are created to cache the minimum and maximum data values from the cells of each tile. These macro cells enable a simple min/max comparison to detect whether a ray intersects an isosurface within a tile; empty macro cells need not be traversed.

Careful hand-tiling of the volume data set can yield much better memory performance, but choosing the optimal number of levels in the spatial hierarchy and sizes for the tiles at each level is difficult, and the resulting code is hard to understand and maintain. Impulse can deliver better performance than hand-tiling at a lower programming cost. There is no need to preprocess the volume data set for good memory performance: the Impulse memory controller can remap it dynamically.

Like many real-world visualization systems, our benchmark uses an orthographic tracer, whose rays all intersect

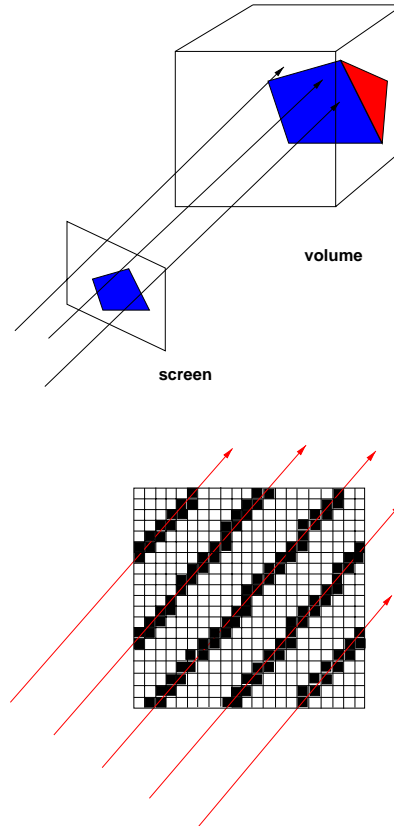


Figure 3. Volume rendering via orthographic ray tracing. The picture on the left shows rays (which are all perpendicular to the viewing screen) being traced through a volume to determine pixel values. The one on the right illustrates how each ray visits a sequence of voxels in the volume; Impulse optimizes voxel fetches from memory via indirection vectors representing the voxel sequences for each ray.

the screen surface at right angles. Such tracers produce images that lack perspective and appear far away, but are relatively simple to compute.

We use Impulse to extract the voxels that a ray potentially intersects when traversing the volume. The right-hand side of Figure 3 illustrates how each ray visits a certain sequence of voxels in the volume. Instead of fetching cache lines full of unnecessary voxels, Impulse can remap a ray to the voxels it requires so that only useful voxels will be fetched.

3.2 Separable Image Warping

Image warping refers to any algorithm that performs an image-to-image transformation. *Separable image warps*

are those that can be decomposed into multiple one-dimensional transformations [4]. For separable warps, Impulse can be used to improve the cache and TLB performance of one-dimensional traversals orthogonal to the image layout in memory. Our three-shear image rotation benchmark is an example of a separable image warp. This algorithm rotates a 2-dimensional image around its center in three stages, each performing a “shear” operation on the image, as illustrated in Figure 4. The algorithm thus implements a two-dimensional image transformation using several one-dimensional transformations, each of which is simpler to implement, faster to run, and has fewer visual artifacts. The underlying math is straightforward. Rotation through an angle θ can be expressed as matrix multiplication:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

The rotation matrix can be broken into three shears as follows:

$$\begin{pmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ -\tan \frac{\theta}{2} & 1 \end{pmatrix} \begin{pmatrix} 1 & \sin \theta \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ -\tan \frac{\theta}{2} & 1 \end{pmatrix}$$

None of the shears requires scaling (since the determinant of each matrix is 1), so each involves just a shift of rows or columns. Not only is this algorithm simple to understand, it is robust in that it is defined over all rotation values from 0° to 90° . Two-shear rotations fail for angles near 90° . We assume a simple image representation of an array of pixel values. The second shear operation (along the y axis) walks along the column of the image matrix, which gives rise to poor memory performance for large images. We use Impulse to create a transposed version of the array in shadow space. As a result, the memory controller performs a (logical) column walk of the original array in response to accesses to the synthetic array’s rows, which improves both cache and TLB performance.

3.3 Image Filtering

Image filtering may be used to attenuate high-frequency components caused by noise in a sampled image, to adjust an image to different geometry, to detect or enhance edges within an image, or to create various special effects. Box, Bartlett, Gaussian, and binomial filters are common in practice. Each modifies the input image in a different way, but all share similar computational characteristics.

We concentrate on a representative class, *binomial filters* [10], in which each pixel in the output image is computed by applying a two-dimensional “mask” to the input image. Binomial filtering is computationally simi-

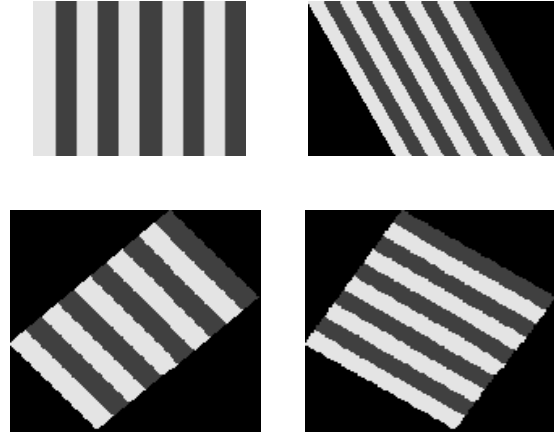


Figure 4. Three-shear rotation of an image counter-clockwise through one radian. The original image (upper left) is first sheared horizontally (upper right). That image is sheared upwards (lower right). The final rotated image (lower left) is generated via one last horizontal shift.

lar to a single step of a successive over-relaxation algorithm for solving differential equations: the filtered pixel value is calculated as a linear function of the neighboring pixel values of the original image and the corresponding mask values. For example, for an order-5 binomial filter, the value of pixel (i, j) in the output image will be $\frac{36}{256} * (i, j) + \frac{24}{256} * (i - 1, j) + \frac{24}{256} * (i + 1, j) + \dots$. To avoid edge effects, the original image boundaries must be extended before applying the masking function. Figure 5 illustrates a black-and-white sample image before and after the application of a small binomial filter.

In practice, many filter functions, including binomial, are “separable,” meaning that they are symmetric and can be decomposed into a pair of orthogonal linear filters. For example, a two-dimensional mask can be decomposed into two one-dimensional masks $([\frac{1}{16}, \frac{4}{16}, \frac{6}{16}, \frac{4}{16}, \frac{1}{16}])$ — the two-dimensional mask is simply the outer product of this one-dimensional mask with its transpose. The process of applying the mask to the input image can be performed by sweeping first along the rows and then the columns, calculating a partial sum at each step. Each pixel in the original image is used only for a short time, which makes filtering a pure streaming application. Impulse can transpose both the input and output image arrays without copying, which gives the column sweep much better cache behavior.

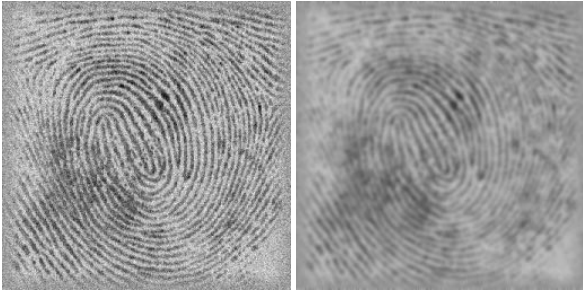


Figure 5. Example of binomial image filtering. The original image is on the left, and the filtered image is on the right.

4. Simulation Environment

Our studies use the Paint simulator [23], which models a variation of a 120 MHz, single-issue, HP PA-RISC 1.1 processor running a BSD-based microkernel, and a 120 MHz HP Runway bus. The 32K L1 data cache is non-blocking, single-cycle, write-back, write-around, virtually indexed, physically tagged, and direct mapped with 32-byte lines. The 256K L2 data cache is non-blocking, write-allocate, write-back, physically indexed and tagged, 2-way set-associative, and has 128-byte lines. Instruction caching is assumed to be perfect. The TLB is unified I/D, single-cycle, and fully associative, with a not-recently-used replacement policy. In addition to the main TLB, a single-entry micro-ITLB holds the most recent instruction translation. Kernel code and data structures are mapped using a single block-TLB entry that is not subject to replacement. The split-transaction bus multiplexes addresses and data. The memory system modeled contains four DRAM buses and 16 banks of SDRAM, and has a total memory latency of 44-46 cycles [3].

The simulated Impulse memory controller (described in Section 2.1) is based on the HP memory controller [11] used in servers and high-end workstations. We model eight shadow descriptors, each of which is associated with a 512-byte SRAM buffer. The controller prefetches the corresponding shadow data into these fully associative buffers of four 128-byte lines. A 4K-byte, SRAM Mcache holds prefetched, non-shadow data within the Scheduler Unit. The Mcache is four-way set associative with 32 lines of 128 bytes. The MTLB is two-way associative, has 128 eight-byte entries (the same size as the entries in the kernel’s page table), and includes two 128-bit buffers used to prefetch consecutive lines of page table entries on an MTLB miss.

References to shadow addresses incur a minimum three-cycle delay, and complex mapping functions cause larger

delays. Prefetching within the memory controller reduces the impact of the extra translation delays. To keep the remapping circuitry simple and fast, we require that the various dimensions of data structures used in strided mappings be powers of two in size. This avoids the need for a divider on the controller. All remapped data structures are page-aligned, and the elements being manipulated must be powers of two in size. We implemented the Impulse system calls within Paint’s microkernel such that applications can use Impulse without violating interprocess protection.

Although Paint’s PA-RISC processor is single-issue, we simulate a quad-issue superscalar machine by issuing four instructions per cycle without checking dependencies. While this model is unrealistic for gathering processor microarchitecture statistics, it stresses the memory system in a manner similar to a real superscalar processor, and thus should provide a useful estimate of Impulse’s performance. We are porting our simulation environment to RSIM[16], which models a true four-way superscalar processor. Preliminary results from this effort indicate that the processor model used for this study understates the value of Impulse.

5. Results

In our experiments we measure the performance benefits of using Impulse to remap physical addresses. These results are conservative, as we have not yet modified the Impulse Scheduler Unit to exploit bank parallelism, sophisticated DRAM interfaces, or device characteristics. We examine memory system performance for at least two versions of each benchmark: a hand-tuned algorithm running on a conventional memory system, and an algorithm hand-modified to run on Impulse.

5.1 Volume Rendering

For simplicity, our current version of this benchmark assumes that the screen plane is parallel to the volume’s z axis. This assumption lets us compute an entire plane’s worth of indirection vector at once, so we no longer need to remap addresses for every ray. It is not a large restriction: it assumes the use of a volume rendering algorithm like Lacroute’s [13], which transforms arbitrary viewing angles into angles that have better memory performance. Integrating our measurements with Lacroute’s algorithm is part of our ongoing work.

The measurements we present are for two particular viewing angles. The table at the top of Figure 6 shows results when the screen is parallel to the y axis, so that the rays exactly follow the layout of voxels in memory (we assume an x - y - z layout order). The table at the bottom shows results when the screen is parallel to the x axis — so that the rays exhibit the worst possible cache and TLB behavior in

	Original	Indirection	Impulse
Time	264	249	185
L1 hit ratio	96.8%	96.5%	96.6%
L2 hit ratio	0.8%	0.9%	0.9%
avg load time	2.0	2.1	1.8
TLB misses	.30	.32	.31
speedup	—	6.0%	42.7%

(A)

	Original	Indirection	Impulse
Time	1030	1000	316
L1 hit ratio	81.8%	83.2%	97.5%
L2 hit ratio	0.8%	1.1%	0.4%
avg load time	8.4	7.6	4.7
TLB misses	8.46	8.47	4.42
speedup	—	3.0%	226%

(B)

Figure 6. Results for volume rendering. Times are in millions of cycles; the average load time is in cycles; TLB misses is user data TLB misses in millions. In (A), the rays follow the memory layout of the image; in (B), they are perpendicular to the memory layout.

the x - y planes. These two sets of data points represent the extremes in memory performance for a given plane.

In our data, the measurements labeled “Original” are for a ray tracer that uses macro-cells to reduce the number of voxels traversed, but that does not tile the volume. The macro-cells are $4 \times 4 \times 4$ voxels in size. The results labeled “Indirection” use macro-cells and address voxels through an indirection vector. Finally, the results labeled “Impulse” perform the indirection lookup at the memory controller.

Even in (A), where the rays are parallel to the array layout, Impulse delivers a substantial performance gain. Precomputing the voxel offsets reduces execution time by approximately 70 million cycles. Without Impulse, precomputing the voxel offsets improves execution time by approximately 15 million cycles, despite an increase in average load time, because it reduces the total number of instructions in the key inner loop. However, the speedup achieved by employing an indirection vector is much smaller without Impulse, because the indirection vector itself must be loaded from memory and the elements must be individually referenced. Using Impulse, the accesses to the indirection vector are performed only within the memory controller, which hides their access latencies.

In Table (B), where the rays are perpendicular to the voxel array layout, Impulse yields a 226% performance gain. Reducing the number of TLB misses (fewer pages are addressed) saves approximately 145 million cycles, and

	Base	Padded	Tiled	Tiled, padded	Impulse
Time	218	228	106	93.5	78.7
L1 hit ratio	77.2%	78.0%	89.1%	91.8%	96.6%
L2 hit ratio	5.0%	5.5	4.8%	3.9%	0.6%
avg load	4.8	4.8	2.0	1.97	2.2
TLB misses	3.62	4.60	.92	.92	.03
speedup	—	-4.3%	106%	133%	177%

Table 1. Simulation results for performing a 3-shear rotation of a 1k-by-1k 24-bit color image. Times are in millions of cycles. The L1 hit ratio is the number of L1 hits/total loads; the L2 hit ratio is the number of L2 hits/total loads; the average load time is the average number of cycles that a load takes; TLB misses is the number of user data misses in millions.

increasing the cache hit ratio (no useless voxels are loaded into the cache) saves the remaining 500 million cycles.

5.2 Three-Shear Image Rotation

Table 1 illustrates performance results for rotating a color image clockwise through one radian. The image contains 24 bits of color information, as in a “.ppm” file. We measure three versions of this benchmark: the original version, adapted from Wolberg [26]; a hand-tiled version of the code, in which the vertical shear’s traversal is blocked; and a version adapted to Impulse, in which the matrices are transposed at the memory controller. The Impulse version requires that each pixel be padded to four bytes, since Impulse operates on power-of-two object sizes. To quantify the performance effect of padding, we measure the results for the non-Impulse versions of the code using four-byte pixels.

The performance differences among the five versions are entirely due to cycles saved during the vertical shear. The horizontal shears exhibit good memory behavior (in row-major layout), and so are not a performance bottleneck. Impulse increases the cache hit rate from roughly 78% to 97% and reduces the number of TLB misses by two orders of magnitude. The latter effect saves approximately 75 million cycles of kernel time, constituting most of Impulse’s benefit.

The tiled version walks through all columns 32 pixels at a time, which yields a hit rate higher than the original program’s, but lower than Impulse’s. The tiles in the source matrix are sheared in the destination matrix, so even though cache performance for the source image is nearly perfect, it is poor for the destination image. For the same reason, the decrease in TLB misses for the tiled code is not as great

	Tiled	Impulse
Time	317	219
L1 hit ratio	98.9%	99.8%
L2 hit ratio	0.9%	0.2%
avg load time	1.2	1.0
TLB misses	3.51	0.00
speedup	—	44.7%

Table 2. Simulated results for image filtering with various memory system configurations. Times are in millions of cycles. The L1 hit ratio is the number of L1 hits/total loads; the L2 hit ratio is the number of L2 hits/total loads; the average load time is the average number of cycles that a load takes; TLB misses measure the number of user data misses in millions; speedup is the “Original, no prefetch” time divided by the time for the system being compared, minus one.

as for the Impulse code. Finally, prefetching is not as effective with tiling, because prefetching moves across rows, whereas the traversal moves down columns.

The Impulse code requires 33% more memory to store a 24-bit color image. We also measured the performance impact of using padded 32-bit pixels with each of the non-Impulse codes. In the original program, padding causes each cache line fetch to load useless pad bytes, further degrading the performance of a program that was already memory-bound. In contrast, for the tiled program, the increase in memory traffic is subsumed by the reduction in load, shift, and mask operations: manipulating word-aligned pixels is faster than manipulating byte-aligned pixels, thus padding improves performance. The padded, tiled version of the rotation code is still slower than Impulse. The tiled version of the shear uses more cycles to recompute (or save and restore) each column’s displacement when traversing each tile. For our input image, this displacement is computed $\frac{1024}{32} = 32$ times, since the column length is 1024 and the tile height is 32. In contrast, the Impulse code only computes each column’s displacement once, since each column is visited only once.

5.3 Image Filtering

Table 2 presents the results of order-121 binomial filter on a 32×1024 color image. As with volume rendering, the Impulse version of the code pads each pixel to four bytes. Performance differences between the hand-tiled and Impulse versions of the algorithm arise from the vertical pass over the data. The tiled version suffers 15 times as many L1 cache misses and 200 times as many TLB faults. It spends 16 million user cycles blocked waiting for memory

and 68 million total cycles in the kernel (mostly handling TLB misses), for a memory system overhead in excess of one-quarter of the total execution time. In contrast, the Impulse version of the algorithm spends only 3.4 million cycles waiting for memory and 1 million cycles performing kernel operations — a negligible memory overhead compared to the total running time of 219 million cycles.

Although both versions of the algorithm touch each data element the same number of times, Impulse improves the memory behavior of the image filtering code in two ways. When the original algorithm performs the vertical filtering pass, it touches more pages per iteration than the processor TLB can hold, which yields the high kernel overhead observed in these runs. Image cache lines conflicting within the L1 cache further degrade performance. Since the Impulse version of the code accesses (what appear to the processor to be) contiguous addresses, it suffers very few TLB faults and has near-perfect temporal and spatial locality in the L1 cache.

6. Related Work

The recent literature abounds with arguments that traditional caching cannot bridge the growing processor-memory performance gap. Burger et al. [2] demonstrate that dynamic caching uses memory inefficiently. They found that the percentage of live data in the cache is generally under 20% and that current cache sizes are often thousands of times larger than an optimal cache. Impulse not only improves cache efficiency for many data structures that traditionally have had extremely poor cache performance, but it improves traffic efficiency by buffering prefetched data within the memory controller until requested.

Many others have investigated memory hierarchies that target streaming applications. Most of these use non-programmable buffers to perform hardware prefetching of consecutive cache lines, such as those introduced by Jouppi [12]. Although such stream buffers are intended to be transparent to the programmer, careful coding is required to ensure good performance. Palacharla and Kessler [17] investigate the use of similar stream buffers to replace the L2 cache (as in the Cray T3E multiprocessor [21]) and Farkas et al. [9] identify performance trends and relationships among the various components of the memory hierarchy in a dynamically scheduled processor. Both studies find that dynamically reactive stream buffers can yield significant performance increases, but prefetching cache lines speculatively brings unneeded data into the processor cache(s), which increases memory system bandwidth requirements and decreases effective bandwidth. Even when a filter is added to reduce the number of false stream accesses, Palacharla and Kessler found that their stream buffers required 25% or more extra bandwidth for more than half of

their scientific benchmarks, and as much as 45-50% in a few instances. Farkas et al. mitigate this problem with an incremental prefetching technique that reduces stream buffer bandwidth consumption by 50%.

In contrast, prefetching within the memory controller itself never wastes system bus bandwidth loading unneeded data onto the processor chip. McKee et al.'s Stream Memory Controller [14] combines programmable stream buffers and prefetching within the memory controller with intelligent DRAM scheduling. This system dynamically reorders vector or stream accesses to exploit parallelism in the memory system and to exploit locality of reference among the DRAM page buffers. In the same vein, Corbal et al. [6] propose a Command Vector Memory System that exploits parallelism and locality of reference to improve effective bandwidth for vector accesses on out-of-order vector processors with SDRAM memories.

The Imagine media processor is a stream-based architecture with a bandwidth-efficient stream register file [20]. The streaming model of computation exposes parallelism and locality in applications, which makes such systems an attractive domain for intelligent DRAM scheduling.

Like Impulse, the Cray T3E multiprocessor [21] allows sparse data to be gathered outside the processor and then transferred on-chip in "dense" bus transactions. The memory interface of the DEC 21164 microprocessor is augmented with a large set of explicitly managed, memory-mapped, external registers. These *E-registers* can perform vector *get* or *put* operations to transfer eight words of arbitrary stride between nodes. The bus interface allows up to four properly aligned *get/put* commands each two-cycle bus transaction, thus the operations can be highly pipelined. Gather operations use strided vector gets to load contiguous *E-registers*, which are then loaded "broadside" in bus-width increments into processor registers. This approach avoids address remapping, but since *E-register* data is accessed via I/O space loads, vector elements gathered this way must be copied to local memory to be cacheable.

Other researchers have explored ways to give software control over the memory hierarchy. Yamada et al. [27] describe memory hierarchy and instruction set changes to support combined data relocation and prefetching into L1 cache. While such an approach yields improved L1 caching efficiency, performing relocation at the processor saves no bus bandwidth — since transformations are performed on virtual addresses, the performance of physically indexed caches is unaffected. The Morph project [28] proposes integrating programmable logic throughout the memory hierarchy, giving applications control over memory interleaving, cache organization, and caching policies. Morph, like Impulse, supports application-specific locality optimizations, but requires radical changes to conventional processor and cache designs.

Finally, researchers have proposed attacking the memory bottleneck from the opposite direction—by selectively moving computation to the memory, rather than moving data to the processor. For example, integrated architectures such as RADram [15] and IRAM [19] integrate some form of processor into the memory system. These architectures are adept at handling dense streaming or vector-style applications when all data reside within a single DRAM chip. However, once the data cross chip boundaries, these systems effectively become a specialized form of distributed memory multiprocessor, with all of the attendant complexities. While processor-in-memory architectures are clearly useful in special-purpose domains, much research remains to determine the extent to which they will benefit a general-purpose compute environment.

7. Conclusions

Image processing is an increasingly important application domain from a commercial standpoint, but it has traditionally not been studied closely by computer architects. In this paper, we have studied representative examples of three major classes of image processing algorithms (volume rendering, image warping, and image filtering). The tremendous pressure these algorithms place on conventional systems has led to the development of a wide range of special-purpose image processing systems. By allowing the applications to control how the image streams are mapped into the processor's cache(s) and TLB, the Impulse memory system achieves performance improvements ranging from 40% to 226% on an otherwise conventional system.

More work remains in terms of developing memory system support for high-performance image processing. We recently initiated a collaboration with the image processing, vision, and visualization groups at Utah, and are studying other classes of image processing algorithms whose performance should improve on Impulse. Texture mapping, image extraction, image encoding-decoding, and image compression-decompression all appear to be attractive candidates for optimization.

We are also studying other application areas. One particularly attractive class of applications consists of vectorized codes. A system that combines an Impulse memory controller (which can prefetch and pack sparse data into dense "vectors") with the large number of functional units in modern superscalar processors begins to resemble a vector machine in its capabilities. Just as a Cray T-90's vector load-store unit exploits the bandwidth of the Cray memory interconnect to gather data into its vector registers, we expect that an Impulse memory controller will be able to exploit the bandwidth of a heavily banked DRAM system to gather vectors that can be loaded into the L1 cache.

8. Acknowledgments

Peter Pike Sloan, Gordon Kindlmann, Peter Shirley, Steven Parker, and Brian Smits contributed to our image processing algorithms. Bob Devine commented on drafts of this paper, and John McCorquodale helped get the figures right.

References

- [1] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera computer system. In *Proceedings of the 1990 International Conference on Supercomputing*, pages 1–6, Sept. 1990.
- [2] D. Burger, J. Goodman, and A. Kagi. Memory bandwidth limitations of future microprocessors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 78–89, May 1996.
- [3] J. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, E. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama. Impulse: Building a smarter memory controller. In *Proceedings of the Fifth Annual Symposium on High Performance Computer Architecture*, pages 70–79, Jan. 1999.
- [4] E. Catmull and A. Smith. 3-D transformations of images in scanline order. *Computer Graphics*, 15(3):279–285, 1980.
- [5] T.-F. Chen and J.-L. Baer. Effective hardware-based data prefetching for high performance multiprocessors. *IEEE Transactions on Computers*, 44(5):609–623, May 1995.
- [6] J. Corbal, R. Espasa, and M. Valero. Command vector memory systems: High performance at low cost. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, pages 68–77, Oct. 1998.
- [7] F. Dahlgren, M. Dubois, and P. Stenstrom. Sequential hardware prefetching in shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 6(7):733–746, July 1995.
- [8] S. Eggers, et al. Simultaneous multithreading: A platform for next-generation processors. *IEEE Micro*, pages 12–19, Oct. 1997.
- [9] K. Farkas, P. Chow, N. Jouppi, and Z. Vranesic. Memory-system design considerations for dynamically-scheduled processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 133–143, June 1997.
- [10] J. Gomes and L. Velho. *Image Processing for Computer Graphics*. Springer-Verlag, 1997.
- [11] T. Hotchkiss, N. Marschke, and R. McClosky. A new memory system design for commercial and technical computing products. *Hewlett-Packard Journal*, 47(1):44–51, Feb. 1996.
- [12] N. Jouppi. Improving direct-mapped cache performance by the addition of a small fully associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373, May 1990.
- [13] P. Lacroute. *Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation*. PhD thesis, Stanford University, Stanford, CA, Sept. 1989.
- [14] S. A. McKee et al. Design and evaluation of dynamic access ordering hardware. In *Proceedings of the 1996 International Conference on Supercomputing*, May 1996.
- [15] M. Oskin, F. Chong, and T. Sherwood. Active pages: A model of computation for intelligent memory. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 192–203, June 1998.
- [16] V. S. Pai, P. Ranganathan, and S. V. Adve. The impact of instruction-level parallelism on multiprocessor performance and simulation methodology. In *Proceedings of the Third Annual Symposium on High Performance Computer Architecture*, pages 72–83, Feb. 1997.
- [17] S. Palacharla and R. Kessler. Evaluating stream buffers as a secondary cache replacement. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 24–33, May 1994.
- [18] S. Parker, P. Shirley, Y. Livnat, C. Hansen, and P.-P. Sloan. Interactive ray tracing for isosurface rendering. *Proceedings of the Visualization '98 Conference*, Oct. 1998.
- [19] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keaton, C. Kozyrakis, R. Thomas, and K. Yelick. A case for Intelligent RAM: IRAM. *IEEE Micro*, 17(2), Apr. 1997.
- [20] S. Rixner, W. Dally, U. Kapasi, B. Khailany, A. Lopez-Lagunas, P. Mattson, and D. Owens. A bandwidth-efficient architecture for media processing. In *Proceedings of IEEE/ACM 31st International Symposium on Microarchitecture*, Dec. 1998.
- [21] S. Scott. Synchronization and communication in the T3E multiprocessor. In *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems*, Oct. 1996.
- [22] A. Sez nec. A case for two-way skewed-associative caches. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 169–178, May 1993.
- [23] L. Stoller, R. Kuramkote, and M. Swanson. PAINT: PA instruction set interpreter. Technical Report UUCS-96-009, University of Utah Department of Computer Science, Sept. 1996.
- [24] M. Swanson, L. Stoller, and J. Carter. Increasing TLB reach using superpages backed by shadow memory. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 204–213, June 1998.
- [25] J.-Y. Tsai, Z. Jiang, E. Ness, and P.-C. Yew. Performance study of a concurrent multithreaded processor. In *Proceedings of the Fourth Annual Symposium on High Performance Computer Architecture*, Feb. 1998.
- [26] G. Wolberg. *Digital Image Warping*. IEEE Computer Society Press, 1990.
- [27] Y. Yamada. *Data Relocation and Prefetching in Programs with Large Data Sets*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana, IL, 1995.
- [28] X. Zhang, A. Dasdan, M. Schulz, R. Gupta, and A. Chien. Architectural adaptation for application-specific locality optimizations. In *Proceedings of the 1997 IEEE International Conference on Computer Design*, 1997.