The Impulse Memory Controller

Lixin Zhang, Zhen Fang, Mike Parker, Binu K. Mathew, Lambert Schaelicke,
John B. Carter, Wilson C. Hsieh, Sally A. McKee

September 24, 2001

Abstract

Impulse is a memory system architecture that adds an optional level of address indirection at the memory controller. Applications can use this level of indirection to remap their data structures in memory. As a result, they can control how their data is accessed and cached, which can improve cache and bus utilization. The Impulse design does not require any modification to processor, cache, or bus designs, since all the functionality resides at the memory controller. As a result, Impulse can be adopted in conventional systems without major system changes.

We describe the design of the Impulse architecture and show how an Impulse memory system can be used in a variety of ways to improve the performance of memory-bound applications. Impulse can be used to dynamically create superpages cheaply, to dynamically recolor physical pages, to perform strided fetches, and to perform gathers and scatters through indirection vectors. Our performance results demonstrate the effectiveness of these optimizations in a variety of scenarios. Using Impulse can speed up a range of applications from 20% to over a factor of 5. Alternatively, Impulse can be used by the OS for dynamic superpage creation; the best policy for creating superpages using Impulse outperforms previously known superpage creation policies.

Keywords: computer architecture, memory systems

1 Introduction

Since 1987, microprocessor performance has improved at a rate of 55% per year; in contrast, DRAM latencies have improved by only 7% per year, and DRAM bandwidths by only 15-20% per year [17]. The result is that the relative performance impact of memory accesses continues to grow. In addition, as instruction issue rates increase, the demand for memory bandwidth grows at least proportionately, possibly even superlinearly [8, 19]. Many important applications (e.g., sparse matrix, database, signal processing, multimedia, and CAD applications) do not exhibit sufficient

Contact information: Prof. John Carter, School of Computing, 50 S Central Campus Drive, Room 3190, University of Utah, SLC, UT 84112-9205. retrac@cs.utah.edu. Voice: 801-585-5474. Fax: 801-581-5843.

locality of reference to make effective use of the on-chip cache hierarchy. For such applications, the growing processor/memory performance gap makes it more and more difficult to effectively exploit the tremendous processing power of modern microprocessors. In the Impulse project, we are attacking this problem by designing and building a memory controller that is more powerful than conventional ones.

Impulse introduces an optional level of address translation at the memory controller. The key insight that this feature exploits is that "unused" physical addresses can be translated to "real" physical addresses at the memory controller. An unused physical address is a legitimate address that is not backed by DRAM. For example, in a conventional system with 4GB of physical address space and only 1GB of installed DRAM, 3GB of the physical address space remains unused. We call these unused addresses *shadow addresses*, and they constitute a *shadow address space* that the Impulse controller maps to physical memory. By giving applications control (mediated by the OS) over the use of shadow addresses, Impulse supports application-specific optimizations that restructure data. Using Impulse requires software modifications to applications (or compilers) and operating systems, but requires no hardware modifications to processors, caches, or buses.

As a simple example of how Impulse's memory remapping can be used, consider a program that accesses the diagonal elements of a large, dense matrix `A`. The physical layout of part of the data structure `A` is shown on the right-hand side of Figure 1. On a conventional memory system, each time the processor accesses a new diagonal element (`A[i][i]`), it requests a full cache line of contiguous physical memory (typically 32–128 bytes of data on modern systems). The program accesses only a single word of each of these cache lines. Such an access is shown in the top half of Figure 1.

Using Impulse, an application can configure the memory controller to export a dense, shadow-space alias that contains just the diagonal elements, and can have the OS map a new set of virtual addresses to this shadow memory. The application can then access the diagonal elements via the new virtual alias. Such an access is shown in the bottom half of Figure 1.

Remapping the array diagonal to a dense alias yields several performance benefits. First, the program enjoys a higher cache hit rate because several diagonal elements are loaded into the caches at once. Second, the program consumes less bus bandwidth because non-diagonal elements are not sent over the bus. Third, the program makes more effective use of cache space because the diagonal elements now have contiguous shadow addresses. In general, Impulse's flexibility allows applications to customize addressing to fit their needs.

Section 2 describes the Impulse architecture. It describes the organization of the memory controller itself, as well as the system call interface that applications use to control it. The operating system must mediate use of the memory controller to prevent applications from accessing each other's physical memory.

Section 3 describes the types of optimizations that Impulse supports. Many of the optimizations that we describe are not new, but Impulse is the first system that provides hardware support for them all in general-purpose computer systems. The optimizations include transposing matrices in memory without copying, creating superpages without copying, and doing scatter/gather through an indirection vector. Section 4 presents the results of a simulation study of Impulse, and shows that these optimizations can benefit a wide range of applications. Various applications see speedups ranging from 20% to a factor of 5. OS policies for dynamic superpage creation using Impulse have around 20% better speedup than those from prior work.

Section 5 describes related work. A great deal of work has been done in the compiler and operating systems communities on related optimizations. The contribution of Impulse is that it provides hardware support for many optimizations that previously had to be performed purely in software. As a result, the tradeoffs for performing these optimizations are different. Section 6 summarizes our conclusions, and describes future work.

2 Impulse Architecture

Impulse expands the traditional virtual memory hierarchy by adding address translation hardware to the memory controller. This optional extra level of remapping is enabled by the fact that not all physical addresses in a traditional virtual memory system typically map to valid memory locations. The unused physical addresses constitute a *shadow address space*. The technology trend is putting more and more bits into physical addresses. For example, more and more 64-bit systems are coming out. One result of this trend is that the shadow space is getting larger and larger. Impulse allows software to configure the memory controller to interpret shadow addresses. Virtualizing unused physical addresses in this way can improve the efficiency of on-chip caches and TLBs, since hot data can be dynamically segregated from cold data.

Data items whose physical DRAM addresses are not contiguous can be mapped to contiguous shadow addresses. In response to a cache line fetch of a shadow address, the memory controller fetches and compacts sparse data into dense cache lines before returning the data to the processor. To determine where the data associated with these compacted shadow cache lines reside in physical memory, Impulse first recovers their offsets within the original data structure, which we call *pseudo-virtual addresses*. It then translates these pseudo-virtual addresses to physical DRAM addresses. The pseudo-virtual address space page layout mirrors the virtual address space, allowing Impulse to remap data structures that lie across non-contiguous physical pages. The shadow$\rightarrow$pseudo-virtual$\rightarrow$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.

2.1 Software Interface and OS Support

To exploit Impulse, appropriate system calls must be inserted into the application code to configure the memory controller. The Architecture and Language Implementation group at the University of Massachusetts is developing compiler technology for Impulse. In response to an Impulse system call, the OS allocates a range of contiguous virtual addresses large enough to map the elements of the new (synthetic) data structure. The OS then maps the new data structure through shadow memory to the corresponding physical data elements. It does so by allocating a contiguous range of shadow addresses and downloading two pieces of information to the MMC: (i) a function that the MMC should use to perform the mapping from shadow to pseudo-virtual space and (ii) a set of page table entries that can be used to translate pseudo-virtual 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. Upon seeing an access to a shadow address in the synthetic diagonal data structure, the memory controller gathers the corresponding diagonal elements from the original array, packs them into a dense cache line, and returns this cache line to the processor. The OS interface allows alignment and offset characteristics of the remapped data structure to be specified, which gives the application some control over L1 cache behavior. In the current Impulse design, coherence is maintained in software: the OS or the application programmer must keep aliased data consistent by explicitly flushing the cache.

2.2 Hardware Organization

The organization of the Impulse controller architecture is depicted in Figure 3. The critical component of the Impulse MMC is the *shadow engine*, which processes all shadow accesses. The shadow engine contains a small SRAM *Assembly Buffer*, which is used to scatter/gather cache lines in the shadow address space; some *shadow descriptors* to store remapping configuration information; an

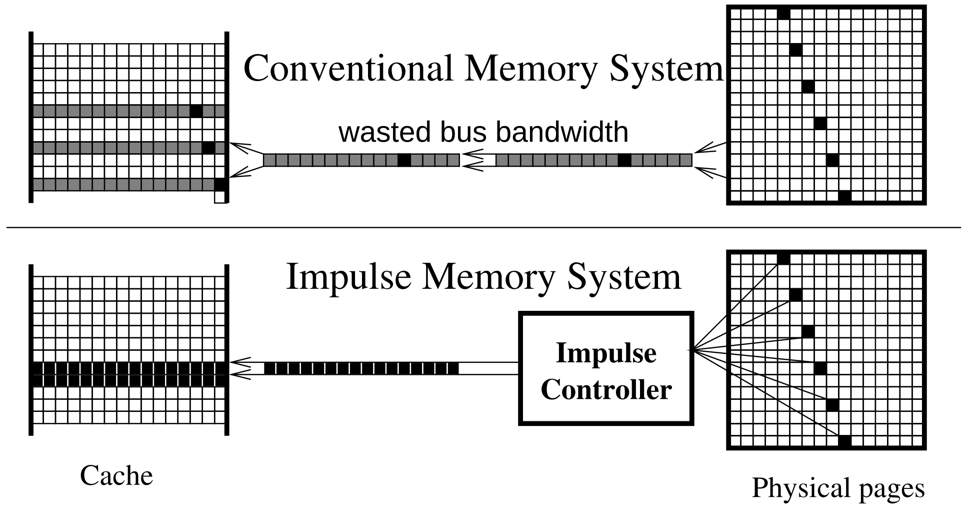


Figure 1: Using Impulse to remap the diagonal of a dense matrix into a dense cache line. The black boxes represent data on the diagonal, whereas the gray boxes represent non-diagonal data.

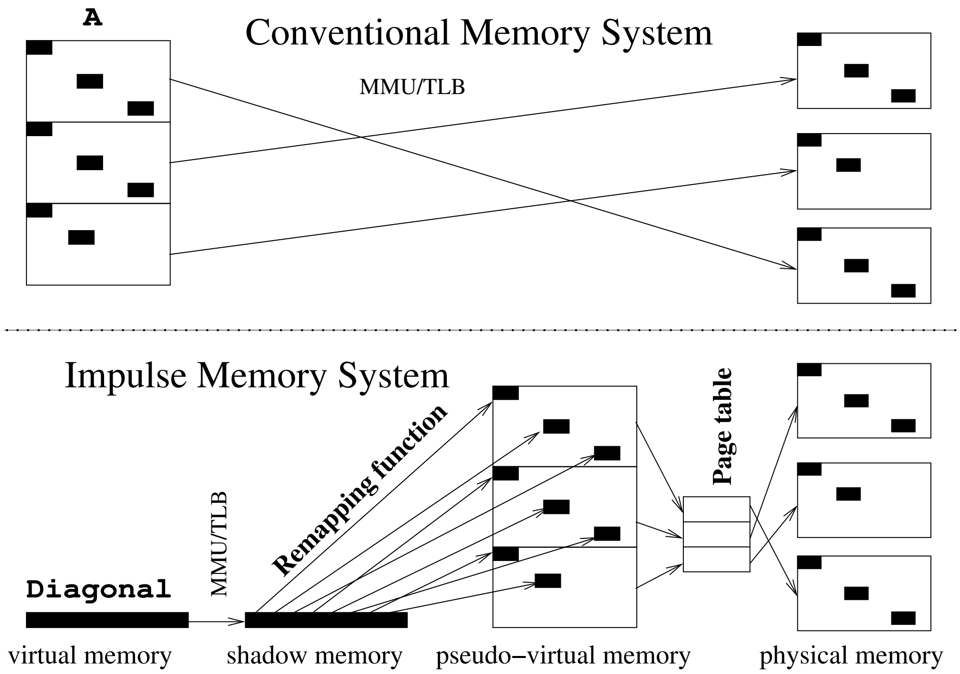


Figure 2: Accessing the (sparse) diagonal elements of an array via a dense `diagonal` variable in Impulse.

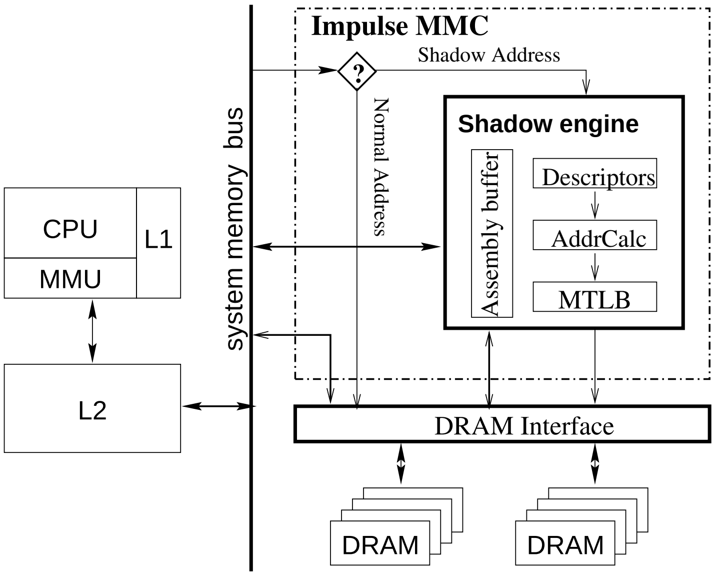


Figure 3: Impulse memory controller organization.

ALU unit *(AddrCalc)* to translate shadow addresses to pseudo-virtual addresses; and a *Memory Controller Translation Lookaside Buffer* (MTLB) to cache recently used translations from pseudo-virtual addresses to physical addresses. The shadow engine contains eight shadow descriptors, each of which is is capable of saving all configuration settings for one remapping. All shadow descriptors share the same ALU unit and the same MTLB.

Since the extra level of address translation is optional, addresses appearing on the memory bus may be in the physical (backed by DRAM) or shadow memory spaces. Valid physical addresses pass untranslated to the DRAM interface.

Shadow addresses must be converted to physical addresses before being presented to the DRAM. To do so, the shadow engine first determines which shadow descriptor to use and passes its contents to the AddrCalc unit. The output of the AddrCalc will be a series of offsets for the individual sparse elements that need to be fetched. These offsets are passed through the MTLB to compute the physical addresses that need to be fetched. To hide some of the latency of fetching remapped data, each shadow descriptor can be configured to prefetch the remapped cache line following the currently accessed one.

Depending on how Impulse is used to access a particular data structure, the shadow address translations can take three forms: 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 or to construct superpages dynamically. *Strided mapping* creates dense cache lines from array elements that are not contiguous. The mapping function maps an address *soffset* in shadow space to pseudo-virtual address *pvaddr* + *stride* $\times$ *soffset*, where *pvaddr* is the starting address of the data structure's pseudo-virtual image. *pvaddr* is assigned by the OS upon configuration. *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]*.

3 Impulse Optimizations

Impulse remappings can be used to enable a wide variety of optimizations. We first describe how Impulse's ability to pack data into cache lines (either using stride or scatter/gather remapping) can be used. We examine two scientific application kernels—sparse matrix-vector multiply (SMVP) and dense matrix-matrix product (DMMP)—and three image processing algorithms—image filtering, image rotation, and ray tracing. We then show how Impulse's ability to remap pages can be used to automatically improve TLB behavior through dynamic superpage creation. Some of these results have been published in prior conference papers [9, 13, 44, 49].

3.1 Sparse Matrix-Vector Product

Sparse matrix-vector product (SMVP) is an irregular computational kernel that is critical to many large scientific algorithms. For example, most of the time in conjugate gradient [3] or in the Spark98 earthquake simulations [33] is spent performing SMVP.

To avoid wasting memory, sparse matrices are generally compacted so that only non-zero elements and corresponding index arrays are stored. For example, the Class A input matrix for the NAS conjugate gradient kernel (CG-A) is 14,000 by 14,000, and contains only 1.85 million non-zeroes. Although sparse encodings save tremendous amounts of memory, sparse matrix codes tend to suffer from poor memory performance because data must be accessed through indirection vectors. CG-A on an SGI Origin 2000 processor (which has a 2-way, 32K L1 cache and a 2-way, 4MB L2 cache) exhibits L1 and L2 cache hit rates of only 63% and 92%, respectively.

The inner loop of the sparse matrix-vector product in CG is roughly:

```
for i := 1 to n do
  sum := 0
  for j := ROWS[i] to ROWS[i+1]-1 do
    sum += DATA[j]*x[COLUMN[j]]
```

```
    b[i] := sum
```

The code and data structures for SMVP are illustrated in Figure 4. Each iteration multiplies a row of the sparse matrix `A` with the dense vector `x`. The accesses to `x` are indirect (via the `COLUMN` index vector) and sparse, making this code perform poorly on conventional memory systems. Whenever `x` is accessed, a conventional memory system fetches a cache line of data, of which only one element is used. The large sizes of `x`, `COLUMN`, and `DATA` and the sparse nature of accesses to `x` inhibit data reuse in the L1 cache. Each element of `COLUMN` or `DATA` is used only once, and almost every access to `x` results in an L1 cache miss. A large L2 cache can enable reuse of `x`, if physical data layouts can be managed to prevent L2 cache conflicts between `A` and `x`. Unfortunately, conventional systems do not typically provide mechanisms for managing physical layout.

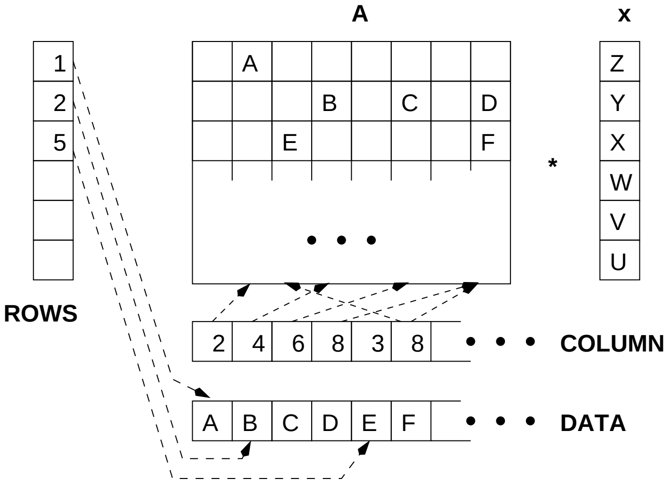
The Impulse memory controller supports scatter/gather of physical addresses through indirection vectors. Vector machines such as the CDC STAR-100 [18] provided scatter/gather capabilities in hardware within the processor. Impulse allows conventional CPUs to take advantage of scatter/gather functionality by implementing the operations at the memory, which reduces memory traffic over the bus.

To exploit Impulse, CG's SMVP code can be modified as follows:

```
// x'[k] <- x[COLUMN[k]]
impulse_remap(x, x', N, COLUMN, INDIRECT, ...)
for i := 1 to n do
  sum := 0
  for j := ROWS[i] to ROWS[i+1]-1 do
    sum += DATA[j] * x'[j]
  b[i] := sum
```

The `impulse_remap` operation asks the operating system to 1) allocate a new region of shadow space, 2) map `x'` to that shadow region, and 3) instruct the memory controller to map the elements of the shadow region `x'[k]` to the physical memory for `x[COLUMN[k]]`. After the remapped array has been set up, the code accesses the remapped version of the gathered structure (`x'`) rather than the original structure (`x`).

This optimization improves the performance of SMVP in two ways. First, spatial locality is improved in the L1 cache. Since the memory controller packs the gathered elements into cache lines, each cache line contain 100% useful data, rather than only one useful element. Second,




```
for i := 1 to n do
  sum := 0
  for j := ROWS[i] to ROWS[i+1]-1 do
    sum += DATA[j] * x[COLUMN[j]];
  b[i] := sum;
```

Figure 4: Conjugate gradient's sparse matrix-vector product. The matrix A is encoded using three dense arrays: `DATA`, `ROWS`, and `COLUMN`. The contents of A are in `DATA`. `ROWS[i]` indicates where the i^{th} row begins in `DATA`. `COLUMN[i]` indicates which column of A the element stored in `DATA[i]` comes from.

the processor issues fewer memory instructions, since the read of the indirection vector `COLUMN` occurs at the memory controller. Note that the use of scatter/gather at the memory controller reduces temporal locality in the L2 cache. The remapped elements of `x'` cannot be reused, since all of the elements have different addresses.

An alternative to scatter/gather is dynamic physical page recoloring through direct remapping of physical pages. Physical page recoloring changes the physical addresses of pages so that reusable data is mapped to a different part of a physically-addressed cache than non-reused data. By performing page recoloring, conflict misses can be eliminated. On a conventional machine, physical page recoloring is expensive: the only way to change the physical address of data is to copy the data between physical pages. Impulse allows physical pages to be recolored *without copying*. Virtual page recoloring has been explored by other authors [6].

For SMVP, the `x` vector is reused within an iteration, while elements of the `DATA`, `ROW`, and `COLUMN` vectors are used only once in each iteration. As an alternative to scatter/gather of `x` at the memory controller, Impulse can be used to physically recolor pages so that `x` does not conflict with the other data structures in the L2 cache. For example, in the CG-A benchmark, `x` is over 100K bytes: it would not fit in most L1 caches, but would fit in many L2 caches.

Impulse can remap `x` to pages that occupy most of the physically-indexed L2 cache, and can remap DATA, ROWS, and COLUMNS to a small number of pages that do not conflict with x. In our experiments, we color the vectors x, DATA, and COLUMN so that they do not conflict in the L2 cache. The multiplicand vector x is heavily reused, so we color it to occupy the first half of the L2 cache. To keep the large DATA and COLUMN structures from conflicting, we divide the second half of the L2 cache into two quarters, and then color DATA and COLUMN so they each occupy one quarter of the cache. In effect, we use pieces of the L2 cache as a set of virtual stream buffers [29] for DATA, ROWS, and COLUMNS.

3.2 Tiled Matrix Algorithms

Dense matrix algorithms form an important class of scientific kernels. For example, LU decomposition and dense Cholesky factorization are dense matrix computational kernels. Such algorithms are *tiled* (or *blocked*) to increase their efficiency. That is, the iterations of tiled algorithms are reordered to improve their memory performance. The difficulty with using tiled algorithms lies in choosing an appropriate tile size [27]. Because tiles are non-contiguous in the virtual address space, it is difficult to keep them from conflicting with each other or with themselves in cache. To avoid conflicts, either tile sizes must be kept small, which makes inefficient use of the cache, or tiles must be copied into non-conflicting regions of memory, which is expensive.

Impulse provides an alternative method of removing cache conflicts for tiles. We use the simplest tiled algorithm, dense matrix-matrix product (DMMP), as an example of how Impulse can improve the behavior of tiled matrix algorithms. Assume that we are computing $C = A \times B$, we want to keep the current tile of the C matrix in the L1 cache as we compute it. In addition, since the same row of the A matrix is used multiple times to compute a row of the C matrix, we would like to keep the active row of A in the L2 cache.

Impulse allows base-stride remapping of the tiles from non-contiguous portions of memory into contiguous tiles of shadow space. As a result, Impulse makes it easy for the OS to virtually remap the tiles, since the physical footprint of a tile will match its size. If we use the OS to remap the virtual address of a matrix tile to its new shadow alias, we can then eliminate interference in a virtually-indexed L1 cache. First, we divide the L1 cache into three segments. In each segment we keep a tile: the current output tile from C, and the input tiles from A and B. When we finish with one tile, we use Impulse to remap the virtual tile to the next physical tile. To maintain cache consistency, we must purge the A and B tiles and flush the C tiles from the caches whenever they are remapped. As Section 4.1.2 shows, these costs are minor.

3.3 Image Filtering

Image filtering applies a numerical filter function to an image to modify its appearance. 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 of filters, *binomial filters* [15], in which each pixel in the output image is computed by applying a two-dimensional "mask" to the input image. Binomial filtering is computationally similar 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 $\frac{36}{256} * (i, j) + \frac{24}{256} * (i - 1, j) + \frac{24}{256} * (i + 1, j) + ...$ 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, linear 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.

3.4 Image Rotation

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 [10]. 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. The three-shear image rotation algorithm is an example of a separable image warp. This algorithm rotates a 2-dimensional image around its center in three stages, each of which performs a "shear" operation on the image, as

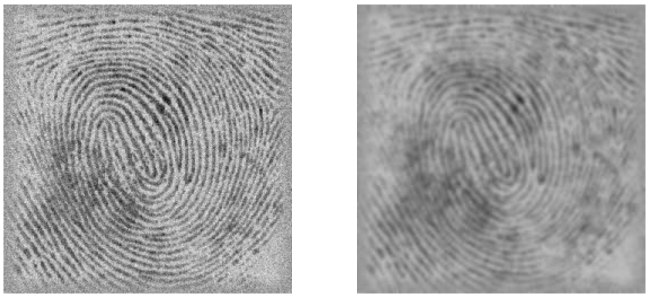


Figure 5: Example of binomial image filtering. The original image is on the left, and the filtered image is on the right.

illustrated in Figure 6. The algorithm is simpler to write, faster to run, and has fewer visual artifacts than a direct rotation. 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 and implement, 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. Impulse improves both cache and TLB performance by transposing the matrix without copying, so that walking along columns in the image is replaced by walking along rows in a transposed matrix.

3.5 Isosurface Rendering Using Ray Tracing

Our isosurface rendering benchmark is based on the technique demonstrated by Parker et al. [37]. This benchmark generates an image of an isosurface in a volume from a specific point of view. In

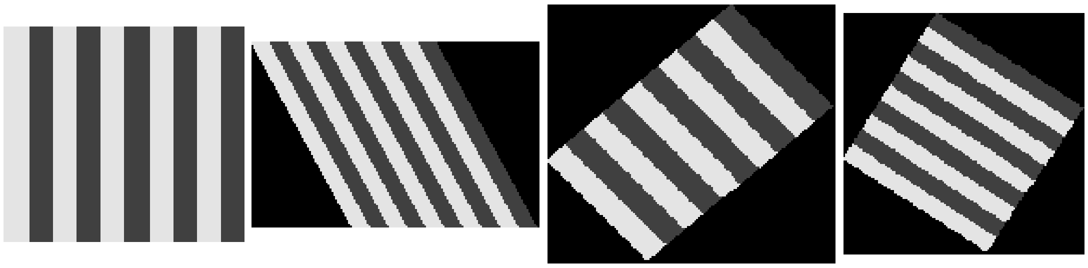


Figure 6: Three-shear rotation of an image counter-clockwise through one radian. The original image (upper left) is first sheared horizontally (upper right). That image is sheared upwards (lower right). The final rotated image (lower left) is generated via one final horizontal shift.

contrast to other volume visualization 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 visualization because it suffers from poor memory behavior when rays do not travel along the direction that data is stored. 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 performance 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 the 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.

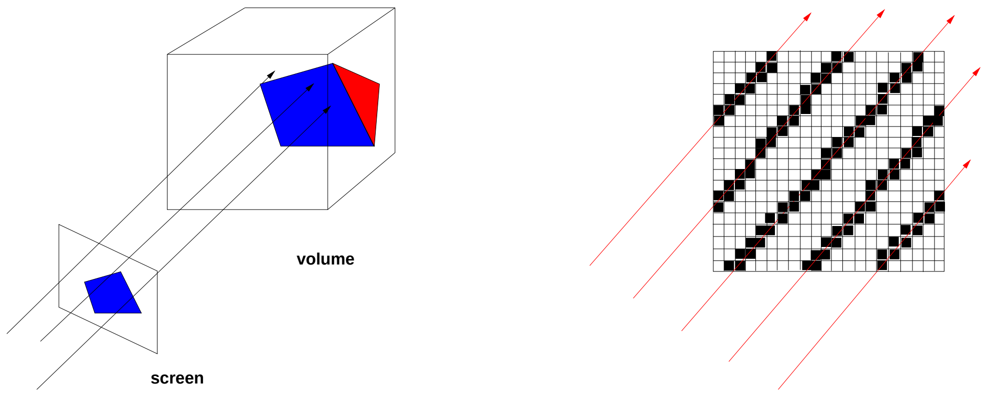


Figure 7: Isosurface rendering using ray tracing. The picture on the left shows rays perpendicular to the viewing screen being traced through a volume. The one on the right illustrates how each ray visits a sequence of voxels in the volume; Impulse optimizes voxel fetches from memory via indirection vectors representing the voxel sequences for each ray.

In addition, the source code retains its readability and modifiability.

Like many real-world visualization systems, our benchmark uses an orthographic tracer whose rays all intersect the screen surface at right angles, producing 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 7 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.

3.6 Online Superpage Promotion

Impulse can be used to improve TLB performance automatically, by having the operating system automatically create *superpages* dynamically. Superpages are supported by the translation lookaside buffers (TLBs) on almost all modern processors; they are groups of contiguous virtual memory pages that can be mapped with a single TLB entry [12, 30, 43]. Using superpages makes more efficient use of a TLB, but the physical pages that back a superpage must be contiguous and properly aligned. Dynamically coalescing smaller pages into a superpage thus requires that all the pages be be coincidentally adjacent and aligned (which is unlikely), or that they be copied so that they become so. The overhead of promoting superpages by copying includes both direct and indirect costs. The direct costs come from copying the pages and changing the mappings. Indi-

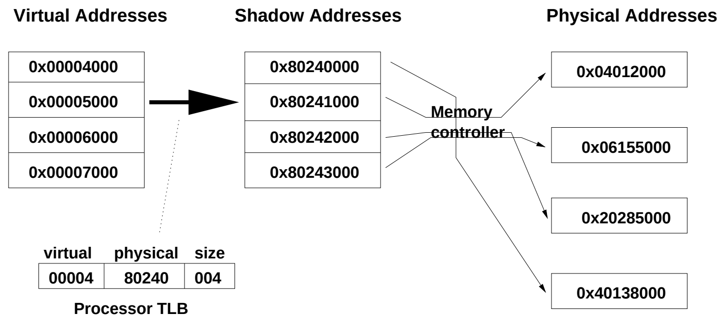


Figure 8: An Example of Creating Superpages Using Shadow Space

rect costs include the increased number of instructions executed on each TLB miss (due to the new decision-making code in the miss handler) and the increased contention in the cache hierarchy (due to the code and data used in the promotion process). When deciding whether to create superpages, all costs must be balanced against the improvements in TLB performance.

Romer *et al.* [40] study several different policies for dynamically creating superpages. Their trace-driven simulations and analysis show how a policy that balances potential performance benefits and promotion overheads can improve performance in some TLB-bound applications by about 50%. Our work extends that of Romer *et al.* by showing how Impulse changes the design of a dynamic superpage promotion policy.

The Impulse memory controller maintains its own page tables for shadow memory mappings. Building superpages from base pages that are not physically contiguous entails simply remapping the virtual pages to properly aligned shadow pages. The memory controller then maps the shadow pages to the original physical pages. The processor's TLB is not affected by the extra level of translation that takes place at the controller.

Figure 8 illustrates how superpage mapping works on Impulse. In this example, the OS has mapped a contiguous 16KB virtual address range to a single shadow superpage at "physical" page frame `0x80240`. When an address in the shadow physical range is placed on the system memory bus, the memory controller detects that this "physical" address needs to be retranslated using its local shadow-to-physical translation tables. In the example in Figure 8, the processor translates an access to virtual address `0x00004080` to shadow physical address `0x80240080`, which the controller, in turn, translates to real physical address `0x40138080`.

4 Performance

We performed a series of detailed simulations to evaluate the performance impact of the optimizations described in Section 3. Our studies use the URSIM [48] execution-driven simulator, which is derived from RSIM [35]. URSIM models a microarchitecture close to MIPS R10000 microprocessor [30] with a 64-entry instruction window. We configured it to issue four instructions per cycle. We model a 64-kilobyte L1 data cache that is non-blocking, write-back, virtually indexed, physically tagged, direct-mapped, and has 32-byte lines. The 512-kilobyte L2 data cache is non-blocking, write-back, physically indexed, physically tagged, two-way associative, and has 128-byte lines. L1 cache hits take one cycle, and L2 cache hits take eight cycles.

URSIM models a split-transaction MIPS R10000 cluster bus with a snoopy coherence protocol. The bus multiplexes addresses and data, is eight bytes wide, has a three-cycle arbitration delay and a one-cycle turn-around time. We model two memory controllers: a conventional high-performance MMC based on the one in the SGI O200 server and the Impulse MMC. The system bus, memory controller, and DRAMs have the same clock rate, which is one third of the CPU clock's. The memory system supports critical word first, i.e., a stalled memory instruction resumes execution after the first quad-word returns. The load latency of the first quad-word is 16 memory cycles.

The unified TLB is single-cycle, fully associative, software-managed, and combined instruction and data. It employs a least-recently-used replacement policy. The base page size is 4096 bytes. Superpages are built in power-of-two multiples of the base page size, and the biggest superpage that the TLB can map contains 2048 base pages. We model a 128-entry TLB.

In the remainder of this section we examine the simulated performance of Impulse on the examples given in Section 3. Our calculation of "L2 cache hit ratio" and "mem (memory) hit ratio" uses the total number of loads executed (not the total number of L2 cache accesses) as the divisor for both ratios. This formulation makes it easier to compare the effects of the L1 and L2 caches on memory accesses: the sum of the L1 cache, L2 cache, and memory hit ratios equals 100%.

4.1 Fine-Grained Remapping

The first set of experiments exploit Impulse's fine-grained remapping capabilities to create synthetic data structures with better locality than in the original programs.

4.1.1 Sparse Matrix-Vector Product

Table 4.1.1 shows how Impulse can be used to improve the performance of the NAS Class A Conjugate Gradient (CG-A) benchmark. The first column gives results from running CG-A on a non-Impulse system. The second and third columns give results from running CG-A on an Impulse system. The second column numbers come from using the Impulse memory controller to perform scatter/gather; the third column numbers come from using it to perform physical page coloring.

On the conventional memory system, CG-A suffers many cache misses: nearly 17% of accesses go to the memory. The inner loop of CG-A is very small, so it can generate cache misses quickly, which leads to there being a large number of cache misses outstanding at any given time. The large number of outstanding memory operations causes heavy contention for the system bus, memory controller, and DRAMs; for the baseline version of CG-A, bus utilization reaches 88.5%. As a result, the average latency of a memory operation reaches 163 cycles for the baseline version of CG-A. This behavior combined with the high cache miss rates causes the average load in CG-A to take 47.6 cycles compared to only 1 cycle for L1 cache hits.

Scatter/gather remapping on CG-A improves performance by over a factor of 3, largely due to the increase in the L1 cache hit ratio and the decrease in the number of loads/stores that go to memory. Each main memory access for the remapped vector `x'` loads the cache with several useful elements from the original vector `x`, which increases the L1 cache hit rate. In other words, retrieving elements from the remapped array `x'` improves the spatial locality of CG-A.

Scatter/gather remapping reduces the total number of loads executed by the program from 493 million from 353 million. In the original program, two loads are issued to compute `x[COLUMN[j]]`. In the scatter/gather version of the program, only one load is issued by the processor, because the load of the indirection vector occurs at the memory controller. This reduction more than compensates for the scatter/gather's increase in the average cost of a load, and accounts for almost one-third of the cycles saved.

To provide another example of how useful Impulse can be, we use it to recolor the pages of the major data structures in CG-A. Page recoloring consistently reduces the cost of memory accesses by eliminating conflict misses in the L2 cache and increasing the L2 cache hit ratio from 19.7% to 22.0%. As a result, fewer loads go to memory, and performance is improved by 17%.

Although page recoloring improves performance on CG-A, it is not nearly as effective as scatter/gather. The difference is primarily because page recoloring does not achieve the two major improvements that scatter/gather provides: improving the locality of CG-A and reducing the number of loads executed. This comparison does not mean that page recoloring is not a useful optimization. Although the speedup for page recoloring on CG-A is substantially less than scatter/gather, page recoloring is more broadly applicable.

4.1.2 Dense Matrix-Matrix Product

This section examines the performance benefits of tile remapping for DMMP, and compares the results to software tile copying. Impulse's alignment restrictions require that remapped tiles be aligned to L2 cache line boundaries, which adds the following constraints to our matrices:

- Tile sizes must be a multiple of a cache line. In our experiments, this size is 128 bytes. This constraint is not overly limiting, especially since it makes the most efficient use of cache space.

- Arrays must be padded so that tiles are aligned to 128 bytes. Compilers can easily support this constraint: similar padding techniques have been explored in the context of vector processors [7].

Table 4.1.2 illustrates the results of our tiling experiments. The baseline is the conventional no-copy tiling. Software tile copying outperforms the baseline code by almost 10%; Impulse tile remapping outperforms it by more than 20%. The improvement in performance for both is primarily due to the difference in cache behavior. Both copying and remapping more than double the L1 cache hit rate, and they reduce the average number of cycles for a load to less than two. Impulse has a higher L1 cache hit ratio than software copying, since copying tiles can incur cache misses: the number of loads that go to memory is reduced by two-thirds. In addition, the cost of copying the tiles is greater than the overhead of using Impulse to remap tiles. As a result, using Impulse provides twice as much speedup.

This comparison between conventional and Impulse copying schemes is conservative for several reasons. Copying works particularly well on DMMP: the number of operations performed on a tile of size $O(n^2)$ is $O(n^3)$, which means the overhead of copying is relatively low. For algorithms where the reuse of the data is lower, the relative overhead of copying is greater. Likewise, as caches (and therefore tiles) grow larger, the cost of copying grows, whereas the (low) cost of Impulse's tile remapping remains fixed. Finally, some authors have found that the performance of copying can vary greatly with matrix size, tile size, and cache size [45], but Impulse should be insensitive to cross-interference between tiles.

	Conventional	Scatter/Gather	Page Coloring
Time	5.48B	1.77B	4.67B
L1 hit ratio	63.4%	77.8%	63.7%
L2 hit ratio	19.7%	15.9%	22.0%
mem hit ratio	16.9%	6.3%	14.3%
avg load time	47.6	23.2	38.7
TLB misses	1.01M	1.05M	0.75M
loads	493M	353M	493M
speedup	—	3.10	1.17

Table 1: Simulated results for the NAS Class A conjugate gradient benchmark, using two different optimizations. Times are in processor cycles.

	Conventional	Software copying	Impulse
Time	664M	610M	547M
L1 hit ratio	49.6%	98.6%	99.5%
L2 hit ratio	48.7%	1.1%	0.4%
mem hit ratio	1.7%	0.3%	0.1%
avg load time	6.68	1.71	1.46
TLB misses	0.27M	0.28M	0.01M
speedup	—	1.09	1.21

Table 2: Simulated results for tiled matrix-matrix product. Times are in millions of cycles. The matrices are 512 by 512, with 32 by 32 tiles.

	Tiled	Impulse
Time	459M	237M
L1 hit ratio	98.95%	99.7%
L2 hit ratio	0.81%	0.25%
mem hit ratio	0.24%	0.05%
avg load time	1.57	1.16
issued instructions (total)	725M	290M
graduated instructions (total)	435M	280M
issued instructions (TLB)	256M	7.8M
graduated instructions (TLB)	134M	3.3M
TLB misses	4.21M	0.13M
speedup	—	1.94

Table 3: Simulated results for image filtering with various memory system configurations. Times are in processor cycles. TLB misses are the number of user data misses.

4.1.3 Image Filtering

Table 3 presents the results of order-129 binomial filter on a 32×1024 color image. 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 more than 3.5 times as many L1 cache misses and 40 times as many TLB faults, and executed 134 million instructions in TLB miss handlers. The indirect impact of the high TLB miss rate is even more dramatic – in the baseline filtering program, almost 300 million instructions are issued but not graduated. In contrast, the Impulse version of the algorithm executes only 3.3 million instructions handling TLB misses and only 10 million instructions are issued but not graduated. Compared to these dramatic performance improvements, the less than 1 million cycles spent setting up Impulse remapping are a negligible overhead.

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, yielding 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.

4.1.4 Three-Shear Image Rotation

Table 4 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 [47]; 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 both three-byte and four-byte pixels.

The performance differences among the different 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 95% to 98.5% and reduces the number of TLB misses by two orders of magnitude. This reduction

	Original	Original padded	Tiled	Tiled padded	Impulse
Time	572M	576M	284M	278M	215M
L1 hit ratio	95.0%	94.8%	98.1%	97.6%	98.5%
L2 hit ratio	1.5%	1.6%	1.1%	1.5%	1.1%
mem hit ratio	3.5%	3.6%	0.8%	0.9%	0.4%
avg load time	3.85	3.85	1.81	2.19	1.50
issued instructions (total)	476M	477M	300M	294M	232M
graduated instructions (total)	346M	346M	262M	262M	229M
issued instructions (TLB)	212M	215M	52M	51M	0.81M
graduated instructions (TLB)	103M	104M	24M	24M	0.42M
TLB misses	3.70M	3.72M	0.93M	0.94M	.01M
speedup	—	0.99	2.01	2.06	2.66

Table 4: Simulation results for performing a 3-shear rotation of a 1k-by-1k 24-bit color image. Times are in processor cycles. TLB misses are user data misses.

in the TLB miss rate eliminates 99 million TLB miss handling instructions and reduces the number of issued but not graduated instructions by over 100 million. These two effects constitute 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 is nearly perfect, it suffers for the destination. For the same reason, the decrease in TLB misses for the tiled code is not as great as that for the Impulse code.

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, which degrades the performance of a program that is already memory-bound. In contrast, for the tiled program, the increase in memory traffic is balanced by the reduction in load, shift, and mask operations: manipulating word-aligned pixels is faster than manipulating byte-aligned pixels. The padded, tiled version of the rotation code is still slower than Impulse. The tiled version of the shear uses more cycles recomputing (or saving and restoring) each column's displacement when traversing the tiles. 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 (which is not tiled) only computes each column's displacement once, since each column is completely traversed when it is visited.

4.1.5 Isosurface Rendering Using Ray Tracing

For simplicity, our benchmark assumes that the screen plane is parallel to the volume's z axis. As a result, we can compute an entire plane's worth of indirection vector at once, and we do not need to remap addresses for every ray. This assumption is not a large restriction: it assumes the use of a volume rendering algorithm like Lacroute's [26], which transforms arbitrary viewing angles into angles that have better memory performance. The isosurface in the volume is on one edge of the surface, parallel to the x-z plane.

The measurements we present are for two particular viewing angles. Table 5(A) shows results when the screen is parallel to the y-z plane, so that the rays exactly follow the layout of voxels in memory (we assume an x-y-z layout order). Table 5(B) shows results when the screen is parallel to the x-z plane, where the rays exhibit the worst possible cache and TLB behavior when traversing the x-y planes. These two sets of data points represent the extremes in memory performance for the ray tracer.

In our data, the measurements labeled "Original" are of 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. The indirection vector stores precomputed voxel offsets of each x-y plane. Finally, the results labeled "Impulse" use Impulse to perform the indirection lookup at the memory controller.

In Table 5(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 9 million cycles. The experiment reported in the Indirection column exchanges the computation of voxels offsets for the accesses to the indirection vector. Although it increases the number of memory loads, it still achieves positive speedup because most of those accesses are cache hits. With Impulse, the accesses to the indirection vector are performed only within the memory controller, which hides the access latencies. Consequently, Impulse obtained a higher speedup. Compared to the original version, Impulse saves the computation of voxels offsets.

In Table 5(B), where the rays are perpendicular to the voxel array layout, Impulse yields a much larger performance gain — a speedup of 5.49. Reducing the number of TLB misses saves approximately 59 million graduated instructions while reducing the number of issued but not graduated instructions by approximately 120 million. Increasing the cache hit ratio by loading no useless voxels into the cache saves the remaining quarter-billion cycles. The Indirection version executes about 3% slower than the original one. With rays perpendicular to the voxel array, accessing voxels

	Original	Indirection	Impulse
Time	74.2M	65.0M	61.4M
L1 hit ratio	95.1%	90.8%	91.8%
L2 hit ratio	3.7%	7.3%	6.3%
mem hit ratio	1.2%	1.9%	1.9%
avg load time	1.8	2.8	2.5
loads	21.6M	17.2M	13M
issued instructions (total)	131M	71.4M	57.7M
graduated instructions (total)	128M	69.3M	55.5M
issued instructions (TLB)	0.68M	1.14M	0.18M
graduated instructions (TLB)	0.35M	0.50M	0.15M
TLB misses	9.0K	13.5K	0.8K
speedup	—	1.14	1.21

(A)

	Original	Indirection	Impulse
Time	383M	397M	69.7M
L1 hit ratio	87.1%	82.6%	93.3%
L2 hit ratio	0.6%	2.2%	5.1%
mem hit ratio	12.3%	15.2%	1.6%
avg load time	8.2	10.3	2.4
loads	32M	27M	16M
issued instructions (total)	348M	318M	76M
graduated instructions (total)	218M	148M	68M
issued instructions (TLB)	126M	156M	0.18M
graduated instructions (TLB)	59M	60M	0.15M
TLB misses	2.28M	2.33M	0.01M
speedup	—	0.97	5.49

(B)

Table 5: Results for isosurface rendering. Times are in processor cycles. TLB misses are user data misses. In (A), the rays follow the memory layout of the image; in (B), they are perpendicular to the memory layout.

generates lots of cache misses and frequently loads new data into the cache. These loads can evict the indirection vector from the cache and bring down the cache hit ratio of the indirection vector accesses. As a result, the overhead of accessing the indirection vector outweighs the benefit of saving the computation of voxel offsets and slows down execution.

4.2 Online Superpage Promotion

To evaluate the performance of Impulse's support for inexpensive superpage promotion, we reevaluated Romer *et al.*'s work on dynamic superpage promotion algorithms [40] in the context of Impulse. Our system model differs from theirs in several significant ways. They employ a form of trace-driven simulation with ATOM [42], a binary rewriting tool. That is, they rewrite their applications using ATOM to monitor memory references, and the modified applications are used to do on-the-fly "simulation" of TLB behavior. Their simulated system has two 32-entry, fully-associative TLBs (one for instructions and one for data), uses LRU replacement on TLB entries, and has a base page size of 4096 bytes. To better understand how TLB size may affect the performance, we model two TLB sizes: 64 and 128 entries.

Romer *et al.* combine the results of their trace-driven simulation with measured baseline performance results to calculate effective speedup on their benchmarks. They execute their benchmarks on a DEC Alpha 3000/700 running DEC OSF/1 2.1. The processor in that system is a dual-issue, in-order, 225 MHz Alpha 21064. The system has two megabytes of off-chip cache and 160 megabytes of main memory.

For their simulations, they assume the following fixed costs, which do not take cache effects into account:

- each 1Kbyte copied is assigned a 3000-cycle cost;
- the **asap** policy is charged 30 cycles for each TLB miss;
- and the **approx-online** policy is charged 130 cycles for each TLB miss.

The performance results presented here are obtained through complete simulation of the benchmarks. We measure both kernel and application time, the direct overhead of implementing the superpage promotion algorithms and the resulting effects on the system, including the expanded TLB miss handlers, cache effects due to accessing the page tables and maintaining prefetch counters, and the overhead associated with promoting and using superpages with Impulse. We present comparative performance results for our application benchmark suite.

4.2.1 Application Results

To evaluate the different superpage promotion approaches on larger problems, we use eight programs from a mix of sources. Our benchmark suite includes three SPEC95 benchmarks (`compress`, `gcc`, and `vortex`), the three image processing benchmarks described earlier (`isosurf`, `rotate`, and `filter`), one scientific benchmark (`adi`), and one benchmark from the DIS benchmark suite (`dm`) [28]. All benchmarks were compiled with Sun cc Workshop Compiler 4.2 and optimization level "-xO4".

`Compress` is the SPEC95 data compression program run on an input of ten million characters. To avoid overestimating the efficacy of superpages, the compression algorithm was run only once, instead of the default 25 times. `gcc` is the `cc1` pass of the version 2.5.3 gcc compiler (for SPARC architectures) used to compile the 306-kilobyte file "1cp-dec1.c". `vortex` is an object-oriented database program measured with the SPEC95 "test" input. `isosurf` is the interactive isosurfacing volume renderer described in Section 4.1.5. `filter` performs an order-129 binomial filter on a 32×1024 color image. `rotate` turns a 1024×1024 color image clockwise through one radian. `adi` implements algorithm *alternative direction integration*. `dm` is a data management program using input file "dm07.in".

Two of these benchmarks, `gcc` and `compress`, are also included in Romer *et al.*'s benchmark suite, although we use SPEC95 versions, whereas they used SPEC92 versions. We do not use the other SPEC92 applications from that study, due to the benchmarks' obsolescence. Several of Romer *et al.*'s remaining benchmarks were based on tools used in the research environment at the University of Washington, and were not readily available to us.

Table 6 lists the characteristics of the baseline run of each benchmark with a four-way issue superscalar processor, where no superpage promotion occurs. *TLB miss time* is the total time spent in the data TLB miss handler. These benchmarks demonstrate varying sensitivity to TLB performance: on the system with the smaller TLB, between 9.2% and 35.1% of their execution time is lost due to TLB miss costs. The percentage of time spent handling TLB misses falls to between less than 1% and 33.4% on the system with a 128-entry TLB.

Figures 9 and 10 show the normalized speedups of the different combinations of promotion policies (**asap** and **approx-online**) and mechanisms (*remapping* and *copying*) compared to the baseline instance of each benchmark. In our experiments we found that the best **approx-online** threshold for a two-page superpage is 16 on a conventional system and is 4 on an Impulse system. These are also the thresholds used in our full-application tests. Figure 9 gives results with a 64-entry TLB; Figure 10 gives results with a 128-entry TLB. Online superpage promotion can improve

Benchmark	Total cycles (M)	Cache misses (K)	TLB misses (K)	TLB miss time
64-entry TLB				
compress	632	3455	4845	27.9%
gcc	628	1555	2103	10.3%
vortex	605	1090	4062	21.4%
isosurf	94	989	563	18.3%
adi	669	5796	6673	33.8%
filter	425	241	4798	35.1%
rotate	547	3570	3807	17.9%
dm	233	129	771	9.2%
128-entry TLB				
compress	426	3619	36	0.6%
gcc	533	1526	332	2.0%
vortex	423	763	1047	8.1%
isosurf	93	989	548	17.4%
adi	662	5795	6482	32.1%
filter	417	240	4544	33.4%
rotate	545	3569	3702	16.9%
dm	211	143	250	3.3%

Table 6: Characteristics of each baseline run

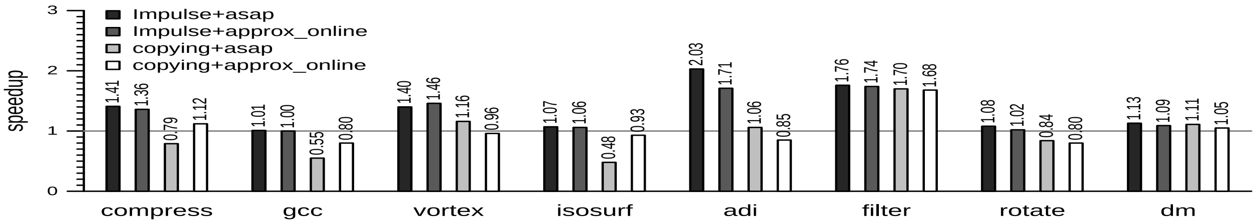


Figure 9: Normalized speedups for each of two promotion policies on a 4-issue system with a 64-entry TLB.

performance by up to a factor of two (on `adi` with remapping **asap**), but it also can decrease performance by a similar factor (when using the copying version of **asap** on `isosurf`). We can make two orthogonal comparisons from these figures: *remapping* versus *copying*, and **asap** versus **approx-online**.

4.2.2 Asap vs. Approx-online

We first compare the two promotion algorithms, **asap** and **approx-online**, using the results from Figures 9 and 10. The relative performance of the two algorithms is strongly influenced by the choice of promotion mechanism, *remapping* or *copying*. Using remapping , **asap** slightly outperforms **approx-online** in the average case. It exceeds the performance of **approx-online** in 14 of the 16 experiments, and trails the performance of **approx-online** in only one case (on `vortex` with a 64-entry tlb). The differences in performance range from *asap+remap* outperforming *aol+remap* by 32% for `adi` with a 64-entry TLB, to *aol+remap* outperforming *asap+remap* by 6% for `vortex` with a 64-entry TLB. In general, however, performance differences between the two policies are small: **asap** is on average 7% better with a 64-entry TLB, and 6% better with a 128-entry TLB.

The results change noticeably when we employ a *copying* promotion mechanism: **approx-online** outperforms **asap** in nine of the 16 experiments, while the policies perform almost identically in three of the other seven cases. The magnitude of the disparity between **approx-online** and **asap** results is also dramatically larger. The differences in performance range from **asap** outperforming **approx-online** by 20% for `vortex` with a 64-entry TLB, to **approx-online** outperforming **asap** by 45% for `isosurf` with a 64-entry TLB. Overall, our results confirm those of Romer *et al.*: the best promotion policy to use when creating superpages via copying is **approx-online**. Taking the arithmetic mean of the performance differences reveals that **asap** is, on average, 6% better with a 64-entry TLB, and 4% better with a 128-entry TLB.

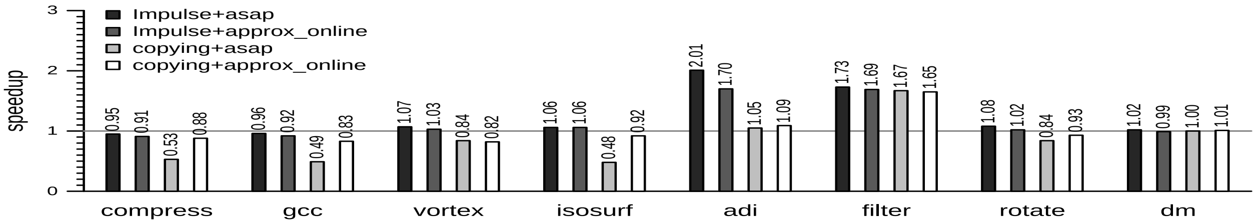


Figure 10: Normalized speedups for each of two promotion policies on a 4-issue system with a 128-entry TLB.

The relative performance of the **asap** and **approx-online** promotion policies changes when we employ different promotion mechanisms because **asap** tends to create superpages more aggressively than **approx-online**. The design assumption underlying the **approx-online** algorithm (and the reason that it performs better than **asap** when copying is used) is that superpages should not be created until the cost of TLB misses equals the cost of creating the superpages. Given that remapping has a much lower cost for creating superpages than copying, it is not surprising that the more aggressive **asap** policy performs relatively better with it than **approx-online** does.

4.2.3 Remapping vs. Copying

When we compare the two superpage creation mechanisms, *remapping* is the clear winner, but by highly varying margins. The differences in performance between the best overall remapping-based algorithm (*asap+remap*) and the best copying-based algorithm (*aonline+copying*) is as large as 97% in the case of `adi` on both a 64-entry and 128-entry TLB. Overall, *asap+remap* outperforms *aonline+copying* by more than 10% in eleven of the sixteen experiments, averaging 33% better with a 64-entry TLB, and 22% better with a 128-entry TLB.

4.2.4 Discussion

Romer *et al.* show that **approx-online** is generally superior to **asap** when copying is used. When remapping is used to build superpages, though, we find that the reverse is true. Using Impulse-style remapping results in larger speedups and consumes much less physical memory. Since superpage promotion is cheaper with Impulse, we can also afford to promote pages more aggressively.

Romer *et al.*'s trace-based simulation does not model any cache interference between the application and the TLB miss handler; instead, that study assumes that each superpage promotion costs a total of 3000 cycles per kilobyte copied [40]. Table 7 shows our measured per-kilobyte cost (in CPU cycles) to promote pages by copying for four representative benchmarks. (Note that

	cycles per 1K bytes promoted	average cache hit ratio	baseline cache hit ratio
gcc	10,798	98.81%	99.33%
filter	5,966	99.80%	99.80%
raytrace	10,352	96.50%	87.20%
dm	6,534	99.80%	99.86%

Table 7: Average copy costs (in cycles) for **approx-online** policy.

we also assume a relatively faster processor.) We measure this bound by subtracting the execution time of *aol+remap* from that of *aol+copy* and dividing by the number of kilobytes copied. For our simulation platform and benchmark suite, copying is at least twice as expensive as Romer *et al.* assumed. For `gcc` and `raytrace`, superpage promotion costs more than three times the cost charged in the trace-driven study. Part of these differences are due to the cache effects that copying incurs.

We find that when copying is used to promote pages, **approx-online** performs better with a lower (more aggressive) threshold than used by Romer *et al.* Specifically, the best thresholds that our experiments revealed varied from four to 16, while their study used a fixed threshold of 100. This difference in thresholds has a significant impact on performance. For example, when we run the `adi` benchmark using a threshold of 32, **approx-online** with copying *slows* performance by 10% with a 128-entry TLB. In contrast, when we run **approx-online** with copying using the best threshold of 16, performance *improves* by 9%. In general, we find that even the copying-based promotion algorithms need to be more aggressive about creating superpages than was suggested by Romer *et al.* Given that our cost of promoting pages is much higher than the 3000 cycles estimated in their study, one might expect that the best thresholds would be higher than Romer's. However, the cost of a TLB miss far outweighs the greater copying costs; our TLB miss costs are about an order of magnitude greater than those assumed in his study.

5 Related Work

A number of projects have proposed modifications to conventional CPU or DRAM designs to improve memory system performance, including supporting massive multithreading [2], moving processing power on to DRAM chips [25], or developing configurable architectures [50]. While these projects show promise, it is now almost impossible to prototype non-traditional CPU or cache designs that can perform as well as commodity processors. In addition, the performance of processor-in-memory approaches are handicapped by the optimization of DRAM processes for capacity (to increase bit density) rather than speed.

The Morph architecture [50] is almost entirely configurable: programmable logic is embedded in virtually every datapath in the system, enabling optimizations similar to those described here. The primary difference between Impulse and Morph is that Impulse is a simpler design that can be used in current systems.

The RADram project at UC Davis is building a memory system that lets the memory perform computation [34]. RADram is a PIM, or *processor-in-memory*, project similar to IRAM [25]. The RAW project at MIT [46] is an even more radical idea, where each IRAM element is almost entirely reconfigurable. In contrast to these projects, Impulse does not seek to put an entire processor in memory, since DRAM processes are substantially slower than logic processes.

Many others have investigated memory hierarchies that incorporate stream buffers. Most of these focus on non-programmable buffers to perform hardware prefetching of consecutive cache lines, such as the prefetch buffers introduced by Jouppi [23]. Even though such stream buffers are intended to be transparent to the programmer, careful coding is required to ensure good memory performance. Palacharla and Kessler [36] investigate the use of similar stream buffers to replace the L2 cache, and Farkas et al. [14] identify performance trends and relationships among the various components of the memory hierarchy (including stream buffers) in a dynamically scheduled processor. Both studies find that dynamically reactive stream buffers can yield significant performance increases.

The Imagine media processor is a stream-based architecture with a bandwidth-efficient stream register file [38]. The streaming model of computation exposes parallelism and locality in applications, which makes such systems an attractive domain for intelligent DRAM scheduling.

Competitive algorithms perform online cost/benefit analyses to make decisions that guarantee performance within a constant factor of an optimal offline algorithm. Romer *et al.* [40] adapt this approach to TLB management, and employ a competitive strategy to decide when to perform dynamic superpage promotion. They also investigate online software policies for dynamically remapping pages to improve cache performance [6, 39]. Competitive algorithms have been used to help increase the efficiency of other operating system functions and resources, including paging, synchronization, and file cache management.

Chen *et al.* [11] report on the performance effects of various TLB organizations and sizes. Their results indicate that the most important factor for minimizing the overhead induced by TLB misses is *reach*, the amount of address space that the TLB can map at any instant in time. Even though the SPEC benchmarks they study have relatively small memory requirements, they find that TLB misses increase the effective CPI (cycles per instruction) by up to a factor of five. Jacob and Mudge [22] compare five virtual memory designs, including combinations of hierarchical and inverted page tables for both hardware-managed and software-managed TLBs. They find that large TLBs are necessary for good performance, and that TLB miss handling accounts for much of the memory-management overhead. They also project that individual costs of TLB miss traps will increase in future microprocessors.

Proposed solutions to this growing TLB performance bottleneck range from changing the TLB structure to retain more of the working set (e.g., multi-level TLB hierarchies [1, 16]), to implementing better management policies (in software [21] or hardware [20]), to masking TLB miss latency by prefetching entries (again, in software [4] or hardware [41]).

All of these approaches can be improved by exploiting superpages. Most commercial TLBs support superpages, and have for several years [30, 43], but more research is needed into how best to make general use of them. Khalidi [24] and Mogul [31] discuss the benefits of systems that support superpages, and advocate static allocation via compiler or programmer hints. Talluri *et al*. [32] report on many of the difficulties attendant upon general utilization of superpages, most of which result from the requirement that superpages map physical memory regions that are contiguous and aligned.

6 Conclusions

The Impulse project attacks the memory bottleneck by designing and building a smarter memory controller. Impulse requires no modifications to the CPU, caches, or DRAMs. It has one special form of "smarts": The controller supports application-specific physical address remapping. This paper demonstrates how several simple remapping functions can be used in different ways to improve the performance of two important scientific application kernels.

Flexible remapping support in the Impulse controller can be used to implement a variety of optimizations. Our experimental results show that Impulse's fine-grained remappings can result in substantial program speedups. Using the scatter/gather through an indirection vector remapping mechanism improves the NAS conjugate gradient benchmark performance by 210% and the volume rendering benchmark performance by 449%; using strided remapping improves performance of image filtering, image rotation, and dense matrix-matrix product applications by 94%, 166%, and 21%, respectively.

Impulse's direct remappings are also effective for a range of programs. They can be used to dynamically build superpages without copying, and thereby reduce the frequency of TLB faults. Our simulations show that this optimization speeds up eight programs from a variety of sources by up to a factor of 2.03, which is 25% better than prior work. Page-level remapping to perform cache coloring improves performance of conjugate gradient by 17%.

The optimizations that we describe should be applicable across a variety of memory-bound applications. In particular, Impulse should be useful in improving system-wide performance. For example, Impulse can speed up messaging and interprocess communication (IPC). Impulse's support for scatter/gather can remove the software overhead of gathering IPC message data from multiple user buffers and protocol headers. The ability to use Impulse to construct contiguous shadow pages from non-contiguous pages means that network interfaces need not perform complex and expensive address translations. Finally, fast local IPC mechanisms like LRPC [5] use shared memory to map buffers into sender and receiver address spaces, and Impulse could be used to support fast, no-copy scatter/gather into shared shadow address spaces.

7 Acknowledgments

We thank Al Davis, Bharat Chandramouli, Krishna Mohan, Bob Devine, Mark Swanson, Arjun Dutt, Ali Ibrahim, Shuhuan Yu, Michael Abbott, Sean Cardwell, and Yeshwant Kolla for their contributions to the Impulse project.

References

[1] Advanced Micro Devices. AMD Athlon processor technical brief. `http://www.amd.com/-products/cpg/athlon/techdocs/pdf/22054.pdf`, 1999.

[2] 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.

[3] D. Bailey et al. The NAS parallel benchmarks. Technical Report RNR-94-007, NASA Ames Research Center, March 1994.

[4] K. Bala, F. Kaashoek, and W. Weihl. Software prefetching and caching for translation buffers. In *Proceedings of the First Symposium on Operating System Design and Implementation*, pages 243–254, Nov. 1994.

[5] B. Bershad, T. Anderson, E. Lazowska, and H. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, Feb. 1990.

[6] B. Bershad, D. Lee, T. Romer, and J. Chen. Avoiding conflict misses dynamically in large direct-mapped caches. In *Proceedings of the 6th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 158–170, Oct. 1994.

[7] P. Budnik and D. Kuck. The organization and use of parallel memories. *ACM Transactions on Computers*, C-20(12):1566–1569, 1971.

[8] 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.

[9] 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.

[10] E. Catmull and A. Smith. 3-D transformations of images in scanline order. *Computer Graphics*, 15(3):279–285, 1980.

[11] J. B. Chen, A. Borg, and N. P. Jouppi. A simulation based study of TLB performance. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 114–123, May 1992.

[12] Compaq Computer Corporation. *Alpha 21164 Microprocessor Hardware Reference Manual*, July 1999.

[13] Z. Fang, L. Zhang, J. Carter, W. Hsieh, and S. McKee. Revisiting superpage promotion with hardware support. In *Proceedings of the Seventh Annual Symposium on High Performance Computer Architecture*, Jan. 2001. To appear.

[14] 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.

[15] J. Gomes and L. Velho. *Image Processing for Computer Graphics*. Springer-Verlag, 1997.

[16] HAL Computer Systems Inc. SPARC64-GP processor. `http://mpd.hal.com/products/-SPARC64-GP.html`, 1999.

[17] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Francisco, CA, second edition, 1996.

[18] R. Hintz and D. Tate. Control Data STAR-100 processor design. In *COMPCON '72*, Boston, MA, Sept. 1972.

[19] A. Huang and J. Shen. The intrinsic bandwidth requirements of ordinary programs. In *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 105–114, Oct. 1996.

[20] Intel Corporation. *Pentium Pro Family Developer's Manual*, Jan. 1996.

[21] B. Jacob and T. Mudge. Software-managed address translation. In *Proceedings of the Third Annual Symposium on High Performance Computer Architecture*, pages 156–167, Feb. 1997.

[22] B. Jacob and T. Mudge. A look at several memory management units, tlb-refill mechanisms, and page table organizations. In *Proceedings of the 8th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 295–306, Oct. 1998.

[23] 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.

[24] Y. Khalidi, M. Talluri, M. Nelson, and D. Williams. Virtual memory support for multiple page sizes. In *Proc. of the 4th Workshop on Workstation Operating Systems*, pages 104–109, Oct. 1993.

[25] C. E. Kozyrakis et al. Scalable processors in the billion-transistor era: IRAM. *IEEE Computer*, pages 75–78, Sept. 1997.

[26] P. G. Lacroute. *Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation*. PhD thesis, Stanford University, Stanford, CA, Sept. 1995. CSL-TR-95-678.

[27] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the 4th ASPLOS*, pages 63–74, Santa Clara, CA, April 1991.

[28] J. W. Manke and J. Wu. *Data-Intensive System Benchmark Suite Analysis and Specification*. Atlantic Aerospace Electronics Corp., June 1999.

[29] S. McKee and W. Wulf. Access ordering and memory-conscious cache utilization. In *Proceedings of the First Annual Symposium on High Performance Computer Architecture*, pages 253–262, Jan. 1995.

[30] MIPS Technologies, Inc. *MIPS R10000 Microprocessor User's Manual, Version 2.0*, Dec. 1996.

[31] J. Mogul. Big memories on the desktop. In *Proc. 4th Workshop on Workstation Operating Systems*, pages 110–115, Oct. 1993.

[32] M.Talluri and M. Hill. Surpassing the TLB performance of superpages with less operating system support. In *Proceedings of the 6th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 171–182, Oct. 1994.

[33] D. R. O'Hallaron. Spark98: Sparse matrix kernels for shared memory and message passing systems. Technical Report CMU-CS-97-178, Carnegie Mellon University School of Computer Science, October 1997.

[34] M. Oskin, F. T. Chong, and T. Sherwood. Active pages: A model of computation for intelligent memory. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 192–203, Barcelona, Spain, June 27–July 1, 1998.

[35] V. Pai, P. Ranganathan, and S. Adve. RSIM reference manual, version 1.0. *IEEE Technical Committee on Computer Architecture Newsletter*, Fall 1997.

[36] 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.

[37] S. Parker, P. Shirley, Y. Livnat, C. Hansen, and P.-P. Sloan. Interactive ray tracing for isosurface rendering. In *Proceedings of the Visualization '98 Conference*, Research Triangle Park, NC, October 1998.

[38] 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 the 31st Annual International Symposium on Microarchitecture*, December 1998.

[39] T. Romer. *Using Virtual Memory to Improve Cache and TLB Performance*. PhD thesis, University of Washington, May 1998.

[40] T. Romer, W. Ohlrich, A. Karlin, and B. Bershad. Reducing TLB and memory overhead using online superpage promotion. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 176–187, June 1995.

[41] A. Saulsbury, F. Dahlgren, and P. Stenstrom. Recency-based TLB preloading. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 117–127, June 2000.

[42] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the 1994 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 196–205, June 1994.

[43] SUN Microsystems, Inc. *UltraSPARC User's Manual*, July 1997.

[44] 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.

[45] O. Temam, E. D. Granston, and W. Jalby. To copy or not to copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In *Proceedings of Supercomputing '93*, pages 410–419, Portland, OR, November 1993.

[46] E. Waingold, et al.˙ Baring it all to software: Raw machines. *IEEE Computer*, pages 86–93, Sept. 1997.

[47] G. Wolberg. *Digital Image Warping*. IEEE Computer Society Press, 1990.

[48] L. Zhang. URSIM reference manual. Technical Report UUCS-00-015, University of Utah, August 2000.

[49] L. Zhang, J. Carter, W. Hsieh, and S. McKee. Memory system support for image processing. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, pages 98–107, Oct. 1999.

[50] X. Zhang, A. Dasdan, M. Schulz, R. K. Gupta, and A. A. Chien. Architectural adaptation for application-specific locality optimizations. In *Proceedings of the 1997 IEEE International Conference on Computer Design*, 1997.