Impulse: Building a Smarter Memory Controller

John Carter, Wilson Hsieh, Leigh Stoller, Mark Swanson, Lixin Zhang,
Erik Brunvand, Al Davis, Chen-Chi Kuo, Ravindra Kuramkote,
Michael Parker, Lambert Schaelicke, and Terry Tateyama

Abstract

Impulse is a new memory system architecture that adds two important features to a traditional memory controller. First, Impulse supports application-specific optimizations through configurable physical address remapping. By remapping physical addresses, applications can control how their data is accessed and cached, which enables them to improve their cache and bus utilization. Second, Impulse supports prefetching at the memory controller. By prefetching data at the controller, the memory system can hide much of the latency of DRAM accesses.

In this paper we describe the design of the Impulse architecture, and show how an Impulse memory system can be used to improve the performance of memory-bound programs. For the specific example of the NAS conjugate gradient benchmark, using Impulse improves performance by over 60%. Because it requires no modification to processor, cache, or bus designs, Impulse can be adopted in conventional systems. In addition to scientific applications, we expect that Impulse will benefit regularly strided, memory-bound applications of commercial importance, such as certain database and multimedia programs.

1 Introduction

Since 1985, microprocessor performance has improved at a rate of 60% per year. In contrast, DRAM latencies have improved by only 7% per year, and DRAM bandwidths by only 15-20% per year. The result is that the relative performance impact of memory accesses continues to grow. In addition, as instruction issue rates continue to increase, the demand for memory bandwidth increases proportionately (and possibly even superlinearly) [7, 13]. For applications that do not exhibit sufficient locality, these trends make it increasingly hard to make effective use of the tremendous processing power of modern microprocessors. It is an unfortunate fact that many important applications (e.g., sparse matrix, database, signal processing, multimedia, and CAD applications) do not exhibit such high degrees of locality. In the Impulse project, we are attacking this problem by designing and building a memory controller that is more powerful than conventional ones.

The Impulse memory controller has two features that are not present in current memory controllers. First, the Impulse controller supports an optional extra stage of address translation: as a result, data can have its addresses remapped *without copying*. This feature allows applications to control how their data is accessed and cached, in order to improve bus and cache utilization. Second, the Impulse controller supports prefetching at the memory controller,

Contact information: retrac,wilson,stoller,swanson,lizhang,elb,ald,chenchi,ravi,map,lambert,ttt@cs.utah.edu. Dept. of Computer Science, 50 S Central Campus Drive, Room 3190, University of Utah, SLC, UT 84112-9205.

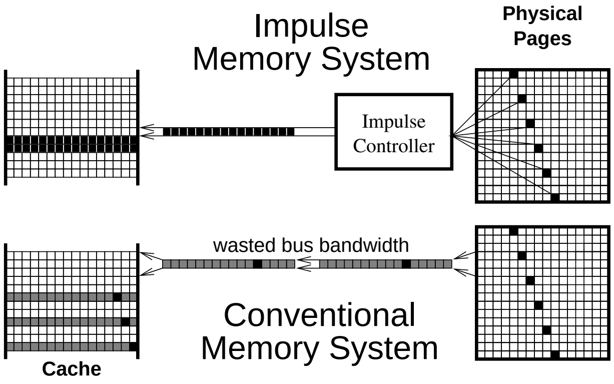

Figure 1: Using Impulse to remap the diagonal of a dense matrix into a dense cache line. The black boxes represent data on the diagonal, whereas the gray boxes represent non-diagonal data.

which reduces the effective latency to memory. Prefetching at the memory controller is important for reducing the latency of Impulse's address translation, and is also a useful optimization for non-remapped data.

The novel feature in Impulse is the addition of another level of address translation at the memory controller. The key insight exploited by this feature is that unused "physical" addresses can undergo translation to "real" physical addresses at the memory controller. An unused physical address is a legitimate address, but one that is not backed by DRAM. For example, in a system with 4GB of physical address space with only 1GB of installed DRAM, there is 3GB of unused physical address space. We call these unused addresses *shadow addresses*, and they constitute a *shadow address space* that is mapped to physical memory by the Impulse controller. 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 modifications to software: applications (or compilers) and operating systems. Using Impulse does not require any modification to other hardware (either processors, caches, or buses).

As a simple example of how Impulse 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 (e.g., `A[i][i]`), it must request a full cache line of contiguous physical memory. On modern systems, a cache line contains 32–128 bytes of data, of which the program accesses only a single word. Such an access is shown in the bottom half of Figure 1.

On an Impulse memory system, an application can configure the memory controller to export a dense shadow space alias that contains just the diagonal elements, and 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 top half of Figure 1. The details of how Impulse performs the remapping is described in Section 2.1.

Remapping the array diagonal to a dense alias results in several performance benefits. First, the processor achieves a higher cache hit rate, because several diagonal elements are loaded into the caches at once. Second, the processor consumes less bus bandwidth, because non-diagonal elements are not sent over the bus. Finally, the processor makes more effective use of cache space, because the non-diagonal elements are not sent. In general, the flexibility that Impulse supports allows applications to customize addressing to fit their needs.

The second important feature of the Impulse memory controller is that it supports prefetching. We include a small amount of SRAM on the Impulse memory controller to store data prefetched from the DRAM's. For non-remapped data, prefetching is useful for reducing the latency of sequentially accessed data. We show that controller-based prefetching of non-remapped data performs as well as a system that uses simple L1 cache prefetching. For remapped data, prefetching enables the controller to hide the cost of remapping: some remappings can require multiple DRAM accesses to fill a single cache line. With both prefetching and remapping, an Impulse controller greatly outperforms conventional memory systems.

In recent years, a number of hardware mechanisms have been proposed to address the problem of increasing memory system overhead. For example, researchers have evaluated the prospects of making the processor cache configurable [26, 27], adding computational power to the memory system [15, 19, 25], and supporting stream buffers [14, 17]. All of these mechanisms promise significant performance improvements; unfortunately, most require significant changes to processors, caches, or memories, and thus have not been adopted in current systems. Impulse supports similar optimizations, but its hardware modifications are localized to the memory controller.

We simulated the impact of Impulse on two benchmarks: the NAS conjugate gradient benchmark, and a dense matrix-matrix product kernel. For both benchmarks, the use of Impulse optimizations significantly improved performance compared to a conventional memory controller. In particular, we found that a combination of address remapping and controller-based prefetching improved the performance of conjugate gradient by over 60%.

Section 2 describes the Impulse architecture in more detail. Section 3 explains how Impulse can be used to optimize sparse matrix-vector and dense matrix-matrix product. Some of the optimizations that we describe are not conceptually new, but the Impulse project is the first system that will provide hardware support for them in general-purpose computer systems. Section 4 presents the results of a preliminary simulation study of Impulse. Although this paper only evaluates two scientific kernels, we expect that Impulse will be useful for optimizing regularly strided non-scientific applications as well. Section 5 describes related work; Section 6 summarizes our conclusions, and describes future work.

2 Impulse Architecture

To illustrate how the Impulse memory controller (MC) works, we describe in detail how it can be used to optimize the simple diagonal matrix example described in Section 1. We describe the internal architecture of the Impulse memory controller, and explain the kinds of address remappings that it currently supports.

2.1 Using Impulse

Figure 2 illustrates the address transformations that Impulse performs to remap the diagonal of a dense matrix. The top half of the figure illustrates how the diagonal elements are accessed on a conventional memory system. The original dense matrix, `A`, occupies three pages of the virtual address space. Accesses to the diagonal elements of `A` are translated into accesses to physical addresses at the processor. Each access to a diagonal element loads an entire cache line of data, but only the diagonal element is accessed, which wastes bus bandwidth and cache capacity.

The bottom half of the figure illustrates how the diagonal elements of `A` are accessed using Impulse. The application reads from a data structure that the OS has remapped to a shadow alias for the matrix diagonal. When the processor issues the read for that alias over the bus, the Impulse controller gathers the data in the diagonal into a single cache line, and sends that data back to the processor. Impulse supports prefetching of memory accesses, so that the latency of the gather can be hidden.

The operating system remaps the diagonal elements to a new alias, `diagonal`, as follows:

1. The application allocates a contiguous range of virtual addresses large enough to map the diagonal elements of `A`, and asks the OS to map it through shadow memory to the actual elements. This range of virtual addresses corresponds to the new variable `diagonal`. To improve L1 cache utilization, an application can allocate virtual

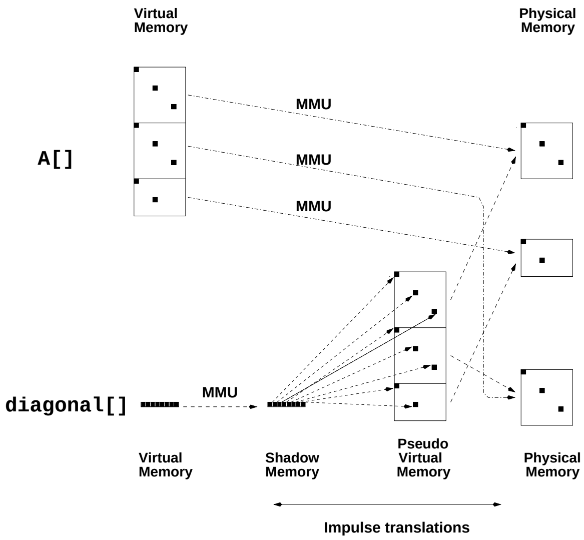

Figure 2: Using Impulse to remap memory: The translation on the top of the figure is the standard translation performed by an MMU. The translation on the bottom of the figure is the translation performed on an Impulse system. The processor translates virtual aliases into what it thinks are physical addresses; however, these physical addresses are really *shadow addresses*. The Impulse MC maps the shadow addresses into *pseudo-virtual addresses*, and then to physical memory.

addresses with appropriate alignment and offset characteristics.

2. The OS allocates a contiguous range of shadow addresses large enough to contain the diagonal elements of `A`. The operating system allocates shadow addresses from a pool of physical addresses that do not correspond to real DRAM addresses (e.g., on a machine with 128MB of DRAM installed, any physical address above 0x08000000).

3. The OS downloads to the memory controller a mapping function from the shadow addresses to offsets within *pseudo-virtual memory space*. An address space that mirrors virtual space is necessary to be able to remap data structures that are larger than a page. We use a pseudo-virtual space in order to save address bits. In our example, the mapping function involves a simple *base* and *stride* function — other remapping functions supported by the current Impulse model are described in Section 2.3.

4. The OS downloads to the memory controller a set of page mappings for pseudo-virtual space for `A`

5. The OS maps the virtual alias `diagonal` to the newly allocated shadow memory, flushes the original address from the caches, and returns.

Currently, we have modified application kernels by hand to perform the system calls to remap data; we are exploring

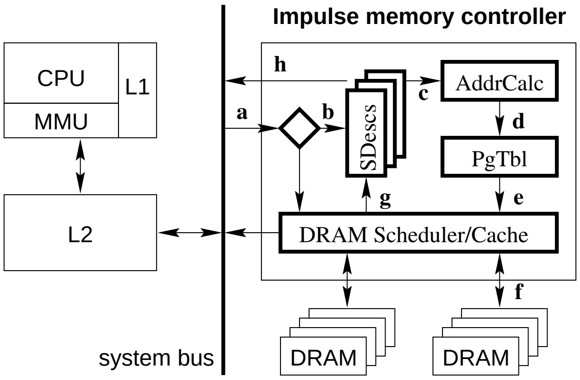

Figure 3: The Impulse memory architecture. The arrows indicate how data flows within an Impulse memory system.

compiler algorithms to automate the process. Both shadow addresses and virtual addresses are system resources, so the operating system must manage their allocation and mapping. We have designed a set of system calls that allow applications to use Impulse without violating inter-process protection.

2.2 Hardware

Figure 3 illustrates Impulse's memory architecture, including the internal organization of the memory controller (MC). The major functional units of the MC are:

- a small number of shadow space descriptors (SDesc) — currently, we assume eight,
- a simple ALU that remaps shadow addresses to pseudo-virtual addresses (AddrCalc), based on information stored in shadow descriptors,
- logic to perform page-grained remapping of pseudo-virtual addresses to physical addresses backed by DRAM (PgTbl), and
- a DRAM scheduler that will optimize the dynamic ordering of accesses to the actual DRAM chips, in order to take advantage of both locality of accesses and parallelism between accesses.

In Figure 3, an address first appears on the memory bus (**a**). This address can be either a physical or a shadow address. If it is physical, it is passed directly to the DRAM scheduler. Otherwise, the matching shadow descriptor is selected (**b**). The remapping information stored in the shadow descriptor is used to translate the shadow address into a set of pseudo-virtual addresses using a simple ALU (AddrCalc) (**c**). Pseudo-virtual addresses are necessary for

Impulse to be able to map data structures that span multiple pages. These addresses are translated into real physical addresses (**d**) using a page table (an on-chip TLB backed by main memory). The real physical addresses are passed to the DRAM scheduler (**e**). The DRAM scheduler orders and issues the reads (**f**), and sends the data back to the shadow descriptors (**g**). Finally, the appropriate shadow descriptor assembles the data into cache lines, and sends it over the bus (**h**).

An important design goal of Impulse is that it should not slow down accesses to non-shadow physical memory, because not all programs will utilize Impulse's remapping functions. Even programs that do remap data will probably contain significant numbers of references to non-remapped data. Therefore, our design for Impulse tries to avoid additional latency on "normal" accesses to memory. In addition, the Impulse controller has a 2K buffer for prefetching non-remapped data. As we show in Section 4, using this prefetch buffer at the controller is competitive with L1 cache prefetching.

Because accesses to remapped memory require a potentially complex address calculation, it is also important that the latency of accesses to remapped memory be kept as low as possible. Therefore, the Impulse controller is designed to support prefetching. Each shadow descriptor contains a 256-byte buffer that can be used to prefetch lines of shadow memory.

We also expect that the controller will be able to schedule remapped memory accesses so that the actual DRAM accesses will occur in parallel. We are designing a low-level DRAM scheduler designed to exploit locality in parallelism between DRAM accesses. First, it will reorder word-grained requests to exploit DRAM page locality. Second, it will schedule requests to exploit bank-level parallelism. Third, it will give priority to requests from the processor over requests that originate in the MC. The design of our DRAM scheduler is not yet complete. Therefore, the simulation results reported in this paper assume a simple scheduler that issues accesses in order. Thus, the results we report should be conservative in terms of the impact of DRAM access time.

2.3 Software Interface

The initial design for Impulse supports several forms of shadow-to-physical remapping:

- *Direct mapping*: Impulse allows applications to map a shadow page directly to a physical page. By remapping physical pages in this manner, applications can recolor physical pages without copying: this optimization is described in Section 3.1.2. In another publication we have described how direct mappings in Impulse can be used to form superpages from non-contiguous physical pages [22].

- *Strided physical memory*: Impulse allows applications to map a region of shadow addresses to a strided data structure. That is, a shadow address at offset *soffset* on a shadow region is mapped to a pseudo-virtual address $pvaddr + stride * soffset$, where *pvaddr* is the starting address of the data structure's pseudo-virtual image. By mapping sparse, regular data items into packed cache lines, applications reduce their bus bandwidth consumption and the cache footprint of the data. An example of such an optimization, tile remapping, is described in Section 3.2.

- *Scatter/gather using an indirection vector*: Impulse allows applications to map a region of shadow addresses to a data structure through an indirection vector. That is, a shadow address at offset *soffset* in a shadow region is mapped to a pseudo-virtual address $pvaddr + stride * vector[soffset]$. By mapping sparse, indirectly addressed data items into packed cache lines, applications reduce their bus bandwidth consumption, the cache footprint of the data, and the number of loads they must issue. An example of this optimization for conjugate gradient is described in Section 3.1.1.

In order to keep the controller hardware simple and fast, Impulse restricts the remappings. For example, in order to avoid the necessity for a divider in the controller, strided mappings must ensure that a strided object has a size that is a power of 2. Also, we assume that an application (or compiler/OS) that uses Impulse ensures data consistency through appropriate flushing of the caches.

3 Impulse Optimizations

In this section we describe how Impulse can be used to optimize two important scientific application kernels: sparse matrix-vector multiply (SMVP) and dense matrix-matrix product. We apply two techniques to optimize SMVP: vector-style scatter/gather at the memory controller and no-copy physical page coloring. We apply a third form of optimization, no-copy tile remapping, to dense matrix-matrix product. We believe this variety of optimizations demonstrates the potential of using Impulse. Other optimizations, some of which we have already explored in other work and some of which are future work, are described in Section 6.

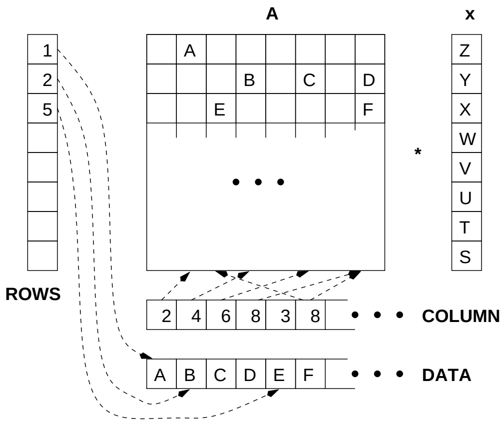

Figure 4: Conjugate gradient's sparse matrix-vector product. The matrix A is encoded using three dense arrays: `DATA`, `ROWS`, and `COLUMN`. The contents of A are in `DATA`. `ROWS[i]` indicates where the i^{th} row begins in `DATA`. `COLUMN[i]` indicates which column of A the element stored in `DATA[i]`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

3.1 Sparse Matrix-Vector Product

Sparse matrix-vector product 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 [18] are spent in SMVP.

To avoid wasting memory, sparse matrices are generally encoded so that only non-zero elements and corresponding index arrays need to be stored. For example, the Class A input matrix for the NAS Conjugate Gradient kernel (CG-A) is 14,000 by 14,000, and contains less than two 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. When we ran CG-A on a Silicon Graphics Origin 2000 processor (which has a 2-way, 32K L1 cache and a 2-way, 4MB L2 cache), the L1 cache hit rate was only 63%, and the L2 cache hit rate was only 92%.

The key inner loop of sparse matrix-product follows:

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

Sparse matrix-vector product is illustrated in Figure 4. Each iteration multiplies a row of the sparse matrix `A` with the dense vector `x`. This code performs poorly on conventional memory systems, because the accesses to `x` are both indirect (via the COLUMN index vector) and sparse. When x is accessed, a conventional memory system will fetch a cache line of data, of which only one element is used. Because of the large sizes of x, COLUMN, and DATA and the sparse nature of accesses to x during each iteration of the loop, there will be very little 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 provide 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.

3.1.1 Scatter/Gather

The Impulse memory controller supports scatter/gather of physical addresses through indirection vectors. Vector machines, such as the CDC STAR-100 [11], have provided scatter/gather capabilities in hardware, but such mechanisms have been provided at the processor. Because Impulse allows scatter/gather to occur at the memory, it can be used to reduce memory traffic over the bus. In addition, Impulse will allow conventional CPU's to take advantage of scatter/gather functionality.

The CG code that would be generated by a compiler designed to exploit the Impulse memory controller is as follows:

```
setup x', where x'[k] = x[COLUMN[k]]
for i := 0 to n-1 do
  sum := 0
  for j := ROWS[i] to ROWS[i+1]-1 do
    sum += DATA[j] * x'[j]
  b[i] := sum
```

The first line asks the operating system to allocate a new region of shadow space, map x' to that shadow region, and have the memory controller 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 the original structure (x).

This optimization improves the performance of sparse matrix-vector product in two ways. First, spatial locality is improved in the L1 cache. Since the memory controller packs the gathered elements into cache lines, the cache lines contain 100% useful data, rather than only one useful element each. Second, fewer memory instructions need to be issued. Since the read of the indirection vector (COLUMN[]) occurs at the memory controller, the processor does not need to issue the read. Note that the use of scatter/gather at the memory controller reduces temporal locality in the L2 cache. The reason is that the remapped elements of `x'` cannot be reused (since all of the elements have different addresses), even though the underlying data structure `x` does get reused. On a conventional memory system, a sufficiently large L2 cache could hold enough of elements of `x` to achieve reuse. The improvement in L1 cache behavior far outweighs the degradation in L2 cache behavior, as we show in Section 4.1.1.

Consider the inner loop of sparse matrix-vector product. It is dominated by the cost for performing three loads (to `DATA[i]`, `COLUMN[i]`, and `x[COLUMN[i]]`). We assume that the L1 cache has 32-byte lines, and the L2 cache has 128-byte lines. Table 1 illustrates the advantage of using Impulse scatter/gather remapping. The table lists the memory references over four iterations of the loop. The initial read of `DATA[i]` hits in the L2 cache $\frac{3}{4}$ of the time, because an L2 cache line is four times larger than an L1 cache line. The read of `COLUMN[i]` is similar: because the elements of `COLUMN` are single-word integers, elements will hit in the L1 cache.

The columns of Table 1 show the difference between a conventional memory system and Impulse. *Best* represents the best case performance of a conventional memory system. In this case, the L2 cache is large enough to hold `x`, and there are no L2 cache conflicts between `x` and any other data. *Worst* represents the worst case performance of a conventional memory system. In this case, either the L2 cache is too small to hold a significant fraction of `x`, or `x` conflicts with other structures in the L2 cache. Because `x` is not accessed directly in Impulse, its best and worst case are identical.

As Table 1 shows, scatter/gather remapping on Impulse can eliminate four memory accesses, each of which are hits in the L2 cache, from the best case for a conventional system. In place of these four accesses, Impulse incurs the miss marked in the table with an asterisk, which is the gathered access to x'. Impulse performs the gathered access by reading `COLUMN[i]` from DRAM at the controller, and then reading `x[COLUMN[i]]`. Compared to the worst case for a conventional system, Impulse eliminates four misses to main memory. If we assume that software pipelining and prefetching hides cold misses to linearly accessed data, the misses to `DATA[i]`, `COLUMN[i]`, and `x'[i]` can be hidden. As a result, using Impulse will allow the processor to perform floating point operations as fast as the memory system can supply two streams of dense data (`x'` and `DATA`), which will have good L1 cache performance. In contrast, the conventional system makes sparse accesses to `x[COLUMN[i]]`, so it will incur frequent L1 cache misses.

Conventional				Impulse	
Value Loaded	Behavior			Value Loaded	Behavior
	Best	Worst			
`DATA[i]`	.75 L2 hit	.75 L2 hit		`DATA[i]`	.75 L2 hit
`COLUMN[i]`	.5 L1 hit, .375 L2 hit	.5 L1 hit, .375 L2 hit		—	—
`x[COLUMN[i]]`	L2 hit	miss		—	—
—	—	—		`x'[i]`	.75 L2 hit*
`DATA[i+1]`	L1 hit	L1 hit		`DATA[i+1]`	L1 hit
`COLUMN[i+1]`	L1 hit	L1 hit		—	—
`x[COLUMN[i+1]]`	L2 hit	miss		—	—
—	—	—		`x'[i+1]`	L1 hit
`DATA[i+2]`	L1 hit	L1 hit		`DATA[i+2]`	L1 hit
`COLUMN[i+2]`	L1 hit	L1 hit		—	—
`x[COLUMN[i+2]]`	L2 hit	miss		—	—
—	—	—		`x'[i+2]`	L1 hit
`DATA[i+3]`	L1 hit	L1 hit		`DATA[i+3]`	L1 hit
`COLUMN[i+3]`	L1 hit	L1 hit		—	—
`x[COLUMN[i+3]]`	L2 hit	miss		—	—
—	—	—		`x'[i+3]`	L1 hit

Table 1: Simple performance comparison of conventional memory systems (best and worst cases) and Impulse for scatter/gather remapping of sparse matrix-vector product. The inner loop has been unrolled four times, and the analysis assumes a 32-byte L1 cache and a 128-byte L2 cache. On a conventional memory system, three reads are performed in each iteration; on Impulse, only two reads are performed in each iteration. The miss that occurs in the starred entry is more expensive on Impulse, because it requires a gather at the memory controller.

3.1.2 Page Recoloring

The Impulse memory controller supports dynamic physical page recoloring through direct remapping of physical pages. Physical page recoloring changes the physical addresses of pages so that data with good locality (e.g., `x` in conjugate gradient) is mapped to a different part of a physically-addressed cache than data with poor locality (e.g., `DATA`, `ROW`, and `COLUMN`). By performing page recoloring, conflict misses due to poor data alignment can be eliminated. Since Impulse can map data structures to shadow physical addresses, data with good locality can be mapped to shadow pages that do not conflict with other data in a physically-addressed cache.

On a conventional machine, physical page recoloring is expensive to exploit. (Note that virtual page recoloring has been explored by other authors [5].) The cost is in copying: the only way to change the physical address of data is to copy the data between physical pages. Impulse allows pages to be recolored *without copying*.

For sparse matrix-vector product, the `x` vector is reused within an iteration, while elements of the `DATA`, `ROW`, and `COLUMN` vectors are used only once each 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 in the L2 cache with the other data structures. For example, in the CG-A benchmark, `x` is over 100K bytes: it would not fit in most processors' L1 caches, but would fit in many L2 caches. Impulse can be used to 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 effect, we can use a small part of the L2 cache as a stream buffer [17] for `DATA`, `ROWS`, and `COLUMNS`.) The resulting performance should approach that of the column labeled *Best* in Table 1.

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") in order to increase their efficiency. That is, the iterations of tiled algorithms are reordered so as to improve their memory performance. The difficulty with using tiled algorithms lies in choosing an appropriate tile size [16]. 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 the caches. 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 another alternative to removing cache conflicts for tiles. We use the simplest tiled algorithm, dense matrix-matrix product, as an example of how Impulse can be used to improve the behavior of tiled matrix algorithms. Assume an `i-j-k` ordering of the loops, as in the following simplified example:

```
for it = 0 to n-1 by 32
  for jt = 0 to n-1 by 32
    for kt = 0 to n-1 by 32
      for i = it to min(n, it+32-1)
        for j = jt to min(n, jt+32-1)
          tmp := 0
          for k = kt to min(n, kt+32-1)
            tmp += A[i,k]*B[k,j]
          C[i,j] := tmp
```

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 remap the virtual tile to the next physical tile by using Impulse. In order 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 4.2 shows, these costs are minor.

4 Performance

We have performed a preliminary simulation study of Impulse using the Paint simulator [21]: it 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. A hit in the L1 cache has a minimum latency of one cycle; a hit in the L2 cache, seven cycles; an access to memory, sixty cycles. The TLB's are 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 holding the most recent instruction translation is also modeled. Kernel code and data structures are mapped using a single *block TLB* entry that is not subject to replacement.

The simulated Impulse memory controller is based on the HP memory controller [12] that is used in their servers and high-end workstations, and models the hardware described in Section 2. References to shadow addresses incur a minimum three cycle delay; complex mapping functions cause larger delays. The microkernel running on Paint supports the system calls necessary to use Impulse. In our experiments we measure the performance benefits of using Impulse to remap physical addresses, as described in Section 3.

Our experiments also measure the benefits of using Impulse to prefetch data. When prefetching is turned on for Impulse, both shadow and non-shadow accesses is prefetched. As a point of comparison, we compare controller

	No prefetching	Prefetching		
		Impulse	L1 cache	Impulse + L1 cache
Conventional memory system				
Time	3.66	3.35	3.39	3.36
L1 hit ratio	68.5%	68.5%	57.3%	57.2%
L2 hit ratio	26.5%	26.5%	41.0%	41.0%
avg load time	6.18	5.22	5.51	5.45
speedup	—	9%	8%	8%
Impulse with scatter/gather remapping				
Time	2.69	2.20	2.46	2.41
L1 hit ratio	88.2%	88.2%	66.5%	66.5%
L2 hit ratio	5.1%	5.1%	32.5%	32.5%
avg load time	6.80	4.39	5.18	4.95
speedup	36%	66%	49%	52%
Impulse with page recoloring				
Time	3.41	3.10	3.27	3.16
L1 hit ratio	68.5%	68.5%	57.2%	57.2%
L2 hit ratio	27.5%	27.5%	41.5%	41.8%
avg load time	5.59	4.63	5.23	4.99
speedup	7%	18%	12%	16%

Table 2: Simulated results for the NAS Class A conjugate gradient benchmark, with various memory system configurations. Times are in billions 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; the speedup is the "Conventional, no prefetch" time divided by the time for the system being compared, minus one.

prefetching against a form of processor-side prefetching: hardware next-line prefetching into the L1 cache, such as that used in the HP PA 7200 [8]. (The primary difference between our model and the PA 7200 is that the PA 7200 has an assist cache.) We show that controller prefetching is competitive (and often better) than this simple form of processor-side prefetching.

In the following sections we show how Impulse's remappings can be used to support optimizations on sparse matrix-vector product (SMVP) and dense matrix-matrix product. Scatter/gather remapping can be used to improve the L1 cache performance of SMVP. Alternatively, page remapping can be used to recolor the physical pages of SMVP data for the L2 cache. Finally, base-stride remapping can be used to remap dense matrix tiles into contiguous shadow addresses.

4.1 Sparse Matrix-Vector Product

To evaluate the performance benefits that Impulse enables, we use the NAS Class A conjugate gradient benchmark as our benchmark for sparse matrix-vector product. Table 2 illustrates the performance of an Impulse system on that benchmark, under various memory system configurations. In the following two sections we evaluate the performance of scatter/gather remapping and page recoloring, respectively. Note that our use of "L2 cache hit ratio" uses the total number of loads (not the total number of L2 cache accesses) as the divisor: the reason is to make it easier to compare the effects of the L1 and L2 caches on absorbing memory accesses.

4.1.1 Scatter/Gather

The first and second parts of Table 2 show that the use of scatter/gather remapping on CG-A improves performance significantly. If we examine the performance without prefetching, Impulse improves performance by 36%, because it increases the L1 cache hit ratio dramatically. The cache hits are due to the fact that accesses to the remapped vector `x'` now fetch several useful elements of `x` at a time. In addition to the increase in cache hits, the use of scatter/gather reduces the total number of loads issued, since the indirection load occurs at the memory. The reduction in the total number of loads outweighs the fact that scatter/gather increases the average cost of a load: almost one-third of the cycles saved are due to this factor. Finally, despite the drop in L2 cache hit ratio, using scatter/gather still improves performance.

The combination of scatter-gather remapping and prefetching is even more effective in improving performance: the speedup is 63%. Prefetching improves the effectiveness of scatter/gather: the average time for a load drops from 6.80 cycles to 4.39 cycles. Even though the cache hit ratios do not change, CG-A runs significantly faster because Impulse hides the latency of the memory system.

Prefetching at the memory controller not only hides the latency of scatter/gather: it is useful as an optimization on its own. Interestingly, the table shows that controller-based prefetching is as effective as next-line prefetching at the L1 cache. The combination of controller prefetching and L1 cache prefetching is not significantly better than either alone — in fact, it tends to be worse than controller prefetching by itself! The reason is that the combination of the two forms of prefetching puts too much of a load on the controller. In addition, L1 cache prefetching can create L1 cache conflicts, as the table shows: it effectively moves L1 hits into the L2 cache, because of the conflicts that it creates. Controller prefetching is more effective, because it does not hurt cache performance. Instead, it just reduces the penalty of accessing memory.

4.1.2 Page Recoloring

The first and third sections of Table 2 show that the use of page recoloring improves performance on CG-A. We color the vectors `x`, `DATA`, and `COLUMN` so that they do not conflict in the L2 cache. The multiplicand vector `x` is reused during SMVP, so it is most important to keep it in the L2 cache. Therefore, we color it to occupy the first half of the L2 cache. We want to keep the two other large data structures, `DATA` and `COLUMN`, from conflicting as well. As a result, we divide the second half of the L2 cache into two quadrants. We then color `DATA` and `COLUMN` so that they each occupy one of these quadrants.

Without prefetching, the speedup of using page recoloring is 7%. The improvement is due to a 1% increase in the L2 cache hit ratio. With the addition of prefetching at the controller, the speedup increases to 18%. Page recoloring consistently reduces the cost of memory accesses.

When comparing controller prefetching with L1 cache prefetching, the effects are similar to those with scatter/gather. Controller prefetching is at least as effective as L1 cache prefetching, and controller prefetching alone is generally as effective as the combination of the two. Finally, both forms of prefetching substantially reduce memory access times, but L1 cache prefetching tends to create cache conflicts. Adding an assist cache for prefetching (as in the PA 7200) would eliminate most of these conflicts.

Although page recoloring does not achieve as great a speedup as scatter/gather remapping, it does provide useful speedups. In addition, page recoloring can probably be applied in more applications than scatter/gather (or other fine-grained types of remappings).

4.2 Dense Matrix-Matrix Product

This section examines the performance benefits of tile remapping for matrix-matrix product, and compares the results to software tile copying. Because Impulse places alignment restrictions on remapping, remapped tiles must 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 contraint 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:

	No prefetching	Prefetching		
		Impulse	L1 cache	Impulse + L1 cache
Conventional memory system				
Time	3.14	3.01	3.20	3.09
L1 hit ratio	49.0%	49.0%	48.9%	48.8%
L2 hit ratio	43.0%	43.0%	43.6%	43.5%
avg load time	8.50	8.00	8.73	8.33
speedup	—	4%	-2%	2%
Conventional memory system with software tile copying				
Time	1.34	1.33	1.33	1.33
L1 hit ratio	98.5%	98.5%	98.1%	98.0%
L2 hit ratio	1.4%	1.4%	1.8%	1.9%
avg load time	1.13	1.11	1.12	1.11
speedup	134%	136%	136%	136%
Impulse with tile remapping				
Time	1.32	1.31	1.30	1.30
L1 hit ratio	99.4%	99.4%	97.8%	97.7%
L2 hit ratio	0.4%	0.4%	2.2%	2.2%
avg load time	1.15	1.11	1.10	1.09
speedup	138%	140%	142%	142%

Table 3: Simulated results for tiled matrix-matrix product. Times are in billions 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; the speedup is the "Conventional, no prefetch" time divided by the time for the system being compared, minus one. The matrices are 512 by 512, with 32 by 32 tiles.

similar padding techniques have been explored in the context of vector processors [6]. The difference is that for Impulse, the padding is necessary to correctly align the rows of the array (modulo the cache line size), whereas previous techniques have used padding to misalign array rows in memory.

Table 3 illustrates the results of our tiling experiments. Copying and tile remapping both outperform the baseline code by 140%, unsurprisingly. The improvement in performance is primarily due to the difference in caching behavior: both copying and remapping more than double the L1 cache hit rate. As a result, the average memory access time is approximately one cycle! Impulse tile remapping is slightly faster than tile copying: the system calls for using Impulse, and the associated cache flushes/purges, are faster than copying tiles.

Note that this comparison between conventional and Impulse copying schemes is conservative for several reasons. Copying works particularly well on matrix product, because the number of operations performed on a tile is $O(n^3)$, where $O(n^2)$ is the size of a tile. Therefore, the overhead of physical copying is fairly low. For algorithms where the reuse of the data is lower (or where the tiles are larger), the relative overhead of copying will be greater. In addition, our physical copying experiment avoids cross-interference between active tiles in both the L1 and L2 cache. Other authors have found that the performance of copying can vary greatly with matrix size, tile size, and cache size [23]. Because Impulse remaps tiles without copying, we expect that tile remapping using Impulse will not be sensitive to cross-interference between tiles. Finally, as caches (and therefore tiles) grow larger, the cost of copying grows, whereas the cost of tile remapping does not.

The comparison between controller and L1 cache prefetching shows similar results to those for CG-A. Because of the effectiveness of copying and tile remapping, prefetching makes almost no difference when using those optimizations. When the optimizations are not being used, controller prefetching improves performance by about 4%. L1 cache prefetching actually hurts performance, due to the very low hit rate in the L1 cache — the effect is that prefetching causes too much contention at the L2 cache.

5 Related Work

A number of projects have proposed modifications to conventional CPU or DRAM designs to overcome memory system performance: supporting massive multithreading [2], moving processing power on to DRAM chips [15], building programmable stream buffers [17], or developing configurable architectures [27]. 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.

We briefly describe the most closely related architecture research projects. The Morph architecture [27] is almost entirely configurable: programmable logic is embedded in virtually every datapath in the system. As a result, optimizations similar to those that we have described are possible using Morph. The primary difference between Impulse and Morph is that Impulse is a simpler design that current architectures can take advantage of.

The RADram project at UC Davis is building a memory system that lets the memory perform computation [19]. RADram is a PIM ("processor-in-memory") project similar to IRAM [15], where the goal is to put processors close to memory. The Raw project at MIT [25] 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.

Several researchers have proposed different forms of hardware to improve the performance of applications that access memory using regular strides (vector applications, for example). Jouppi proposed the notion of a stream buffer [14], which is a device that detects strided accesses and prefetches along those strides. McKee et al. [17] proposed a programmable variant of the stream buffer that allows applications to explicitly specify when they make vector accesses. Both forms of stream buffer allow applications to improve their performance on regular applications, but they do not support irregular applications.

Yamada [26] proposed instruction set changes to support combined relocation and prefetching into the L1 cache. Because relocation is done at the processor in his system, no bus bandwidth is saved. In addition, because relocation is done on virtual addresses, the utilization of the L2 cache cannot be improved. With Impulse, the utilization of the L2 cache can directly be improved; the operating system can then be used to improve the utilization of the L1 cache.

A great deal of research has gone into prefetching into the cache [20]. For example, Chen and Baer [9] describe how a prefetching cache can outperform a non-blocking cache. Fu and Patel [10] describe how cache prefetching can be used to improve the performance of caches on vector machines, which is somewhat related to Impulse's scatter/gather optimization. Although our research is related, cache prefetching is orthogonal to Impulse's controller prefetching. In addition, we have shown that controller prefetching can outperform simple forms of cache prefetching.

One memory-based prefetching scheme, described by Alexander and Kedem [1], can improve the performance of some benchmarks significantly. They use a prediction table to store up to four possible predictions for any given memory address. All four predictions are prefetched into SRAM buffers. The size of their prediction table is kept small by using a large prefetch block size.

Finally, the Impulse DRAM scheduler that we are designing has goals that are similar to other research on dynamic access ordering. McKee et al. [17] show that reordering of stream accesses can be used to exploit parallelism in multi-bank memories, as well as locality of reference in page-mode DRAM's. Valero et al. [24] show how reordering of strided accesses on a vector machine can be used to eliminate bank conflicts. On Impulse, the set of addresses to be reordered will be more complex: for example, the set of physical addresses that is generated for scatter/gather is much more irregular than strided vector accesses.

6 Conclusions

In summary, the Impulse project is attacking the memory bottleneck by designing and building a smarter memory controller. The Impulse controller requires no modifications to the CPU, caches, or DRAM's, and it has two forms of "smarts":

- The controller supports application-specific physical address remappings. The paper demonstrates that several simple remapping functions can be used in different ways to improve the performance of two important scientific application kernels.
- The controller supports prefetching at the memory. The paper demonstrates that controller-based prefetching performs as well as simple next-line prefetching in the L1 cache.

Both of these features can be used to improve performance. The combination of these features can result in good speedups: using scatter/gather remapping and prefetching improves performance on the NAS conjugate gradient benchmark by over 60%. Speedups should be greater on superscalar machines (our simulation model was single-issue), because non-memory instructions will be effectively cheaper. That is, on superscalars, memory will be even more of a bottleneck, and Impulse will therefore be able to improve performance even more.

Flexible remapping support in the Impulse controller can be used to support a variety of optimizations. Although our simulation study has only examined two scientific kernels, the optimizations that we have described should be usable across a variety of memory-bound applications. In addition, despite the fact that we use conjugate gradient as our application for two optimizations, we are not comparing optimizations: the two optimizations are usable on different sets of different applications.

In previous work [22], we have shown that the Impulse memory remappings can be used to dynamically build superpages, which can reduce the frequency of TLB faults. Impulse can create superpages from non-contiguous user pages: simulations show that this optimization improves the performance of five SPECint95 benchmark programs by 5-20%. This paper builds on our previous work by demonstrating a range of other optimizations that Impulse supports.

Finally, an Impulse memory system will be useful in improving system-wide performance. For example, Impulse can improve messaging and interprocess communication (IPC) performance. A major chore of remote IPC is collecting message data from multiple user buffers and protocol headers. Impulse's support for scatter/gather can remove the overhead of gathering data in software, which should significantly reduce IPC overhead. 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 translation. Finally, fast local IPC mechanisms, such as LRPC [4], 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.

Acknowledgments

We thank Sally McKee, Massimiliano Poletto, and Llewellyn Reese for comments on drafts of this paper, and Chris Johnson for his assistance in providing us information on on conjugate gradient.

References

[1] T. Alexander and G. Kedem. Distributed prefetch-buffer/cache design for high performance memory systems. In *Proceedings of the Second Annual Symposium on High Performance Computer Architecture*, pages 254–263, Feb. 1996.

[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 272–277, Amsterdam, The Netherlands, June 1990.

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

[4] B. Bershad, T. Anderson, E. Lazowska, and H. Levy. Lightweight remote procedure call. In *Proceedings of the 12th Symposium on Operating Systems Principles*, pages 102–113, Litchfield Park, AZ, Dec. 1989.

[5] 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.

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

[7] 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.

[8] K. Chan, C. Hay, J. Keller, G. Kurpanek, F. Schumacher, and J. Zheng. Design of the HP PA 7200 CPU. *Hewlett-Packard Journal*, 47(1):25–33, February 1996.

[9] T.-F. Chen and J.-L. Baer. Reducing memory latency via non-blocking and prefetching caches. In *Proceedings of the 5th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 51–61, Oct. 1992.

[10] J. Fu and J. Patel. Data prefetching in multiprocessor vector cache memories. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 54–65, Toronto, Canada, May 1991.

[11] R. Hintz and D. Tate. Control Data STAR-100 processor design. In *IEEE Computer Society International Conference*, Boston, MA, Sept. 1972.

[12] 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, February 1996.

[13] 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.

[14] 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.

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

[16] 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, Apr. 1991.

[17] S. McKee et al. Design and evaluation of dynamic access ordering hardware. In *Proceedings of the 10th ACM International Conference on Supercomputing*, Philadelphia, PA, May 1996.

[18] 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, Oct. 1997.

[19] 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.

[20] A. Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, Sept. 1982.

[21] L. Stoller, R. Kuramkote, and M. Swanson. PAINT: PA instruction set interpreter. Technical Report UUCS-96-009, University of Utah - Computer Science Department, Sept. 1996.

[22] 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*, June 1998.

[23] 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, Nov. 1993.

[24] M. Valero, T. Lang, J. Llaberia, M. Peiron, E. Ayguade, and J. Navarro. Increasing the number of strides for conflict-free vector access. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 372–381, Gold Coast, Australia, 1992.

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

[26] Y. Yamada. *Data Relocation and Prefetching in Programs with Large Data Sets*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana, IL, 1995.

[27] 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.