Impulse: Memory System Support for Scientific Applications

John B. Carter, Wilson C. Hsieh, Leigh B. Stoller,

Mark Swanson†, Lixin Zhang, and Sally A. McKee

Department of Computer Science
University of Utah
Salt Lake City, UT 84112

†Intel Corporation
Dupont, WA 98327

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 control how their data is accessed and cached, improving their cache and bus utilization. Second, Impulse supports prefetching at the memory controller, which can hide much of the latency of DRAM accesses. Because it requires no modification to processor, cache, or bus designs, Impulse can be adopted in conventional systems.

In this paper we describe the design of the Impulse architecture, and show how an Impulse memory system can improve the performance of memory-bound scientific applications. For instance, Impulse decreases the running time of the NAS conjugate gradient benchmark by 67%. We expect that Impulse will also benefit regularly strided, memory-bound applications of commercial importance, such as database and multimedia programs.

Keywords: irregular applications, memory systems, computer architecture, memory bandwidth, hardware prefetching

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 [14]. 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 super-linearly) [7, 18]. Many important applications (e.g., sparse matrix, database, signal processing, multimedia, and CAD applications) do not exhibit sufficient 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.

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: data addresses can be remapped *without copying*. This feature improves bus and cache utilization by allowing applications to control how their data is accessed and cached. Second, the Impulse controller supports prefetching at the memory controller, which reduces the effective latency to memory. Prefetching at the memory controller

Contact information: Prof. John Carter, Dept. of Computer Science, 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.

helps hide the latency of Impulse's address translation, and is also a useful optimization for non-remapped data.

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 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 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 top half of Figure 1. The details of how Impulse performs the remapping are described in Section 2.1.

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.

The second important feature of the Impulse memory controller is that it supports prefetching. The controller includes a small amount of SRAM to store data prefetched from the DRAMs. For non-remapped data, prefetching can reduce 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 costs associated with remapping (some remappings can require multiple DRAM accesses to fill a single cache line). With both prefetching and remapping, an Impulse controller significantly 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 [34, 35], adding computational power to the memory system [20, 25, 33], and supporting stream buffers [19]. 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 mainstream 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. Although we evaluate only scientific kernels here, we expect that Impulse will be useful for optimizing non-scientific applications, as well. Some of the optimizations that we describe are not conceptually new, but the Impulse project is the first system that provides hardware support for them in general-purpose computer systems. For both benchmarks, the use of Impulse optimizations significantly improves performance compared to a conventional memory controller. In particular, we find that a combination of address remapping and controller-based prefetching improves the performance of conjugate gradient by 67%.

2 Impulse Architecture

To illustrate how the Impulse memory controller 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 3 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, wasting bus bandwidth and cache capacity by loading the adjacent, non-diagonal elements that won't be used.

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 diagonal data into a single cache line, and sends that data back over the processor bus. Impulse supports prefetching of remapped data within the controller 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 these virtual addresses 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 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.

3. The OS downloads to the memory controller a function to map shadow addresses to offsets within *pseudo-virtual memory space*, which mirrors virtual space in its layout. This pseudo-virtual space is necessary to be able to remap data structures that are larger than a page. 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 2.2.

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 hand-modify application kernels to perform the system calls to remap data, but we are exploring 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 implemented a set of system calls that allow applications to use Impulse without violating inter-process protection.

2.2 Hardware

The organization of the Impulse controller architecture is depicted in Figure 2. The memory controller includes:

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

Since the extra level of address translation is optional, addresses appearing on the memory bus may be to physical (backed by actual DRAM) or shadow memory space. Valid physical addresses pass untranslated to the DRAM scheduler. The Page Table Unit uses the corresponding shadow descriptor to turn shadow addresses into physical DRAM addresses. Currently, this translation can take three forms, depending on how Impulse is used to access a particular data structure: 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 [8] or to construct superpages dynamically [30]. We discuss no-copy page coloring further in Section 3.1. *Strided mapping* creates dense cache lines from array elements that are not contiguous in physical memory. The mapping function maps an address *soffset* in shadow space to pseudo-virtual address $pvaddr + stride \times soffset$, where *pvaddr* is the starting address (assigned by the OS) of the data structure's pseudo-virtual image. *Scatter/gather mapping* uses an indirection vector *ivec* to translate an address *soffset* in shadow space to pseudo-virtual address $pvaddr + stride \times ivec[soffset]$. Investigating support for other mappings is part of ongoing work.

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 two. Also, we assume that an application (or compiler/OS) that uses Impulse ensures data consistency through appropriate flushing of the caches. Note that Impulse in no way affects the virtual memory system — paging and address translation are handled by the OS and on-chip TLB just as in a non-Impulse system.

3 Impulse Optimizations

In this section we describe how Impulse can be used to optimize two scientific application kernels: sparse matrix-vector multiply (SMVP) and dense matrix-matrix product (DMMP). 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 optimization, no-copy tile remapping, to DMMP.

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 [24] 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 2.19 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) exhibited L1 and L2 cache hit rates of only 63% and 92%, respectively.

The inner loop of sparse matrix-product looks like:

```
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
```

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.

Scatter/gather. The Impulse memory controller supports scatter/gather of physical addresses through indirection vectors. Vector machines such as the CDC STAR-100 [15] 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.

The CG code that an Impulse compiler would generate looks like:

```
setup x', where x'[k] = x[COLUMN[k]]
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 first line 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, 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.

Consider the inner loop of SMVP. Its cost is dominated by three loads (to `DATA[i]`, `COLUMN[i]`, and `x[COLUMN[i]]`). 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 75% 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, where 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, where either the L2 cache is too small to hold a significant fraction of `x`, or `x` conflicts with other structures in the L2 cache. `x` is not accessed directly in Impulse and therefore its best and worst cases are identical.

As Table 1 shows, scatter/gather remapping on Impulse can eliminate four L2 accesses 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 hide cold misses to linearly accessed data, the misses to `DATA[i]`, `COLUMN[i]`, and `x'[i]` can be overlapped with processor activity. 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`) with good L1 cache performance. In contrast, the conventional system makes sparse accesses to `x[COLUMN[i]]`, and will incur frequent L1 cache misses.

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 reusable data is mapped to a different part of a physically-addressed cache from non-reused data. By performing page recoloring, conflict misses can be eliminated. On a conventional machine, physical page recoloring is expensive. 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*. Virtual page recoloring has been explored by other authors [5].

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 effect, we can use a small part of the L2 cache as a set of virtual stream buffers for `DATA`, `ROWS`, and `COLUMNS` [23]. 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 to improve their memory performance. The difficulty with using tiled algorithms lies in choosing an appropriate tile size [21]. 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 want to compute $C = A \times B$. 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. 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 [29]. We model 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. In addition, we model a synthetic four-way superscalar version of the same processor. The 32K L1 data cache in both models 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, and 2-way set-associative, with 128-byte lines. Since Impulse is only intended to improve data cache performance, 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, 44-46 cycles. The TLBs 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 (described in Section 2.2) is based on the HP memory controller [17] used in servers and high-end workstations. We model eight shadow descriptors, each of which is associated with a 512-byte SRAM buffer. The controller prefetches the corresponding shadow data into these fully associative buffers of four 128-byte lines. A 4K SRAM Mcache holds prefetched, non-shadow data within the Scheduler Unit. The Mcache is four-way set associative with 32 lines of 128 bytes. The MTLB is two-way associative, has 128 eight-byte entries (the same size as the entries in the kernel's page table), and includes two 128-bit buffers used to prefetched consecutive lines of page table entries on an MTLB miss. Prefetched page table entries are also stored in the Mcache. If an MTLB miss hits in the buffers, the required entry can be transferred into the MTLB in one cycle. Otherwise the MTLB initiates an Mcache access, and if that misses, it initiates a DRAM access to retrieve the entry.

References to shadow addresses incur a minimum three-cycle delay, with complex mapping functions causing larger delays. Prefetching within the memory controller reduces the impact of the extra translation overhead. To keep the remapping circuitry simple and fast, we require that all remapped data structures be page-aligned and that various dimensions of data structures used in strided mappings be powers of two in size. This avoids including a divider in Impulse. We implemented the Impulse system calls within Paint's microkernel such that applications can use Impulse without violating interprocess protection.

The memory system modeled contains four DRAM buses and 16 banks of SDRAM, and has a total memory latency of 44-46 cycles, broken down as follows:

- L1 cache latency is one cycle,
- L2 cache latency is six cycles,
- address latency on system bus is one cycle,

- memory controller latency is two cycles,
- DRAM access latency is 24 cycles,
- DRAM demultiplexer latency is two cycles,
- bus arbitration latency is zero to two cycles, and
- latency for returning data is eight cycles.

In our experiments we measure the performance benefits of using Impulse to remap physical addresses, as described in Section 3. We also measure the benefits of using Impulse to prefetch data. When prefetching is turned on for Impulse, both shadow and non-shadow accesses are prefetched. As a point of comparison, we evaluate controller 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 [9]. Our results show that controller prefetching is competitive with this simple form of processor-side prefetching. Finally, we modify our simulator to approximate a four-way superscalar processor so that we can estimate the speedups that Impulse should provide on a more modern architecture.

4.1 Sparse Matrix-Vector Product

Table 2 illustrates the performance of the NAS Class A Conjugate Gradient (CG-A) benchmark on various configurations of an Impulse system. In the following two sections we evaluate the performance of scatter/gather remapping and page recoloring, respectively. Note that our calculation of "L2 cache hit ratio" uses the total number of loads (not the total number of L2 cache accesses) as the divisor to make it easier to compare the effects of the L1 and L2 caches on memory accesses.

Scatter/gather. The first and second parts of Table 2 show that scatter/gather remapping on CG-A improves performance significantly. Without prefetching, Impulse improves performance by 1.33, largely due to the increase in the L1 cache hit ratio. Each main-memory access for the remapped vector `x'` now loads the cache with several useful elements from the original `x`, increasing the cache hit rate. In addition, scatter/gather reduces the total number of loads issued, since loads of the indirection vector occur 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 in this instance. The drop in the L2 cache hit ratio does not negatively impact performance.

The combination of scatter/gather remapping and prefetching is even more effective, speeding up execution time by a factor of 1.67. With prefetching, the average time for a load drops from 5.24 cycles to 3.53 cycles. Even though the cache hit ratios do not change, CG-A runs significantly faster because Impulse hides the latency of the memory system.

We introduced controller-based prefetching to Impulse primarily to hide the latency of scatter/gather operations, but it has proved useful on its own. Without scatter/gather support, controller-based prefetching improves performance by 4%, compared to the 12% performance improvement that can be achieved for this benchmark by performing a simple one-block-ahead prefetching mechanism at the L1 cache. However, controller-based prefetching requires no changes to the processor core, and thus can benefit processors with no integrated hardware prefetching.

Page recoloring. The first and third sections of Table 2 show that 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 heavily reused during SMVP, 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, and then color `DATA` and `COLUMN` so they each occupy one section.

Page recoloring consistently reduces the cost of memory accesses. Without prefetching, recoloring speeds execution time by a factor of 1.04. With the addition of prefetching at the controller, the speedup increases to 1.09. The effects of controller prefetching compared to L1 cache prefetching are similar to those with scatter/gather. Controller prefetching alone is about half as effective as either L1 cache prefetching or the combination of the two. Although the speedups for page recoloring are more modest than scatter/gather remapping, this optimization is nonetheless worthwhile. In addition, page recoloring benefits a much wider range of applications than scatter/gather (or any other fine-grained type of remapping).

4.2 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 [6].

Table 3 illustrates the results of our tiling experiments. The baseline is the conventional no-copy tiling. Software tile copying and tile remapping both outperform the baseline code by more than 95%, 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, giving rise to an average memory access time of approximately one cycle. Impulse tile remapping is slightly faster than tile copying, even when the overheads of the Impulse system calls and the associated cache flushes are taken into account.

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 $O(n^3)$, making the overhead of physical copying 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. In addition, our physical copying experiment avoids cross-interference between active tiles in both the L1 and L2 caches. Other authors have found that the performance of copying can vary greatly with matrix size, tile size, and cache size [31], but Impulse should be insensitive to cross-interference between tiles.

All forms of prefetching performed approximately equally for this application. The effectiveness of copying and tile remapping diminish the effects of prefetching. When the tiling optimizations are not being used, controller prefetching improves performance by about 2%. In contrast, L1 cache prefetching actually hurts performance slightly, due to the very low hit rate in the L1 cache and to the contention that prefetching introduces at the L2 cache.

4.3 Impact of Superscalar Processors

To measure the expected performance benefit of using Impulse on more modern processors, we modified our simulator to approximate a four-way superscalar machine. We do not change Paint's PA-RISC processor model, but we approximate a quad-issue superscalar machine by issuing up to four instructions each cycle (without checking dependencies). We leave the cache and bus models unchanged, thereby enforcing realistic limits on the rate of memory requests. While this model is unrealistic for gathering processor microarchitecture statistics, it stresses the memory system in a manner similar to a real superscalar processor [26].

Tables 4 and 5 summarize the results of running CG-A and DMMP on Impulse. The numbers in those two tables correspond directly to those in Tables 2 and 3, respectively. Note that even though the synthetic superscalar processor is nominally four times faster than the single-issue processor, it is less than 1.5 faster on our memory-bound benchmarks. As expected, the hit ratios on the superscalar are essentially the same as those on the single-issue processor.

Since the performance of our benchmarks is dominated by memory latency, Impulse delivers greater speedups on the superscalar processor. For example, the speedup for scatter/gather remapping using Impulse is 1.40, compared with 1.33 on the single-issue processor. Similarly, the speedup for CG-A with page recoloring is 1.07 (vs. 1.04 for the single-issue machine), and the speedup for DMMP with tile remapping is 3.41 (vs. 1.98 for the single-issue machine).

The speedup due to using Impulse's controller-based prefetching increases with issue width: 1.96 vs. 1.67 for CG-A using scatter/gather, and 1.14 vs. 1.09 for CG-A using page recoloring. The performance benefits of prefetching on the non-memory-bound versions of DMMP (the two tiled versions) are negligible: the raw execution times for this benchmark are nearly identical.

The performance benefits of L1 cache prefetching do not increase with instruction issue width for CG-A. With L1 prefetching, a miss to main memory takes 63 cycles to satisfy; with Impulse prefetching, a miss takes only 33 cycles. Even though the average time for a load is less with L1 prefetching (since the L1 hit rate is higher), those loads that go to memory cannot be hidden as effectively by the processor. As a result, Impulse prefetching outperforms L1 prefetching by 11%.

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 [20], or developing configurable architectures [35]. 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 [35] 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 profitably exploited by current processor architectures.

The RADram project at UC Davis is building a memory system that lets the memory perform computation [25]. RADram is a PIM, or *processor-in-memory*, project similar to IRAM [20]. The RAW project at MIT [33] 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 [19]. 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 [27] investigate the use of similar stream buffers to replace the L2 cache, and Farkas et al. [12] 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. All of these mechanisms prefetch cache lines speculatively, so they may bring unneeded data into the processor cache(s). This increases memory system bandwidth requirements, and decreases effective bandwidth. Farkas et al. mitigate this problem by implementing an incremental prefetching technique that reduces stream buffer bandwidth consumption by 50% without decreasing performance.

In contrast, systems that prefetch within the memory controller itself never waste bus bandwidth fetching unneeded data onto the processor chip. The Dynamic Access Ordering systems studied by McKee et al. [22] and Hong et al. [16] combine programmable stream buffers and prefetching within the memory controller with intelligent DRAM scheduling. For vector or streaming applications with predictable memory reference patterns, these systems dynamically reorder stream accesses to improve bus utilization, to exploit parallelism in the memory system (e.g., from multi-bank memories or sophisticated command interfaces), and to increase locality of reference with respect to the DRAM page buffers. In the same vein, Corbal et al. [11] propose a *Command Vector Memory System* that exploits parallelism and locality of reference to improve effective bandwidth for vector accesses on out-of-order vector processors with SDRAM memories. For SRAM memory systems, Valero et al. [32] show how reordering of strided accesses can be used to eliminate bank conflicts on a vector machine.

The Impulse DRAM scheduler that we are designing has similar goals to these other studies of dynamic access ordering. With Impulse, though, 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.

The Imagine media processor is a stream-based architecture with a bandwidth-efficient stream register file [28]. The streaming model of computation exposes parallelism and locality in applications, which makes such systems an attractive domain for intelligent DRAM scheduling.

A great deal of research has gone into prefetching into cache. For example, Chen and Baer [10] describe how a prefetching cache can outperform a non-blocking cache. Fu and Patel [13] use cache prefetching to improve memory hierarchy performance on vector machines, which is somewhat related to Impulse's scatter/gather optimization. Cache prefetching is orthogonal to Impulse's controller-based prefetching. In addition, our results show that controller prefetching can outperform simple forms of cache prefetching.

Yamada [34] proposed instruction set changes to support combined relocation and prefetching into the L1 cache. Relocation is done at the processor in this system, and thus no bus bandwidth is saved. In addition, because relocation is done on virtual addresses, the L2 cache utilization cannot be improved. With Impulse, the L2 cache utilization increases directly, and the operating system can then be used to improve L1 cache utilization.

Alexander and Kedem [1] describe a memory-based prefetching scheme that can significantly improve the performance of some applications. They use a prediction table to store up to four possible "next-access" predictions for any given memory address. When an address is accessed, the targets of the associated predictions are prefetched into SRAM buffers.

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, and it has two forms 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.

- The controller supports prefetching at the memory. Our results demonstrate that controller-based prefetching often performs as well as simple next-line prefetching in the L1 cache.

The combination of these features can result in substantial program speedups: using scatter/gather remapping and prefetching improves performance on the NAS conjugate gradient benchmark by 67%. Impulse's performance impact should be even greater on superscalar machines, where memory becomes a bigger bottleneck, and where non-memory instructions are effectively cheaper.

Flexible remapping support in the Impulse controller can be used to implement a variety of optimizations. In previous work [30], we showed that the Impulse memory remappings can be used to dynamically build superpages and thereby reduce the frequency of TLB faults. Impulse creates superpages from non-contiguous user pages. Our simulations show that this optimization improves the performance of five SPECint95 benchmark programs by 5-20%.

Although this simulation study focuses on two scientific kernels, 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 [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.

7 Acknowledgments

We thank Erik Brunvand, Al Davis, Chris Johnson, Chen-Chi Kuo, Ravindra Kuramkote, Mike Parker, Lambert Schaelicke, Terry Tateyama, Massimiliano Poletto, and Llewellyn Reese for their assistance in preparing earlier drafts of this paper. This work was sponsored in part by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under agreement number F30602-98-1-0101 and DARPA Order Numbers F393/00-01 and F376/00. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official polices or endorsements, either express or implied, of DARPA, AFRL, or the U.S. Government.

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, February 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 1–6, September 1990.

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

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

[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, October 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] 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, January 1999.

[9] 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.

[10] 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, October 1992.

[11] J. Corbal, R. Espasa, and M. Valero. Command vector memory systems: High performance at low cost. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, pages 68–77, October 1998.

[12] 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.

[13] 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, Ontario (Canada), May 1991.

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

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

[16] S. I. Hong, S. A. McKee, M. H. Salinas, R. H. Klenke, J. H. Aylor, and W. A. Wulf. Access order and effective bandwidth for streams on a direct rambus memory. In *Proceedings of the Fifth Annual Symposium on High Performance Computer Architecture*, pages 80–89, January 1999.

[17] 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.

[18] 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, October 1996.

[19] 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.

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

[21] 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.

[22] S. A. 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.

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

[24] 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.

[25] 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.

[26] V. S. Pai, P. Ranganathan, and S. V. Adve. The impact of instruction-level parallelism on multiprocessor performance and simulation methodology. In *Proceedings of the Third Annual Symposium on High Performance Computer Architecture*, pages 72–83, February 1997.

[27] 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.

[28] 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.

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

[30] 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.

[31] 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.

[32] 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.

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

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

[35] 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.

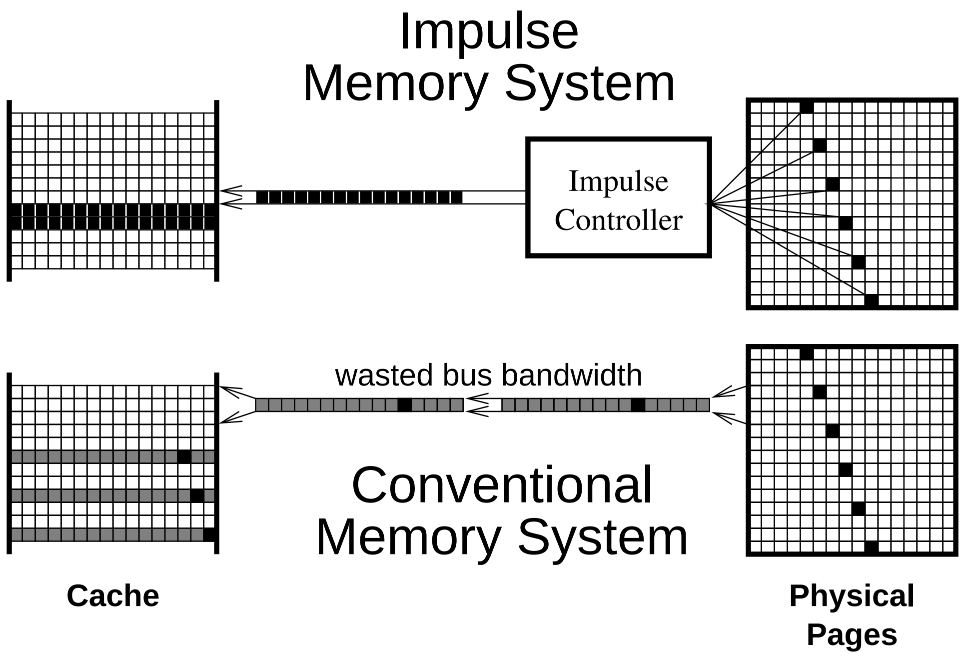


Figure 1: Using Impulse to remap the diagonal of a dense matrix into a dense cache line. The black boxes represent data on the diagonal, whereas the gray boxes represent non-diagonal data.

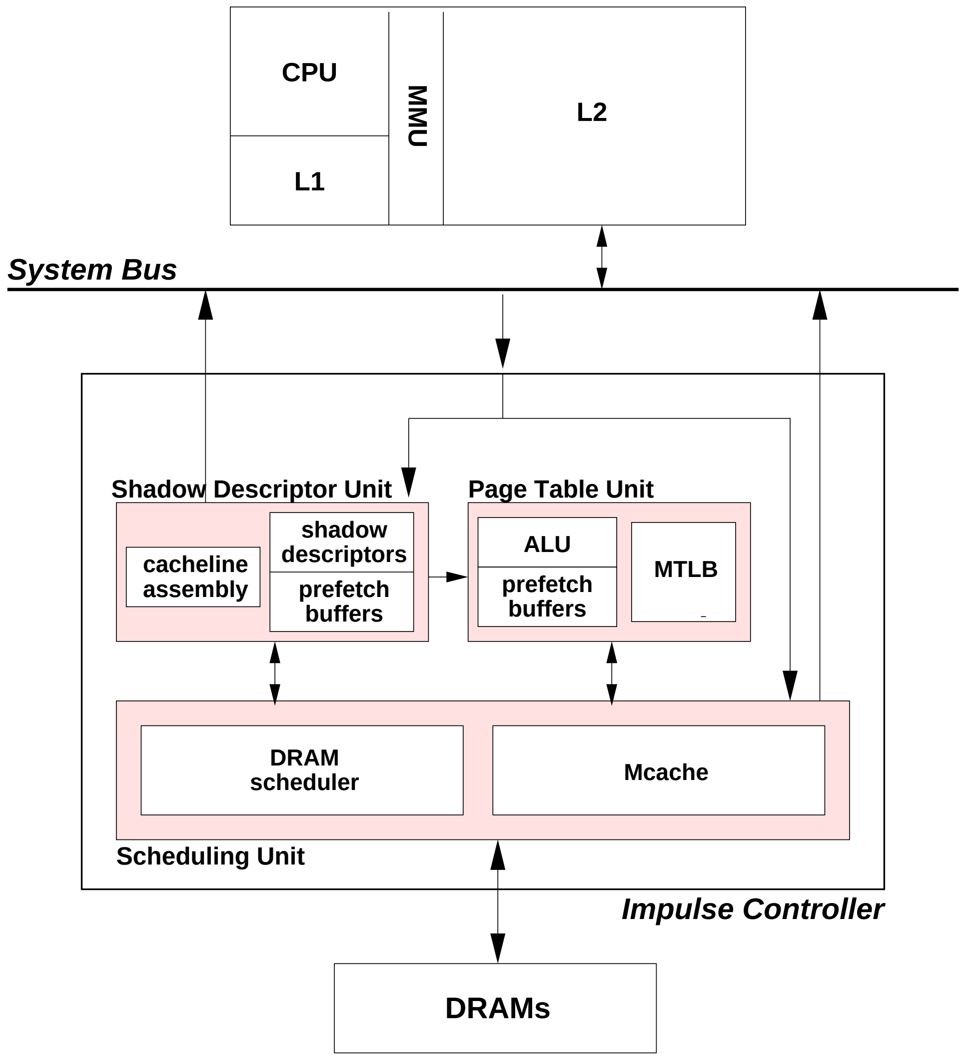


Figure 2: Impulse memory controller organization.

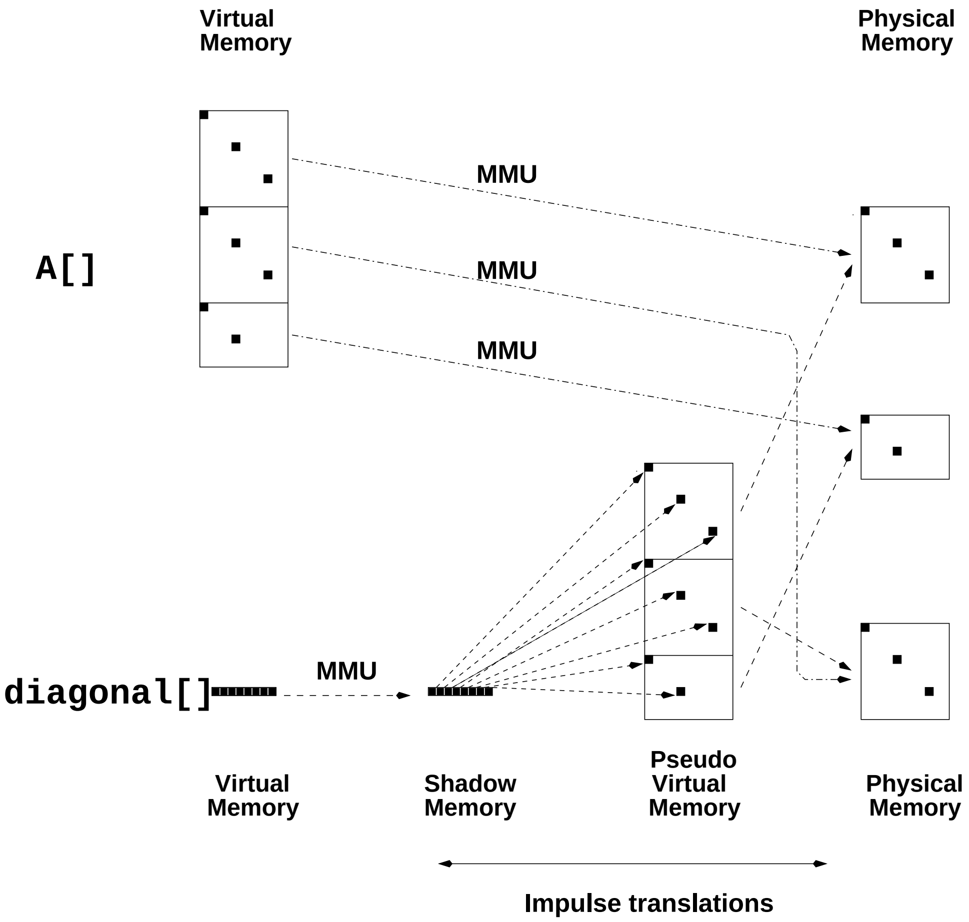
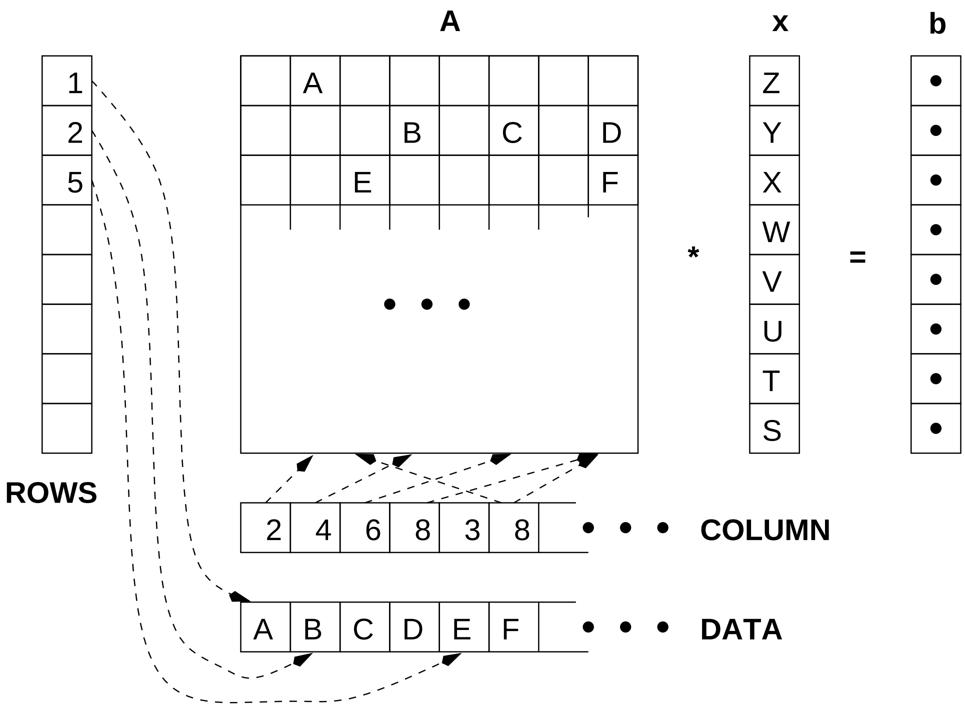


Figure 3: Using Impulse to remap memory: The translation on the top of the figure is the standard translation performed by an MMU. The translation on the bottom of the figure is the translation performed on an Impulse system. The processor translates virtual aliases into what it thinks are physical addresses; however, these physical addresses are really *shadow addresses*. The Impulse MC maps the shadow addresses into *pseudo-virtual addresses*, and then to physical memory.




```
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.

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

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.

	Standard	Prefetching		
		Impulse	L1 cache	both
Conventional memory system				
Time	2.81	2.69	2.51	2.49
L1 hit ratio	64.6%	64.6%	67.7%	67.7%
L2 hit ratio	29.9%	29.9%	30.4%	30.4%
mem hit ratio	5.5%	5.5%	1.9%	1.9%
avg load time	4.75	4.38	3.56	3.54
speedup	—	1.04	1.12	1.13
Impulse with scatter/gather remapping				
Time	2.11	1.68	1.51	1.44
L1 hit ratio	88.0%	88.0%	94.7%	94.7%
L2 hit ratio	4.4%	4.4%	4.3%	4.3%
mem hit ratio	7.6%	7.6%	1.0%	1.0%
avg load time	5.24	3.54	2.19	2.04
speedup	1.33	1.67	1.86	1.95
Impulse with page recoloring				
Time	2.70	2.57	2.39	2.37
L1 hit ratio	64.7%	64.7%	67.7%	67.7%
L2 hit ratio	30.9%	31.0%	31.3%	31.3%
mem hit ratio	4.4%	4.3%	1.0%	1.0%
avg load time	4.47	4.05	3.28	3.26
speedup	1.04	1.09	1.18	1.19

Table 2: Simulated results for the NAS Class A conjugate gradient benchmark, with various memory system configurations. Times are in billions of cycles; the hit ratios are the number of loads that hit in the corresponding level of the memory hierarchy divided by 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.

	Standard	Prefetching		
		Impulse	L1 cache	both
Conventional memory system				
Time	2.57	2.51	2.58	2.52
L1 hit ratio	49.0%	49.0%	48.9%	48.9%
L2 hit ratio	43.0%	43.0%	43.4%	43.5%
mem hit ratio	8.0%	8.0%	7.7%	7.6%
avg load time	6.37	6.18	6.44	6.22
speedup	—	1.02	1.00	1.02
Conventional memory system with software tile copying				
Time	1.32	1.32	1.32	1.32
L1 hit ratio	98.5%	98.5%	98.5%	98.5%
L2 hit ratio	1.3%	1.3%	1.4%	1.4%
mem hit ratio	0.2%	0.2%	0.1%	0.1%
avg load time	1.09	1.08	1.06	1.06
speedup	1.95	1.95	1.95	1.95
Impulse with tile remapping				
Time	1.30	1.29	1.30	1.28
L1 hit ratio	99.4%	99.4%	99.4%	99.6%
L2 hit ratio	0.4%	0.4%	0.4%	0.4%
mem hit ratio	0.2%	0.2%	0.2%	0.0%
avg load time	1.09	1.07	1.09	1.03
speedup	1.98	1.99	1.98	2.01

Table 3: Simulated results for tiled dense matrix-matrix product. Times are in billions of cycles; the hit ratios are the number of loads that hit in the corresponding level of the memory hierarchy divided by 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. The matrices are 512×512, with 32×32 tiles.

	Standard	Prefetching		
		Impulse	L1 cache	both
Conventional memory system				
Time	2.06	1.94	1.84	1.83
L1 hit ratio	64.7%	64.7%	67.6%	67.6%
L2 hit ratio	29.9%	29.9%	30.5%	30.4%
mem hit ratio	5.4%	5.4%	1.9%	2.0%
avg load time	4.73	4.35	3.74	3.72
speedup	—	1.06	1.12	1.13
Impulse with scatter/gather remapping				
Time	1.47	1.05	1.17	0.93
L1 hit ratio	88.1%	88.0%	94.7%	94.7%
L2 hit ratio	4.4%	4.4%	4.4%	4.4%
mem hit ratio	7.5%	7.6%	0.9%	0.9%
avg load time	5.13	3.44	2.11	1.95
speedup	1.40	1.96	1.76	2.22
Impulse with page recoloring				
Time	1.94	1.82	1.75	1.75
L1 hit ratio	64.8%	64.7%	67.7%	67.7%
L2 hit ratio	30.9%	31.0%	31.0%	30.9%
mem hit ratio	4.3%	4.3%	1.3%	1.4%
avg load time	4.38	4.00	3.50	3.49
speedup	1.07	1.14	1.18	1.18

Table 4: Simulated results for the NAS Class A conjugate gradient benchmark, with various memory system configurations and a synthetic four-way superscalar. Times are in billions of cycles; the hit ratios are the number of loads that hit in the corresponding level of the memory hierarchy divided by 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.

	Standard	Prefetching		
		Impulse	L1 cache	both
Conventional memory system				
Time	2.01	1.96	2.10	2.04
L1 hit ratio	49.0%	49.0%	48.9%	48.9%
L2 hit ratio	43.0%	43.0%	43.4%	43.4%
mem hit ratio	8.0%	8.0%	7.7%	7.7%
avg load time	6.46	6.27	6.81	6.61
speedup	—	1.03	0.96	0.99
Conventional memory system with software tile copying				
Time	0.60	0.60	0.59	0.59
L1 hit ratio	98.5%	98.5%	98.5%	98.5%
L2 hit ratio	1.3%	1.3%	1.4%	1.4%
mem hit ratio	0.2%	0.2%	0.1%	0.1%
avg load time	1.09	1.08	1.07	1.07
speedup	3.35	3.35	3.41	3.41
Impulse with tile remapping				
Time	0.59	0.59	0.59	0.58
L1 hit ratio	99.4%	99.4%	99.5%	99.6%
L2 hit ratio	0.4%	0.4%	0.3%	0.3%
mem hit ratio	0.2%	0.2%	0.2%	0.1%
avg load time	1.09	1.07	1.09	1.03
speedup	3.41	3.41	3.41	3.47

Table 5: Simulated results for tiled matrix-matrix product on a synthetic four-way superscalar. Times are in billions of cycles; the hit ratios are the number of loads that hit in the corresponding level of the memory hierarchy divided by 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. The matrices are 512 by 512, with 32 by 32 tiles.