Memory System Support for Irregular Applications

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

Department of Computer Science, University of Utah

Abstract. Because irregular applications have unpredictable memory access patterns, their performance is dominated by memory behavior. The Impulse configurable memory controller will enable significant performance improvements for irregular applications, because it can be configured to optimize memory accesses on an application-by-application basis. In this paper we describe the optimizations that the Impulse controller supports for sparse matrix-vector product, an important computational kernel, and outline the transformations that the compiler and runtime system must perform to exploit these optimizations.

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. One result of these trends is that it is becoming increasingly hard to make effective use of the tremendous processing power of modern microprocessors because of the difficulty of providing data in a timely fashion. For example, in a recent study of the cache behavior of the SQLserver database running on an Alphastation 8400, the system achieved only 12% of its peak memory bandwidth [11]; the resulting CPI was 2 (compared to a minimum CPI of 1/4). This factor of eight difference in performance is a compelling indication that caches are beginning to fail in their role of hiding the latency of main memory from processors. Other studies present similar results for other applications [4, 5].

Fundamentally, modern caches and memory systems are optimized for applications that sequentially access dense regions of memory. Programs with high degrees of spatial and temporal locality achieve near 100% cache hit rates, and will not be affected significantly as the latency of main memory increases. However, many important applications do not exhibit sufficient locality to achieve such high cache hit rates, such as sparse matrix, database, signal processing, multimedia, and CAD applications. Such programs consume huge amounts of memory bandwidth and cache capacity for little improvement in performance. Even applications with good spatial locality often suffer from poor cache hit rates, because of conflict misses caused by the limited size of processor caches and large working sets.

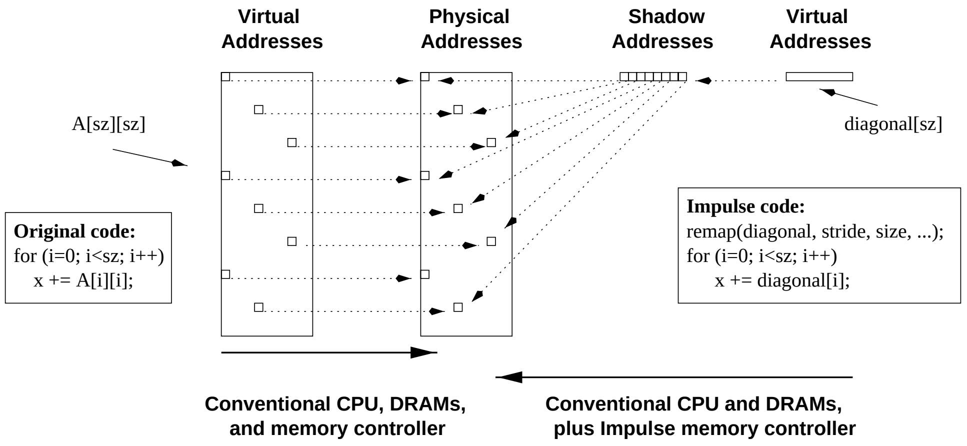


Fig. 1. Remapping shadow addresses using the Impulse memory controller. For clarity of exposition, we leave out how the Impulse controller supports non-contiguous physical pages for `A`.

A number of ongoing projects have proposed significant modifications to conventional CPU or DRAM designs to attack this memory problem: supporting massive multithreading [1], moving processing power on to the DRAM chips [6], or building completely programmable architectures [14]. While these projects show promise, unconventional CPU or DRAM designs will likely be slower than conventional designs, due to slow industry acceptance and the fact that processors built on current DRAM processes are 50% slower than conventional processors. In the Impulse project, we address the memory problem without modifying conventional CPUs, caches, memory busses, or DRAMs. Instead, we are building an *adaptable memory controller* that will enable programs to control how data is moved between cache and main memory at a fine grain, which will result in significantly improved memory performance for irregular applications. The Impulse memory controller implements this functionality by supporting an extra level of address translation in the memory controller.

The Impulse controller allows applications to make use of unused physical addresses, which it then translates into real physical addresses. Suppose a processor exports 32 address lines across the memory bus, but has only 1GB of memory installed. The remaining 3GB of physical address normally would be considered invalid. The Impulse controller makes use of these otherwise unused physical addresses by letting software specify *mapping functions* between these so-called *shadow addresses* and physical addresses directly backed by DRAM.

Consider a simple function that calculates the sum of the diagonal of a dense matrix, as illustrated in Figure 1. The left-hand side of the figure represents how the data would be accessed on a conventional system, where the desired diagonal elements are spread across physical memory. Each diagonal element is on a different cache line, and each such cache line contains only one useful element.

The right-hand side of the figure shows how data is accessed on an Impulse system. The application would specify that the contents of the `diagonal` vector are *gathered* using a simple strided function, using the `remap` operation. Once the `remap` operation has been performed, the memory controller knows to respond to requests for data in this shadow region by performing a gather-read operation from the physical memory storing `A`. The code then accesses the synthetic data structure, `diagonal`, which is mapped to a region of shadow memory. By accessing a dense structure, the application will see fewer cache misses, suffer less cache pollution, and more effectively utilize scarce bus bandwidth and cache capacity.

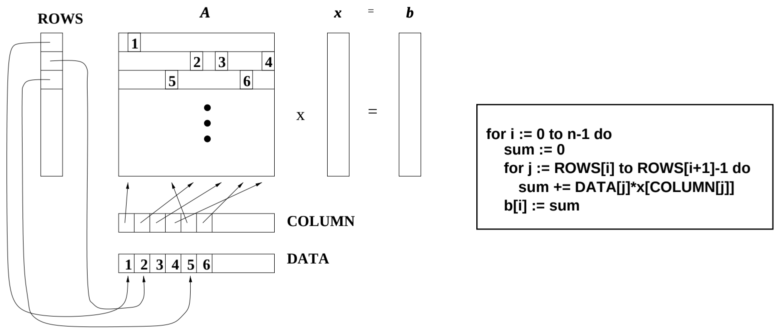
In Section 2, we use the sparse matrix-vector product algorithm from conjugate gradient to illustrate in detail two Impulse optimizations: (i) efficient scatter-gather access of sparse data structures and (ii) no-copy page recoloring to eliminate conflict misses between data with good locality and data with poor locality. In Section 3, we briefly discuss several other optimizations that are enabled by an Impulse memory system, describe related projects, and summarize our conclusions.

2 Sparse Matrix-Vector Multiplication

Sparse matrix-vector product is an irregular computational kernel critical to many large scientific algorithms. For example, most of the time spent performing a conjugate gradient computation [2] is spent in a sparse matrix-vector product. Similarly, the Spark98 [9] kernels are all sparse matrix-vector product codes. In this section, we describe several optimizations that Impulse enables to improve the performance of sparse matrix-vector muliplication, and sketch the transformations that the compiler and runtime system must perform to exploit these optimizations.

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 B input for the NAS Conjugate Gradient kernel involves a 75,000 by 75,000 sparse matrix with only 13,000,000 non-zeroes - far less than 1% of the entries. Although these encodings save tremendous amounts of memory, sparse matrix codes tend to suffer from poor memory performance because of the use of indirection vectors and the need to access sparse elements of the dense vector. In particular, when we profiled the NAS Class B conjugate gradient benchmark on a Silicon Graphics Origin 2000, we found that its L1 cache hit rate was only 58.5% and its L2 cache hit rate was only 92.7%.

Figure 2 illustrates the key inner loop and matrix encodings for conjugate gradient. Each iteration multiplies a row of the sparse matrix A with the dense vector x. On average, less than 1% of the elements of each row are non-zeroes, and these non-zeroes are "randomly" distributed. While matrix-vector product is trivially parallelizable or vectorizable, this code performs quite 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



```
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
```

Fig. 2. Conjugate gradient's sparse matrix-vector product. The sparse matrix A is encoded using three arrays: `DATA`, `ROWS`, and `COLUMN`. A is encoded densely 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]` is in, and thus which value of x must be fetched when performing the inner product.

system will fetch a cache line of data, of which only one element is used. Because of the large size of x and A and the sparse nature of accesses to x during each iteration of the loop, there will be very little reuse in the L1 cache — almost every access to x results in an L1 cache miss. If data layouts are carefully managed to avoid L2 cache conflicts between A and x, a large L2 cache can provide some reuse of x, but conventional memory systems do not provide simple mechanisms for avoiding such cache conflicts.

Because the performance of conjugate gradient is dominated by memory performance, it performs poorly on conventional architectures. Two previously studied solutions to this problem are to build main memory exclusively from fast SRAM or support massive multithreading. The Cray T916, whose main memory consists solely of fast (and expensive) SRAM, achieved 759.9 Mflops/second on the NAS CG benchmark in July 1995 [12]. The Tera MTA prototype achieves good single-node performance on the NAS conjugate gradient benchmark because its support for large numbers of cheap threads allows it to tolerate almost arbitrary memory latencies [3]. However, both of these solutions are very expensive compared to conventional hardware.

In Impulse, we take a different position — we employ off-the-shelf CPUs and memories, but build an adaptable memory controller. Our memory controller supports two major optimizations for improving the memory behavior of sparse matrix-vector product. First, it can perform scatter/gather operations, which trades temporal locality in the second-level cache for spatial locality in the first-level cache. Second, it can be used to color pages so as to better utilize physically indexed, second-level caches. We discuss these optimizations in the following two subsections.

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

Fig. 3. Code for sparse matrix-vector product on Impulse

2.1 Scatter/Gather

As described in Section 1, the Impulse memory controller will support scatter/gather reads and writes. We currently envision at least three forms of scatter/gather functions: (i) simple strides (ala the diagonal example), (ii) indirection vectors, and (iii) pointer chasing. The type of scatter/gather function that is relevant to conjugate gradient is the second, the use of indirection vectors.

The compiler technology necessary to exploit Impulse's support for scatter/gather is similar to that used in vectorizing compilers. Figures 3 and 4illustrate the code generated for the inner loop of conjugate gradient by a compiler designed to exploit the Impulse memory controller. The compiler must be able to recognize the use of indirection vectors, and download to the memory controller a description of the data structure being accessed (x) and the indirection vector used to specify which elements of this structure to gather (`COLUMN[]`). This code can be safely hoisted outside of the inner loops, as shown in Figure 3. After performing this setup operation, the compiler will then emit accesses to the remapped version of the gathered structure (x') rather than accesses to the original structure (x).

Scatter/gather memory operations take longer than accessing a contiguous cache line, since the memory controller needs to perform multiple actual DRAM accesses to fill the x' structure. The low-level design details of the Impulse DRAM scheduler are beyond the scope of this paper – suffice it to say that it is possible to pipeline and parallelize many of the DRAM accesses required to read and write the sparse data through careful design of the controller. For optimal performance, the compiler should tell the memory controller to sequentially prefetch the elements of `x'` (i.e., pre-gather the next line of `x'`), since it can tell that `x'` is accessed sequentially. When all of these issues are considered, the compiler should unroll the inner loop according to the cache line size and software pipeline accesses to the multiplicand vector, as shown in Figure 4. In this example, we assume a cache line contains four doubles; for clarity we assume that each row contains a multiple of eight doubles.

These optimizations improve performance in two ways. First, fewer memory instructions need to be issued. Since the read of the indirection vector (`COLUMN[]`) occurs at the memory, the processor does not need to issue the read. Second, spatial locality is improved in the L1 cache. Since the memory controller packs

```
setup x'[k] = x[COLUMN[k]]
for i := 0 to n-1
  lo := ROWS[i]
  hi := ROWS[i+1]-1
  sum := DATA[lo]*x'[lo]
  for j := lo to hi step 8 do
    sum += DATA[j+4]*x'[j+4]
    sum += DATA[j+1]*x'[j+1]
    sum += DATA[j+2]*x'[j+2]
    sum += DATA[j+3]*x'[j+3]
    sum += DATA[j+8]*x'[j+8]
    sum += DATA[j+5]*x'[j+5]
    sum += DATA[j+6]*x'[j+6]
    sum += DATA[j+7]*x'[j+7]
  sum -= DATA[hi+1]*x'[hi+1]
  b[i] := sum
```

Fig. 4. Code for sparse matrix-vector product on Impulse with strip mining and software pipelining

the gathered elements into cache lines, the cache lines contain 100% useful data, rather than only one useful element each.

Since a detailed simulation of the scatter/gather capability is not yet available, we performed a simple analysis of its impact. If we examine just the inner loop of the original algorithm, it is dominated by the cost for performing three loads (to `DATA[i]`, `COLUMN[i]`, and `x[COLUMN[i]]`). If we assume a 32-byte cache line that can hold four doubles, we can see where Impulse wins in Table 1. Conv. (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. Conv. (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 frequent conflicts between x and other structures cause the useful elements of x to be conflicted out of the L2 cache before they can be reused in a subsequent iteration. Because x is not accessed directly in Impulse, its best and worst case are identical.

As we see in Table 1, Impulse eliminates 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'. Compared to the worst case for a conventional system, Impulse eliminates both the four L2 hits and four misses to main memory. Note that except on machines with large L2 caches and very careful data layout, the worst case is also the expected case because of the large size of x and its poor locality. If we assume that software pipelining and prefetching will hide cold misses to linearly accessed data, the misses to `DATA[i]`, `COLUMN[i]`, and `x'[i]` can be hidden. In this case, using Impulse will allow the processor to perform floating point operations as fast as the memory system can

Load	Conv. (Best)	Conv. (Worst)	Impulse
`DATA[i]`	miss	miss	miss
`COLUMN[i]`	.5 miss	.5 miss	
`x[COLUMN[i]]`	L2 hit	miss	
`x'[i]`			miss*
`DATA[i+1]`	L1 hit	L1 hit	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	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	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. The starred miss requires a gather at the memory controller.

supply two streams of dense data (x' and `DATA`). However, since the non-linear acccess to `x[COLUMN[i]]` cannot be hidden, a conventional machine will suffer frequent high latency read misses, thereby dramatically reducing performance.

It is important to note that the use of scatter/gather at the memory controller reduces temporal locality in the second-level cache. The remapped elements of x' are themselves never reused, whereas a carefully tuned implementation of CG would be able to reuse many elements of x cached in a large L2 cache. Such a situation would approach the best case column of Figure 1, modulo conflict effects in the L2 cache. In the next section, we propose a way to achieve the best case using Impulse.

2.2 Page Coloring

As an alternative to gathering elements of `x`, which achieves the performance indicated under the Impulse column of Table 1, Impulse allows applications to manage the layout of data in the L2 cache. In particular, on an Impulse machine, we can achieve the best-case memory performance for a conventional machine on conjugate gradient by performing *page coloring*.

Page coloring optimizes the physical memory layout of data structures so that data with good temporal locality (e.g., x) is mapped to a different part of a physically-indexed cache than data with poor temporal locality (e.g., `DATA`, `ROW`, and `COLUMN`). In a physically-indexed second level cache, physical pages from different data structures can wind up being mapped to the same location. As a result, data in these pages will tend to conflict with one another. For conjugate gradient, 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. Thus, we would like to ensure that x does not conflict in the L2 cache with the other data structures. In the NAS CG benchmarks, x ranges from 600 kilobytes to 1.2 megabytes, and the other structures range from 100-300 megabytes. Thus, x will not fit in most processor's L1 caches, but can fit in many L2 caches.

On a conventional machine, page coloring is hard to exploit. Coloring requires that the system not map data to physical pages that have the same "color" (i.e., map to the same portion of the cache) as data with good locality. If you wish to devote a large percentage of your cache to data with good temporal locality, this effectively means that you cannot use large portions of your physical memory, which is impractical for problems with large data sets. In addition, coloring often requires data to be copied between physical pages to eliminate conflicts, which is expensive.

Impulse eliminates both of these problems, which makes page coloring simple and efficient. Since Impulse can map data structures to *shadow* physical addresses, rather than real physical addresses, and because wasting shadow address space does not waste physical memory, data with good locality can be mapped to pages that do not conflict with other data. And, because of the extra level of translation, this recoloring can be done without copying. In the case of conjugate gradient, Impulse can be used to remap x to pages that map to 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 either x or each other. In effect, we can use a small part of the second-level cache, e.g., two pages each for `DATA`, `ROWS`, and `COLUMNS`, as a stream buffer [8].

3 Conclusions

We have described two optimizations that the Impulse controller supports for irregular applications. An Impulse system can also improve the performance of dense matrix kernels. Dense matrices are tiled to improve cache behavior, but the effectiveness of tiling is limited by the fact that tiles do not necessarily occupy contiguous blocks in caches. Tile copying [7] can improve the performance of tiled algorithms by reducing cache conflicts, but the cost of copying is fairly high. The Impulse controller allows tiles to be copied *virtually.* The cost that virtual copying incurs is that read-write sharing between virtual copies requires cache flushing to maintain coherence.

An Impulse memory controller can be used to dynamically build superpages, so as to save processor TLB entries [13]. This optimization can dramatically improve the performance of applications with large working sets. Unfortunately, because the physical memory associated with a superpage must be contiguous and aligned, superpages cannot easily be used to map user data on conventional memory systems. However, by using shadow memory to map superpages, and then remapping the shadow addresses back to the non-contiguous physical addresses corresponding to the user data, Impulse can create superpages for arbitrarily shuffled user data.

An Impulse memory system also helps support efficient message passing. First, its support for scatter/gather means that the memory controller, rather than software, can handle the chore of gathering data into messages. As a result, the Impulse controller can reduce the overhead of sending a message. Second, its support for building superpages means that network interfaces need not perform complex and expensive address translation. Instead, an Impulse controller could handle the translations necessary for messages that span multiple pages.

Other research projects are looking at similar issues in designing memory systems; we briefly describe a few of them. The Morph architecture [14] 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 Impulse project is designing hardware that will be built in an academic environment, which requires that we attack hardware outside of the processor. The RADram project at UC Davis is building a memory system that lets the memory perform computation [10]. RADram is a PIM ("processor-in-memory") project similar to IRAM [6], where the goal is to put processors close to memory. In contrast, Impulse does not seek to put a processor in memory; instead, its memory controller is programmable.

In summary, the Impulse project is developing a memory system that will help to improve the memory performance of irregular applications. We have used sparse matrix-vector product as our motivating example, but we expect that the ability to configure an Impulse memory system to various memory access patterns will be useful for a wide range of irregular problems. We are investigating both manual and automatic mechanisms for taking advantage of the flexibility that Impulse provides.

References

1. R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera computer system. In *Proceedings of the International Conference on Supercomputing*, pages 272–277, Amsterdam, The Netherlands, June 1990.
2. D. Bailey et al. The NAS parallel benchmarks. Technical Report RNR-94-007, NASA Ames Research Center, Mar. 1994.
3. J. Boisseau, L. Carter, K. S. Gatlin, A. Majumdar, and A. Snavely. NAS benchmarks on the Tera MTA. In *Proceedings of the Multithreaded Execution Architecture and Compilation*, Las Vegas, NV, Jan. 31–Feb. 1, 1998.
4. 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.
5. 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.

6. C. E. Kozyrakis et al. Scalable processors in the billion-transistor era: IRAM. *IEEE Computer*, pages 75–78, Sept. 1997.
7. 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.
8. S. McKee and W. A. Wulf. Access ordering and memory-conscious cache utilization. In *Proceedings of the First IEEE Symposium on High Performance Computer Architecture*, pages 253–262, Raleigh, NC, Jan. 1995.
9. 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.
10. 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*, June 1998. To appear.
11. S. E. Perl and R. Sites. Studies of Windows NT performance using dynamic execution traces. In *Proceedings of the Second Symposium on Operating System Design and Implementation*, pages 169–184, October 1996.
12. S. Saini and D. H. Bailey. NAS parallel benchmark (version 1.0) results. Technical Report NAS-96-18, NASA Ames Research Center, Moffett Field, CA, Nov. 1996.
13. 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.
14. 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.