

Memory System Support for Dynamic Cache Line Assembly

Lixin Zhang, Venkata K. Pingali, Bharat Chandramouli, and John B. Carter

School of Computing
University of Utah
Salt Lake City, UT 84112
{lizhang, kmohan, bharat, retrac}@cs.utah.edu
<http://www.cs.utah.edu/impulse/>

Abstract. The effectiveness of cache-based memory hierarchies depends on the presence of spatial and temporal locality in applications. Memory accesses of many important applications have predictable behavior but poor locality. As a result, the performance of these applications suffers from the increasing gap between processor and memory performance. In this paper, we describe a novel mechanism provided by the Impulse memory controller called *Dynamic Cache Line Assembly* that can be used by applications to improve memory performance. This mechanism allows applications to gather on-the-fly data spread through memory into contiguous cache lines, which creates spatial data locality where none exists naturally. We have used dynamic cache line assembly to optimize a random access loop and an implementation of Fast Fourier Transform (FFTW). Detailed simulation results show that the use of dynamic cache line assembly improves the performance of these benchmarks by up to a factor of 3.2 and 1.4, respectively.

1 Introduction

The performance gap between processors and memory is widening at a rapid rate. Processor clock rates have been increasing 60% per year, while DRAM latencies have been decreasing only 7% per year. Computer architects have developed a variety of mechanisms to bridge this performance gap including out-of-order execution, non-blocking multi-level caches, speculative loads, prefetching, cache-conscious data/computation transformation, moving computation to DRAM chips, and memory request reordering. Many of these mechanisms achieve remarkable success for some applications, but none are particularly effective for irregular applications with poor spatial or temporal locality. For example, no

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

systems based on conventional microprocessors can handle the following loop efficiently if the array `A` is sufficiently large:

```
float A[SIZE];
for (i = 0; i < itcount; i++) {
    sum += A[random()%SIZE];
}
```

We are developing a memory system called Impulse that lets applications control how, when, and what data is placed in the processor cache [2]. We do this by adding an optional extra level of *physical-to-physical* address translation at the main memory controller (MMC). This extra level of translation enables optimizations such as “gathering” sparse data into dense cache lines, no-copy page coloring, and no-copy superpage creation. In this paper, we describe a new mechanism called *dynamic cache line assembly* that we are considering for Impulse. This mechanism allows applications to request that a cache line be loaded on-the-fly with data from disjoint parts of memory. Applications that can determine the addresses that they will access in the near future can request that data from those addresses be fetched from memory. This mechanism lets applications create spatial locality where none exists naturally and works in situations where prefetching would fail due to bandwidth constraints. Simulation indicates that dynamic cache line assembly improves the performance of the random access loop above by a factor of 3.2 and the performance of the dominant phase of FFTW by a factor of 2.6 to 3.4.

The rest of the paper is organized as follows. Section 3 briefly describes the basic technology of Impulse. Section 4 presents the design details of dynamic cache line assembly. Section 5 studies the performance evaluation of the proposed mechanism. And finally, Section 6 discusses future work and concludes this paper.

2 Related Work

Much work has been done to increase the spatial and temporal locality of regular applications using static analysis. Compiler techniques such as loop transformations [1] and data transformations [3] have been useful in improving memory locality of applications. However, these methods are not well suited to tackle the locality problem in irregular applications where the locality characteristics are not known at compile time.

A hybrid hardware/software approach to improving locality proposed by Yamada et al. [12] involves memory hierarchy and instruction set changes to support combined data relocation and prefetching into the L1 cache. Their solution uses a separate relocation buffer to translate array elements’ virtual addresses into the virtual relocation buffer space. The compiler inserts code to initiate the remapping, and it replaces the original array references with corresponding relocation buffer references. However, this approach can only relocate strided array references. Also, it saves no bus bandwidth because it performs relocation

at the processor. Contention for cache and TLB ports could be greatly increased because the collecting procedure of each relocated cache line must access the cache and CPU/MMU multiple times. This approach is also not designed for irregular applications.

There has been some work in developing dynamic techniques for improving locality. Ding and Kennedy [5] introduce the notion of dynamic data packing, which is a run time optimization that groups data accessed at close intervals in the program into the same cache line. This optimization is efficient only if the gathered data is accessed many times to amortize the overhead of packing and if the access order does not change frequently during execution. DCA setup incurs much lesser overhead because it does not involve data copying, and it allows frequent changes to the indirection vector.

To the best of our knowledge, hardware support for general-purpose cache line gathering such as is supported by DCA is not present in any architecture other than Cray vector machines. For example, the Cray T3E [10] provides special support for single-word load. Sparse data can be gathered into contiguous E-registers and the resulting blocks of E-registers can then be loaded “broad-side” into the processor in cache line sized blocks, thus substantially reducing unnecessary bus bandwidth that would have been used in normal cache line fills. Dynamic cache line assembly provides similar scatter gather capability for conventional microprocessors without the need for special vector registers, vector memory operations in the instruction set, or an SRAM main memory.

3 Impulse Architecture

The Impulse adaptable memory system expands the traditional virtual memory hierarchy by adding address translation hardware to the main memory controller (MMC) [2, 11, 14]. Impulse uses physical addresses unused in conventional systems as remapped aliases of real physical addresses. For instance, in a system with 32-bit physical addresses and one gigabyte of installed DRAM, the physical addresses inside $[0x40000000 - 0xFFFFFFFF]$ normally would be considered invalid.¹ Those, otherwise unused, physical addresses refer to a *shadow address space*.

Figure 1 shows how addresses are mapped in an Impulse system. The real physical address space is directly backed up by physical memory; its size is exactly the size of installed physical memory. The shadow address space does not directly point to any real physical memory (thus the term *shadow*) and must be remapped to real physical addresses through the Impulse MMC. How the MMC interprets shadow addresses presented to it is configured by the operating system.

This virtualization of unused physical addresses can provide different views of data stored in physical memory to programs. For example, it can create cache-

¹ It is common to have I/O devices mapped to special “high” addresses. This problem can be easily avoided by not letting shadow address space overlap with I/O devices addresses.

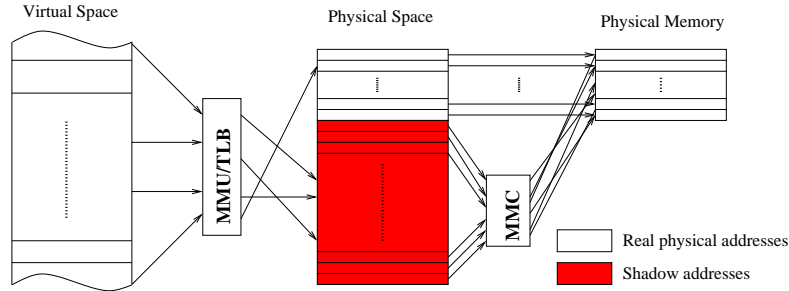


Fig. 1. Address mapping in an Impulse system.

friendly data structures to improve the efficiency of the processor caches. The operating system manages all of the resources in the expanded memory hierarchy and provides an interface for the application to specify optimizations for particular data structures. The programmer (or the compiler) inserts appropriate system calls into the application code to configure the memory controller.

To map a data item in the shadow address space to the physical memory, the Impulse MMC must first recover its virtual address. To avoid directly handling virtual addresses at the MMC, we require that the virtual address must be located inside a special virtual region. The OS creates a dense, flat page table in contiguous physical addresses for the special virtual region. We call the page table the *memory controller page table*. The OS then pins down this page table in main memory and sends its starting physical address to the memory controller so that the MMC can access this page table without interrupting the OS. Since data items in a shadow region are mapped to a special virtual region, the MMC only need compute offsets relative to the starting address of the virtual region. We call such an offset a *pseudo-virtual address*. For each shadow data item, the MMC first computes its pseudo-virtual address, then uses the memory controller page table to determine the data item’s real physical address. To speed up the translation from pseudo-virtual to physical addresses, the MMC uses an TLB to store recently used translations. We call this TLB the *MTLB*.

Figure 2 shows a simplified block diagram of the Impulse memory system. The critical component of the Impulse MMC is the shadow engine, which processes all shadow accesses. The shadow engine contains a small *scatter/gather SRAM buffer* used as a place to scatter/gather cache lines in the shadow address space, some *control registers* to store remapping configuration information, an ALU unit (*AddrCalc*) to translate shadow addresses to pseudo-virtual addresses, and a *Memory Controller Translation Lookaside Buffer (MTLB)* to cache recently used translations from pseudo-virtual addresses to physical addresses. The control registers are split into eight different sets and are capable of saving configuration information for eight different mappings. However, all mappings share the same ALU unit and the same MTLB.

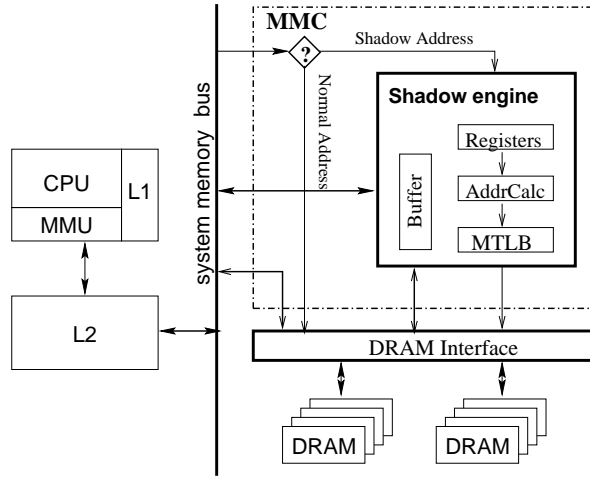


Fig. 2. Impulse architecture.

4 Design

The proposed dynamic cache line assembly mechanism is an extension of Impulse’s *scatter/gather through an indirection vector* remapping mechanism. Its main goal is to enable applications to access data spread through memory as if it were stored sequentially. In this section, we first talk about scatter/gather through an indirection vector, then describe the design of dynamic cache line assembly.

4.1 Scatter/Gather through An Indirection Vector

The Impulse system supports a remapping called *scatter/gather through an indirection vector*. For simplicity, we refer to it as *IV remapping* throughout the rest of this paper. IV remapping maps a region of shadow addresses to a data structure such that a shadow address at offset *soffset* in the shadow region is mapped to data item addressed by *vector[soffset]* in the physical memory.

Figure 3 shows an example of using IV remapping on a sparse matrix-vector product algorithm. In this example, P_i is an alias array in the shadow address space. An element $P_i[j]$ of this array is mapped to element $P[\text{ColIdx}[j]]$ in the physical memory by the Impulse memory controller. For a shadow cache line containing elements $P_i[j], P_i[j+1], \dots, P_i[j+k]$, the MMC fetches elements $P[\text{ColIdx}[j]], P[\text{ColIdx}[j+1]], \dots, P[\text{ColIdx}[j+k]]$ one by one from the physical memory and packs them into a dense cache line.

Figure 4 illustrates the gathering procedure. The shadow engine contains a one cache line SRAM buffer to store indirection vectors. We call this SRAM buffer the *IV buffer*. When the MMC receives a request for a cache line of $P_i[j]$, it loads the corresponding cache line of $\text{ColIdx}[j]$ into the IV buffer, if the IV buffer

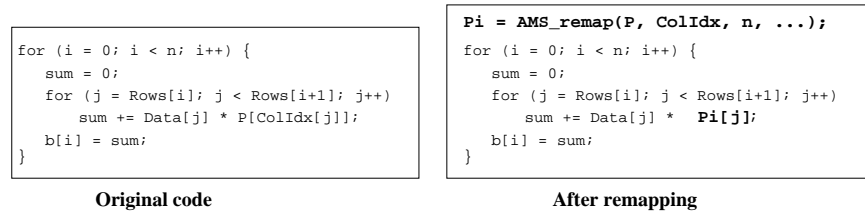


Fig. 3. Scatter/gather through an indirection vector changes indirect accesses to sequential accesses.

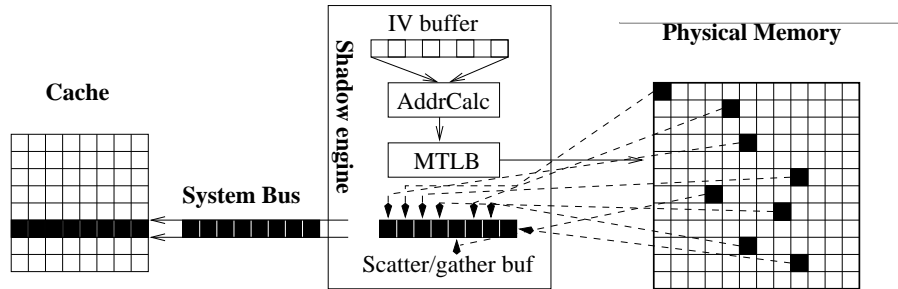


Fig. 4. Visualize the gathering procedure through an indirection vector.

does not already contain it. The MMC then can interpret one element of the indirection vector per cycle. The indirection vector may store virtual addresses, array indices, or even real physical addresses. What it stores (addresses or indices) is specified when the remapping is configured; without loss of generality, we will refer to the contents of the IV buffer generically as “addresses” throughout the rest of this paper. If the IV buffer stores virtual address or array indices, the MMC passes each entry to the AddrCalc unit to generate a pseudo-virtual address and translates the pseudo-virtual address to a physical address using the MTLB. Once the MMC has a physical address for a data element, it uses this address to access physical memory. When a data item returns, it is packed into a dense cache line in the scatter/gather buffer.

By mapping sparse, indirectly addressed data items into packed cache lines, *scatter/gather through an indirection vector* enables applications to replace indirect accesses with sequential accesses. As a result, applications reduce their bus bandwidth consumption, the cache footprint of their data, and the number of memory loads they must issue.

A naive implementation of IV remapping requires that the indirection vector exists in the program and its number of elements be the same as the number of data items being gathered, e.g., `Rows[n]` in Figure 3. We extend IV remapping to implement *dynamic cache line assembly*, which can be used by programs where *no indirection vector exists naturally* and where the size of an indirection vector need not be equal to the number of data items being gathered.

4.2 Dynamic Cache Line Assembly

The basic idea of *dynamic cache line assembly* is to create indirection vectors dynamically during program execution and to access them using Impulse's IV remapping mechanism. The indirection vectors typically are small, usually smaller than a page. We choose small indirection vectors because accessing them and the resulting small alias arrays leaves a very small footprint in the cache. The small indirection vectors and alias arrays can be reused to remap large data structures.

To use DCA, the application performs a system call to allocate two special ranges of shadow addresses and have the operating system map new virtual addresses to these two ranges. The first range is used to store the addresses from which the application wishes to load data (we call this the *address region*), while the second range is used to store the requested data (we call this the *data region*). The number of addresses that can be stored in the address region is the same as the number of data items that can be stored in the data region. There is a one-to-one mapping between elements of the two ranges: the i^{th} element of the address region is the address of the i^{th} element of the data region. The operating system also allocates a contiguous real physical memory to back up the address region in case an indirection vector is forced out of the IV buffer before the MMC has finished using it.

After setting up the regions, the operating system informs the MMC of their location, as well as the size of each address and the size of the object that needs to be loaded from each address. For simplicity, we currently require that both the address and data regions are a multiple of a cache line size, and that the data objects are a power of two bytes.

After setup, the application can exploit dynamic cache line assembly by filling a cache line in the address region with a set of addresses and writing it back to memory through a cache flush operation. When the MMC sees and recognizes this write-back, it stores the write-back cache line into both the IV buffer and the memory. Storing the write-back into the memory is necessary because the IV buffer may be used by another writeback. When the MMC receives a load request for a cache line in the data region, it checks to see if the IV buffer contains the corresponding cache line in the address region. If the required cache line is not in the IV buffer, the shadow engine loads it from memory. The MMC then interprets the contents of the IV buffer as a set of addresses, passes these addresses through the AddrCalc unit and the MTLB to generate the corresponding physical addresses, fetches data from these physical addresses, and stores the fetched data densely into the scatter/gather buffer inside the shadow engine. After an entire cache line has been packed, the MMC supplies it to the system bus from the scatter/gather buffer.

Figure 5 shows how dynamic cache line assembly can be used to improve the performance of the random access loop presented in Section 1. In this example, we assume that L2 cache lines are 128 bytes, so each cache line can hold 32

```

float *aliasarray;
int *idxvector;
/* aliasarray[i] <= A[idxvector[i]] */
setup_call(A, SIZE, 32, &aliasarray, &idxvector);
for (i = 0; i < itcount/32; i++) {
    for (k = 0; k < 32; k++)
        idxvector[k] = & (A[random()%SIZE]);
    flush_cache_line(idxvector);
    memory_barrier();
    for (k = 0; k < 32; k++)
        sum += aliasarray[k];
    purge_cache_line(aliasarray);
}

```

Fig. 5. Using *dynamic cache line assembly* on the random access loop

addresses or 32 floats². The program first allocates a 32-element address region `idxvector` and a 32-element data region `aliasarray` through the system call `setup_call()`. In each iteration, the application fills a cache line's worth of the address region with addresses, flushes it, and then reads from the corresponding shadow data region to access the data. Traditional microprocessors with out-of-order execution usually give reads higher priority than writes, so the read request for the data may be issued before the flush occurs. To ensure this does not happen, we insert a *memory barrier* after the flush. Since the same data region is used in every iteration, the old data in the cache must be invalidated so that the next fetch will go to the MMC to retrieve the right data.

The main overhead of using dynamic cache line assembly is that the program must handle the address regions (i.e., indirection vectors) and the MMC must gather a cache line using multiple DRAM fetches. In this example, filling a cache line of the indirection vector introduces 32 sequential memory accesses that do not exist in the original code. Fortunately, those 32 accesses generate only one cache miss. Accessing a cache line of the data region results in another cache miss. As a result, the Impulse version has 2 cache misses for every 32 data accesses. In the original code, however, the same 32 data accesses generate roughly $(32 * (1 - \text{sizeof}(\text{cache}) / \text{sizeof}(A)))$ cache misses when the array `A` is larger than the cache. Dynamically gathering a cache line in the MMC is much more expensive than fetching a single dense region of memory. Consequently, the code in Figure 5 will be memory-bound because there is a long waiting time in each

² It is not required that the objects being fetched are the same size as an address. If, for example, you want to dynamically fetch quad-precision floating point numbers (16 bytes), each time the application flushes a line of addresses, the MMC will fetch four cache lines full of data.

iteration for the MMC packing and returning data. However, its performance can be improved using unroll-and-jam.

```

#define PrecomputeAddresses(start, end)          \
    for (k = start; k < end; k++)              \
        idxvector[k] = &(A[random()%SIZE]);    \
    flush_cache line(&(idxvector[start]));      \
    memory_barrier();                           \
    prefetch_cache line(&(aliasarray[start]));

#define AccessData(start, end)                  \
    for (l = start; l < end; l++)              \
        sum += aliasarray[l];                  \
    purge_cache line(&(aliasarray[start]));

float *aliasarray;
int *idxvector;
/* aliasarray[i] <= A[idxvector[i]] */
setup_call(A, SIZE, 64, &aliasarray, &idxvector);
PrecomputeAddresses(0, 32);
for (i = 0; i < itcount/64 - 1; i++) {
    PrecomputeAddresses(32, 64);
    AccessData(0, 32);
    PrecomputeAddresses(0, 32);
    AccessData(32, 64);
}
.....

```

Fig. 6. Using unroll-and-jam with dynamic cache line assembly.

Figure 6 illustrates how unroll-and-jam can be used to improve performance. By using two cache lines for each of the address and data regions, we can overlap computation on one line's worth of data while prefetching the next line, thereby hiding the long memory latency of dynamic cache line gathering. To support this optimization, we need to increase the size of the IV buffer to two cache lines. With software unroll-and-jam, the processor may flush back one cache line of the address region while the MMC is gathering data from another set of address. With two cache lines in the IV buffer, the second write-back can be saved in the buffer instead of being written back to DRAM and then reloaded when needed.

5 Performance Evaluation

We evaluated the performance of dynamic cache line assembly using execution-driven simulation. We compared the performance of two benchmarks using dynamic cache line assembly with the same benchmarks unmodified. The first benchmark is the synthetic “random walk” microbenchmark described in Section 1. The second benchmark is a three-dimensional FFT [6] program from DIS benchmark suite [7]. Both benchmarks were compiled using the SPARC SC4.2 compiler with the `-xO4` option to produce optimized code.

5.1 Simulation Environment

Our studies use the execution-driven simulator URSIM [13] derived from RSIM [9]. URSIM models a microprocessor close to MIPS R1000 [8] and a split-transaction MIPS R10000 cluster bus with a snoopy coherence protocol. It also simulates the Impulse adaptable memory system in great detail. The processor is a four-way, out-of-order superscalar with a 64-entry instruction window. The D/I unified TLB is single-cycle, fully associative, software-managed, and has 128 entries. The instruction cache is assumed to be perfect. The 64-kilobyte L1 data cache is non-blocking, write-back, virtually indexed, physically tagged, direct-mapped, and has 32-byte lines and one-cycle latency. The 512-kilobyte L2 data cache is non-blocking, write-back, physically indexed, physically tagged, two-way associative, and has 128-byte lines and eight-cycle latency. The split-transaction bus multiplexes addresses and data, is eight bytes wide, has a three-cycle arbitration delay and a one-cycle turn-around time. The system bus, memory controller, and DRAMs have the same clock rate, which is one third of the CPU clock. The memory supports critical word first. It returns the critical quad-word for a load request 16 bus cycles after the corresponding L2 cache miss occurs. The memory system contains 8 banks, pairs of which share an eight-byte wide bus between DRAM and the MMC.

The address translation procedure in the shadow engine is fully pipelined. For each shadow access entering the pipeline, the engine generates the first physical address four cycles later and one physical address per cycle afterwards, provided that no MTLB miss occurs. On an MTLB miss, the pipeline is stalled until the required page table entry has been loaded into the MTLB. The MTLB is configured to be four-way associative, with 256 entries and a one-memory-cycle lookup latency.

5.2 Results

The performance results presented here are obtained through complete simulation of the benchmarks, including both kernel and application time, the overhead of setting up and using dynamic cache line assembly, and the resulting effects on the memory system.

	Elapsed cycles	TLB hit rate	L1 hit rate	L2 hit rate	Miss rate	Memory latency	Speedup
Base	196M	73.67%	65.68%	7.76%	26.56%	51 cycles	
Impulse	61M	99.97%	93.37%	6.21%	0.42%	113 cycles	3.22

Table 1. Performance results for the microbenchmark.

Microbenchmark In the synthetic microbenchmark, `A[]` holds one million elements and two million random accesses are performed. In theory, its cache hit rate should be the size of the cache divided by the size of `A[]`. So larger `A[]` has smaller cache hit rate and likely yields better performance improvement with dynamic cache line assembly. We choose `A[]` to contain one million elements simply because it is large enough to prove the effectiveness of dynamic cache line assembly and its simulation can complete in a reasonable amount of time.

Table 1 presents the results of this experiment. The unrolled version of the microbenchmark shown in Figure 6, which uses DCA, executes 3.2 times faster than the baseline version. The Impulse version increases the L1 cache hit rate from 65.68% to 93.37% and reduces the number of accesses that are handled by the main memory from 26.56% to 0.42%. One nice side effect of dynamic cache line assembly is the improved TLB performance. The base version of this benchmark has very bad TLB behavior because the TLB is not big enough to hold the translations for the entire array `A[]`. After using dynamic cache line assembly, the TLB needs at most two entries to hold the translations for the address and data regions. Table 1 shows that the TLB hit rate has indeed been greatly improved (from 73.67% to 99.97%).

Average memory latency increases from 51 cycles to 113 cycles, because dynamic cache line assembly requires more work than a simple dense cache line fill, but the improved cache performance overwhelms the effect of the increased memory latency. The memory latency reported here is the average latency for all load accesses, excluding prefetch accesses. The average memory latency of prefetch accesses reaches around 600 cycles due to high MTLB miss rate (82.97%). The MTLB is configured to be four-way set associative with 256 entries. The simulated system uses four-kilobyte base page, so the MTLB’s maximum reach is only one megabyte, much less than the `A[]`, which is four megabytes. In the real hardware we are building, the MTLB has 1024 entries. We reduced the MTLB size to 256 entries in our simulations to generate high MTLB miss rates, while leaving the MTLB larger than the CPU TLB (an important feature of Impulse [11]). The good performance of `/dynamiciv/` even with such high MTLB miss rates gives us confidence that our results will hold, and perhaps even improve, on larger data structures. Despite the high latency of dynamic cache line assembly, the use of prefetching results in an average latency of demand requests of 113 cycles. In this microbenchmark, not enough work is done on each piece of data for prefetching to completely hide the load latency, so we still see a high aver-

age memory latency. If more work were performed per data item, the memory latency perceived by the processor would drop.

FFT Fast Fourier Transform(FFT) is generally characterized by poor temporal and spatial locality. FFTW [6] is a specific implementation of FFT whose self-optimizing approach lets it outperform most other FFT implementations. We chose this FFT implementation as our baseline and modified it to use dynamic cache line assembly.

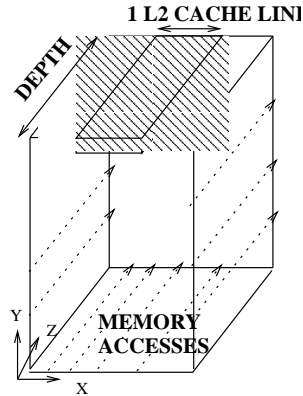


Fig. 7. Shows memory access pattern of depth phase and a *cache-column*

In general, 3D FFTW operates in two phases. The 3D input array is accessed along x and y axes in the first phase of the computation. In the second phase, which we call the *Depth Phase*, data is accessed along the z axis. For large arrays, row-major array layout causes poor locality when the array is accessed along the y and z dimensions. The memory performance accesses along the y dimension is usually acceptable because (1) the preceding the x dimension access load much of the necessary data into the cache and (2) the amount of data accessed per *plane* is usually smaller than the cache. As a result, most y accesses hits in the cache. However, most z accesses during the depth phase suffer cache misses, which accounts for 40-70% of total execution time. Accesses along the y -dimension and z -dimension load into the cache columns of data whose length is the length of the array dimension being traversed (y or z) and whose width is one cache line. We call each such block of data a *cache column*, one of which is highlighted in Figure 7. Each *cache column*, once loaded, is reused for as many column accesses as possible.

For FFTW, array elements are 16 bytes (a pair of double precision floating point numbers). Eight elements can fit into each 128-byte L2 cache line, so a single cache column will be able to service as many as seven adjacent column accesses before a cache miss will occur. However, if a cache column is larger than

Input	Type	Elapsed Cycles	TLB hit rate	L1 hit rate	L2 hit rate	Miss rate	Speedup	
							Depth	Overall
567x61x51	Base	3.1B	98.92%	92.66%	4.97%	2.37%		
	Impulse	2.2B	99.99%	93.50%	5.50%	1.00%	2.64	1.40
576x57x31	Base	1.8B	99.06%	93.03%	4.90%	2.07%		
	Impulse	1.2B	99.99%	94.55%	4.42%	1.03%	2.74	1.47
576x7x11	Base	36.5M	99.46%	92.53%	4.70%	2.77%		
	Impulse	18.6M	99.98%	93.92%	5.48%	0.60%	3.38	2.29

Table 2. Performance results for FFTW benchmark

the L2 cache, which is the case for input arrays with large y or z dimensions, then almost every access during the depth phase will be a cache miss. The reason for this is that each cache line in the cache column will be evicted before it can be reused. For this benchmark, prefetching is ineffective because the amount of work performed per element is dwarfed by the time required to load a cache line from memory. Because FFTs of interest are performed on fairly large input arrays, we evaluated the performance of DCA only for such arrays where the *cache column* size exceeds cache sizes. Also, we only consider arrays with large z dimensions and thus restrict our optimization to the depth phase. This makes our results conservative, as additional performance benefits could be had by applying DCA to accesses along the y accesses in the first phase of the FFT. To reduce simulation overhead, we simulated an 8-kilobyte L1 cache and a 64-kilobyte 64K L2 cache. For these cache sizes, the height of a cache column to be at least 512. We arbitrarily chose input’s z dimension value to be 567 and 576. The x and y dimension sizes also were chosen arbitrarily.

The FFTW library consists of highly optimized code for computing parts of the transform, called *codelets* [6]. Every multidimensional FFT is translated into a series of calls to these codelets. As part of optimizing FFTW to exploit Impulse’s DCA mechanism, we modified each codelet to create dynamic indirection vectors pointing at the array elements being accessed by that codelet. As in the random access benchmark, we unrolled-and-jammed the resulting code and added prefetching instructions to overlap computation with the DCA operation. Conventional compiler-directed prefetching or data reordering techniques will not work because the addresses and strides are input parameters for the codelets, and thus only known at runtime.

Table 2 presents our results for the FFT benchmark for various input sizes. DCA improves the performance of the depth phase by 2.64 to 3.48, depending on the input size. Total application speedup ranged from 1.40 to 2.29. The reason for these performance improvements can also be seen in Table 2. The use of DCA reduces the miss rate by more than a factor of two, and TLB performance improves significantly. The TLB performance improvement is due to the fact that all DCA accesses are to a small range of addresses, rather than the entire range of the input array.

6 Conclusions and Future Work

The development of the dynamic cache line assembly mechanism is still in its infancy. Future work includes applying it to more applications and further optimizing its performance. We believe the proposed mechanism can be effective for many applications with poor locality. To confirm this hypothesis, we are evaluating DCA's potential on a mix of pointer-intensive programs from the Olden benchmark suite, image processing programs, and irregular scientific application kernels (Moldyn and NBF).

The performance of dynamic cache line assembly can be improved in a number of ways. The current implementation loads each cache line of the address region from the memory before overwriting it with new addresses. If the processor we modeled had supported for *write, no-allocate*, such as is possible via the Alpha 21264 WH64 instruction [4], we could eliminate this unnecessary cache miss. Along the same lines, without support from the ISA, we must synchronize flushes of the address region and the subsequent accesses of the data region using memory barriers. A memory barrier serializes accesses before and after it and can impede the processor access streams. One way to eliminate this effect would be to extend the ISA with a special "indirection gather" instruction. The instruction would combine a flush back of the address region with a prefetch of the corresponding data region.

In conclusion, we believe that as the performance gap between processors and DRAM grows, a more flexible memory interface will be necessary to hide memory latency. Simply building larger on-chip caches will not suffice, and will increase cache access latency. We have begun investigating the potential benefits of allowing applications to selectively read/write data from/to random locations in memory efficiently when conventional caching does not suffice. Our initial experimental results have shown that for applications with poor spatial locality, such a mechanism can improve performance by a factor of three or more.

References

1. S. Carr, K. McKinley, and C.-W. Tseng. Compiler optimizations for improving data locality. In *Proceedings of the 6th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 252–262, Oct. 1994.
2. J. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, E. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama. Impulse: Building a smarter memory controller. In *Proceedings of the Fifth Annual Symposium on High Performance Computer Architecture*, pages 70–79, Jan. 1999.
3. M. Cierniak and W. Li. Unifying data and control transformations for distributed shared memory machines. Technical Report TR-542, University of Rochester, November 1994.
4. Compaq Computer Corporation. *Alpha 21264 Microprocessor Hardware Reference Manual*, July 1999.
5. C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Proceedings of the*

- 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 229–241, May 1999.
6. M. Frigo and S. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proceedings of ICASSP Conference*, 1998.
 7. J. W. Manke and J. Wu. *Data-Intensive System Benchmark Suite Analysis and Specification*. Atlantic Aerospace Electronics Corp., June 1999.
 8. MIPS Technologies Inc. *MIPS R10000 Microprocessor User's Manual, Version 2.0*, Dec. 1996.
 9. V. Pai, P. Ranganathan, and S. Adve. RSIM reference manual, version 1.0. *IEEE Technical Committee on Computer Architecture Newsletter*, Fall 1997.
 10. S. Scott. Synchronization and communication in the T3E multiprocessor. In *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems*, Oct. 1996.
 11. 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.
 12. Y. Yamada. *Data Relocation and Prefetching in Programs with Large Data Sets*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana, IL, 1995.
 13. L. Zhang. URSIM reference manual. Technical Report UUCS-00-015, University of Utah, August 2000.
 14. L. Zhang, J. Carter, W. Hsieh, and S. McKee. Memory system support for image processing. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, pages 98–107, Oct. 1999.