Reducing Cache Misses in Numerical Applications Using Data Relocation and Prefetching

Yoji Yamada[*] Teresa L. Johnson[†] Grant E. Haab[‡] John C. Gyllenhaal[§]
Wen-mei W. Hwu Josep Torrellas

Coordinated Science Laboratory
University of Illinois
Urbana, IL 61801

Abstract

Numerical applications frequently contain nested loops that process large arrays of data. The execution of these loop structures often produces memory reference patterns that utilize data caches poorly. Indeed, poor reuse of the data, large working set sizes, and frequent non-unit stride accesses all combine to cause many cache misses. To improve cache performance, data copying has been proposed. However, this technique has high overhead.

In this paper, instead, we propose a combined hardware and software technique called data relocation and prefetching which eliminates much of the overhead of data copying through the use of special hardware. Furthermore, by relocating the data while performing software prefetching, the overhead of copying the data can be reduced further. This technique performs better than prefetching alone because it reduces cache misses through relocation, and it reduces overhead by prefetching multiple elements at once. The hardware is designed to overlap relocation and prefetching with normal execution, and to highly utilize the available bus bandwidth. Simulation results show that this technique greatly reduces data cache miss rates. As a result, large applications including PERFECT and SPEC benchmarks achieve up to 2.5 times speedup. The hardware support required by this technique has been greatly refined over that presented in an earlier paper.

Index terms - Cache conflicts, data copying, data relocation, program optimization, software prefetching.

1 Introduction

Numerical applications frequently contain nested loops that process large arrays . The execution of these loop structures has been shown to produce memory reference patterns that under-utilize data caches [1][2]. This is caused by at least three problems. First, large working set sizes cause

[*]Supported by Mazda Motor Corporation.
[†]Supported by a National Science Foundation Graduate Fellowship.
[‡]Supported by a Fannie and John Hertz Foundation Graduate Fellowship.
[§]Supported by a National Science Foundation Graduate Fellowship.

cache overflow, resulting in cache misses. Secondly, non-unit stride access patterns can cause low utilization of cache lines, which increases cache conflicts, in addition to wasted bus and memory cycles [3]. Finally, low reuse of data results in poor cache use, and therefore more cache misses.

Potentially, one could use a larger cache size and higher cache associativity to reduce some of these effects. This brute force approach, however, does not scale well with larger problem sizes. Moreover, it would result in significant hardware cost and increased cache access latency, both of which could be avoided via the more cost-effective approach proposed in this paper.

The use of loop blocking transformations is often effective in improving performance of caches [2] [4] [5]. By partitioning the iteration space, loop blocking transformations reduce the amount of data referenced between two references to the same datum, thereby increasing the potential for data reuse. In practice, however, it has been shown [4] that loop blocking transformations suffer from cache mapping conflicts. Additionally, blocking alone is not effective for singly-nested loops since the data accesses are not reordered. Data copying has been proposed to reduce the cache conflict misses [1] [4] [6], however the overhead is significant.

Data prefetching has also been proposed to reduce cache misses by fetching data into the cache before it is referenced [7] [8]. When used in conjunction with small cache-block sizes, one can potentially eliminate the problem of low utilization of cache blocks and wasted bus cycles [3]. However, data prefetching may increase the size of the working set, introducing capacity misses. Also, prefetched data may conflict with the current working set in the cache, introducing more conflict misses [9] [10]. In order for data prefetching to improve performance in a reliable manner, one must ensure that both current and future working sets can fit into the cache.

We propose an approach, called Data Relocation and Prefetching (*DRP*), that prefetches the array element read references into consecutive, and therefore non-conflicting, locations within the cache. This is implemented with combined hardware and compiler support which has less overhead than the traditional data copying approach, because the relocation is integrated with prefetching. Also, compression and prefetching of the next working set is overlapped with the computation for the current working set in order to hide the latency of the relocation. With this technique, if the original data access pattern is of non-unit stride, unused data are not brought into the cache during compression and prefetch, resulting in improved cache-line utilization.

The initial version of this technique was proposed in an earlier paper [11], which focused on compiler support and gave a rough description of the hardware support. Preliminary results were also presented. This paper presents a more detailed description of the architectural and hardware support, which has been redesigned for improved efficiency. Improvements include greatly reduced instruction set modifications required, a much wider range of hardware issues that are discussed, and even more encouraging experimental results.

Overlap of program execution with *DRP*, high utilization of the memory bus bandwidth, and support of out-of-order return of requests from the memory system are important objectives the hardware should satisfy. Four of the main components of the hardware unit were designed to satisfy these objectives: the Precollect Status Store, Instruction Queue, Outstanding Memory Request Store and Block Assembly Cache. The Precollect Status Store, which provides synchronization between the *DRP* unit and the *CPU*, and the Instruction Queue, which allows the *DRP* unit to operate asynchronously with the *CPU* by buffering *DRP* requests, combine to help fully overlap program execution and *DRP* instruction execution. Additionally, the Outstanding Memory Request Store, which allows multiple outstanding memory requests, provides high utilization of the memory bus, and the Block Assembly Cache handles out-of-order return of requests from the memory system. The operation of each component will be discussed in detail in Section 2.4.

Using the IMPACT compiler [12], an emulation tool, and a simulation tool, we show that this extension to the cache and processor architecture along with the requisite compiler support greatly improves the data cache performance for array-based applications. Cache miss rates are reduced for all the applications tested, and are nearly eliminated for several applications. Up to 2.5 times speedup is achieved, and the average speedup is over 1.5.

1.1 Related Work

The technique proposed here is conceptually similar to the *gather* operation used in the Cray-1 [13]. In the Cray-1, the array elements are "gathered" from memory into the vector registers before performing vector operations, and "scattered" back to memory after the vector operations are complete. However, the hardware necessary to support data relocation and prefetching would be much easier to add to an existing processor than the hardware to support vectorization.

Chen and Baer [7] presented a hardware approach which preloads blocks for accesses with constant strides. However, their method does not attempt to reduce conflict misses because it does not change the layout of the prefetched data within the cache. On the other hand, our technique relocates only data elements which will be referenced, and maps them into sequential locations within the cache.

Several techniques have been proposed to help reduce cache conflict misses, and can be used in conjunction with *DRP*. The victim cache [14], the column-associative cache [15], and the assist cache [16] can be used with *DRP* to reduce conflict misses for untransformed accesses. These strategies avoid some of the cost of an associative cache by providing a fast access for cache hits and a slightly slower path for references that conflict in a direct-mapped cache. Additionally, the Cache Miss Lookaside buffer [17] can be used to reduce conflict misses in a large second-level cache, while *DRP* is used for first-level caches.

The remainder of this paper is organized into four sections. Section 2 presents the proposed technique and describes the necessary architecture, compiler, and hardware support. In Section 3, the simulation environment is detailed, and in Section 4 simulation-based experimental results are provided to demonstrate the effectiveness of the proposed technique. Finally, Section 5 offers concluding remarks and future directions.

2 Data Relocation and Prefetching

2.1 Overview

We propose a hardware-based, compiler-supported technique called data relocation and prefetching in order to improve the data cache performance . In this method, the array elements read in the inner loop of a nest are sequentially mapped into special relocation buffers within the cache before they are accessed. Special hardware that is attached to the cache unit performs this relocation of data into sequential locations while prefetching the data from the memory to the cache, without stalling the *CPU*. The relocation operations are invoked by an explicit *precollect* instruction inserted by the compiler. The compiler also inserts a declaration into the original code to allocate a relocation buffer in memory for the relocated data.

Because the array data is relocated, the prefetch is binding. During the computation, the newly assigned address in the relocation buffer is used to access the data rather than the original address. Consequently, the relocation must be completed before the computation on the same data begins.

a) Original Code

```
for (i = 0; i < N; i++)
   for (j = 0; j < 3; j++)
      ... = A[i][2*j] + B[j][i];
```

b)Transformed Code

```
for (i = 0; i < N; i++)
   for (j = 0; j < 3; j++)
      ... = A'[j] + B'[j];
```

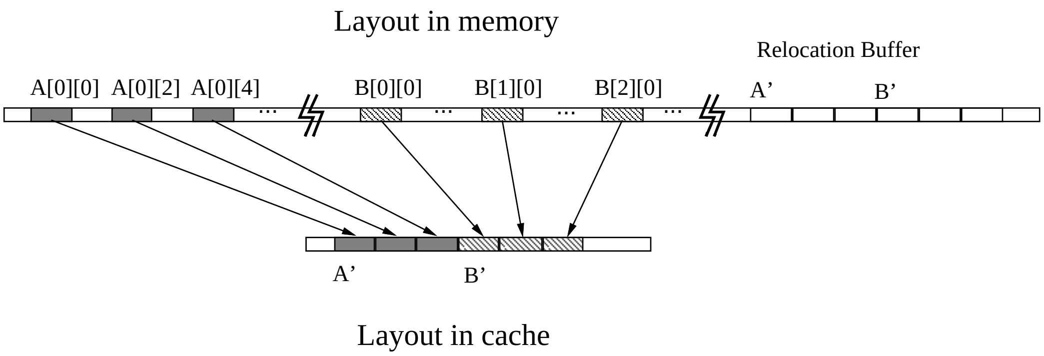

Figure 1: Concept of Data Relocation

So if the relocated, cached data is replaced by some other data, the relocated data is automatically written back to the relocation buffer in memory since the accesses in the computation use the address of the relocated data. To insure that this write-back occurs, the dirty bit is set when the cache line for the data is allocated. Modifications to this algorithm necessary for the use of a write-through cache, instead of a write-back cache, will be discussed in Section 2.4. Array references that are written in the loop are never transformed, as will be discussed in Section 2.2.2, so the relocated data will never need to be written back to the original address.

DRP can improve the spatial locality of array accesses for a loop nest. Figure 1 shows how array data elements accessed in the first iteration of the outer loop are copied to sequential cache locations that map to the relocation buffer in memory. Array **A** is accessed with a stride of two, and array **B** is accessed in column order during the execution of the inner loop. Accesses to these array elements can result in poor performance because:

1. The accesses may have low spatial locality because of the non-unit access stride, resulting in wasted cache capacity which may lower the cache hit rate.

2. The sets of accesses for different arrays may conflict with each other because they may be mapped to some of the same locations in the cache.

3. The accesses for a single array may conflict with each other because of a large access stride.

If the accessed elements of these arrays are relocated into the cache, spatial locality can be improved by packing elements of the arrays into contiguous locations. This increased spatial locality can eliminate conflicts between elements of the same array, as well as between those of different arrays. Also, since only necessary elements are brought into the cache, the extra memory requests and time to fill the cache line due to the non-unit stride accesses are reduced. Furthermore, if the total size of the relocated array elements is smaller than the cache size, the compression guarantees that the references to the relocated data do not conflict with each other in the cache. Finally, cache space is conserved by packing elements of the arrays.

In order to reduce the instruction-fetch overhead due to the inserted *precollect* instructions, each *precollect* instruction contains enough information to operate on several elements of the array in sequence. Also, in order to accommodate the latency of the relocation of array data, the relocation and computation phases are separated in time by software-pipelining the outer loop.

2.2 Architectural Support

Implementing the mechanism for the *DRP* technique requires an extra instruction as well as extra hardware. In the following, we first describe the new instruction, and then discuss instructions eliminated from the original design [11].

2.2.1 Precollect Instruction

To support *DRP*, a new *precollect* instruction is added to the instruction set of the processor. This instruction compacts the array data referenced in a computation into consecutive memory locations in the cache before the data is needed for the computation. These locations together are called a *relocation buffer*. The *precollect* instruction also sets the dirty bit of the cache line so that if the relocated data is displaced from the cache, it is backed up in a relocation buffer in memory. The *precollect* instruction accesses the cache to find the source data first. If it misses then a request is sent to the memory system for the data. In either case, the processor does not stall from cache accesses caused by a *precollect* instruction.

The *precollect* instruction has five operands. The first operand is the address of the first element of the array to be relocated. The second operand is the address of the first element of the relocated array in the relocation buffer. Finally, the third through the fifth operands are the size of each array element in bytes, the stride of the array accesses in bytes, and the number of array elements to be collected, respectively.

Since most RISC architectures do not allow five operands, the *precollect* instruction is actually implemented with two machine instructions. The first machine instruction is an immediate load, which encodes the element size, stride and number of data elements in a register. Then the new machine instruction called *precollect* has three register operands: the array address, the buffer address, and the encoded register. If the instruction set architecture only supports two source register operands, then the buffer address may be specified as an immediate value, since it requires the fewest bits of the three operands. Because the data element size, stride, and number of data elements are likely to be loop-invariant, the immediate load is usually moved out of the outer loop by our compiler. Therefore, the overhead of this instruction is minimal.

2.2.2 Instructions Eliminated from Previous Work

The previous version of this technique that we proposed [11] required five extra instructions, not only *precollect*, but also *await*, *preallocate*, *distribute* and *finishup*. The *await* was used as a synchronization mechanism to avoid accessing the relocated array data before the *precollect* was completed. We were able to remove this extra instruction by adding hardware to enforce this constraint, as will be explained in Section 2.4.

For data that is written only, a write-no-allocate cache is used so that these references are simply sent to the write buffer if cache misses occur, thereby avoiding cache conflicts. The *preallocate*, *distribute* and *finishup* instructions were used to support relocation and prefetching of data that is written and later read within the inner loop. Experimental evidence suggests that this support is unwarranted, for only two loop nests in all of the applications that we tested contained references of this nature that were being transformed. When we did not transform these references, less than a 1% degradation in performance occurred. The fact that few loads follow stores to the same array within the same inner-most loop in our applications, coupled with the use of a write-no-allocate cache as described above, results in this small performance degradation. Therefore, we have eliminated the *preallocate*, *distribute* and *finishup* instructions.

2.3 Compiler Support

We apply the data relocation and prefetching optimization to loops nested two or more deep. For loop nests which are nested more deeply than two, the transformation is applied to the two inner-most loops. In this case, we refer to the outer of the two inner-most loops as the outer loop, and the inner-most loop as the inner loop. Singly nested loops are transformed into doubly-nested loops via loop strip-mining, discussed in Section A.1, so that relocation and prefetching can be applied to them.

Before each execution of the inner loop, the array data accessed in the inner loop are relocated and prefetched. In order to allow sufficient time for *precollect* instructions to complete, we overlap the execution of one outer loop iteration with the precollection of data used in the next outer loop iteration. Therefore, two relocation buffers are required for each array we precollect (prefetch and relocate), one for the executing iteration and one for the next iteration which is precollecting.

The code transformations employed for data relocation and prefetching are explained in detail in [18]. The compiler algorithms have been improved over those in previous work [11] so that the *DRP* technique can be applied more broadly and effectively. The most important of the compiler transformations are explained in Appendix A, and an example is given.

2.4 Hardware Support

The hardware support required for *DRP* is an on-chip module called the *Data Relocation and Prefetching unit*. The *DRP* unit receives *precollect* instructions from the processor. It fetches data from the cache or main memory, one element at a time, and then assembles each cache block in an assembly buffer and writes it into the cache. The *DRP* unit needs to access both the bus and the data cache. Therefore, the *DRP* unit shares the cache with the *CPU* and shares the bus with the cache (Figure 2). In this configuration, both the processor and the cache have higher priority than the *DRP* unit to access shared resources. This priority hierarchy helps to ensure that the *DRP* unit does not interfere with the processor accesses to the memory system while a *precollect* instruction is being executed. We verified via simulation that the *DRP* unit should have lower priority in accessing the bus [18].

In our current implementation, we use a virtually-addressed write-back data cache. However, a physically-addressed cache can also be used, as discussed in Section 2.4.6. If a write-through cache

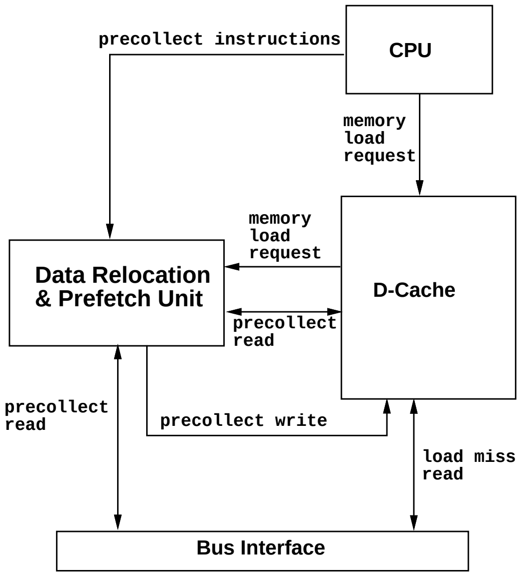

Figure 2: Data Relocation and Prefetch Unit Interfaces

is used instead, when the precollected data is written to the data cache, it must also be placed in the write buffer. This is to ensure that the relocated data is also initially placed in the relocation buffer in memory, because there is no provision to ensure that the relocated data will be written back to memory when it is replaced from a write-through cache. A dedicated read port into the data cache is required by the *DRP* unit. It is used to check if data to be precollected resides in the cache. This read port should not trigger a cache miss if the data is not found in the cache: If the data is not found in the cache, the *DRP* unit accesses memory using its connection to the memory bus. Furthermore, the *DRP* unit itself does not block when fetching data from a memory location. If a high-bandwidth memory system like a split-transaction bus system is used, many read and write requests issued by the *DRP* unit can be in progress at the same time.

The structure of the *DRP* unit is shown in Figure 3. It has five main parts: Precollect Status Store, Instruction Queue, Address Generator, Outstanding Memory Request Store, and *DRP* Block Assembly Cache. As discussed in Section 1, the Precollect Status Store and Instruction Queue satisfy the objective of overlapping program execution with *DRP*, the Outstanding Memory Request Store helps fully utilize the memory bus bandwidth, and Block Assembly Cache supports out-of-order return of data from memory. In the following, we consider each component in turn.

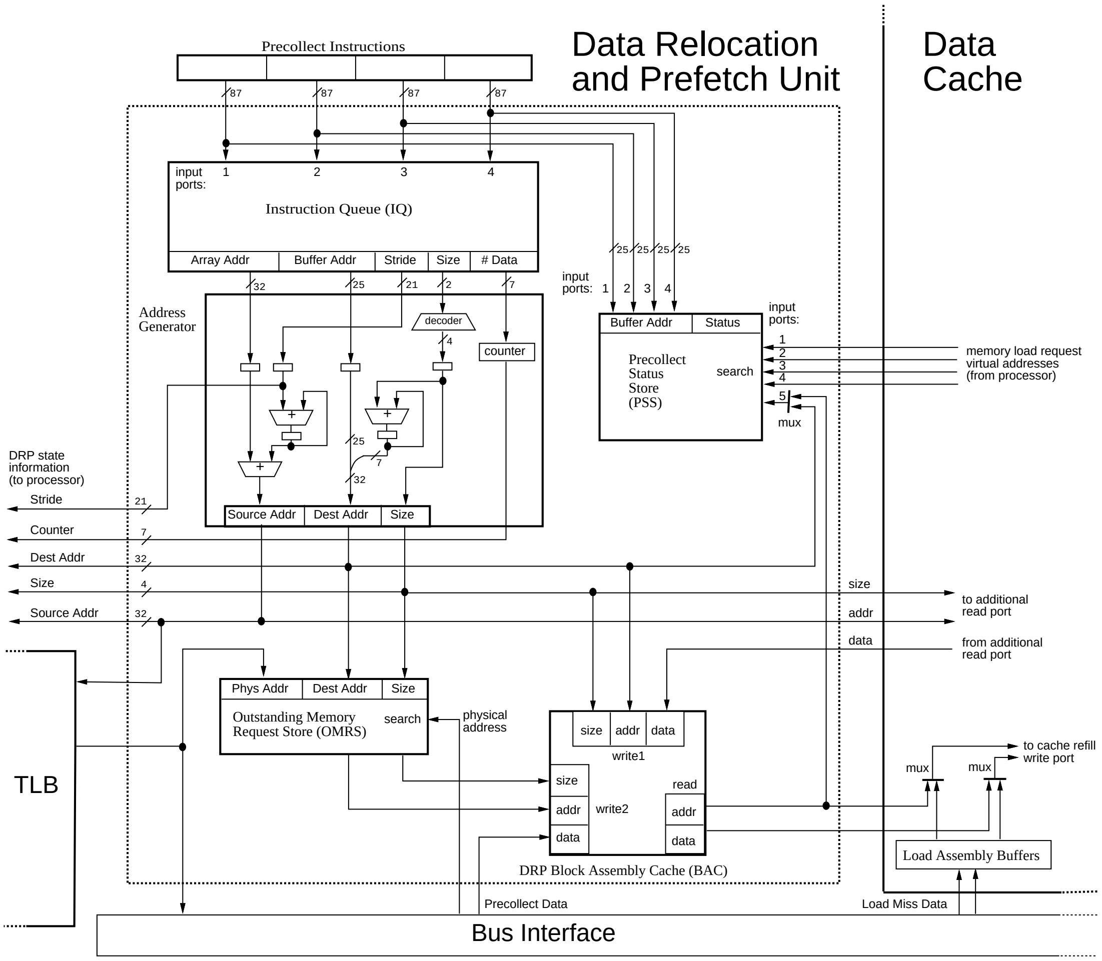

Figure 3: Data Relocation and Prefetch Unit Data Path

2.4.1 Precollect Status Store

When a *precollect* instruction is executed, we allocate an entry in the Precollect Status Store (PSS). Each PSS entry has a bit for each cache block of the relocation buffer into which we are precollecting. All the bits are initialized to zero when the entry is first allocated. As each cache block of relocated data is finally copied into the data cache, the corresponding bit in the PSS is set. Once all bits for an entry are set, the *precollect* instruction is complete, and we remove the entry from the PSS. Note that more than one precollect may be in progress if the memory can return data out-of-order.

The cache controller must check the PSS each time the cache is accessed by the processor to ensure that the block it is attempting to access has been precollected. If the corresponding bit is zero then the cache is blocked until the bit for that block is set. If an entry is not found in the PSS for the block being accessed, the cache can proceed because either the corresponding *precollect* instruction is completed, or the data being accessed was not transformed for relocation. As discussed in Section 1, the PSS contributes to the desired overlap of program execution with *DRP* by providing a mechanism to synchronize the *CPU* and the *DRP* unit when necessary.

Entries in the PSS must be allocated immediately upon the issue of a *precollect* instruction so that an access to the precollected data issued the immediately following cycle will correctly cause the cache to block. Therefore, the number of write ports into the PSS should equal the number of precollects allowed to issue in the same cycle.

The PSS is fully-associative and is accessed from one of three sources: the data cache, the Block Assembly Cache (BAC) (discussed in Section 2.4.5), and the Address Generator (discussed in Section 2.4.3). There should be a dedicated port into the PSS from the data cache for each of the memory load requests that the processor can issue simultaneously. The other two requests are multiplexed into one port, as shown in Figure 3. All of these ports perform an associative search on the buffer address. Priority to access the shared port is given to the Address Generator, since stalls may impact performance significantly. The algorithm flow chart for the PSS access port is illustrated in Figure 4.

2.4.2 Instruction Queue

Once the *precollect* instruction is decoded, the operands are fetched from the register file and placed in the Instruction Queue (IQ). As with the PSS, the IQ must have as many write ports as the number of precollects allowed in the same cycle. In addition, to avoid stalling the processor, both the PSS and the IQ need enough entries to hold the maximum number of "in-flight" precollects. If either the IQ or the PSS is full when a new precollect is sent from the processor, the *DRP* unit stalls the processor until there is an empty entry. By buffering *precollect* instructions as they are received from the *CPU*, the IQ allows asynchronous operation of the *CPU* and *DRP* unit.

An entry is dequeued from the IQ as soon as it is latched into the Address Generator, discussed in Section 2.4.3.

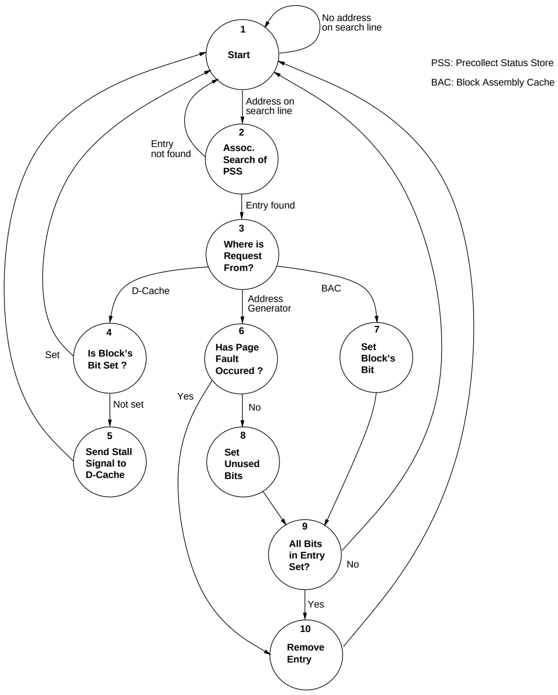

Figure 4: Algorithm Flow Chart for the Precollect Status Store Access Port

2.4.3 Address Generator

The Address Generator simultaneously calculates, for each array element, its original address and its destination address. To do this it uses the starting array address, stride, starting relocation buffer address and element size. The compiler forces all relocation buffers to align to virtual page boundaries, and therefore, memory block boundaries. Consequently, the lower bits of the relocation buffer addresses are zero and we can simply append the computed element offset to the high-order bits of the relocation buffer address. Also, the size field is encoded, since the data element size (in bytes) can be any power of two up to the bus transfer width. Step 5 of Figure 5 corresponds to the generation of these addresses, as described above. Additionally, a counter is used to determine how many element addresses remain to be generated.

After the source and destination addresses are generated, but before attempting to access the corresponding data, an entry must exist in the BAC, described in Section 2.4.5. The BAC contains the logic to align the requested element within the received block of data into the corresponding relocation buffer cache block. If the element to be precollected will reside at the start of a relocation buffer cache block, an entry must be allocated in the BAC. To determine if this element is the start

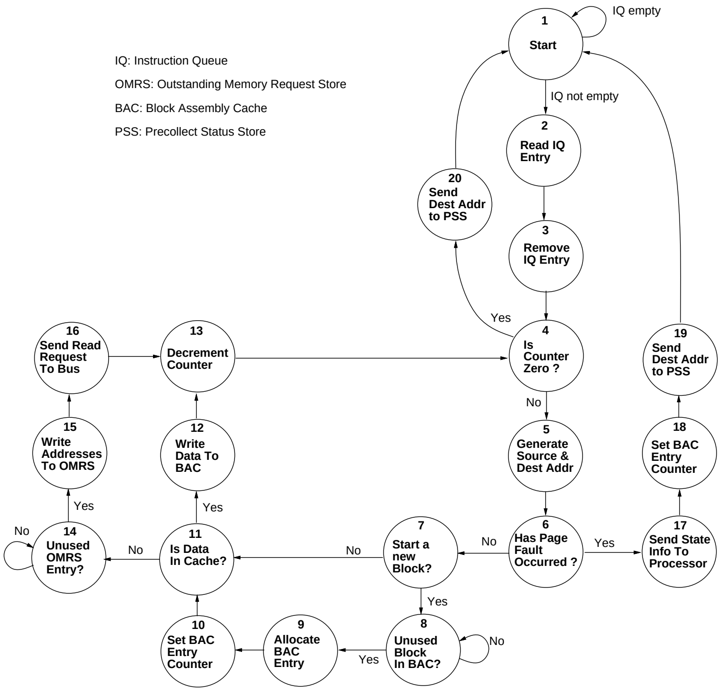

Figure 5: Algorithm Flow Chart for the Address Generator

of a new cache block, the Address Generator checks if the destination address low order bits are zero, and if so, allocates an entry in the BAC once there is space. Furthermore, a counter associated with each BAC entry is set to the number of precollected elements that will reside in that cache block, which is determined using the element size, the block size and the Address Generator counter value. Steps 7 through 10 of Figure 5 correspond to these actions.

Next, a read request is sent to the cache. If data is present in the cache, it is written to the BAC, using the destination address and size. These actions are shown in Steps 11 and 12 of Figure 5.

If the data is not present in the cache, the cache does not send a read request to fetch the data. Instead, the *DRP* unit sends the read request to memory using the source address. Step 16 of Figure 5 corresponds to this action. Before sending the request, the address is translated by the *TLB*. Because the *DRP* unit needs to access the *TLB* for every element, a dedicated port into the *TLB* is required. The destination address and size are sent to the Outstanding Memory Request Store (OMRS), discussed in Section 2.4.4, for later use. Similarly, the physical address is sent from the *TLB* to the OMRS. Steps 14 and 15 of Figure 5 correspond to these actions.

Because we may not be precollecting into the maximum number of blocks allowed per relocation

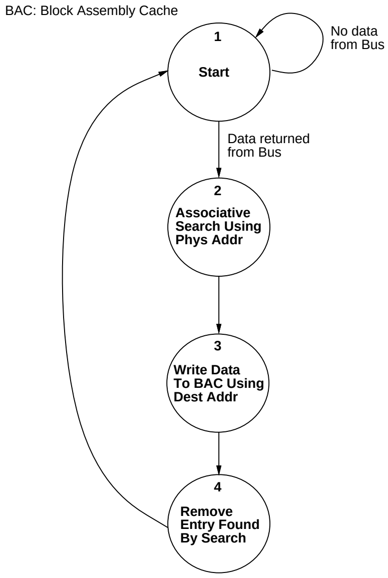

Figure 6: Algorithm Flow Chart of the Outstanding Memory Request Store Logic

buffer, some status bits in the PSS may be unused and will never be set by the data returning from the *precollect*. Since an entry is removed from the PSS only when all of its status bits are set, these unused bits must somehow be set. Therefore, when the Address Generator counter reaches zero all of the addresses have been generated, so the bits corresponding to unused blocks in the relocation buffer are set. Steps 4 and 20 of Figure 5 correspond to these actions.

Each *precollect* instruction is processed completely by the Address Generator before the next may proceed. When the Address Generator's counter has reached zero, requests have been generated for every element, and the next *precollect* instruction may be latched from the IQ.

2.4.4 Outstanding Memory Request Store

As mentioned in Section 2.4.3, when a request is sent to the memory system by the Address Generator, the physical address, destination address and size are saved in the Outstanding Memory Request Store. The algorithm flow chart of the OMRS's logic is shown in Figure 6. Each entry in the OMRS corresponds to one outstanding request.

If the OMRS has enough entries, the store does not block while it waits for a memory access to complete. In order to minimize the probability that the OMRS blocks, the store should have as many entries as the maximum memory latency divided by the Address Generator cycle time. By keeping track of multiple outstanding requests, the OMRS exploits available bus bandwidth, one of the objectives discussed in Section 1.

When the data returns from memory, the physical address fields of the OMRS are searched associatively using the physical address from memory to find the corresponding entry. The requested data is then written to the BAC using the destination address and size from the OMRS. Additionally, that entry is removed from the OMRS. These last three actions are represented by Steps 2, 3 and 4, respectively, of Figure 6.

2.4.5 DRP Block Assembly Cache

In order to handle out-of-order return of data from the memory system, we use a small fully-associative cache in the *DRP* unit, called the *DRP* Block Assembly Cache, to align and assemble the precollected elements into cache blocks. Each entry of the BAC holds a cache block that is a portion of a relocation buffer. The assembly of each destination cache block is performed in the BAC instead of in the data cache, to minimize the interference with regular cache operation. The BAC also eliminates the need for a dedicated write port into the cache, because the cache refill write port used by memory to write load miss data can be shared with the *DRP* unit. After an entire block is assembled, it is written into the cache. Then, the corresponding bit in the PSS is set and the block is removed from the BAC. For each block in the BAC, a counter keeps track of how many elements have yet to be received. This counter is initially set by the Address Generator, as described in Section 2.4.3. Consequently, the BAC knows when the block is fully assembled.

Because the BAC is small, there may not be enough entries and the OMRS may have to stall. To avoid stalls, the BAC should be fully-associative and have enough entries to hold the maximum number of blocks "in-flight". The number of blocks required depends on the memory latency, bus bandwidth and the average element size. For our configuration, which has a 10-cycle memory latency, five entries are enough. Note that the *DRP* unit can be easily interfaced with a banked memory system since the BAC handles out-of-order data return.

2.4.6 Virtual Versus Physical Cache Considerations

Our implementation thus far uses a virtual data cache. Therefore, we attach the process identifier to the data cache tags so that the data cache need not be drained when a context switch occurs.

For the same reason, entries in the IQ, PSS, OMRS and BAC also need to have process identifiers attached. Otherwise, we must drain the *DRP* unit completely before performing a context switch. This approach, of course, would result in prolonged context switching latencies. Assuming that we attach process identifiers, then the actions taken in response to a context switch depend on the cause of the context switch. If the context switch was caused by anything other than a page fault, no actions need be taken, and the *DRP* unit can continue normal execution.

If a context switch is due to a page fault triggered by a *precollect* instruction, then a small part of the *DRP* state must be saved to the processor, since the *DRP* unit is in the process of executing the faulting *precollect* instruction (Step 17 of Figure 5). After the page fault has been handled and execution of the faulting process resumes, the saved state is used to restart the precollect instruction at a point where the fault occurred. *Precollect* instructions in the IQ which are issued after the faulting *precollect* instruction may begin execution immediately after the context switch. In order to simplify the implementation, immediately before the faulting process resumes execution, the operating system emulates the execution of the faulting *precollect* instruction using ordinary processor load and store instructions.

At the point of page fault detection for a *precollect* instruction, we save the precollect's state information from the Address Generator. As shown in Figure 3, the state information saved consists of the source address that faulted, the number of remaining elements (from the counter), the destination address, stride, and size. To avoid having a single partially filled block remain in the BAC, the destination address of the element that caused the fault, the element size and the block size are used to set the counter of that BAC entry to the number of elements precollected before the page fault (Step 18 of Figure 5). Therefore, after the OMRS drains, this partially completed block of relocated data will be written into the data cache. Finally, the entry in the PSS corresponding to the faulting *precollect* instruction is removed using the destination address from the Address Generator (Step 19 of Figure 5, Steps 6 and 10 of Figure 4).

If the data cache is physically addressed, we assume that address translation occurs in parallel with access of the cache tag store using the virtual address. The BAC is also physically addressed using the same assumptions. In Figure 3, the read and write ports for the data cache and the write ports for the BAC use virtual addresses since the address translation is initiated by the caches. The *DRP* unit should still use virtual addresses, since the PSS, IQ, Address Generator and OMRS use the virtual addresses provided by the *precollect* instruction.

For a physically addressed cache, on any type of a context switch, the same process described above for a virtually addressed cache must be performed, but the entire IQ and PSS must also be saved as part of the state. However, in this case, the *DRP* state is restored in hardware when the faulting process resumes execution, rather than performing software emulation, since software emulation will not simplify hardware.

2.4.7 Relocation Buffer and *DRP* Component Size Considerations

Another issue that needs to be considered is the relocation buffer size. The buffer space should be the same size or smaller than the data cache so that there are no cache conflicts between relocation buffers. Within this area, we will allocate several relocation buffers corresponding to different arrays in the loop nest. The relocation buffer size imposes a maximum on the number of arrays per loop nest which may be transformed. For our experiments, we used an 8K-byte cache and 128-byte relocation buffers, which allowed us 64 relocation buffers per loop nest. However, as explained in Section 2.3, two relocation buffers are needed per array to accommodate software pipelining, so we can transform 32 arrays per loop nest. Although 64 entries is the maximum needed for both the IQ and PSS, software pipelining allows us to use 32 entries for each with only a small possibility of stalling.

If more than 128 bytes of array data are accessed in the inner loop, multiple precollects can be executed. The compiler aligns the corresponding relocation buffers sequentially in memory so that they appear as one large relocation buffer. Therefore, the size of the relocation buffer does not inherently limit the amount of data one can precollect for the inner loop execution.

3 Experimental Environment

In this section, the environment used for experimental evaluation of the *DRP* technique is presented . The applications for this study consist of seven numeric benchmark programs: *ARC2D*, *ADM*, *BDNA*, and *OCEAN* from *PERFECT* [19], *MATRIX300* from *SPEC'89*, and *NASA7* and *TOMCATV* from *SPEC'92*. The experimental environment includes the compiler support, emulation to verify transformation correctness, and the simulation techniques used to generate experimental

Function	Latency	Function	Latency
Int ALU	1	FP ALU	2
memory load	2	FP multiply	2
memory store	1	FP divide (single prec.)	8
branch	1 + 1 slot	FP divide (double prec.)	15

Table 1: Instruction latencies for simulation experiments.

results.

3.1 Compiler Support

In the high-level *IMPACT* compiler phases, the applications were profiled at the loop-level to obtain the number of invocations and the number of iterations for all loops in order to apply the *DRP* transformations selectively and effectively. Data dependence analysis is performed using the Omega test [20][21] to exclude inner loops with specific cross-iteration dependences which can prevent the necessary transformations.

In order to provide a realistic evaluation of the *DRP* technique, we first optimized the code using the machine-specific phases of *IMPACT* compiler. Classical optimizations were applied, then optimizations were performed which increase instruction level parallelism such as loop unrolling and superblock formation [22]. The code was scheduled, register allocated, and optimized for a four-issue, scoreboarded, superscalar processor with register renaming. Each of the four functional units are pipelined and can execute any type of instruction. The register file contains 64 integer registers and 64 double-precision floating-point registers.

3.2 Transformation Correctness Verification via Emulation

To verify the correctness of the code transformations, emulation of the generated code was performed on a Hewlett-Packard *PA-RISC 7100* workstation. The *precollect* instruction is emulated using a machine language subroutine that performs the data relocation from memory to memory instead of to and from the cache. Thus, the transformed code must relocate the data and reference it using the correct addresses for the emulation to produce valid results.

3.3 Simulation Parameters and Techniques

The emulator drives the simulator that models the processor and the *DRP* unit to determine application execution time, cache performance, bus utilization, and processor stall overhead due to *precollect* instructions. The simulation latencies used are those of a Hewlett-Packard *PA-RISC 7100* microprocessor, as given in Table 1.

The processor model includes separate instruction and data caches that are direct-mapped, 8k-byte blocking caches with a 16-byte block size. In order to increase the accuracy of the results, the small data cache size was chosen to match the application data set size, which is reduced from the data set size of the actual applications. The data cache is a multiported, write-back, no write-allocate cache that satisfies up to four load or store requests per cycle from the processor. Any single load miss blocks the processor, but up to four load misses (one per cache port) can be outstanding simultaneously, because the data fetches are pipelined on the split-transaction memory bus. An 8-entry write buffer combines write requests to the same cache block. The instruction cache and data cache share a common, split-transaction memory bus, with a 8 bytes/cycle data bandwidth. A pipelined memory model is used with a 10-cycle latency. The latency of a cache block fetch from memory is 13 cycles, one for the request, ten for the memory latency, and two for the return of two 8-byte quantums of data. However, since the 8-byte quantum containing the requested data is fetched from memory before the other 8-byte quantum in the cache block, the load miss penalty is actually 12 cycles.

A direct-mapped branch target buffer with 1024 entries is used to perform dynamic branch prediction using a 2-bit counter. Hardware speculation is supported, and the branch misprediction penalty is approximately two cycles.

The simulation model for the *DRP* unit is based on the hardware description in Section 2.4. For instance, the number of queue and buffer entries are chosen as indicated in the hardware description so that no blocking occurs in the *DRP* unit due to insufficient entries.

Since simulating the entire applications at this level of detail would be impractical, uniform sampling is used to reduce simulation time [23], however emulation is still performed between samples. The samples are 200,000 instructions in length and are spaced evenly every 20,000,000 instructions, yielding a 1% sampling ratio. Most of the applications used have more than a billion dynamic instructions, at least 50 samples, and thus, more than 10,000,000 instructions are

Benchmark	Number of Transformed Loop Nests	Percentage of Execution Time
ADM	4	3.1%
BDNA	1	65.8%
OCEAN	13	20.2%
ARC2D	36	89.2%
MATRIX300	1	97.6%
NASA7	17	70.4%
TOMCATV	4	89.6%

a) Number of *DRP*-transformed loop nests for each application, and percentage of original code execution time for all transformed loop nests per application.

Benchmark	Transformed Loop Nest	Percentage of Execution Time
ADM	LEAPFR.30	1.9%
BDNA	ACTFOR.350	65.8%
OCEAN	IN.10	9.8%
	OUT.10	5.9%
ARC2D	STEPFX.232	9.9%
	STEPFX.435J	7.0%
	XPENTA.11	5.5%
	STEPFX.212	5.4%
	STEPFY.430	5.4%
MATRIX300	SAXPY.10	97.6%
NASA7	CFFT2D2.30K	16.6%
	GMTRY.8K	11.2%
	CFFT2D1.130	9.9%
	VPENTA.11	7.9%
	VPENTA.15	5.6%
TOMCATV	MAIN.250	42.3%
	MAIN.401I	22.5%
	MAIN.501I	15.7%
	MAIN.290I	9.0%

b) Original code loop nest execution time percentages for important *DRP*-transformed loop nests.

Table 2: *DRP* transformation statistics.

simulated. For smaller applications, the time between samples is reduced to maintain at least 50 samples (10,000,000 instructions). From experience with the emulation-driven simulator, we have determined that sampling with at least 50 samples introduces typically less than 1% error in generated performance statistics.

4 Experimental Evaluation and Analysis

In order to show the full performance benefit of the *DRP* technique, experimental results are presented for all application loop nests which are transformed by the *DRP* technique . Results for the entire applications are also presented.

4.1 Individual Loop Nest Results

Performance statistics for individual loop nests are obtained by marking the *DRP*-transformed loop nests as regions for gathering simulation statistics. Table 2a) lists the total number of loop nests that were *DRP*-transformed, as well as the percentage of total original code execution time they represent, for each application. Table 2b) shows the original code execution time for each of the most important loop nests transformed by the *DRP* technique as a percentage of the total original code execution time. The execution time percentages are determined through simulation, and therefore, include all memory system and processor model effects discussed in Section 3.3. The transformed loop nests are identified by function name, the *Fortran* outer **DO**-loop number, and the loop iteration variable if necessary. The most important loop nests are those which account for over 5% of the application execution time, with the exception of ADM. ADM has no transformed loop nests which account for over 5% of the application execution time, so the only loop nest which accounts for over 1% of the execution time is shown. As shown in the table, the total execution percentage of the transformed loop nests for each application varies widely from 3.1% to 97.6%.

Figure 7 presents the measured speedup of the *DRP* technique which is calculated by dividing the original loop nest execution time by the *DRP*-transformed loop nest execution time. Measured speedup for most loop nests is relatively large, demonstrating the high performance improvement obtainable using the *DRP* technique. The smallest speedup in execution time obtained for any important loop nest is 1.21.

The speedup obtained using the *DRP* technique is affected by several factors. First, the number of cache misses decreases with the application of the *DRP* technique, which will be quantified using cache miss rates from simulation. Second, the increase in bus contention caused by extra *DRP* accesses can reduce speedup. Bus utilization data will be shown to quantify the effect of bus contention on speedup. Finally, the processor stalls due to unfinished precollect operations can cause severe degradation of execution time, which will be quantified using simulation data. The effect of stalls due to insufficient entries in the *DRP* queues and buffers is not evaluated in this paper due to space constraints.

Figure 7 shows the data cache miss rate for the original code and the *DRP*-transformed code for the loop nests. Since a no write-allocate cache is used for these experiments, the miss ratios for both original and transformed code are calculated by dividing the number of cache read misses by the number of cache read requests in the original code. This method of calculating the cache misses assures a fair comparison if the number of cache accesses for the transformed code is different from the number for the original code, which may occur due to *DRP* compiler transformations.

Note that cache misses are nearly eliminated for most of the loop nests. Since the compiler relocates nearly all array data read in the loop nest, there are few possibilities for capacity and conflict cache misses to occur. Furthermore, prefetching the array data eliminates the compulsory misses.

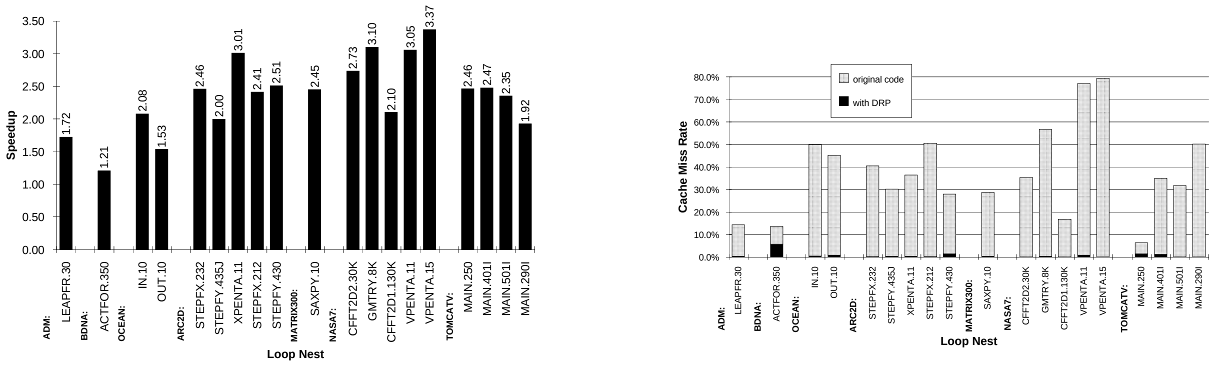

Figure 7: Speedup of the *DRP*-transformed code over the original code, and cache miss rate of the *DRP*-transformed code and the original code, for loop nests.

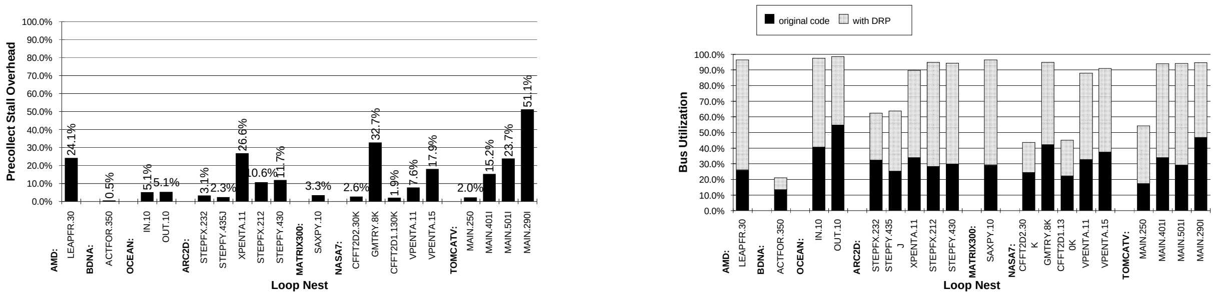

Figure 8: Bus utilization and *Precollect* stall overhead for the *DRP*-transformed code and the original code for loop nests.

Scalar variables accessed in the loop nest are not relocated and may conflict with relocated array data, causing cache misses. For instance, the *STEPFY.430* loop nest in the *ARC2D* application contains inner-loop scalar accesses which contribute to the non-zero cache miss rate. In the loop nest *ACTFOR.350* in *BDNA*, only half the array reads are relocated due to data dependences and complicated subscript expressions, which results in a relatively large percentage of remaining cache misses. Most of the remaining cache misses are due to additional memory accesses introduced by register spill code.

Figure 8 displays the percentage of execution cycles for which the processor is stalled waiting for a *precollect* to complete in the transformed loop nests, and the memory bus utilization for the transformed and original loop nests. For the loop nests with small stall overhead (less than 5%), the stalls are due almost entirely to the software-pipeline startup overhead in the first outer loop iteration. The *DRP* unit utilizes the unused bus cycles because it has lower priority than

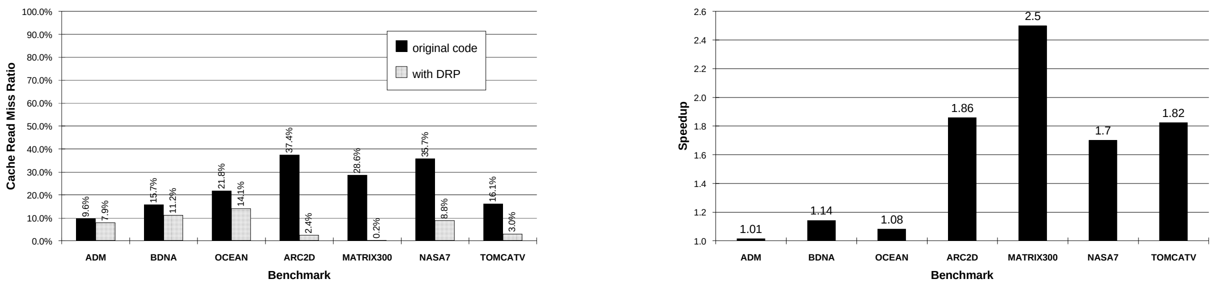

Figure 9: Data cache miss ratio for original code and *DRP*-transformed code, and speedup of the *DRP*-transformed code over the original code.

the cache when accessing the bus. However, the *DRP* transformation does not cause more data to be brought to the CPU, but instead overlaps the data fetches to take full advantage of the bus bandwidth. Consequently, bus utilization increases because the program execution is sped up. Since the application programs tend to be limited by data access rather than computation, large stall overhead indicates that that no more performance can be gained unless the bus bandwidth is increased.

4.2 Entire Benchmark Results

The data cache read miss ratios show promising improvement for several applications shown in Figure 9. However, applications *ADM* and *OCEAN* show little improvement in cache performance since only a small percentage of the application execution is transformed by the *DRP* technique. In addition, application *BDNA* contains a single transformed loop with only half the array references relocated, so conflict misses are not effectively reduced. However, the cache miss reductions are large compared to the transformed loop nest execution percentage for *ARC2D* and *NASA7* indicating that the loop nests with the worst cache behavior were indeed transformed.

Speedups for the simulated execution of the *DRP*-transformed code over the original code for the six applications are shown in Figure 9. The total speedups for *ARC2D*, *MATRIX300*, *NASA7* and *TOMCATV* are high, although the rest of the applications show much smaller gains in total performance. For *ADM* and *OCEAN* the small speedup is attributable to the fact that percentage of the execution time spent in the transformed loop nests is relatively small, (3.1% and 20.2% respectively). In the case of *BDNA*, a single loop nest dominates the entire execution time. As

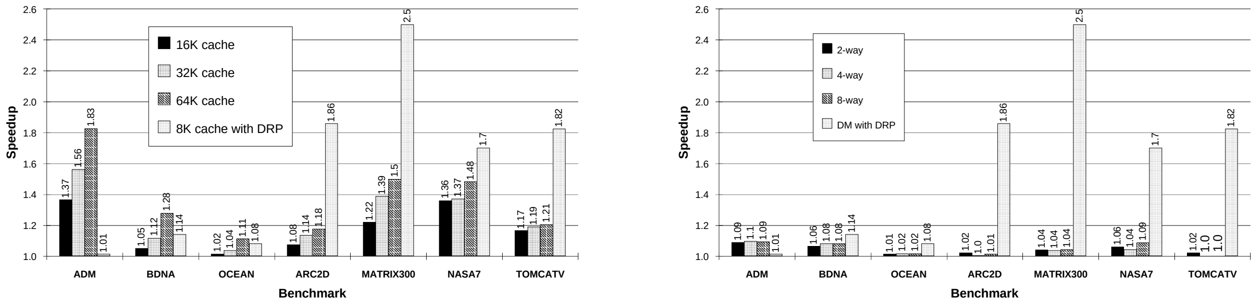

Figure 10: Speedup comparison of the *DRP* technique with 8K-byte direct-mapped caches to original code with caches of larger capacity and set-associativity.

mentioned earlier, only half of the array references in the loop nest were transformed for *DRP*, resulting in the small speedup.

In order to evaluate the effectiveness of the *DRP* technique as opposed to just increasing the cache size or associativity, we simulated the execution of the original code using various cache design parameters. Figure 10 shows the application execution speedup for the *DRP* technique using an 8K-byte direct-mapped (DM) cache plotted with the speedup for 16K, 32K, and 64K-byte DM caches as well as for 2-way, 4-way and 8-way set-associative 8K-byte caches. All speedups shown are with respect to the execution of the original code using an 8K-byte DM cache.

For *ARC2D*, *MATRIX300*, *NASA7*, and TOMCATV, the *DRP* technique with an 8K-byte cache outperforms caches with up to 64K-byte capacity by a large margin. *DRP* still outperforms up to 32K-byte caches for *BDNA* and *OCEAN*, even though the application of *DRP* is limited. For *ADM*, however, *DRP* is ineffective in reducing the execution time, so larger caches are the clear winner.

The *DRP* technique is much more effective than increasing the set associativity for all the applications but *ADM*. Since most of the cache misses are due to limited cache capacity rather than mapping conflicts, increasing the associativity is of limited benefit.

5 Conclusions

5.1 Summary

An architectural extension, referred to as data relocation and prefetching, is proposed to perform data relocation and compression during prefetching. Data relocation is employed to remove array-data mapping conflicts by compressing the accesses in a loop nest into sequential locations in the cache. Compression also improves utilization of the cache by transforming non-unit stride and array column accesses into sequential accesses that require fewer cache lines for storage. Furthermore, reduction of the cache space used to hold the data in the loop nest can increase the data reuse across transformed loop nests.

The hardware design meets the objectives of asynchronous operation of the CPU and DRP unit by allowing overlap of program execution and DRP, high utilization of memory bus bandwidth, and support for out-of-order return of data from the memory system. The Precollect Status Store and Instruction Queue components realize the first objective, the Outstanding Memory Request Queue realizes the second, and the Block Assembly Cache realizes the third.

By combining the data relocation and prefetching hardware with supporting compiler transformations, the performance of loop nests is greatly improved for a set of array-based benchmarks. Furthermore, we have shown that application of the data relocation and prefetching technique greatly improves the cache performance. Finally, the DRP technique compares very favorably to increasing the cache capacity or set-associativity in terms of application execution speedup.

5.2 Future Research

For the programs that contain many transformed arrays, some technique is necessary to reduce the bus traffic. Use of separate request and return data busses could be investigated for this purpose, as well as some technique to combine multiple bus accesses into a single access. In addition, the compiler transformations can be expanded and improved in order to transform more loop nests effectively. Further experiments are warranted to study the performance of this technique by varying implementation parameters for the DRP hardware.

References

[1] D. Gannon and W. Jalby, *The characteristics of parallel programs*, ch. The influence of memory hierarchy on algorithm organization: Programming FFTs on a vector multiprocessor. MIT press, 1987.

[2] K. Gallivan, W. Jalby, U. Meier, and A. Sameh, "The impact of hierarchical memory systems on linear algebra design," Tech. Rep. CSRD-625, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, 1987.

[3] J. W. C. Fu and J. H. Patel, "Data prefetching in multiprocessor vector cache memories," in *Proc. 18th Ann. Int'l Symp. Computer Architecture*, (Toronto, Canada), pp. 54–63, June 1991.

[4] M. S. Lam, E. E. Rothberg, and M. E. Wolf, "The cache performance and optimizations of blocked algorithms," in *Proc. Fourth Int'l Conf. on Architectural Support for Prog. Lang. and Operating Systems*, pp. 63–74, Apr. 1991.

[5] S. G. Abraham and D. E. Hudak, "Compile-time partitioning of iterative parallel loops to reduce cache coherence traffic," *J. Parallel and Distributed Computing*, vol. 2, pp. 318–328, 1991.

[6] 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*, (Los Alamitos, California), pp. 410–419, IEEE Computer Society Press, Nov. 1993.

[7] J.-L. Baer and T.-F. Chen, "An effective on-chip preloading scheme to reduce data access penalty," in *Proceeding of Supercomputing '91*, pp. 176–186, Nov. 1991.

[8] T. C. Mowry, M. S. Lam, and A. Gupta, "Design and evaluation of a compiler algorithm for prefetching," in *Proc. Fifth Int'l Conf. on Architectural Support for Prog. Lang. and Operating Systems*, pp. 62–73, Oct. 1992.

[9] W. Y. Chen, S. A. Mahlke, P. P. Chang, and W. W. Hwu, "Data access microarchitectures for superscalar processors with compiler-assisted data prefetching," in *Proc. 24th Ann. Workshop on Microprogramming and Microarchitectures*, (Albuquerque, NM.), Nov. 1991.

[10] W. Y. Chen, S. A. Mahlke, W. W. Hwu, T. Kiyohara, and P. P. Chang, "Tolerating data access latency with register preloading," in *Proceedings of the 6th International Conference on Supercomputing*, July 1992.

[11] Y. Yamada, J. C. Gyllenhaal, G. E. Haab, and W. W. Hwu, "Data relocation and prefetching for programs with large data sets," in *Proc. 27th Ann. Workshop on Microprogramming and Microarchitectures*, Nov. 1994.

[12] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu, "IMPACT: An architectural framework for multiple-instruction-issue processors," in *Proc. 18th Ann. Int'l Symp. Computer Architecture*, (Toronto, Canada), pp. 266–275, June 1991.

[13] R. M. Russell, "The Cray-1 computer system," *Communications of the ACM*, vol. 21, pp. 63–72, Jan. 1978.

[14] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *Proc. 17th Ann. Int'l Symp. Computer Architecture*, (Seattle, WA), pp. 364–373, May 1990.

[15] A. Agarwal and S. D. Pudar, "Column-associative caches: A technique for reducing the miss rate of direct mapped caches," in *Proc. 20th Ann. Int'l Symp. Computer Architecture*, pp. 179–190, May 1993.

[16] G. Kurpanek, K. Chan, J. Zheng, E. DeLano, and W. Bryg, "PA7200: A PA-RISC processor with integrated high performance MP bus interface," in *Digest of Papers. Spring COMPCON '94*, pp. 375–382, 1994.

[17] B. N. Bershad, D. Lee, T. H. Romer, and J. B. Chen, "Avoiding conflict misses dynamically in large direct-mapped caches," in *Proc. 21st Ann. Int'l Symp. Computer Architecture*, (San Jose, California), pp. 158–170, Oct. 1994.

[18] Y. Yamada, *Data Relocation and Prefetching for Programs with Large Data Sets.* PhD thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1995.

[19] M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orzag, F. Seidl, O. Johnson, G. Swanson, R. Goodrum, and J. Martin, "The PERFECT club benchmarks: Effective performance evaluation of supercomputers," Tech. Rep. CSRD-827, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, May 1989.

[20] W. Pugh, "A practical algorithm for exact array dependence analysis," *Communications of the ACM*, vol. 35, pp. 102–114, Aug. 1992.

[21] W. Pugh and D. Wonnacott, "Eliminating false data dependences using the omega test," in *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pp. 140–151, June 1992.

[22] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The superblock: An effective technique for VLIW and superscalar compilation," *Journal of Supercomputing*, vol. 7, pp. 229–248, Jan. 1992.

[23] J. W. C. Fu and J. H. Patel, "How to simulate 100 billion references cheaply," Tech. Rep. CRHC-91-30, Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, 1991.

a) Original Loop Nest

```
for (i=0; i<N; i++)
  for (j=0; j<N; j++) {
    C[i][j] = A[i][2*j] + B[j][i];
    D[i][j] = C[i][j] - B[j+1][i];
  }
```

b) After Loop Unrolling

```
for (i'=0; i'<N; i'+=2) {
  /* first outer loop body */
  i = i';
  for (j=0; j<N; j++)
    C[i][j] = A[i][2*j] + B[j][i];
    D[i][j] = C[i][j] - B[j+1][i];

  /* second outer loop body */
  i++;
  if (i < N) {
    for (j=0; j<N; j++)
      C[i][j] = A[i][2*j] + B[j][i];
      D[i][j] = C[i][j] - B[j+1][i];
  }
}
```

c) After Transformations

```
/* Bd = dimension(B[]) * 8 bytes/element */

/* prologue: for first outer loop body */
precollect(&A[0][0], A', 8, 16, N);
precollect(&B[0][0], B', 8, Bd, N+1);

for (i'=0; i'<N; i'+=2) {
  /* first outer loop body */
  i = i';
  if (i+1 < N) {
    /* for second outer loop body */
    precollect(&A[i+1][0],A'',8,16,N);
    precollect(&B[0][i+1],B'',8,Bd,N+1);
  }
  for (j=0; j<N; j++) {
    C[i][j] = A'[j] + B'[j];
    D[i][j] = C[i][j] - B'[j+1];
  }

  /* second outer loop body */
  i++;
  if (i < N) {
    if (i+1 < N) {
      /* for first outer loop body */
      precollect(&A[i+1][0],A',8,16,N);
      precollect(&B[0][i+1],B',8,Bd,N+1);
    }
    for (j=0; j<N; j++) {
      C[i][j] = A''[j] + B''[j];
      D[i][j] = C[i][j] - B''[j+1];
    }
  }
}
```

Figure 11: Transformation Example for *DRP*

A Compiler Transformations

A.1 Loop Strip-mining

For the *DRP* technique, strip-mining of the inner loop is performed in order to create a doubly nested loop if the loop nest consists of only a singly nested loop. This is most beneficial for singly nested loops that issue strided references to array elements. Strip-mining is also used in order to reduce the amount of data relocated for the inner-loop computation if that amount is too large to fit in the cache. This is necessary since all elements that are relocated and prefetched for the inner loop computation must be allocated to unique cache locations to prevent possible conflict misses.

A.2 Declaration of New Variables for *DRP*

Next, the buffer space is declared as an array that has the same size as the cache size. This guarantees that the relocated arrays in the buffer space have no cache line conflicts among them.

A.3 Loop Unrolling

In order to overlap the data relocation and prefetching for the next outer-loop iteration with the computation for the current iteration, the relocation and prefetching phase is software-pipelined with the computation phase. The original loop nest of Figure 11a) is shown after loop unrolling in Figure 11b). This software-pipelining scheme requires two relocation buffers. The inner loop is duplicated by unrolling the outer loop once. In the first outer-loop body, the data relocation proceeds into the second relocation buffer, while the computation is performed using the data already relocated in the first buffer. For the second outer-loop body, the same method is used as for the first outer-loop body except that the relocation buffers are switched.

A.4 Insertion of *Precollect* Operations

After creating a doubly-nested loop by strip-mining, *precollect* operations are inserted in the high-level code. These high-level operations are replaced by the corresponding machine instructions at the assembly code level.

Before the outer loop, *precollect* operations are inserted for the first inner loop computation. Since these operations cannot be overlapped with any other computation in the loop body, these operations constitute the start-up overhead of software pipelining. Also, *precollect* operations for each inner loop within the unrolled outer loop are inserted before the inner loop in which they are used. Each inner loop needs to use a different relocation buffer since precollecting the data for the next loop is overlapped with the computation for the current loop by software pipelining.

A.5 Replacement of Array References with Relocation Buffer References

Once the *DRP* operations have been inserted, the array references for the computation within the inner loop are modified so that the relocation buffer locations are accessed instead of the original array locations. Some array references that do not need to be relocated are left as the original array references. Replaced array references are all one-dimensional even if the original array references are multi-dimensional. If more than one reference within a loop nest is to the same array, the reference patterns are analyzed to generate the minimum number of precollects necessary. For example, references B[j+1][i] and B[j][i] in Figure 11 are replaced by references to a single relocation buffer B' (in the first outer loop body), since the two references overlap in most elements they access. We only need to precollect B[j][i] for j from 0 to N+1. Therefore relocation buffer B' contains N+1 elements.

The final transformed code of the example in Figure 11a) is shown in Figure 11c).