Data Relocation and Prefetching for Programs with Large Data Sets

Yoji Yamada, John Gyllenhaal, Grant Haab, Wen-mei Hwu

Center for Reliable and High-Performance Computing
University of Illinois
Urbana-Champaign, IL 61801

Correspondent: Wen-mei W. Hwu
Tel: (217)-244-8270
Fax: (217)-333-5579
Email: hwu@crhc.uiuc.edu

Abstract

Numerical applications frequently contain nested loop structures that process large arrays of data. The execution of these loop structures often produces memory preference patterns that poorly utilize data caches. Limited associativity and cache capacity result in cache conflict misses. Also, non-unit stride access patterns can cause low utilization of cache lines. Data copying has been proposed and investigated in order to reduce the cache conflict misses [1][2], but this technique has a high execution overhead since it does the copy operations entirely in software.

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. Experimental results for data relocation and prefetching are encouraging and show a large improvement in cache performance.

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

1 Introduction

Numerical applications frequently contain nested loop structures that process large arrays. The execution of these loop structures has been shown to produce memory preference patterns that poorly utilize data caches [3][4]. The first of three problems involves an insufficient capacity of the cache: The data accessed by each loop may exceed the cache size, resulting in cache misses. Limited associativity of the cache presents a second problem: accesses to different arrays, or even to different elements of a single array, may conflict. The final problem involves non-unit stride access patterns that can cause low utilization of cache lines and wasted bus and memory cycles [5].

Potentially, one could use a larger cache size and higher cache associativity to eliminate cache capacity misses and cache conflict misses. This brute force approach, however, does not scale well with the rapidly increasing amount of data used in sophisticated numerical applications. 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 blocking transformations could reduce the working-set size of of data accessed in loop nests [4] [1] [6]. By reordering the execution of iterations, blocking transformations reduce the amount of data referenced between two references to the same data. Once the data accessed between two references to the same data is reduced to an amount smaller than the cache size, capacity misses are eliminated. In practice, however, blocking transformations may not reduce cache misses because of cache mapping conflicts. Additionally, blocking alone does not reduce the working-set size of data accessed in single loops since the data accesses are not reordered.

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 [5]. 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. The proposed approach achieves this goal by compressing the current and future working sets into a localized region in the virtual address space such that no cache mapping conflicts exist among locations in the region.

This paper presents an approach, however, to solve all three cache performance problems for array-based applications, rather than a solving these problems singly. The first phase in this technique consists of loop blocking inner loops to reduce the number of array accesses in the working set. In the second phase, the insertion of special hardware instructions compress the working set into a localized region in the virtual address space and prefetch the compressed working set into the cache. The compiler also modifies the working set accesses so that all references will be made to the compressed data in the cache. Since array data are sequentialized in the localized region, most conflict misses are eliminated. Also, 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. After the computation is completed, additional instructions decompress the modified data and relocate it back to the original program arrays.

In order to minimize the overhead of compressing and decompressing data, compression is performed as the data is prefetched from the memory into the cache. Also, through the use of compiler transformations, compression and prefetch of the next working set is overlapped with the current computation in order to

hide the latency of the relocation.

Using a prototype compiler, an emulation tool, and a simulation tool, we show that this extension to the cache architecture along with the requisite compiler support greatly improves the data cache performance for array-based applications.

The remainder of this paper is organized into six sections. Section 2 describes the proposed method and describes the necessary architecture and hardware support. Section 3 explains the compiler transformations for the data relocation and prefetching. In Section 4, simulation-based experimental results are provided to demonstrate the effectiveness of the proposed architecture. Related work is discussed in Section 5. Finally, Section 6 offers concluding remarks and future directions.

2 Data Relocation and Prefetching

2.1 Method

We propose a compiler-supported, hardware-based technique called data relocation and prefetching in order to improve the data cache performance. In this method, the array references in the inner loop of a nest are sequentially mapped in the cache before they are accessed. The relocation operations are invoked by explicit instructions that the compiler inserts. The compiler also inserts a declaration into the original code for the relocation buffer that allocates space for the relocated data in memory. Special hardware that is attached to the cache unit maps and compresses the data into the virtual buffer space so that the relocation can be performed while prefetching the data from the memory to the cache and without stalling the *CPU*. In order to access the relocated data (instead of the original array data) during the computation, the compiler replaces the original array references with corresponding relocation buffer references.

Because the array data is relocated, the prefetch is binding. During the computation, the newly assigned address in the relocation buffer space 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. If the relocated, cached data is replaced by some other original address accesses, such as scalar accesses, the relocated data must be written back to the relocation buffer in memory since the accesses in the computation use the address of the relocated data. To insure write-back of the relocated data, the dirty bit is set when the cache line for the data is allocated. When the computation that uses the relocated data is finished, all modified, relocated data are written back from the relocation buffer to their original memory locations using an explicit machine instruction.

Data relocation and prefetching can improve the 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

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

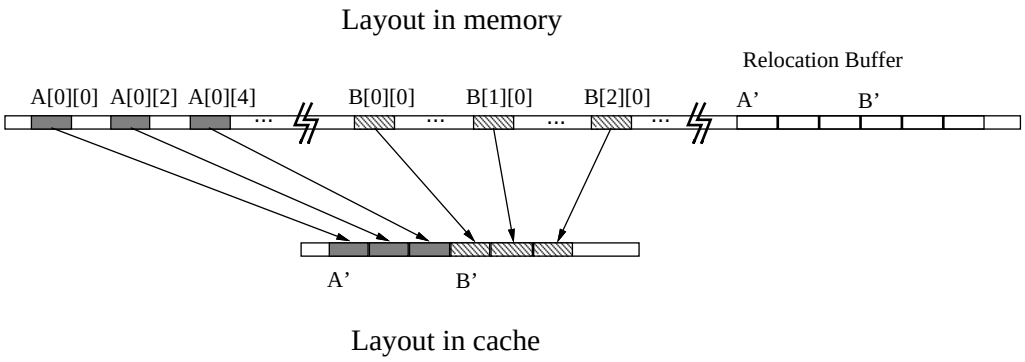

Figure 1: Concept of data relocation

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 not exhibit any 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 happen to 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 in the cache, spatial locality can be improved by packing elements of the arrays into contiguous locations. 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 relocation instructions, each 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. Further details are given in Section 3.

2.2 Architectural Support

Implementing the mechanism for data relocation and prefetching require extra instructions as well as extra hardware. Five instructions to support the data relocation and prefetching are added to the instruction set: *precollect*, *preallocate*, *distribute*, *await* and *finishup*.

2.2.1 Precollect

The *precollect* instruction, which has five operands, collects the array data referenced in a computation into consecutive locations in the cache before the data is needed for the computation. The first operand is the address of the first element of the array to be relocated, whereas the second operand is the address of the first element of the relocated array in the relocation buffer. 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. The information given by the third to the fifth operands is necessary since our scheme does not relocate entire cache lines but rather relocates array elements that will be accessed in the computation. Using these operands, the *precollect* instruction fetches the data at the original addresses and then stores the data into the cache with the tag of the relocated addresses. This instruction is non-blocking: It does not stall the processor even if it causes cache misses. Therefore, the execution of this instruction can be overlapped with the execution of the instructions which perform computations in the loop nest.

2.2.2 Preallocate

The *preallocate* instruction only allocates the necessary cache lines instead of collecting the array data into consecutive locations within the cache and can be used if the array data is not read before being written during the computation. In this case, there is no need to collect the data into the cache before the computation, eliminating the overhead of fetching the data. The *preallocate* instruction has the same operands as the *precollect* instruction, except the first operand (the original starting address) and the fourth operand (the stride of the original array accesses) are not needed. This instruction is also non-blocking.

2.2.3 Distribute

The *distribute* instruction writes the relocated data which were updated by the computation back to their original array locations in memory. The format of the *distribute* instruction is the same as that of the *precollect* instruction: The first operand specifies the starting address of the array elements before relocation, and the second operand specifies the starting address of the array data in the relocation buffer. Like the *precollect* and *preallocate* instructions, the *distribute* instruction does not stall the processor.

2.2.4 Await

The *await* instruction provides a simple synchronization mechanism to avoid accessing the relocated array data before the *precollect* or *preallocate* operation is completed. This instruction has a single operand that must match the second operand of the associated *precollect* or *preallocate* instruction. If the *precollect* or *preallocate* operation has not been completed, the *await* operation causes the data cache to block until the operation is finished.

2.2.5 Finishup

After the loop nest finishes execution, there may be unfinished *distribute* operations which must be completed before the array data is accessed in subsequent code. The *finishup* instruction provides a synchronization mechanism to insure that all *distribute* operations are completed by blocking the data cache. No operands are necessary.

2.3 Hardware Support

The execution of the data relocation and prefetch instructions is handled by special hardware called the *Data Relocation and Prefetch* (*DRP*) unit which can be attached to the existing cache unit (Figure 2). The *DRP* unit shares the *MMU* with the cache and also shares the cache itself with the *CPU*. In this configuration, both the processor and the cache have higher priority than the *DRP* unit in accessing the shared resources. This priority hierarchy helps to ensure that the *DRP* unit doesn't significantly slow down the execution of an application program while executing *precollect*, *preallocate* and *distribute* instructions unless an *await* or *finishup* instruction is executed.

If a high-bandwidth memory system like a split-transaction bus system is used, the *DRP* unit can utilize the bandwidth by pipelining the read and write requests to the memory system. The data cache configuration necessary to support the *DRP* unit is a write-back cache does not allocate on a write miss. Figure 3 illustrates the components of the *DRP* unit data path.

The *DRP* unit is designed in such a way that the cache need not block due to a fetch generated by the *DRP* instructions, which is insured by handling these fetches in the *DRP* unit. Since the cache does not block while writes are serviced unless the write buffer is full, writes misses generated by the *DRP* instructions are handled by the cache instead of dedicated *DRP* hardware. Furthermore, the *DRP* unit itself does not block when fetching from a memory location.

As program execution proceeds, *precollect*, *preallocate*, and *distribute* instructions from the processor are placed in the instruction queue. If the instruction queue is full, the *DRP* unit stalls the processor until there is an empty entry in the queue. Each instruction at the head of the queue is processed completely by the

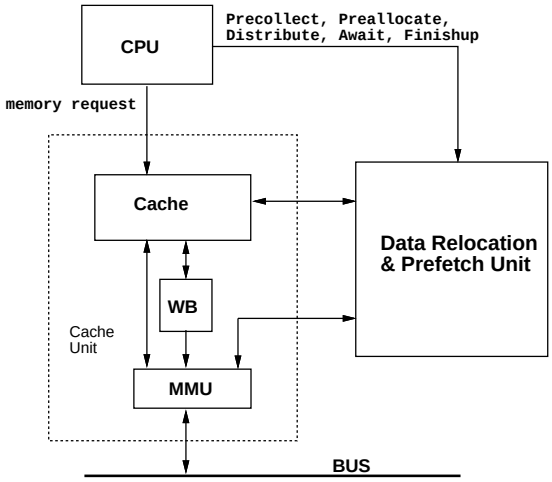

Figure 2: Data relocation and prefetch unit interfaces

address generator before the next may proceed.

The address generator calculates the original address of each array element using the starting array address and the stride information. The address generator also uses the starting relocation buffer address and the element size to calculate the relocation buffer address for each array element,

As each pair of addresses is generated, an entry is stored in the sub-operation queue for processing.

For the *precollect* and *preallocate* instructions, the original addresses are stored in the source address field, and the buffer addresses are stored in the destination field of the sub-operation queue. For *distribute* instructions, the addresses are reversed.

To begin processing a *precollect* sub-operation, the source address field is used to send a special read request to the cache. If the data is present in the cache, it is first stored in the *DRP* unit data buffer then written to the cache using the destination address. Also, the sub-operation is removed from the queue since it is finished.

If the data is not present in the cache, the cache does not send a read request to fetch the data, but instead, the *DRP* unit sends the read request to the *MMU* using the source address. At this point, the next sub-operation in the sub-operation queue begins processing, but the pending sub-operation is left in the queue for further processing. If the sub-operation queue has enough entries, the queue does not block while a sub-operation is waiting for memory access to be completed.

When the data returns from memory to the read buffer, the source address fields of the sub-operation queue are searched associatively using the address from the read buffer to obtain the destination address and size of the data from the same sub-operation. Then, the appropriate cache line is allocated, and the data from the read buffer is written to the cache using the destination address and size. All blocks written with relocated data in the cache are marked dirty.

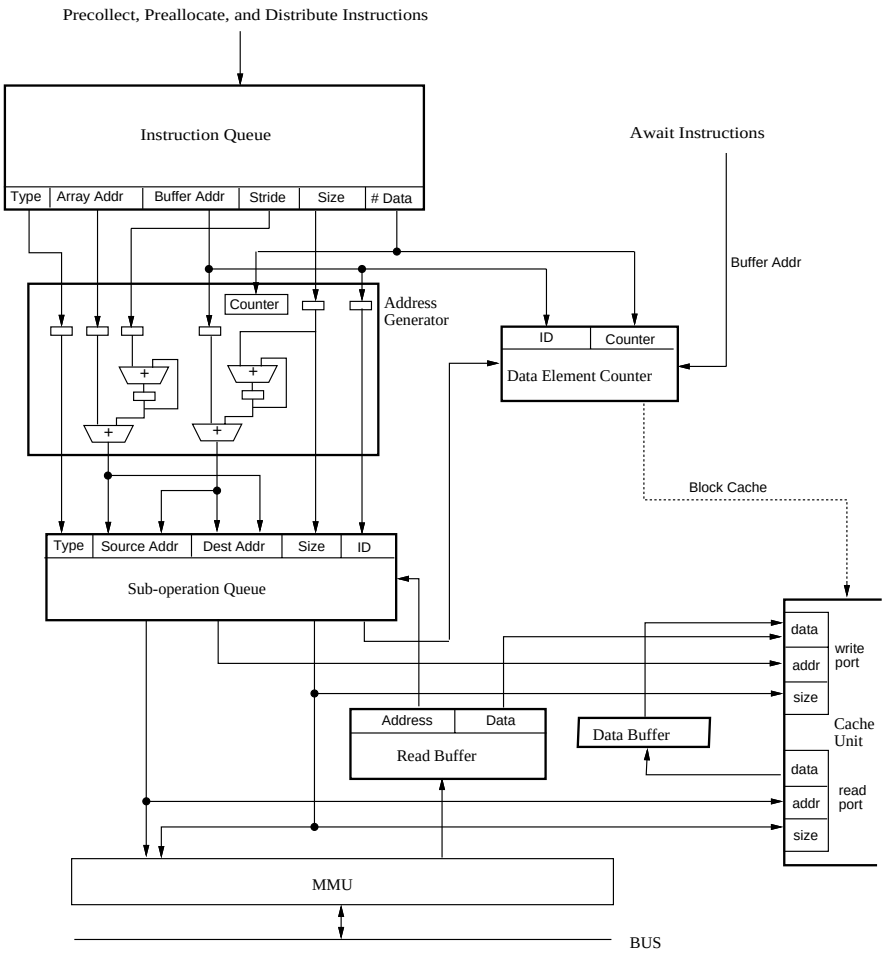

Figure 3: Data relocation and prefetch unit data path

To begin processing a *preallocate* sub-operation, the source address field is used to access the cache tag store. For each address that misses in the cache, a cache line is allocated for the relocated data, but the array data is not fetched into the cache. After the line is allocated, it is marked clean. Then, the sub-operation is removed from the sub-operation queue. The *preallocate* sub-operation must finish before the relocated address is written by the computation in order to insure correct results are obtained for program execution.

For a *distribute* sub-operation, the relocated data is read from the cache using the source address field from the entry at the head of the distribute queue. If the data is present in the cache, it is placed in the *DRP* unit data buffer. If the data is not present in the cache (which can occur if non-relocated data conflicts with relocated data in the cache), the cache does not fetch the data; but instead, the *DRP* unit sends a read request to the *MMU* using the source address. At this point, the next sub-operation in the queue begins processing, but the pending sub-operation is left in the queue for further processing.

When the data returns from memory, the address in the read buffer is used in an associative search of

the source address fields of the sub-operation queue for the destination address and size fields. Next, a write request is sent to the cache using the destination address and data in either the read buffer or the data buffer, depending on whether or not the read of the relocated data produced a cache miss or hit, respectively. The sub-operation is removed from the queue, and the data is written to the original address in the cache if a write hit occurs or in the memory if a write miss occurs.

When each instruction is passed from the instruction queue to the address generator, an entry is allocated in the data element counter for it, and the total number of data elements is used to initialize the counter. As the sub-operation queue finishes the processing of each data element, the element counter is decremented by one. Therefore, a *preallocate*, *precollect*, or *distribute* instruction has a corresponding non-zero data counter entry during execution. An *await* instruction causes the data cache to block until the data counter entry with the matching buffer address identifier reaches zero. A *finishup* instruction causes the data cache to block until all the data counters contain zero.

An interlock mechanism insures that a *precollect* or *preallocate* instruction will not begin execution until after all previously issued *distribute* instructions which use the same relocation buffer have completed execution. The identifier (buffer address) of the instruction at the head of the instruction queue is used to search the data counter associatively. If an instruction with the same identifier is still being executed, address generation is delayed until the previous instruction has been completed.

3 Compiler Support

Compiler support is essential to transforming the source code so that the proposed scheme can improve the cache performance. The *IMPACT* research prototype compiler [11] supports high-level transformations and optimizations, superscalar and *VLIW* optimizations and scheduling [12], as well as classical and machine-specific optimizations.

The data relocation and prefetching optimization can be applied most directly to loops nested at least two deep. Figure 4a shows an example loop nest that illustrates different array access patterns. For each invocation of the inner-most loop, all array data accessed in the inner-most loop are first relocated and prefetched, after which the computation proceeds until the inner loop is finished. The transformation is always applied for the two inner-most loops for loop nests which are nested more deeply than two. For 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. The high-level code transformations employed for data relocation and prefetching are loop unrolling, insertion of *DRP* operations, replacement of array references with relocation buffer references, and loop blocking.

3.1 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. This software-pipelining scheme requires two relocation buffers. The inner loop is duplicated by unrolling the outer loop once, as shown in Figure 4b. 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 buffers are switched. Data dependence information, provided by the Omega Test [13][14], is used to insure the validity of software-pipelining the data relocation and computation phases.

3.2 Insertion of Operations

New operations are inserted in the source-level code to perform the *precollect*, *preallocate*, *await*, *distribute*, and *finishup* operations. (See Figure 4c.) These high-level operations are replaced by the corresponding machine instructions at the assembly code level. One or more *precollect* and/or *preallocate* operations are inserted before the inner loop in order to relocate and, in the case of the *precollect* operation, fetch the data that will be accessed in the next iteration of the original outer loop. These two operations are non-blocking so that their execution can be overlapped with the computation in the subsequent inner loop.

In Figure 4c, $\mathbf{A}'$, $\mathbf{B}'$, and $\mathbf{C}'$ are pointers to the relocation buffer starting addresses for the elements of arrays $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ (respectively) accessed in the first unrolled outer-loop body, and $\mathbf{A}''$, $\mathbf{B}''$, and $\mathbf{C}''$ point to the relocation buffer starting addresses for array elements accessed in the second unrolled outer-loop body. A single-dimensional data-cache-sized array which is divided into two equal parts to represent the two relocation buffers is declared before the entire loop nest. The pointer variables to the new locations are then initialized to the starting addresses for the relocated arrays.

One or more *await* operations are inserted just before the inner loop to insure that each of the *precollect* and *preallocate* operations is completed before the inner-loop array computation begins. *Distribute* operations are inserted just after the inner loop in order to restore all updated data in the buffers to their original location in the memory. Finally, a *finishup* operation is placed just after the outer loop in order to insure that all *distribute* operations are completed before execution proceeds to other computations which may involve the same array data.

3.3 Replacement of array references with buffer references

Once the relocation and prefetching operations have been inserted, the array references for the computation within the inner loop are modified so that the buffer locations are accessed instead of the original array

locations. The final version of the example loop after all the transformations are completed is shown in Figure 4c.

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];
```

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];

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

c) After relocation operation insertion

```
/* Bd = dimension(B[]) * 8 bytes/element */
/* for first outer loop body */
precollect(&A[0][0], A', 8, 16, N);
precollect(&B[0][0], B', 8, Bd, N);
preallocate(C', 8, N);

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);
     preallocate(C'', 8, N);
  }
  await(A');
  await(B');
  await(C');
  for (j=0; j<N; j++)
    C'[j] = A'[j] + B'[j];
  distribute(&C[i][0], C', 8, 8, N);

  /* 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);
      preallocate(C', 8, N);
    }
    await(A'');
    await(B'');
    await(C'');
    for (j=0; j<N; j++)
      C''[j] = A''[j] + B''[j];
    distribute(&C[i][0], C'', 8, 8, N);
  }
}
finishup();
```

Figure 4: Example of loop unrolling and insertion of relocation and prefetching operations.

3.4 Loop Blocking

Loop blocking is an optimization technique that, by partitioning the iteration space, can increase the data reuse in the cache and thereby reduce the number of cache misses. For the data relocation technique, blocking of the inner loop is used to reduce the amount of data relocated for the inner-loop computation if that amount is too large to fit in the cache. As a consequence, single loops are often blocked even though

the access pattern of the data in the loop body is not affected by the blocking transformation. Again, data dependence information provided by the Omega Test is used to determine if loop blocking is a valid transformation for a loop nest.

4 Experimental Evaluation

In this section, the effectiveness of the data relocation and prefetching optimization technique is evaluated through simulations for a set of array-based benchmarks.

4.1 Methodology

4.1.1 Benchmark Programs

The benchmarks for this study consist of six numeric programs: ADM, OCEAN, and ARC2D from *PERFECT* [15], MATRIX300 from *SPEC'89* and TOMCATV and NASA7 from *SPEC'92*. All benchmarks 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 transformations selectively and effectively. Loop nests that contain multiple inner loops, return, goto, or break statements were excluded, as well as loopnests which contain subroutine calls with possible side-effects. Data dependence analysis is also used to exclude loops with specific cross-iteration dependences which can prevent the necessary transformations.

4.1.2 Transformation Correctness Verification via Emulation

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

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* and *distribute* instructions were emulated using machine language subroutines that perform 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. Timing simulation is also needed to verify correctness of the detailed operation of the *DRP* unit since the emulated *DRP* instructions stall the processor until they are completed.

Function	Latency	Function	Latency
Int ALU	1	FP ALU	2
memory load	2	FP multiply	2
memory store	1	FP divide (SGL)	8
branch	1 / 1 slot	FP divide (DBL)	15

Table 1: Instruction latencies for simulation experiments.

4.1.3 Simulation Experiments

The emulator drives the simulator that models the processor and the *DRP* unit to determine execution time, cache performance, and bus utilization. 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. The data cache is a multiported, write-back, no write-allocate cache that satisfies four load or store requests per cycle from the processor. The 8-entry write buffer combines write requests to the same cache line. Streaming of data from load misses minimizes the load miss penalty. The instruction cache and data cache share a common, split-transaction memory bus, with a 64 bits/cycle data bandwidth. A pipelined memory model is used with a 10-cycle latency.

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 description in Section 2. However, infinitely-sized queues and buffers are modeled so that no blocking occurs in the *DRP* unit due to insufficient entries. Also, the data cache has one read and one write port dedicated to service *DRP* unit accesses.

Since simulating the entire benchmark programs at this level of detail would be impractical, uniform sampling is used to reduce simulation time [16]. 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 benchmarks used have more than a billion dynamic instructions, at least 50 samples, and thus, more than 10,000,000 instructions are simulated. For smaller benchmarks, the spacing is reduced to maintain at least 50 samples (10,000,000 instructions). From experience with our emulation-driven simulator, we have determined that sampling with at least 50 samples introduces very little error in our performance estimates. Typically, the statistics generated with sampling are within 1% of those generated without sampling.

4.2 Experimental Results

In order to show the full performance benefit of the *DRP* technique, experimental results are presented for all benchmark program loop nests which are modified by the *DRP* transformation. Results for the entire

Benchmark	Transformed Loop Nest	Execution Percentage
ARC2D	FILERY.35J	2.4%
	FILERY.36J	1.4%
	FILERY.38J	2.3%
	STEPFY.430	5.3%
	SPECT.100J	1.2%
	SCALDT.10J	1.4%
	TK.1J	2.6%
	TKINV.1J	2.7%
	RHSY.20J	2.2%
	RHSY.30J	2.1%
	RHSX.200J	2.2%
	EIGVAL.100J	3.3%
	YPENTA.13	1.4%
	YPENT2.13	1.4%
OCEAN	IN.10	11.2%
	OUT.10	8.6%
ADM	LEAPFR.30	2.2%
TOMCATV	MAIN.401I	25.7%
	MAIN.501I	14.4%
MATRIX300	SAXPY.10	96.3%
NASA7	CHOLSKY.3L	1.3%
	CHOLSKY.8L	1.0%
	CHOLSKY.7L	4.7%
	COPY.100	3.1%
	VPENTA.15	5.4%

Table 2: Original code loop-nest execution time as a percentage of entire program execution time for *DRP*-transformed loopnests.

benchmarks are also presented to evaluate the current state of our compiler techniques for loop nest selection and application of the *DRP* technique.

4.2.1 Individual Loop Nest Results

Performance statistics for individual loop nests are obtained by marking the *DRP*-transformed loop nests as execution regions for the simulation. Consequently, simulating the execution of the entire program once is sufficient to gather results for transformed loop nests in the context of the entire program execution. Table 2 shows the original code execution time for each of the loop nests selected by the *DRP* technique as a percentage of the total original code execution time. The transformed loop nests are identified by function name, the *Fortran* outer DO-loop number, and the loop iteration variable if necessary.

Figure 5 illustrates the speedups of the original loop nest execution time over the *DRP*-transformed loop nest execution time. Measured speedup for most loop nests is relatively large, demonstrating the high performance improvement obtainable using the data relocation and prefetch technique.

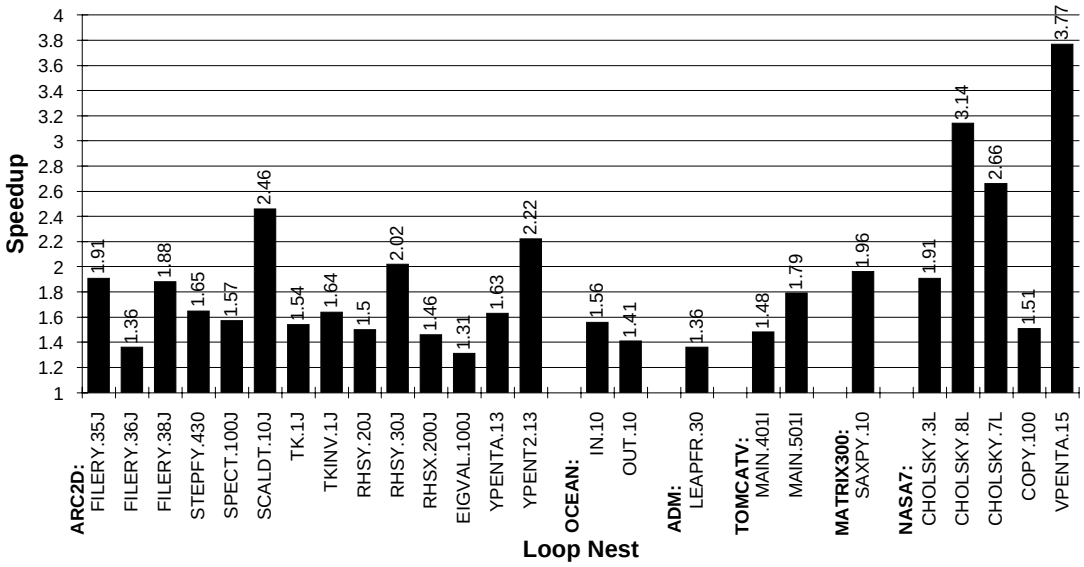

Figure 5: Speedup of the *DRP*-optimized code over the original code for the loop nests.

Figure 6 shows the data cache miss ratio for the original code and the *DRP*-optimized code for the loop nests. Since a no write-allocate cache is used for these experiments, the miss ratios for both original and optimized 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. This ratio does not include requests initiated by the *DRP* unit. Since these cache reads do not stall the *CPU*, they have a minimal impact on performance.

Note that cache misses are nearly eliminated for most of the loop nests. Since the compiler relocates all array data referenced in the loop nest, there are few possibilities for cache misses to occur. Scalar variables accessed in the loop nest are not prefetched and may conflict with relocated data, causing cache misses. Several loop nests in the *ARC2D* benchmark contain inner-loop scalar accesses which contribute to non-zero cache miss rations: *STEPFY.430, TK.1J, TKINV.1J, RHSX.200J, RHSX.200J* and *EIGVAL.100J*. Another cause of cache misses is additional memory accesses introduced by register spill code, as illustrated by the loop nests *EIGVAL.100J* and *YPENTA.13*. The execution speedup obtained for the loop nests is not always highly correlated to the reduction in cache miss ratio because of the complexity of scheduling a multiple-issue processor.

The memory bus utilization for the transformed and original loop nests is displayed in Figure 7. The *DRP* unit utilizes the unused bus cycles because it has lower priority than the cache when accessing the bus. In a few cases, however, the precollect and preallocate instructions have not been completed by the time the await instruction is issued.

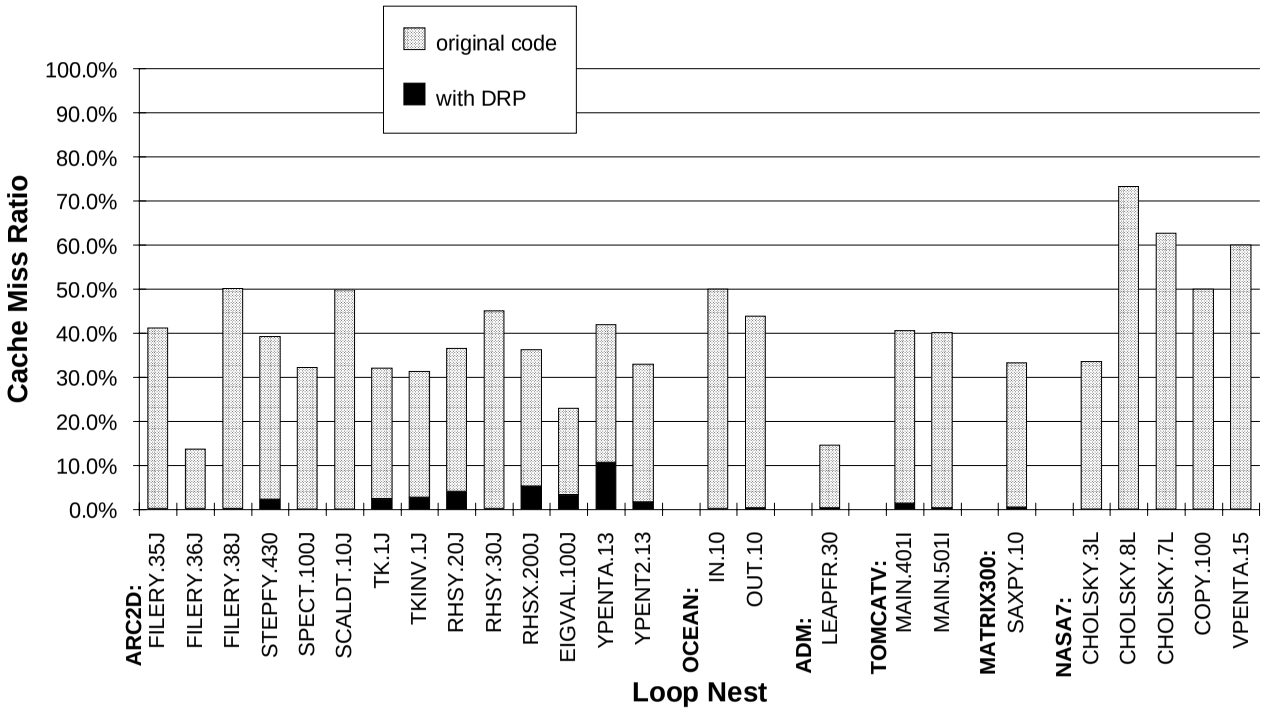

Figure 6: Cache miss ratio of the *DRP*-transformed code and the original code for loop nests.

In some loops, there is insufficient computation to overlap the latency of the *precollect* operations, even though the operations have been moved back across one iteration of the outer loop by the software pipelining. Thus, a *precollect* is always waiting to use the bus when it is free, resulting in near 100% bus utilization for these loops.

Figure 8 provides evidence for this effect, by displaying the percentage of execution cycles for which the *await* instruction stalls the processor for the transformed loop nests. Notice that the loop nests which have more than 95% bus utilization also have a large percentage of cycles stalled on *awaits*, while those with below 95% bus utilization have fewer stalled cycles. To reduce the *await* cycles and improve performance, the *precollect* and *preallocate* operations must be executed earlier than one iteration before the corresponding computation.

4.2.2 Entire Benchmark Results

Speedups for the simulated execution of the *DRP*-transformed code over the original code for the six benchmarks are given in Table 3. The program execution speedup of all benchmarks except for *MATRIX300* and *TOMCATV* is relatively small. For *ADM*, *NASA7*, and *ARC2D* the small speedup is attributable to the fact that percentage of the execution time spent in the transformed loop nests is relatively small since we used an overly-conservative loop nest transformation criterion to guarantee the correctness of the transformation.

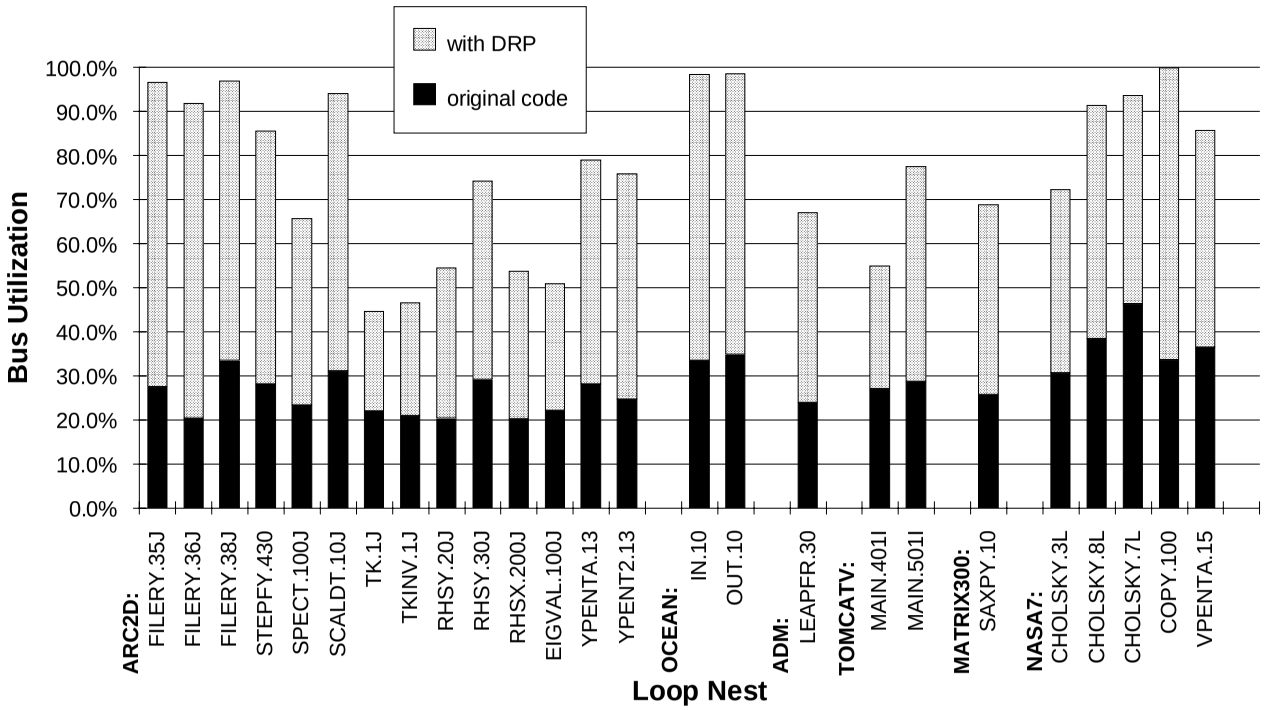

Figure 7: Bus utilization for the *DRP*-transformed code and the original code for loop nests.

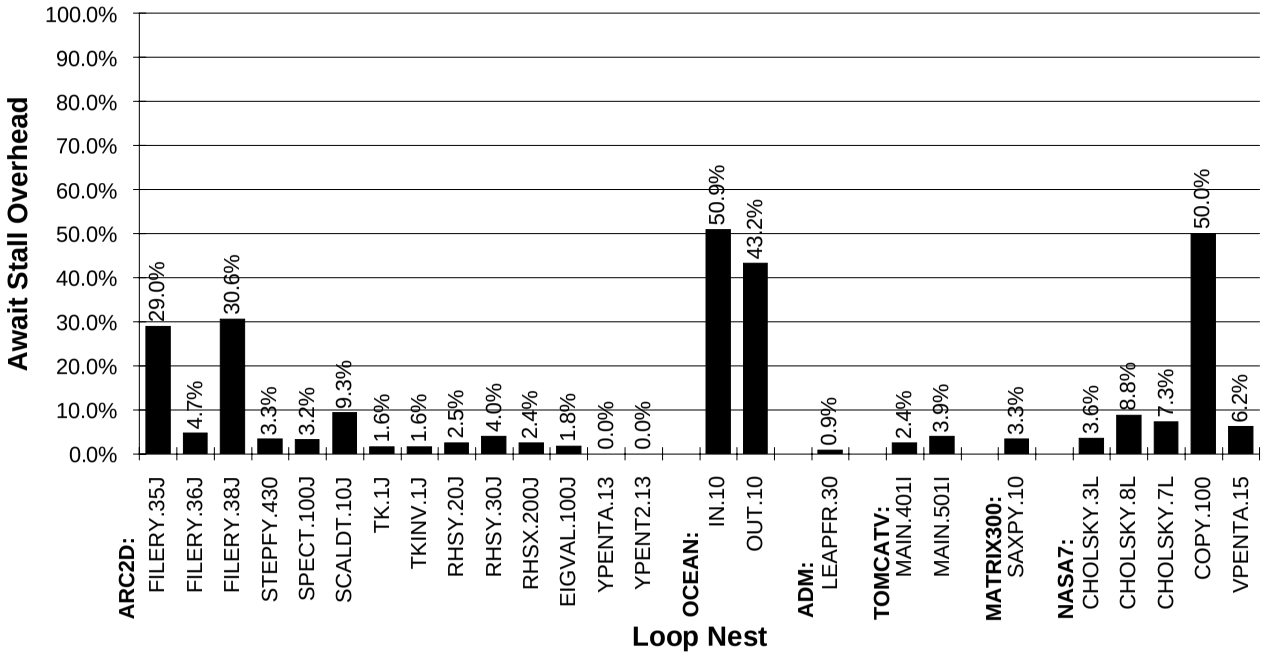

Figure 8: *Await* stall overhead for the *DRP*-transformed code and the original code for loop nests.

Benchmark	Transformed Loop Nest Execution %	Execution Time in Billions of Cycles		Execution Speedup
		Base	DRP	
ARC2D	31.9%	9.248	8.300	1.11
OCEAN	19.8%	11.165	10.512	1.06
ADM	2.2%	1.732	1.703	1.02
TOMCATV	40.1%	1.442	1.065	1.35
MATRIX300	96.3%	3.181	1.586	2.01
NASA7	15.5%	13.010	11.788	1.10

Table 3: Speedup of the *DRP*-transformed code over the original code for entire benchmarks.

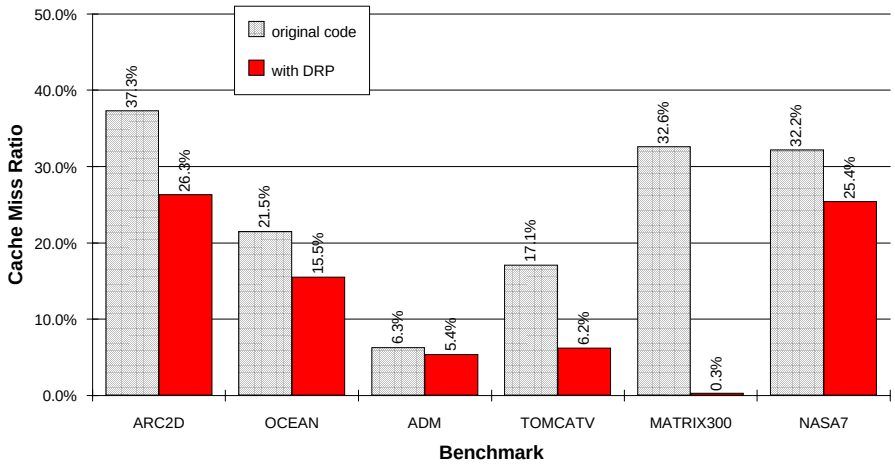

Figure 9: Data cache miss ratio for original code and *DRP*-transformed for the benchmarks

For *OCEAN*, the processor stalls due to the *await* instructions severely restrict the performance benefit of the *DRP* technique, as discussed in Section 4.2.1.

The speedup measurements given in Table 3 match very closely the entire benchmark speedups calculated using the loop speedups in Figure 5 and loop execution percentages in Table 2 except for the *TOMCATV* benchmark. In *TOMCATV*, the whole program speedup exceeds the weighted sum of the loop speedups because some of the non-transformed loops benefit from the cache effects of the transformed loops. Although the transformed loops accesses enough array data to replace the data in the entire cache, the transformation substantially reduces the space used to cache the array data. Since the data replaced in the cache by the original code had extensive reuse opportunities, the transformed code allowed most of the this data to be effectively reused. This reuse results in much better cache performance for an important non-transformed loop nest, due to a reduction in inter-array conflict misses.

The data cache miss ratios show promising improvement for almost all benchmarks shown in Figure 9. The binding-prefetching mechanism of the *DRP* technique guarantees that most of the necessary data for

the computation resides in the cache. Also, the compression of the data in the relocation buffer increases the utilization of the cache so that more data can reside in the cache at the same time.

5 Related Work

A technique called data copying has been proposed and investigated in order to reduce the cache conflict misses [1][2]. Data copying, however is beneficial only if the performance improvement outweighes the overhead of copying by reusing the data many times. The overhead of copying data from array to array is significant in general. Our proposed method, data relocation and prefetching, has the same benefits as data copying while reducing the overhead of copying. Furthermore, unlike data copying, data relocation and prefetching relocates data into the cache during prefetching, which minimizes the overhead of the relocation.

The *precollect* and *distribute* operations are conceptually similar to the *Gather* and *Scatter* operations used in the Cray-1 [17]. 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 to necessary to support data relocation and prefetching would be much easier to add to an existing processor than the hardware to support vectorization.

6 Conclusions

An architectural extension, referred to as the data relocation and prefetching, is proposed to perform data relocation and compression during prefetching. Data relocation is used 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.

By combining the data relocation and prefetch hardware with supporting compiler transformations, the performance of loop nests is greatly improved for a set of array-based benchmarks. Also, we have shown that application of the data relocation and prefetching optimization greatly improves the cache performance.

Performance measurements for entire benchmarks motivate the need for future research and tuning of the data relocation and prefetch techniques. *Precollect* and *preallocate* instructions need to be moved back across more than one iteration of the outer loop in order to reduce the processor stalls caused by the *await* instruction. 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 optimization using various implementation parameters for the DRP hardware.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[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. 24st 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 *Proceeding of 6th International Conference on Supercomputing*, July 1992.

[11] 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.

[12] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. W. R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. H. 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.

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

[14] 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.

[15] 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. J. nson, 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.

[16] 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.

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