Compiler Techniques for Software Prefetching of Cache-Coherent Shared-Memory Multiprocessors

Nathaniel McIntosch
Katherine Fletcher
Keith Cooper
Ken Kennedy

CRPC-TR96675-S
October 1996

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

Compiler Techniques for Software Prefetching on Cache-Coherent Shared-Memory Multiprocessors

Nathaniel McIntosh[*], Katherine Fletcher[†], Keith Cooper, Ken Kennedy
Department of Computer Science
[†]Department of Electrical and Computer Engineering
Rice University

Abstract

In this paper we present a comprehensive compiler framework for improving the efficiency of compiler-directed software prefetching on cache-coherent distributed shared-memory multiprocessors. The key component of our work is a form of global data-flow analysis that predicts at compile-time the sets of array references that are likely to cause coherence activity at run-time. The data-flow framework accurately analyzes the cache behavior in a parallel program by combining array section analysis with knowledge about the cache configuration and an encoding of the target machine's cache coherence protocol. Existing prefetching algorithms have problems issuing prefetches for coherence misses, resulting in late prefetches and latency penalties. Our compiler identifies the particular variable references and loop iterations that cause coherence misses, and schedules prefetches for these references farther in advance, effectively hiding the latency that they incur. In other situations where existing prefetching techniques encounter difficulties, such as false sharing and many-procesor read sharing, we use data-flow information to apply optimizations that decrease interconnect traffic and reduce the memory latency penalties incurred by the program.

1 Introduction

Cache-coherent shared-memory multiprocessors are an attractive platform for small- to medium-scale parallel programming, since they provide a convenient programming model based on a hardware-supported shared address space. Modern multiprocessors often use a distributed shared-memory (DSM) architecture, in which main memory is divided up into modules and distributed across all of the processors; the processors are then linked together with a high-speed interconnection network [1, 6, 17]. If a processor in a DSM machine accesses a location that is not already in its cache, the underlying hardware may have to send a request across the interconnect to the processor that owns the data (and possibly other processors as well); these remote memory accesses can take many hundreds of cycles.

Because of the penalties associated with remote memory accesses, cache utilization is a critical component of overall performance for programs running on these machines; techniques that improve cache behavior and/or

[*]Corresponding author. Email: mcintosh@cs.rice.edu, phone 713-527-8101 x2731, fax 713-285-5930.

hide the effects of memory latency are an effective way to improve program performance.

1.1 Compiler-Directed Software Prefetching

Compiler-directed software prefetching is a hybrid hardware/software technique for hiding the latency of cache misses [20, 22]. There are two primary components for software prefetching: hardware support for a *prefetch* instruction, and a compiler that inserts prefetches into the programs it compiles.

The most successful prefetching schemes make use of advisory prefetches. This type of prefetch is essentially a hint; the semantics of the program are unchanged if a prefetch is discarded. Prefetch instructions have the same format and addressing mode as `LOAD` instructions, except that no destination register is specified. When a prefetch is executed, the memory subsystem checks to see whether the line containing the specified address is in cache, and if it is not, it begins fetching the correct line into the cache. The prefetch operation executes asynchronously– the processor continues to execute other instructions while the prefetch is in progress. If the prefetch for a given data item is scheduled sufficiently far in advance of the actual load of the item, then the location will be in cache when the actual reference occurs, avoiding any cache miss latency. In compiler-directed software prefetching, the compiler analyzes the application program to determine the set of variable references likely to result in cache misses at run-time. It then transforms the program, inserting a prefetch instruction prior to any load instruction that is predicted to cause a cache miss. The compiler uses a specialized type of software pipelining to insure that prefetch instructions are issued sufficiently far in advance of their corresponding loads [22].

1.2 Prefetching for parallel programs

The critical difference between uniprocessors and DSM multiprocessors with regard to prefetching is that the time required to satisfy a cache miss varies tremendously depending on data sharing patterns.

Existing uniprocessor prefetching strategies schedule prefetches assuming a worst-case memory latency; if the prefetched data arrive early in such cases, the consequences are not very serious. In a DSM multiprocessor, however, cache misses can take many hundreds of cycles; if the compiler schedules *all* prefetches for the worst-case latency, data arriving too early may displace the current working set from the cache, eliminating any benefits of the prefetching. In previous studies of software prefetching on multiprocessors, researchers assumed a uniform "average" miss penalty [20, 21, 24]. This assumption was necessary because existing compilers had no accurate way to predict coherence misses, and could not take steps to avoid coherence miss latency.

In this paper, we describe a new form of compiler analysis that predicts coherence activity at important points within a program. The analysis identifies the references and loop iterations that incur coherence misses, allowing the compiler to generate prefetches for them effectively (see section 3.1). In addition, the analysis identifies and handles other situations that cause difficulties for prefetching, such as false sharing (see Section 3.2.3), many-processor read-sharing (Section 3.2.1), and read-modify-write access patterns (Section 3.2.2).

An outline for the remainder of this paper is as follows. In Section 2.1, we specify our target programming model and architectural assumptions. In Sections 2.3 and 2.4, we lay the groundwork for our data-flow framework, showing how we use array section analysis to capture a parallel loop's data access patterns. In Section 2.5, we give a set of conditions that predict coherence misses for reads and writes. In Sections 2.6 through 2.8, we describe the data-flow analysis framework that realizes these conditions. In Section 3, we show how our compiler uses the information generated by the flow analysis to improve program performance. In Section 4, we discuss our plans for experimentally validating our techniques. Section 5 describes related research. In Section 6, we offer concluding remarks.

2 Description

2.1 Target architecture, programs

This work targets shared-memory, cache-coherent DSM multiprocessors that support invalidation-based coherence protocols [1, 6, 17]. Examples of machines that support such protocols include the Alewife machine [1], the FLASH multiprocessor [17], and the Convex Exemplar [6].

We have chosen to target scientific Fortran programs, since there is a large existing base of such applications, and since these programs often place heavy demands on the cache and memory subsystems of the machines they run on. Our compiler accepts sequential Fortran programs that have been annotated with "DOALL" directives, along with private/shared classifications for each variable. We currently require that loops use barrier synchronization only, and we assume that the iterations of a given parallel loop are scheduled (assigned to processors) either statically by blocks, or through the use of an *affinity scheduler* [19]. Although our compiler does not currently incorporate data distribution directives (of the sort provided in High Performance Fortran [14]), our techniques could easily be extended to exploit information of this nature.

These restrictions and assumptions are intended to coincide with the parallel programs currently generated by commercially available automatic parallelizers such as KAP [16] or by an advanced workstation compiler.

2.2 Invalidation-based coherence protocols

When a processor writes to a location that other processors may have cached, or when it reads a location that another processor may have recently written, the underlying hardware takes action to ensure that all processors see a "coherent" view of memory. In an invalidation-based coherence protocol, when a processor writes to a location, the hardware brings an "exclusive" copy of the line containing the location into the cache and invalidates any copies of the line on remote processors. The state of the line is marked as "exclusive", indicating that no other processor is caching the line. A "shared" state indicates that other processors may have a copy of the line, but the contents of all the copies are the same as in main memory. Finally, an "invalid" state indicates that the data in the line is stale, and should not be used. Figure 1 shows an example coherence protocol state diagram.

The coherence protocol impacts prefetching in several important ways. First, coherence misses (accesses to locations that have been invalidated since they were cached) may add significantly to the total miss count. Second, the time to retrieve a line from a remote processor's cache can be much greater than when simply reading the line from memory. Finally, if a processor reads a line and then immediately writes the same line, the hardware may have to issue two remote requests (one to obtain a shared copy of the line, and one to obtain an exclusive copy of the line), delaying the write and resulting in additional network traffic.

We address each of these concerns. Our analysis tracks coherence misses in addition to capacity misses; we identify long-latency misses and issue prefetches for them farther in advance, and we issue "exclusive" prefetches for lines that will be written in the near future.

2.3 Framework for predicting coherence misses

We have developed a global data-flow analysis method that predicts coherence activity within a parallel program. We give a brief outline of the procedure here and then describe each component in detail in subsequent sections of the paper. Our compiler uses array section analysis to identify portions of shared arrays accessed in specific program regions (typically loop nests). We augment array sections with a *parallel mapping* component that describes how a region within an array is accessed by the processors in the machine. We develop a series of equations that compute where a particular data access will be satisfied (from cache, from local memory, from remote memory, from a remote cache); the sets used in the equations are in terms of the augmented array sections. Our compiler solves these equations using interval-based data-flow analysis; the solutions to the equation for a given control-flow graph node are then used to predict whether particular references within the node will access data that resides in cache, and if not, whether retrieving the data will require coherence activity. To account for cache size constraints, we enhance the analysis by incorporating an "age" function on array sections that approximates the number of capacity misses since the section was cached. The data-flow machinery updates the ages of each section during the analysis; when a section's age reaches a machine-dependent cutoff point, it is eliminated from the set that contains it.

2.4 Array section analysis

Most scientific Fortran programs spend the bulk of their execution time performing computations on arrays in loops; in order to characterize the memory usage patterns for these programs, the compiler must analyze how arrays are accessed within loops. In our framework, we capture information on array access patterns using *array section analysis* [3, 4, 13, 18]. When applied to a portion of the program (typically a basic block, loop, or loop nest), array section analysis produces a summary representation of the region accessed within each array.

Each of the summaries (referred to hereafter as "sections") contains a component that describes the geometric region accessed within the array, in addition to other bookkeeping data and information about the order in which array dimensions are traversed. In order to be useful for our work, the sections must also capture information on how the accesses to a given array region are distributed among the available processors at run-time. For example, in Figure 2, the "`do` k_3" loop nest and the "`do` j_2" loop nest both access the region `a(1:100,1:100)`. In the first case, however, each processor accesses a block of columns, whereas in the second case, each processor accesses a block of rows [recall that for each "doall" loop, a single contiguous block of iterations is assigned to each processor].

To distinguish between these cases, we augment each section with a *parallel mapping* component, or PMAP, that contains information on how the elements within the section are accessed by the available processors at run time. The PMAP is a restricted version of the "mapping function descriptor" used in the *Available Section Descriptor* abstraction [10].

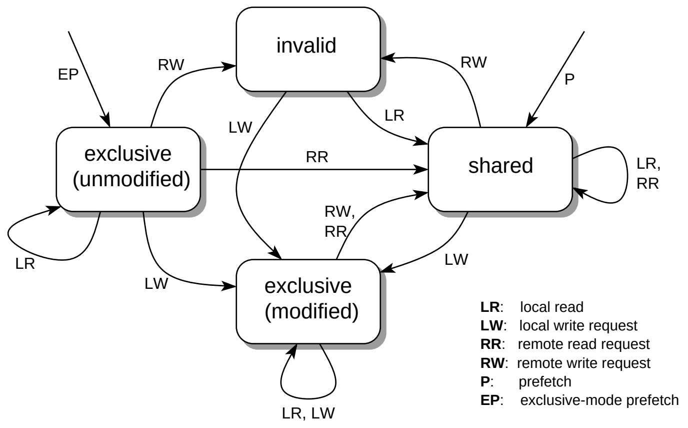


Figure 1: State diagram for example MESI cache coherence protocol

```
do k1 = 1, n
   doall k2 = 1, 100
      do k3 = 1, 100
         ... = ... a(k3,k2) + b(k3,k2,k1) + d(k3)
      enddo
   enddoall
   doall k4 = 1, 50
      do k5 = 1, 50
         ... = ... b(k5,k4,k1)
      enddo
   enddoall
enddo
```

```
do i1 = 1, 100
   ... = ... c(i1,2)
enddo

doall j1 = 1, 100
   do j2 = 1, 100
      ... = ... a(j1,j2)
   enddo
   do j3 = 1, 50
      ... = ... a(j3,j1)
   enddo
enddoall
```

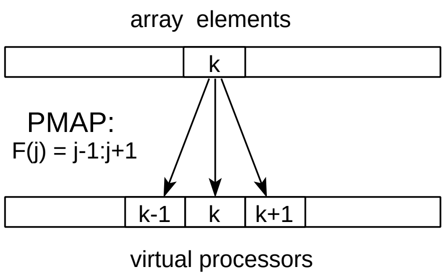
Section	loop	REGION	PMAP
S_1	do k_3	a(1:100,1:100)	$\langle 2, F(j) = j : j : 1\rangle$
S_2	do k_3	b(1:100,1:100,1:n)	$\langle 2, F(j) = j : j : 1\rangle$
S_3	do k_3	d(1:100)	$\langle 0, \top\rangle$
S_4	do k_5	b(1:100,1:50,1:n)	$\langle 2, F(j) = j : j : 1\rangle$
S_5	do i_1	c(1:100,2)	$\langle 0, \bot\rangle$
S_6	do j_2	a(1:100,1:100)	$\langle 1, F(j) = j : j : 1\rangle$
S_7	do j_3	a(1:50,1:100)	$\langle 2, F(j) = j : j : 1\rangle$

Figure 2: Example loop nests with section information

We assume an unbounded virtual processor grid with N dimensions (note that since our program model currently permits only a single level of parallelism, N is currently 1). For a given section S_1, the mapping function is of the form $\langle P, F\rangle$, where P and F are vectors of length N. Element i within P (denoted P_i) is the dimension of the array mapped to grid dimension i, and element i within F (denoted F_i) is a mapping function that gives the position(s) along the processor grid that array elements are mapped to. The mapping function is of the form

$$F_i(j) = (c * j + l : c * j + u : s)$$

where l and u are invariants, s (stride) is an integer constant, and c is either 1 or 0.[1] This triplet-style formulation allows one-to-one mappings (when $l = u$), one-to-many mappings (when $u \geq l + s$) and constant mappings (when $c = 0$). The following picture illustrates the effects of an example mapping function:



There are two special-case mapping functions as well. We use the mapping function $\langle 0, \perp\rangle$ to indicate accesses that take place outside parallel loops, and the mapping function $\langle 0, \top\rangle$ to indicate that that the section in question is accessed by all processors.

The table appearing at the bottom of Figure 2 shows each of the regions accessed within the loop nests, along with their parallel mapping functions.

2.5 Predicting coherence misses

Figure 3 shows a program fragment containing a parallel loop, along with its corresponding control flow graph (CFG). We would like to predict whether some or all of the array accesses in the loop will result in coherence misses. This will depend on the access patterns within and prior to the loop (i.e. node X and its predecessors). Suppose that a loop prior to node X writes to a section S_X in the array "`b`", and that the `doall j` loop accesses a section S_j in array "`b`". When the `doall j` loop executes, there will be coherence activity if regions of the two sections overlap, but the common elements are cached on different processors (we will formalize this notion shortly, with the definition of *conformability* of array sections). In order to predict coherence activity, therefore, the compiler must symbolically compare the regions of the arrays accessed within a given loop with the sections that are likely to be in cache on entry to the loop.

Cache size is an important factor; if none of the array elements accessed within the `doall j` loop are resident in cache upon entry to the loop, then there can be no coherence misses. For simplicity we ignore cache size constraints initially; in Section 2.8 we show how we take cache size into account.

[1] In the mapping function that appears in Gupta and Schonberg's ASD, c can take on a rational value, provided that the function evaluates to a range over integers. This is more general than is needed for our work.

2.5.1 Conformability of sections

We say that section S_1 *conforms* to section S_2 if and only if all of the following conditions are met:

1. $\text{REGION}(S_1)$ contains $\text{REGION}(S_2)$, or $\text{REGION}(S_2)$ contains $\text{REGION}(S_1)$
2. $\text{PMAP}(S_1) = \text{PMAP}(S_2)$
3. if $\text{PMAP}(S_1) = \langle N, F \rangle$ where $N > 0$, then the bounds on dimension N in $\text{REGION}(S_1)$ are identical to the bounds on dimension N in $\text{REGION}(S_2)$

For example, in Figure 2, section S_1 conforms to S_7, since these two sections satisfy all of the conditions above. However S_1 does not conform to S_6, since $\text{PMAP}(S_1) \neq \text{PMAP}(S_6)$. Similarly, S_2 does not conform to S_4, since the third condition above is not met.

For the purposes of this analysis, we relax the definition of containment to include situations where there is not strict geometric containment, but the two regions substantially overlap (i.e. the boundaries match except for a small constant). For example, we would treat a section S_1 = `a(1:1000,2:999)` as containing the section S_2 = `a(1:1000,1:998)`, and vice versa.

We use the following notation in the remainder of this paper:

notation	interpretation
$X \approx Y$	X conforms to Y
$X \not\approx Y$	X does not conform to Y
$X \sqsupseteq Y$	X contains Y

2.5.2 Read outcomes

Figure 4 shows the possible outcomes for a load (read) of shared data on a DSM multiprocessor, from the perspective of the processor issuing the load. There are three possible scenarios; the particular outcome depends on the state of the local cache and the caches of the other processors. For a given scenario, the latency incurred by the read is given in terms of M, the main memory latency, and N, the network latency (time required to send a message over the interconnect). For example, in scenario "B", the processor issuing the read is accessing a location that is currently not cached by any processor. In this scenario, the issuing processor sends a request message to the processor whose memory contains the location in question, which then sends a message back with the data. The scenarios shown in Figure 4 are worst-case latencies (we assume that the data being accessed is owned by a remote processor, not the local processor).

Figure 5 gives a set of conditions that are sufficient to predict each of the three scenarios for a read operation. In the cases where there are multiple conditions for a given outcome, they should be interpreted as being "and'ed" together to form the condition for the outcome. In the figure, X refers to the node containing the read we want to analyze (note that for the remainder of this paper, the term "node" will be used to refer to a CFG node, not to a processor within the DSM machine). The superscript notation for array sections specifies the type of access: S^w corresponds to a write, S^r corresponds to a read, and S^{rw} is either a read or a write. The subscripts for sections indicate the CFG node where the access takes place; S^r_X is the read operation at node X whose outcome

```
...X...
doall j = 1, n
   do i = 1, n
      a(i,j) = ... b(i,j) ...
      ...
   enddo
enddoall
...Y...
```

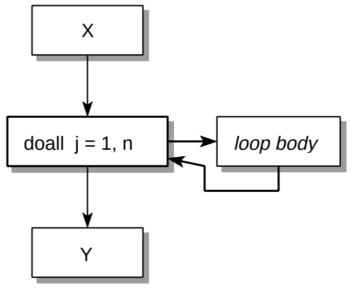


Figure 3: Parallel loop with corresponding CFG

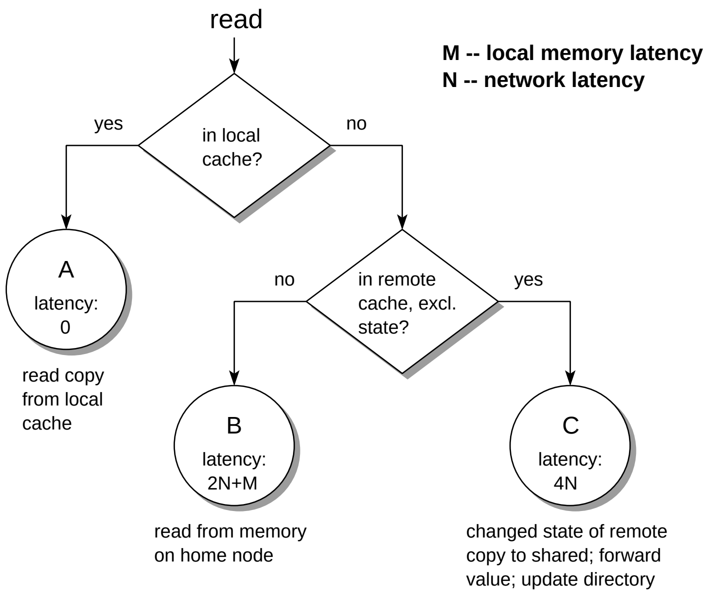

read
M -- local memory latency
N -- network latency
in local cache?
yes
no
A
latency: 0
read copy from local cache
in remote cache, excl. state?
no
yes
B
latency: 2N+M
read from memory on home node
C
latency: 4N
changed state of remote copy to shared; forward value; update directory

Figure 4: Possible outcomes for a cache-coherent read (worst-case)

Scenario	Conditions	
A	1	a section S_Y^{rw} reaches node X from some predecessor node Y, and
	2	$S_Y^{rw} \sqsupseteq S_X^r$ and $S_Y^{rw} \approx S_X^r$
B	1	No S_Y^w (s.t. $S_Y^w \sqsupseteq S_X^r$) reaches X for any Y
C	1	a section S_Y^w reaches X from some predecessor Y, and
	2	$S_Y^w \sqsupseteq S_X^r$ and $S_Y^w \not\approx S_X^r$

Figure 5: Conditions for predicting read outcomes

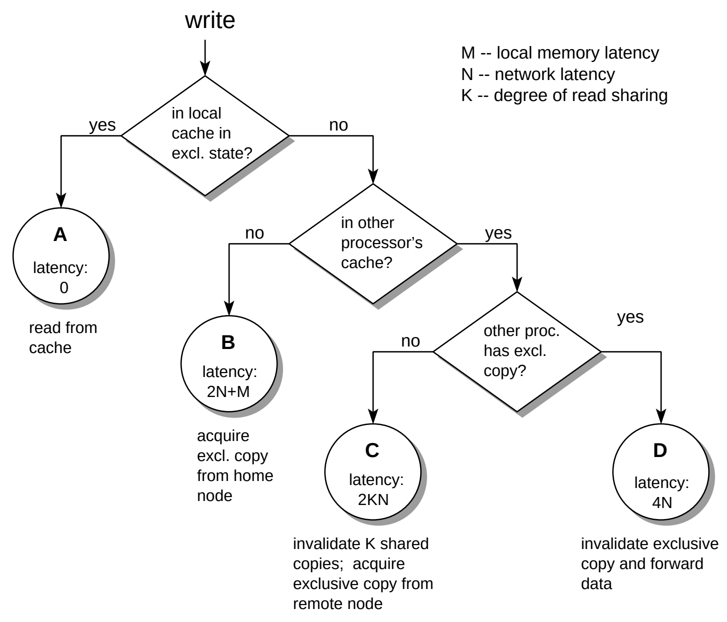

write
M -- local memory latency
N -- network latency
K -- degree of read sharing
in local cache in excl. state?
yes
no
A
latency: 0
read from cache
in other processor's cache?
no
yes
B
latency: 2N+M
acquire excl. copy from home node
other proc. has excl. copy?
no
yes
C
latency: 2KN
invalidate K shared copies; acquire exclusive copy from remote node
D
latency: 4N
invalidate exclusive copy and forward data

Figure 6: Possible outcomes of cache-coherent write (worst-case)

Scenario	Conditions	
A	1	a section S_Y^w reaches X from a predecessor Y, where $S_Y^w \approx S_X^w$ and $S_Y^w \sqsupseteq S_X^w$, and
	2	there is no intervening section S_Y^r such that $S_Y^r \sqsupseteq S_X^w$, and
	3	there is no intervening section S_Z^w such that $S_Z^w \not\approx S_X^w$ but $(S_Z^w \cap S_X^w) \neq \emptyset$
B	1	No section S_Y^{rw} reaches X
C	1	a section S_Y^r reaches X, where $S_Y^r \sqsupseteq S_X^w$, and
	2	there is no intervening section S_Z^w such that $(S_Z^w \cap S_Y^r) \neq \emptyset$
D	1	a section S_Y^w reaches X, where $S_Y^w \sqsupseteq S_X^w$ but $S_Y^w \not\approx S_X^w$, and
	2	there is no intervening section S_Z^r such that $(S_Z^r \cap S_Y^w) \neq \emptyset$

Figure 7: Conditions for predicting write outcomes

we want to predict. In the figure, the term "a section S reaches X from Y" should be interpreted as follows: a region S of an array is accessed in node Y, and this section is still in the cache(s) of the accessing processor(s) when node X is subsequently executed.

2.5.3 Write outcomes

Figure 6 shows the possible outcomes for a write (store) to shared data; the section being written is S_X^w. For writes, the cost of some of the outcomes is no longer fixed: the latency depends on the degree of sharing prior to the execution of the loop. In particular, if a processor tries to write a cache line that is currently read-shared by K processors, the latency will be proportional to K.[2]

Figure 7 gives a set of conditions that are sufficient to predict each of the scenarios for a write operation. As before, all of the numbered conditions for a given outcome must be satisfied.

2.6 Interval analysis

Our data-flow framework is based on *interval* analysis. We refer the reader to previous works for a complete description of the terminology and mechanics of this form of data-flow analysis [2, 8, 9, 12, 15].

Interval analysis proceeds in two steps: an "interval contraction" phase, followed by an "interval expansion" phase. In the contraction phase, intervals are processed from innermost to outermost; an interval is only processed after all the intervals it contains are completed. We solve a set of equations for the nodes in the current interval, and then the nodes in the interval are summarized and collapsed into a single node. The contraction phase proceeds until all intervals have been contracted; the final graph (a DAG) is then analyzed as if it were an interval. In the expansion phase, the process is reversed: summary nodes are expanded into their original intervals and then re-analyzed.

As with the work of Gross and Steenkiste [9] and of Granston and Veidenbaum [8], we present two sets of equations, the first for computing information within an interval, and the second for collapsing the nodes in an interval into a single summary node.

2.7 Data-flow equations

Figure 8 gives the names and definitions for the sets of sections used in our flow analysis. For a given basic block n, initial values of "UREF(n)" and "CREF(n)" are computed by simply inspecting n; these initial sets are used as the inputs to the framework. The remainder of the variables are computed during the flow analysis. In the equations, we parameterize set names according to access type ("r" and "w" superscript notation indicates read and written sections, respectively; an "rw" superscript indicates that the section is either read or written).

During the analysis we maintain the invariant that for every node n, $\mathrm{UREF}^w(n) \cap \mathrm{UREF}^r(n) = \emptyset$; if a region of an array is both read and written within a node, then we place the region in $\mathrm{UREF}^w(n)$ and not in $\mathrm{UREF}^r(n)$.

Figure 9 and 10 show the data-flow equations that we solve to obtain information about coherence activity. The

[2]It should be noted that machines that combine a relaxed memory consistency model with an aggressive write buffer design may be able to hide most write latency.

equations in Figure 9 are computed for each of the nodes within an interval. When processing the interval, we visit the nodes in the interval in reverse postorder; for each node, we compute the "*IN*" sets and then the "*OUT*" sets. All sets are initially set to $\emptyset$.

Note the use of the "$-_{nc}$" operator in equations 4 and 8. Intuitively, $X -_{nc} Y$ subtracts sections in Y from X, but removes only those sections that do *not* conform to the sections in X. In other words, if there is some section $S_1 \in X$ and $S_2 \in Y$ such that $(S_1 \cap S_2) \neq \emptyset$, but $S_1 \approx S_2$, then S_1 will not be removed from X when forming $X -_{nc} Y$. Similarly, the "$-_c$" operator removes only the sections that do conform.

Intuitively, if a section $S \in \mathrm{UIN}^r(n)$, then this means that the cache lines containing S will be in a shared state (see Figure 1) on entry to node n, whereas if $S \in \mathrm{UIN}^w(n)$, the lines containing S will be in a modified state.

The "$\mathrm{UREF}^r(\mathrm{n}) - \mathrm{UIN}^w(\mathrm{n})$" term in equation 3 seems counterintuitive, but it is necessary due to the way the coherence protocol works. If a previously written line is already in the cache in a modified state when a read takes place, the line will remain in the exclusive state (thus a section S already in $\mathrm{UIN}^w(\mathrm{n})$ should not be added to $\mathrm{UOUT}^r(\mathrm{n})$, even if $S \in \mathrm{UREF}^r(\mathrm{n})$).

Figure 10 shows the data-flow equations used to summarize an interval into a single representative node. On the left hand side of these equations, the node S refers to the summary node being created; on the right hand side of the equations, node E refers to the exit node of the interval being summarized. The $\uplus$ operator is defined as "loop translation"; when applied to a section within a given loop, it substitutes in the bounds of the loop for the loop induction variable [3]. For example, in Figure 2, the section accessed on a given iteration of the "`doall` j_1" loop is `a(`j_1`,1:100)`. Applying the operation $\uplus_{j_1}$ to this section will produce the section `a(1:100,1:100)`.

Figures 11 and 12 show how the results of the flow analysis correspond to the scenarios in Figures 4 and 6; they are analogous to Figures 5 and 7, except that the conditions have been rewritten in terms of the sets generated by the data-flow framework.

2.8 Incorporating cache size constraints

The data-flow framework we have described thus far does not take processor cache size into account; in order for the analysis to generate useful information, it must model capacity effects. For example, if the set $\mathrm{UREF}^r(n)$ for a given node n contains a section S^r_X, this tells us that the section in S was accessed some time in the past, but not whether the section is still resident in cache on entry to node n. In this section we summarize the modifications to our framework that incorporate cache size and organization. This work builds on the techniques described in a previous paper [5].

2.8.1 Array section age

First, we introduce the concept of the "age" of an array section with respect to a particular point in the program. We define the age of a given section as the number of capacity misses since the first element of the section was brought into the cache.

During the analysis, we associate age values with each of the sections in our data-flow sets. When an array section is first added to one of the "*OUT*" sets for a node (corresponding to the point where it is first brought

Set name	Definition	Remarks
UREF(n)	the sections accessed within node n that unconditionally downwardly reach the end of node n.	computed for each basic block as part of the initial information.
UIN(n)	the sections accessed by some predecessor of node n that unconditionally downwardly reach the start of node n	computed during data-flow propagation
UOUT(n)	the sections accessed within node n or node n's predecessors that unconditionally downwardly reach the end of node n	computed during data-flow propagation
CREF(n)	the sections accessed in node n that conditionally downwardly reach the end of n (i.e. sections that *may* be accessed in n)	computed for each basic block as part of the initial information.
CIN(n)	the sections accessed by some predecessor of node n that conditionally downwardly reach the start of node n	computed during data-flow propagation
COUT(n)	the sections accessed within node n or node n's predecessors that conditionally downwardly reach the end of node n	computed during data-flow propagation

Figure 8: Data-flow sets

$$\mathrm{UIN}^r(\mathrm{n}) = \bigcap_{p\in PRED(n)} (\ \mathrm{UOUT}^r(\mathrm{p})\ ) \tag{1}$$

$$\mathrm{UIN}^w(\mathrm{n}) = \bigcap_{p\in PRED(n)} (\ \mathrm{UOUT}^w(\mathrm{p})\ ) \tag{2}$$

$$\mathrm{UOUT}^r(\mathrm{n}) = (\ \mathrm{UIN}^r(\mathrm{n}) \cup (\ \mathrm{UREF}^r(\mathrm{n}) -_c \mathrm{UIN}^w(\mathrm{n})\ )\ ) - \mathrm{CREF}^w(\mathrm{n}) \tag{3}$$

$$\mathrm{UOUT}^w(\mathrm{n}) = (\ \mathrm{UIN}^w(\mathrm{n}) \cup \mathrm{UREF}^w(\mathrm{n})\ ) -_{nc} \mathrm{CREF}^{rw}(\mathrm{n}) \tag{4}$$

$$\mathrm{CIN}^r(\mathrm{n}) = \bigcup_{p\in PRED(n)} (\ \mathrm{COUT}^r(\mathrm{p})\ ) \tag{5}$$

$$\mathrm{CIN}^w(\mathrm{n}) = \bigcup_{p\in PRED(n)} (\ \mathrm{COUT}^w(\mathrm{p})\ ) \tag{6}$$

$$\mathrm{COUT}^r(\mathrm{n}) = (\ \mathrm{CIN}^r(\mathrm{n}) \cup (\ \mathrm{CREF}^r(\mathrm{n}) - \mathrm{UIN}^w(\mathrm{n})\ )\ ) - \mathrm{UREF}^w(\mathrm{n}) \tag{7}$$

$$\mathrm{COUT}^w(\mathrm{n}) = (\ \mathrm{CIN}^w(\mathrm{n}) \cup \mathrm{CREF}^w(\mathrm{n})\ ) -_{nc} \mathrm{UREF}^r(\mathrm{n}) \tag{8}$$

Figure 9: Data flow equations computed within an interval

$$\mathrm{UREF}^r(\mathrm{S}) = \biguplus_{loop} (\ \mathrm{UOUT}^r(\mathrm{E})\ ) - \biguplus_{loop} (\ \mathrm{COUT}^w(\mathrm{E})\ ) \tag{9}$$

$$\mathrm{UREF}^w(\mathrm{S}) = \biguplus_{loop} (\ \mathrm{UOUT}^w(\mathrm{E})\ ) -_{nc} \biguplus_{loop} (\ \mathrm{COUT}^{rw}(\mathrm{E})\ ) \tag{10}$$

$$\mathrm{CREF}^r(\mathrm{S}) = \biguplus_{loop} (\ \mathrm{COUT}^r(\mathrm{E})\ ) - \biguplus_{loop} (\ \mathrm{UOUT}^w(\mathrm{E})\ ) \tag{11}$$

$$\mathrm{CREF}^w(\mathrm{S}) = \biguplus_{loop} (\ \mathrm{COUT}^w(\mathrm{E})\ ) -_{nc} \biguplus_{loop} (\ \mathrm{UOUT}^{rw}(\mathrm{E})\ ) \tag{12}$$

Figure 10: Data-flow equations for interval summarization

Scenario	Conditions
A	$\exists\ S_Y^{rw}$ in UIN^w(X) or UIN^r(X) such that $S_Y^{rw} \sqsupseteq S_X^r$ and $S_Y^{rw} \approx S_X^r$
B	No S_Y^{rw} in UIN^r(X) or in UIN^w(X) such that $S_Y^{rw} \sqsupseteq S_X^r$
C	$\exists\ S_Y^w$ in UIN^w(X) such that $S_Y^w \sqsupseteq S_X^r$ but $S_Y^w \not\approx S_X^r$

Figure 11: Read outcomes based on data-flow sets

Scenario	Conditions
A	$\exists\ S_Y^w$ in UIN^w(X) such that $S_Y^w \approx S_X^w$ and $S_Y^w \sqsupseteq S_X^w$
B	No S_Y^{rw} in UIN^r(X) or in UIN^w(X) such that $S_Y^{rw} \sqsupseteq S_X^w$
C	$\exists\ S_Y^r$ in UIN^r(X) such that $S_Y^r \approx S_X^w$ and $S_Y^r \sqsupseteq S_X^r$
D	$\exists\ S_Y^w$ in UIN^w(X) such that $S_Y^w \sqsupseteq S_X^w$ but $S_Y^w \not\approx S_X^w$

Figure 12: Write outcomes based on data-flow sets

into the cache), we assign it an initial age value based on its volume. As the section is propagated to other points in the program, other accesses will start to displace the section from the cache;[3] when this displacement takes place, the age of the section is incremented. Eventually the age of the section reaches a cutoff, at which point we consider the section "dead" (i.e. totally displaced from the cache) and we remove it from the set that contains it.

We conservatively assume that cache conflicts (due to limited associativity) will reduce the amount of reuse that takes place by a fixed factor. We currently estimate the "effective" size of the cache (used in the analysis) by multiplying the actual cache size by $1 - \frac{1}{2^S}$, where S is the set associativity of the cache. More details can be found in [5].

2.8.2 Data-flow framework modifications

The equations in 13 compute the same information as those in Figure 9, except that they incorporate several important changes that take cache size into account.

First, when computing $\mathrm{UIN}^r(n)$ and $\mathrm{CIN}^r(n)$, we use the $\vee$ and $\wedge$ operators in place of $\cup$ and $\cap$. $\vee$ and $\wedge$ still form the intersection and union of their arguments, however when a given section S appears in both arguments, the age value of S in the result set will be the maximum of the ages of S in the arguments.

Second, we introduce a new operator $\odot$, which models the "aging" effects of passing through a given block. Let X be the set of sections that reaches the start of node n, and let Y be the set of sections accessed within n. To form the set $X \odot Y$, we compute the number of capacity misses that will be caused by executing n (by computing the volume of the set $Y - (X \cap Y)$), then "age" each of the sections in X by the resulting miss count, and finally remove any sections whose age values indicate that they have been displaced from the cache.

The *OUT* sets are then computed using the aged ("Δ") versions of the various *IN* sets (i.e. $\mathrm{UIN}^r_\Delta(n)$, $\mathrm{UIN}^w_\Delta(n)$, $\mathrm{CIN}^r_\Delta(n)$, and $\mathrm{CIN}^w_\Delta(n)$).

The equations for interval summarization are the same as those in Figure 10, except that the $\uplus$ operator must take section volume effects into account.

3 Optimizations

In this section we show how our compiler uses the data-flow information to improve prefetching for parallel programs. Section 3.1 outlines our strategy for prefetching of references that incur coherence misses. Section 3.2 discusses additional optimizations for false sharing and widely shared data.

3.1 Exploiting coherence miss information

Our framework provides information about the sets of references likely to cause coherence misses, but in order to derive an effective prefetching strategy, our compiler combines this knowledge with dependence information and with knowledge about the iteration space of the loop nest being optimized.

[3]For the purpose of age calculations, we ignore coherence misses that *only* change the state of a cached line from shared to modified. Only misses that cause evictions are counted when updating age values.

$$\mathrm{UIN}^r(\mathrm{n}) = \bigwedge_{p \in PRED(n)} (\ \mathrm{UOUT}^r(\mathrm{p})\ ) \tag{13}$$

$$\mathrm{UIN}^r_\Delta(\mathrm{n}) = \mathrm{UIN}^r(\mathrm{n}) \odot (\ \mathrm{UREF}^r(\mathrm{n}) \cup \mathrm{UREF}^w(\mathrm{n})\ ) \tag{14}$$

$$\mathrm{UIN}^w(\mathrm{n}) = \bigwedge_{p \in PRED(n)} (\ \mathrm{UOUT}^w(\mathrm{p})\ ) \tag{15}$$

$$\mathrm{UIN}^w_\Delta(\mathrm{n}) = \mathrm{UIN}^w(\mathrm{n}) \odot (\ \mathrm{UREF}^r(\mathrm{n}) \cup \mathrm{UREF}^w(\mathrm{n})\ ) \tag{16}$$

$$\mathrm{UOUT}^r(\mathrm{n}) = (\ \mathrm{UIN}^r_\Delta(\mathrm{n}) \cup (\ \mathrm{UREF}^r(\mathrm{n}) - \mathrm{UIN}^w(\mathrm{n})\ )\ ) - \mathrm{CREF}^w(\mathrm{n}) \tag{17}$$

$$\mathrm{UOUT}^w(\mathrm{n}) = (\ \mathrm{UIN}^w_\Delta(\mathrm{n}) \cup \mathrm{UREF}^w(\mathrm{n})\ ) -_{nc} \mathrm{CREF}^{rw}(\mathrm{n}) \tag{18}$$

$$\mathrm{CIN}^r(\mathrm{n}) = \bigvee_{p \in PRED(n)} (\ \mathrm{COUT}^r(\mathrm{p})\ ) \tag{19}$$

$$\mathrm{CIN}^r_\Delta(\mathrm{n}) = \mathrm{CIN}^r(\mathrm{n}) \odot (\ \mathrm{CREF}^r(\mathrm{n}) \cup \mathrm{CREF}^w(\mathrm{n})\ ) \tag{20}$$

$$\mathrm{CIN}^w(\mathrm{n}) = \bigvee_{p \in PRED(n)} (\ \mathrm{COUT}^w(\mathrm{p})\ ) \tag{21}$$

$$\mathrm{CIN}^w_\Delta(\mathrm{n}) = \mathrm{CIN}^w(\mathrm{n}) \odot (\ \mathrm{CREF}^r(\mathrm{n}) \cup \mathrm{CREF}^w(\mathrm{n})\ ) \tag{22}$$

$$\mathrm{COUT}^r(\mathrm{n}) = (\ \mathrm{CIN}^r_\Delta(\mathrm{n}) \cup \mathrm{CREF}^r(\mathrm{n})\ ) - \mathrm{UREF}^w(\mathrm{n}) \tag{23}$$

$$\mathrm{COUT}^w(\mathrm{n}) = (\ \mathrm{CIN}^w_\Delta(\mathrm{n}) \cup \mathrm{CREF}^w(\mathrm{n})\ ) -_{nc} \mathrm{UREF}^{rw}(\mathrm{n}) \tag{24}$$

Figure 13: Data flow equations computed within an interval (with cache constraints)

```
doall j = 2, 99
   do i = 1, 100
      a(i,j) = ...
      b(i,j) = ...
   enddo
enddoall
doall k = 2, 99
   do i = 1, 100
      ... = a(i,k) + b(k,i) ...
      ... = a(i,k+1) + a(i,k-1)
   enddo
enddoall
```

Figure 14: Example with coherence misses

```
doall j = 1, 100
   do i = 1, 100
      ... = x(i,1)
   enddo
enddoall
doall k = 1, 100
   do i = 1, 100
      x(i,k) = ...
   enddo
enddoall
```

Figure 15: Example with many-processor read-sharing

```
doall j = 2, 99
   do i = 1, 100
      a(i,j) = ...
   enddo
enddoall
doall i = 1, 100
   a(i,1) = ...
   a(i,100) = ...
enddoall
```

Figure 16: Example with false sharing

Consider the second loop nest in Figure 14. The data-flow framework will predict coherence activity for 3 of the 4 references in this nest: `a(i,k+1)`, `a(i,k-1)`, and `b(k,i)`. Our compiler then further classifies the references into two sets: those that incur coherence misses on every loop iteration (*unconditional* coherence misses) and those that incur coherence misses only on some small subset of iterations (*conditional* coherence misses).

3.1.1 Unconditional coherence misses

Our main mechanism for handling the long latencies of coherence misses is to issue prefetches for the references in question farther in advance than for other data. We do this by increasing the *prefetching distance* for the references. The prefetching distance for a given loop nest is the number of iterations in advance that data items are fetched; it is dependent on the number of instructions in the loop and on the expected miss latency.

Since most loop nests contain a mixture of cache hits, coherence misses, and capacity misses, we use a double prefetch pipeline, in which the long-latency references are prefetched farther in advance that the rest of the data.

3.1.2 Conditional coherence misses

Consider the references `a(i,k+1)` and `a(i,k-1)` in Figure 14. In the `doall k` loop, if processor P is assigned iterations K_p through K_p+b, then it will read the set of elements `a(1:100,`K_p-1`:`K_p+b+1`)`. Of these elements, `a(1:100,`K_p`:`K_p+b`)` were written previously on the same processor (and thus will not cause coherence misses), whereas the elements `a(1:100,`K_p-1`)` and `a(1:100,`K_p+b+1`)` were written on a neighboring processor and will probably result in coherence misses. This type of "nearest neighbor" communication occurs quite often in matrix-based scientific programs.

In such situations, we don't want to treat all the references as long-latency misses, since coherence activity only takes place on a subset of the iterations (and since prefetches that arrive too early may displace useful data). Instead, our compiler applies loop peeling to isolate the coherence-causing loop iterations.

We identify peeling opportunities as follows. Given a section S^r corresponding to read reference r in CFG node N, peeling is applicable if there is a section $S^w_X \in \mathrm{UIN}^w(\mathrm{n})$ such that $S^w_X \not\approx S^r$ where

1. $\textsc{pmap}(S^r) = \textsc{pmap}(S^w_X)$, and
2. $\textsc{pmap}(S^r) = \langle N, F \rangle$ where $N > 0$, and
3. the bounds on dimension N in $\textsc{region}(S_r)$ are equal to the the bounds on dimension N in $\textsc{region}(S^w_X)$ shifted by a small constant c.

If any references meet these criteria, then the compiler peels the first and/or last c iterations of the parallel loop, effectively isolating the coherence misses within the peel loop. We use a similar process for write references.

3.2 Additional optimizations

3.2.1 Many-processor read sharing

As previously discussed, the worst-case latency for write misses takes place when a process needs to write a line that is being read-shared by K different processors (requiring the writing processor to wait for K invalidation messages and acknowledgements). Figure 15 shows an example.

Consider the `doall k` loop in this example. Suppose that processor 1 is assigned iteration 1 of the loop. Each time it writes a cache line in the section `x(1:100,1)`, the hardware must invalidate all K copies of the line, resulting in very long latency for the write. Depending on the circumstances, an advanced write buffer might be able to hide the latency for such a write. However due to the nature of the invalidation protocol, there would still be considerable additional network traffic.

One solution would be to determine the parallel nest where the line is multiply shared and apply a local invalidation operation following the last read of the data. This would cut the total network traffic in half (provided that the invalidations did not require acknowledgements) and would spread the traffic out over a longer period of time.

The data-flow analysis required to drive such an optimization is very similar to that presented in this paper, but requires backward instead of forward propagation. We are unable to present the details of the analysis here, due to lack of space.

3.2.2 Exclusive-mode prefetching

When a processor issues a prefetch for a line that is about to be written, it makes sense to fetch the line in an exclusive (as opposed to shared) state (see Figure 1). Selecting the type of prefetch based solely on the type of operation (load or store) is not always the best strategy, however. If a location is going to be written shortly after being read, it pays to issue an exclusive-mode prefetch for the initial read, since exclusive ownership will be needed for the subsequent write.

Previous researchers have studied this problem; Mowry, Lam, and Gupta developed a scheme for selecting the prefetch mode using an algorithm in which references in a loop nest are divided into *equivalence classes* based on a form of vector-space reuse analysis [20].

The information provided by our analysis framework can be used to improve on previous methods. By comparing the array sections for a given pair of read/write references, for example, we can issue exclusive-mode prefetches only when we know that they will be profitable. Due to space constraints, we are unable to present the complete details of this process.

3.2.3 False sharing

The parallel loops in our model are synchronization-free, thus we can be guaranteed that if one processor writes an array element within a parallel loop, no other processor will read that element. With non-unit cache line sizes, however, there may be false sharing. For references to false-shared data, prefetching may actually make matters worse, not better. Figure 16 illustrates this situation.

In the first parallel loop ("`doall j`") each processor writes a block of columns of the array. In the second parallel loop, however, each processor is assigned a chunk of the first and last columns in the array. This will result in contention for the cache lines that are on the boundary between processors– if a line is prefetched too far in advance, it may very well be invalidated before it can be used, since another processor may be trying to write it.

Our framework provides a simple way recognize such situations: we examine all of the sections $S \in \mathrm{UREF}^w(\mathrm{N})$ (where N is a node contained in an inner loop), and suppress prefetching for references whose sections have mapping functions of the form $\mathrm{PMAP}(\mathrm{S}) = \langle 1, \ldots \rangle$. While this will not eliminate the false sharing, it does get rid of the additional network traffic and thrashing caused by the prefetching.

4 Experiments

4.1 Implementation

We have implemented the data flow framework described previously in our research compiler. Our compiler uses *Data Access Descriptors* to represent array sections [3]. In addition to the multiprocessor enhancements, the compiler fully supports uniprocessor software prefetching, including reuse analysis, loop peeling, and loop unrolling [22]. In order to avoid having to implement an instruction-level optimizing "back end" that supports prefetch instructions, we instead create simulated prefetches at the Fortran level: for each array "A" within the application program, we introduce a *shadow* array "SA" of the same type, length, etc. The compiler then issues prefetches using stores to the shadow array; a store to location N in the shadow array is caught by our simulator and treated as a prefetch of location N in the actual array.

4.2 Experimental strategy

For the full version of this paper, we intend to provide an experimental study that will measure how well our techniques work in practice, using a set of parallelized Fortran programs taken from the SPEC benchmark suite [25]. We compare the performance of these programs when run without prefetching, with prefetching, and with the enhanced prefetching (i.e. adding the optimizations enabled by our analysis framework).

We use an execution driven parallel architecture simulator derived from RPPT [7]. Our simulated parallel machine consists of a series of clusters of RISC processors connected by a high speed interconnection network; the cache, memory, and network subsystems are modeled in detail. The data collected by the simulator include overall execution time, cache miss rates, average miss latency, total prefetches issued, useless prefetches, and late prefetches, among others.

5 Related work

Mowry, Lam, and Gupta's work provides a study of compiler-directed software prefetching for uniprocessors and for DSM-style shared-memory multiprocessors [20, 21, 22]. The double pipelining we use for prefetching coherence misses is to their strategy for prefetching references that use indirection arrays. Mowry's compiler analysis is limited to individual loop nests, however, and it stops short of identifying specific sets of references that are likely to cause coherence misses, which prevents the compiler from tailoring the prefetching strategy for these references.

Our interval analysis framework was inspired by that of Granston and Veidenbaum [8], which in turn was based on the framework of Gross and Steenkiste [9]. It differs from the work of Granston and Veidenbaum in several important respects. Their work was geared towards a multiprocessor without hardware support for cache coherence, whereas we specifically incorporate support to hardware cache coherence in our data-flow framework. Their techniques were designed for software-controlled local memories and not caches; they have no mechanism for taking into account cache replacement effects.

The data-flow analysis we use is also related to the techniques developed by Gupta, Schonberg, and Srinivasan for optimizing communication placement for programs running on distributed-memory multiprocessors [11]. Communications placement is a fairly different problem, however and analyzes the flow of *values* within the program, whereas our analysis focuses on the use of *locations*. Their framework also does not handle resource constraints (such as cache size) or architectural features (such as hardware coherence protocols).

6 Conclusions

Applying software prefetching on DSM multiprocessors is more difficult than on uniprocessor machines. Cache miss latencies vary tremendously, and artifacts of the cache coherence protocol can result in miss latencies up to many hundreds of cycles. Such factors cause problems for existing prefetching techniques, which were originally developed for machines with uniform miss latencies.

In this paper we have presented a new tool for addressing these challenges, in the form of a novel data-flow framework that provides accurate predictions of coherence activity within a parallel program. This framework is the first to incorporate knowledge of both the program's memory access patterns and the important characteristics of the cache subsystem on the target machine. We demonstrate methods for using the resulting data-flow information to improve the efficiency of software prefetching, including better prefetching for coherence misses, and handling of false-shared and heavily shared data.

Our methods are designed to work in concert with existing prefetching techniques; we use the same loop transformation tools (i.e. loop pipelining, peeling, and unrolling), but our compiler achieves better prefetch scheduling by identifying and handling long-latency cache misses. We have implemented our ideas in our research compiler, and we are in the process of conducting a simulation study to measure the effectiveness of our techniques in practice.

References

[1] A. Agarwal, R. Bianchini, D. Chiaken, K. Johnson, D. Kratz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife machine: Architecture and performance. In *Proceedings of the 22th International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, June 1995.

[2] A. V. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, second edition, 1986.

[3] V. Balasundaram. A mechanism for keeping useful internal information in parallel programming tools: The data access descriptor. *Journal of Parallel and Distributed Computing*, 9(2):154–170, June 1990.

[4] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, CA, June 1986.

[5] K. Cooper, K. Kennedy, and N. McIntosh. Cross-loop reuse analysis and its application to cache optimizations. In *Proceedings of the Ninth Workshop on Languages and Compilers for Parallel Computing*, San Jose, CA, August 1996. Springer-Verlag (Available on-line prior to publication at ftp://cs.rice.edu/public/mcintosh/lcpc96paper.ps.gz).

[6] CONVEX Computer Corporation. *Exemplar Architecture.* CONVEX Press, Richardson, Texas, first edition, 1993.

[7] Sandhya Dwarkadas, John R. Jump, and James B. Sinclair. Execution-driven simulation of multiprocessors: Address and timing analysis. In *Journal of Transactions on Modeling and Computer Simulation*, October 1994.

[8] E. Granston and A. Veidenbaum. Detecting redundant accesses to array data. In *Proceedings of Supercomputing '91*, Albuquerque, NM, November 1991.

[9] T. Gross and P. Steenkiste. Structured dataflow analysis for arrays and its use in an optimizing compiler. *Software—Practice and Experience*, 20(2):133–155, February 1990.

[10] M. Gupta and E. Schonberg. A framework for exploiting data availability to optimize communication. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[11] M. Gupta, E. Schonberg, and H. Srinivasan. A unified data-flow framework for optimizing communication. In *Proceedings of the Seventh Workshop on Languages and Compilers for Parallel Computing*, Ithaca, NY, August 1994.

[12] R. v. Hanxleden. *Compiler Support for Machine-Independent Parallelization of Irregular Problems.* PhD thesis, Dept. of Computer Science, Rice University, December 1994.

[13] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[14] High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming*, 2(1-2):1–170, 1993.

[15] K. Kennedy. A survey of data flow analysis techniques. In S. Muchnick and N. Jones, editors, *Program Flow Analysis*, pages 5–54. Prentice-Hall, 1981.

[16] Kuck & Associates, Inc. *KAP User's Guide*. Champaign, IL 61820, 1988.

[17] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simon, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH multiprocessor. In *Proceedings of the 21th International Symposium on Computer Architecture*, April 1994.

[18] Z. Li and P. Yew. Efficient interprocedural analysis for program restructuring for parallel programs. In *Proceedings of the ACM SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages, and Systems (PPEALS)*, New Haven, CT, July 1988.

[19] E. Markatos and T. LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 5(4):379–400, April 1994.

[20] T. Mowry. *Tolerating Latency Through Software Controlled Data Prefetching*. PhD thesis, Dept. of Computer Science, Stanford University, March 1994.

[21] T. Mowry and A. Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 12(2):87–106, June 1991.

[22] T. Mowry, M. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 62–73, Boston, MA, October 1992.

[23] M. Papamarcos and J. Patel. A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories. In *Proceedings of the 11th International Symposium on Computer Architecture*, pages 340–347. IEEE, 1984.

[24] D. Tullsen and S. Eggers. Limitations of cache prefetching on a bus-based multiprocessor. In *Proceedings of the 20th International Symposium on Computer Architecture*, San Diego, CA, May 1993.

[25] J. Uniejewski. SPEC Benchmark Suite: Designed for today's advanced systems. SPEC Newsletter Volume 1, Issue 1, SPEC, Fall 1989.