Software Assistance for Directory-Based Caches

Zhiyuan Li
Department of Computer Science
University of Minnesota at Minneapolis-St. Paul
200 Union St. SE, Minneapolis 55455

Abstract

We investigate the benefit of combining directory-based schemes with software schemes as a method for maintaining cache coherence on multiprocessors. The main idea is to maintain the directory hardware while allowing eligible write references to bypass the invalidation process. Static analysis is applied to parallel programs in order to mark those eligible write references. The sample results suggest that such reference marking can reduce invalidation requests significantly when it is combined with locality-preserving task scheduling and when the array subscripts are not complex.

1 Introduction

Private caches have been an attractive mechanism for reducing data access latency on shared memory multiprocessors. However, efficient maintenance of *cache coherence* remains a great challenge. Different processors often fetch the same data item from the shared memory and store a copy in each of their own caches. When one processor updates the data, other cached copies become invalid. Solving the cache coherence problem ensures that no processor ever uses an invalid data copy.

For multiprocessors that are connected with multistage interconnection networks (MIN), solutions for the cache coherence problem have fallen into two categories, namely, compiler-assisted software solutions (see [8] and others) and directory-based solutions (see [3] and others). Each category is known to have limitations which cause unnecessary network traffic and data access delay. This paper investigates the benefit of combining the two approaches. The main idea is to retain the original directory hardware while allowing eligible write references to bypass the invalidation process. Static analysis is applied to parallel programs in order to mark those eligible write references. Such reference marking is most effective when combined with program transformations and with task scheduling that preserves data locality.

1.1 Background

Directory-based solutions associate additional memory bits with each shared data item and use those bits to indicate, conservatively, which processors may have a cached copy of the data. When a processor changes its cached copy of the data, it informs the memory directory. The directory then propagates this change to the involved processors. Depending on implementation, the memory directory may either send the updated copy to the processors to replace the invalid copies (the *write-update* scheme), or may simply send a signal that directs the processors to invalidate their copies (the *write-invalidate* scheme). In this work, we mainly consider the write-invalidate scheme, but the essence of our work applies to the write-update scheme as well.

In a typical invalidation process [3], when a processor writes to a variable *for the first time* in the current task, it sends out an invalidation request to the memory directory. If there are no copies of that variable in other caches, then the memory directory simply acknowledges the receipt of the invalidation signal to the processor. However, if copies exist in other caches, then the memory directory must forward the invalidation request to each processor which has a copy. Upon receipt of the request, each processor resets the *valid bit* on its cache copy and sends acknowledgement to the memory directory. After receiving acknowledgement from all the informed processors, the memory directory sends acknowledgement back to the processor which initiated the invalidation request. The later updates to the same data within the same task do not need to generate further invalidation. The invalidation effort produces extra network traffic which increases the average latency of the network. It is clear that the more cache copies need to be invalidated, the heavier the network traffic will be. When there are no outstanding cache copies to be invalidated, invalidation traffic involves only the requesting processor and the memory directory. Our simulation study [17] suggests that invalidation traffic can be quite significant, relative to data traffic. The benchmarks in the Perfect Suite [2] that we studied generate around 3 bytes of network traffic per variable reference, including the traffic for global data movement (with a word size of 4 bytes) and the traffic due to invalidation requests and acknowledgement. On the other hand, the miss ratio has been below 10% for those programs. Therefore, the invalidation overhead is clearly high. This traffic rate would translate to 800 Mbytes per second on the network of a 16 processor system in which each processor produces a variable reference every two cycles and the clock frequency is 33 MHz. Thus, the demand on the network is quite high for these benchmarks.

It would be ideal if each processor could be aware, without needing to communicate with other processors, of which data within its own cache has become invalid. The software solutions insert cache invalidation instructions which, when executed, either invalidate the whole data cache of the executing processor (termed *indiscriminate invalidation* in the literature [6]) or invalidate a selected cache block. Such *local* invalidating action normally occurs at the beginning of a new task which is created at the run time of a parallel program.

Indiscriminate invalidation tends to grossly invalidate data which may actually be valid. On the other hand, selective invalidation requires executing as many invalidating instructions as the number of cache blocks to be invalidated, which is quite a high overhead. Recently, a couple of remedies for indiscriminate invalidation have been proposed to reduce the degree of over-invalidation [6, 20]. Such remedies, termed *fast selective invalidation*, can be quite effective if the compiler is able to analyze the program's data flow with high precision. Unfortunately, a precise static analysis is known to be difficult, due to such complications in data access patterns as subscripted arrays and pointers. The inability of the compilers to predict run time execution paths also poses a major obstacle. Finally, dynamic task scheduling brings further difficulties to the static analysis. Some of these problems may be reduced by techniques such as *version comparison* [7] or *time-stamp comparison* [15]. However, it is still unclear whether the extra execution time introduced by such techniques would be practically acceptable. Hence, we do not consider these types of techniques here. Overall, previous studies indicate that the over-invalidation by the software schemes increases both the miss ratio and the network traffic, compared to a mechanism that has complete run-time information [14, 5, 19]

The software solutions are attractive because they do not generate invalidation traffic, but they risk a high miss-ratio and heavy overall traffic due to over-invalidation. This dilemma motivates our effort to find a combination of software solutions and directory-based solutions. We pursue this by applying compiler analysis similar to that in software solutions, to find write references that need not request invalidation.

1.2 Overview

The invalidation traffic is caused by the need to maintain cache coherence. A good static scheduling algorithm can assign tasks to the processors in such a way that the number of cache incoherence events, and thus the number of invalidation requests, can be significantly reduced. This is typically done by letting a processor execute consecutive tasks that reference the same data. Dynamic task scheduling, in contrast, cannot guarantee what tasks may be assigned to the same processor, and hence, it may result in more cache incoherence events than static scheduling. Nonetheless, dynamic scheduling may achieve a more balanced work load than static scheduling.

The work presented in this paper assumes that the compiler has made a choice between using static scheduling or dynamic scheduling for a given parallel construct. If the compiler chooses static scheduling, then we further assume that it has assigned the tasks in the parallel construct to the processors. Given either scheduling policy, our work is responsible for identifying and marking the write references that do not need to initiate invalidation. By not sending unnecessary invalidation requests, our work reduces invalidation traffic further,

In next section, we describe the program model used in this work upon which the compiler does the reference marking. In Section 3, we present our cache coherence mechanism. Subsequently, we present a scheme to mark read and write references based on our mechanism. We give sample results in Section 4 and make concluding remarks in Section 5.

2 The program model

We use the *fork-join* model to represent parallel programs. Under this model, a program contains *sequential regions* (to be defined below) which are executed by a dedicated processor, say p_0, and *parallel regions* which are executed by every processor. Here we focus on DOALL loops as the parallel regions. However, we consider our method applicable to other types of parallel regions, as well as other shared-memory programming models. A DOALL loop is in essence a DO loop (as in Fortran) or a `for` loop (as in C) whose iterations can be executed independently by different processors. A DOALL loop does not terminate until all processors have finished their assigned work. We assume that the compiler has annotated each DOALL loop with a directive that specifies the scheduling policy and, in the case of static scheduling, which processor executes which loop iterations. (A related paper describes a programming model that provides such a precise specification [13].)

The sequential regions in our fork-join model include all the code segments outside the DOALL loops, including the sequential DO loop headers. A parallel region consists of tasks derived from the DOALL loop iterations. Before executing the tasks in a parallel region, all processors except p_0 must wait for the **fork** synchronization signal sent by p_0. After executing the tasks in a parallel region, all processors synchronize at the **join** point, after which p_0 continues to execute the code in the next sequential region. Figure 1 shows an example of fork-join programs.

```
jmax = ...
nr = ...
nits = 100 / nr
DO kits = 1, nits
DO n = 1, nr
DO j = 2, jmax - 1
  uu(1) = ...                        (s1)
DO l = 2, jmax - 1
  uu(l) = uu(l - 1) + ...            (s2)
ENDDO
DOALL l = 2, jmax - 1                (s3)
 ... = uu(l - 1)                     (s4)
psi(j,l) = -psi(j,l) - uu(l - 1)     (s5)
 ...
ENDDOALL
ENDDO ENDDO ENDDO
```

Figure 1: An example of fork-join programs

3 Reference Marking

In this section, we briefly discuss the cache coherence mechanism used in this work and then focus on the compiler analysis for identifying write references that do not need to request invalidation. In order to do so, the read references need to be examined as well.

We slightly modify the mechanism in a conventional directory scheme [3] by distinguishing two types of read operations (*m-read* and *c-read*) and three types of write operations (*n-write*, *t-write*, and *i-write*). The compiler marks on each reference to indicate which operation should be used when generating the machine code.

An *m-read* always fetches the data from the shared memory and loads it into the cache, regardless of whether a copy of the data exists in the cache prior to the reference. A *c-read*, on the other hand, always attempts to read the cache first. An m-read is used when the compiler can guarantee that the read always takes a value written by other processors. In other cases, c-reads are issued. Certain software solutions [6] also use these two different types of reads. However, whenever it is unclear whether a read takes a value written by other processors, the software solutions let the read bypass the cache. Our scheme, in contrast, marks such a read as a c-read and resolves the potential incoherence by marking the writes by other processors as i-writes. Thus, unlike software solutions, our method never results in a higher miss-ratio than the conventional hardware invalidation scheme [3].

An *n-write* writes into the private cache and performs no coherence actions. It is used when the compiler can guarantee that no other processor will use the values in the updated block. A *t-write* writes into both the private cache and the shared memory. It is used when the compiler determines that the updated value will be used by some other processors and those processors will voluntarily fetch the value from the memory using m-reads. An *i-write* is used when the updated value *may* be used by other processors using c-reads. An i-write acts the same way as a write to a shared value in a conventional directory scheme [3]: A special bit on each cache block indicates whether an invalidation request has been sent out for that cache block. If not, then the i-write triggers an invalidation request to the directory. The updated value is not written into the memory until a cache replacement or a c-read by another processor forces write-back. Details of the hardware implementation of these read/write operations are provided in a subsequent paper [16].

In this work, we assume the cache block size of one word. It is known that a cache block size greater than a single word may or may not bring better performance than the size of one word, due to the difficult tradeoff between spatial locality and the negative effect of false sharing [9, 11]. If the cache block size is greater than one word, then the compiler should mark i-writes conservatively: all write references to the same cache block must be marked as i-writes if any of these should be marked as an i-write. This may increase the number of i-writes due to false sharing, but our scheme should suffer from false sharing no more than a conventional invalidation scheme. Our scheme can benefit from any optimization technique that reduces the effect of false sharing [4, 23].

3.1 A marking example

Before giving a formal presentation of the marking algorithm, we first illustrate the ideas through the program segment in Figure 1. In that segment, the scalars *nr*, *nits* and *jmax* are written only once, by processor p_0. Therefore, all the reads of these variables can be marked as c-reads, and the writes to these variables are marked as t-writes. The read of array *uu* at statement s2 is marked as a c-read because it reads the value written by the same processor, p_0. The read of *uu* at statement s4, in contrast, should be marked as an m-read, because it is the initial read of the value in a new, parallel task and because the value was written by p_0. The next read of *uu* at statement s5, however, is marked as a c-read, because any invalid copy of the *uu* elements should have already been updated by the preceding read. As a result, the writes of *uu* at statements s1 and s2 can be marked as t-writes which write to the memory but do not initiate invalidation.

The marking of the read of *psi* at statement s5 depends on scheduling policy. With static scheduling, the compiler can make each processor execute the same DOALL loop iterations whenever the DOALL loop is executed. As a result, each processor always reads and writes the same *uu* elements, even within different tasks. Hence the writes are marked as n-writes and the reads are c-reads. With dynamic scheduling, in contrast, the assignment of DOALL loop iterations may not be the same each time the DOALL loop is executed. Hence, either the read of *psi* should be marked as an m-read or the write of *psi* should be marked as an i-write. As mentioned in the beginning, our conservative choice is to mark the write of *psi* as an i-write and to mark the read as a c-read. The marking of loop indices requires special, but simple, treatment [13]. We do not address this trivial matter here.

3.2 The marking scheme

The compiler partitions a fork-join program into sequential regions and parallel regions. The program control may transfer between sequential regions, bypassing parallel regions. To facilitate the reference marking, the compiler first construct a *task control flow graph* in which each node represents a *task block* which is defined below.

Definition A transfer statement is called a *jump point* if it may transfer the execution from one region to another. The target of the transfer is called a *jump target.* A sequential or parallel region is partitioned by jump points and jump targets into *task blocks.* If a region has neither jump points nor jump targets, then the region itself is a task block. □

Each task block, B, is annotated with the following information which is collected by the compiler: (1) the **maybe-mod** set, $mod(B)$, which is the set of variables that may be modified in B. (2) the **maybe-use** set, $use(B)$, which is the set of variables that may be read in B. (3) the **last-write** set, which is a set of writes in B that may reach the bottom of B without being overwritten. (For example, the writes of *uu* at s1 and s2 in Figure 1 are last-writes in their task blocks, respectively. For convenience, we collect the last-writes to the same variable into a *last-write group*;) (4) the **kill** set, $kill(B)$, which is the set of variables that must be modified in B; and (5) the **definitely-exposed-read** set and the **maybe-exposed-read** set which are defined below.

Definition A read of variable v in a task block B is *exposed* in B if and only if there exists an acyclic control path from the beginning of B to the read of v and the path does not contain any writes to v or any other read of v. The read is *definitely exposed* in B if and only if none of the acyclic control paths from the beginning of B to the read of v could contain any write to v or any other read of v. An exposed, but not definitely exposed, read is *maybe-exposed.* □

For example, the read of *psi* at statement s5 in Figure 1 is a definitely exposed read in its task block.

The computation of the maybe-mod set, the maybe-use set, the kill set, the maybe-exposed-read set, and the last-write set is a familiar matter when only scalars are involved. When arrays are involved, unfortunately, no algorithms are known to be able to

Algorithm 1

Input: 1) The control flow graph G for task block B; 2) The set of exposed reads of each node n in G, denoted by $er(n)$.
Output: $DE(B)$, the set of definitely exposed reads in B.
Steps:
1. For each basic block, n, in G, $de(n) := \emptyset$;
/* $de(n)$ represents the set of reads in B that are definitely exposed in any path from n to the end of B */
2. While $de(n)$ is changed for some n, do
For every n in G, do
$de(n) := er(n) \bigcup (\bigcap_{m \in succ(n)} de(m) -$ {reads of var $\in mod(m)$ or var $\in use(m)$}).
3. $DE(B) := de(start)$, where $start$ is the start basic block of B.

Figure 2: The algorithm for computing the definitely exposed reads

compute the kill set and the last-write set precisely for all cases. For simple cases such as the one in Figure 1, however, these sets can be analyzed precisely (see [10, 12] and others). If these sets cannot be analyzed precisely, then reaching definitions (see below) involving certain last-writes and certain exposed reads will become obscure. In such cases, the involved last-writes should be conservatively marked as i-writes and the exposed reads marked as c-reads, which, in effect, leaves the invalidation hardware to resolve potential cache incoherence.

Given exposed reads in the basic blocks [1], Algorithm 1 in Figure 2 determines the definitely exposed reads of a given task block. Note that any exposed read in a basic block is definitely exposed in that basic block.

Given the maybe-mod sets and the kill sets, the compiler uses the *reaching definition* algorithm [1] to decide whether a last-write group of one task block *reaches* the top of another task block. Three different reaching relations are possible: (1) reaching from a sequential block (i.e. any task block executed by p_0 only) to another sequential block; (2) from a sequential block to a parallel block; and (3) from one parallel block to another parallel block. Only (2) and (3) may introduce cache incoherence. (As a matter of fact, the effect of item (3) depends on the choice of task scheduling.) A basic reference marking scheme is now formulated as follows.

Table 1: Sample Programs for Reference Marking

Name	Problem	No. of writes
AMOEBA	multi-dim. minimization	120k
BALANC	balance a matrix	50k
COVSRT	sort a covariance matrix	15k
GAUSSJ	Gauss-Jordan	272k
SPLIE2	1-D spline for 2-D function	90k

The Basic Marking Scheme

1. For each task block $B1$ and for each last-write group, say for variable v, in $B1$, check to see if the last-write group reaches another task block $B2$ ($B2$ may equal $B1$) and if $B2$ has exposed reads of v. If so, do 1.1 – 1.3.

1.1 If $B1$ and $B2$ will definitely be executed by different processors and the reads of v in $B2$ are definitely exposed, then those reads are marked as m-reads, and the reaching last-writes are marked as t-writes..

1.2. If the reads of v in $B2$ are maybe-exposed, or $B1$ and $B2$ may or may not be executed by the same processor, then the reads are marked as c-reads, and the reaching last-writes are marked as i-writes.

2. If an i-write is the unique write to a variable and will be issued only once, then change the i-write to a t-write (because the variable never causes incoherence).

3. Mark all the unmarked writes as n-writes.

4. Mark all the unmarked, i.e. unexposed, reads as c-reads. □

A related paper [13] discusses a few simple but useful methods to improve the basic scheme, which include the following: (1) When p_0 participates in executing a DOALL loop, mark the writes by p_0 as n-writes if they reach only the reads in sequential regions (which will be executed by p_0); (2) Using *pre-loading* and *loop peeling* to avoid redundant m-reads (caused by repeated execution of the same statement which contains an m-read).

4 Sample Results

We have chosen five programs from the *Numerical Recipes* [22] with which to examine the effect of our scheme. Table 1 lists the program names, the problems they solve, and the total number of writes at run time. Each program is chosen to meet two criteria. First, it should perform a reasonablly large amount of computation and have a reasonable degree of DOALL parallelism. Second, the code should not be too long or too complicated for our semi-automatic experiment (described below).

Table 2: Distribution of Different Marking (S)

Name	%mr	%cr	%iw	%tw	%nw
AMOEBA	12.5	71.2	~0	4.9	11.3
BALANCE	0.9	82.3	~0	0.9	15.9
SPLIE2	3.8	79.5	~0	1.2	15.5

Table 3: Distribution of Different Marking (D)

Name	%mr	%cr	%iw	%tw	%nw
AMOEBA	3.9	79.8	4.1	1.3	10.7
BALANCE	0.9	82.3	0.04	0.8	15.9
SPLIE2	3.8	79.5	0.1	1.1	15.5

The Fortran programs are first parallelized by using the Parafrase 2 translator [21], and they are instrumented with statements which generate reference traces [18]. The instrumented programs are then executed to produce the traces. We use a simulator for a multiprocessor with private caches which takes the traces as input and reports the results [16]. We arbitrarily choose the number of processors to be 16. We locate the program segments which generate over 95% of the total write references. These segments are small and generally simple, allowing us to mark the references in the program by hand. We are not able to fully automate the experiment because of the lack of array dataflow analysis in the translator we use.

The results are good for three programs but are poor for two other programs, namely GAUSSJ and COVSRT, in which the most frequent references have complex array subscripts that force a conservative marking: almost all writes must be marked as i-writes and almost all reads must be marked as c-writes. The results of the other three programs are shown in the following tables. Table 2 shows the distribution of different types of reads/writes, as marked by our scheme, that occur at run time when static scheduling is used. The percentages are calculated over all read/write references. The title *mr* is a short name for m-read, and so on. Table 3 shows the same kind of data for dynamic scheduling. Table 4 shows the network traffic (in bytes per reference) for program AMOEBA. Each row shows the data for static scheduling, dynamic scheduling and for hardware invalidation scheme, respectively. The details include the traffic due to read misses, write misses, and due to coherence actions (for write back and for invalidation). Table 3 also shows the percentage of invalidation requests over all references. Tables 4 and 5 show the same kind of data for programs BALANCE and SPLIE2, respectively.

In these three programs, reference marking with static scheduling obtains the best results. For three programs (AMOEBA, BALANC and SPLIE2), almost no invalidation requests are generated. In these programs, the array reference patterns and the control flow are simple, and thus the data movement requirement under the static scheduling is clear.

Table 4: Network Traffic for AMOEBA

Case	rm-net	wm-net	coh-net	% inv
static	2.51	0.002	0.98	~0
dynamic	3.08	0.15	3.15	9.43
hardware	3.08	1.90	3.49	10.74

Table 5: Network Traffic for SPLIE2

Case	rm-net	wm-net	coh-net	% inv
static	0.76	0.14	0.24	~0
dynamic	0.78	0.14	0.35	0.56
hardware	0.78	0.69	0.97	3.06

5 Conclusions

We have presented a compiler scheme for marking those write references which do not need to generate invalidation requests. The preliminary results suggest that the network traffic can be significantly reduced when static scheduling is employed and when the array subscripts are easy to analyze. We are currently implementing an interprocedural compiler to perform a reasonably precise array dataflow analysis and plan to test our ideas on more programs through these compilers and simulators.

Acknowledgement

We thank Trung Nguyen for providing the traces and thank Farnaz Mounes-Toussi for running the simulation. We also thank the reviewers for their helpful comments. This work is supported in part by the National Science Foundation, Grant NSF/CCR-9210913.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Mass., 1986.

Table 6: Network Traffic for BALANCE

Case	rm-net	wm-net	coh-net	% inv
static	0.19	0.66	0.04	~0
dynamic	0.19	0.66	0.06	0.006
hardware	0.19	0.66	0.19	0.08

[2] M. Berry, D. Chen, P. Koss, D. Kuck, and S. Lo. The perfect club benchmarks: Effective performance evaluation of supercomputers. Technical Report CSRD-827, University of Illinois, Urbana, IL, May 1989.

[3] L. M. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, C-27(12):1112–1118, December 1978.

[4] Y. Chen and M. Dubois. Cache protocol with partial block invalidation. In *Proc. 7th International Parallel Processing Symposium*, 1993.

[5] Y. Chen and A. V. Veidenbaum. Comparison and analysis of software and directory coherence schemes. In *Proc. 1991 ACM International Conference on Supercomputing*, pages 818–829, 1991.

[6] H. Cheong and A. V. Veidenbaum. A cache coherence scheme with fast selective invalidation. In *Proc. 15th Annual International Symposium on Computer Architecture*, pages 299–307, June 1988.

[7] H. Cheong and A. V. Veidenbaum. A version control approach to cache coherence. In *Proc. 1989 ACM International Conference on Supercomputing*, pages 322–330, 1989.

[8] H. Cheong and A. V. Veidenbaum. Compiler-directed cache management in multiprocessors. *Computer*, 23(6):39–47, June 1990.

[9] S. J. Eggers and R. H. Katz. A characterization of sharing in parallel programs and its application to coherency protocol evaluation. In *Proc. International Symposium on Computer Architecture*, 1988.

[10] 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.

[11] A. Gupta and W. Weber. Analysis of cache invalidation patterns in multiprocessors. In *Proc. International Symposium on Computer Architecture*, 1989.

[12] Z. Li. Array privatization for parallel execution of loops. In *Proceedings of the 1992 International Conference on Supercomputing*, July 1992.

[13] Z. Li. Analyzing scheduled fork-join programs to reduce cache invalidation traffic. Technical Report CSci-93-86, Computer Science Department, University of Minnesota, Minneapolis, MN, 1993.

[14] S. L. Min and J. Baer. A performance comparison of directory-based and timestamp-based cache coherence schemes. In *Proc. 1990 International Conference on Parallel Processing*, pages 305–311, 1990.

[15] S. L. Min and J. Baer. A timestamp-based cache coherence scheme. In *Proc. 1989 International Conference on Parallel Processing, Vol. I: Architecture*, pages 23–32, 1990.

[16] F. Mounes-Toussi, D. J. Lilja, and Z. Li. Compiler support for reducing the network traffic and the miss ratio in a directory-based cache coherence mechanism. Technical Report CSci-93-89, Computer Science Department, University of Minnesota, Minneapolis, MN, 1993.

[17] T. N. Nguyen, Z. Li, and D. J. Lilja. Efficient use of cache coherence directories through compiler analysis. Technical Report CSci-93-64, Computer Science Department, University of Minnesota, Minneapolis, MN, 1993.

[18] T. N. Nguyen, Z. Li, and D. J. Lilja. Efficient use of dynamically tagged directories through compiler analysis. In *Proc. 1993 International Conference on Parallel Processing*, St. Charles, IL, 1993.

[19] S. Owicki and A. Agarwal. Evaluating the performance of software cache coherence. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 230–242, 1989.

[20] J. Peir, K. So, and J.H.Tang. Techniques to enhance cache performance across parallel program sections. In *Proc. 1993 International Conference on Parallel Processing, Vol. I: Architecture*, pages 12–19, 1993.

[21] C. D. Polychronopoulos, M. B. Girkar, M. R. Haghighat, C. L. Lee, B. P. Leung, and D. A. Schouten. Parafrase-2: An environment for parallelizing, partitioning, synchronizing, and scheduling programs on multiprocessors. In *Proc. 1989 International Conference on Parallel Processing*, St. Charles, IL, August 1989.

[22] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes: The Art of Scientific Computing (Fortran Version)*. Cambridge University Press, 1989.

[23] J. Torrellas, M. S. Lam, and J. L. Hennessy. Shared data placement optimizations to reduce multiprocessor cache miss rates. In *Proc. International Conference on Parallel Processing*, 1990.