CICO: A Practical Shared-Memory Programming Performance Model

James R. Larus
Satish Chandra
David A. Wood

Technical Report #1171

August 1993

CICO: A Practical Shared-Memory Programming Performance Model*

James R. Larus, Satish Chandra, David A Wood†
larus@cs.wisc.edu
Computer Sciences Department
University of Wisconsin–Madison
1210 West Dayton Street
Madison, WI 53706 USA
608-262-9519

August 16, 1993

Abstract

A programming performance model provides a programmer with feedback on the cost of program operations and is a necessary basis to write efficient programs. Many shared-memory performance models do not accurately capture the cost of interprocessor communication caused by non-local memory references, particularly in computers with caches. This paper describes a simple and practical programming performance model—called *check-in, check-out (CICO)*—for cache-coherent, shared-memory parallel computers. CICO consists of two components. The first is a collection of annotations that a programmer adds to a program to elucidate the communication arising from shared-memory references. The second is a model that calculates the communication cost of these annotations. An annotation's cost models the cost of the memory references that it summarizes and serves as a metric to compare alternative implementations. Several examples demonstrate that CICO accurately predicts cache misses and identifies changes that improve program performance.

1 Introduction

Shared memory in parallel computers provides the valuable abstraction of a shared address space in which processors communicate by reading and writing memory locations. This shared name space simplifies many programs by making data structures processor-independent, which facilitates load balancing [26]; allowing pointer-based data structures, which are necessary for sophisticated algorithms; and freeing programs from per-processor memory limits, which permits effective use of a machine's total memory. The shared-memory abstraction, however, hides

*Presented at: Workshop on Portability and Performance for Parallel Processing, Southampton University, England, July 13–15, 1993. To appear: Ferrante & Hey eds., *Portability and Performance for Parallel Processors*.

†This work is supported in part by NSF Grants CCR-9101035 and MIP-9225097, NSF Presidential Young Investigator Award CCR-9157366, a University of Wisconsin Graduate School Grant, and by donations from Digital Equipment Corporation, Xerox Corporation, and Thinking Machines Corporation. Our Thinking Machines CM-5 was purchased through NSF Institutional Infrastructure Grant No. CDA-9024618 with matching funding from the Univ. of Wisconsin Graduate School.

important details of how physical memory is partitioned and how values are communicated among processors, which leads programmers to write inefficient programs.

Shared memory is implemented either with software, hardware, or a combination of the two. Purely software systems, such as Li's shared virtual memory [29], implement a shared address space without shared-memory hardware by using the virtual-memory hardware available in most machines to detect non-local memory references. These references cause traps, which invoke system software that copies pages of data among processors. Another approach is to have compiled code provide a shared address space on a message-passing machine. Compilers, such as Vienna Fortran [18, 41] and Fortran D [24], implement a shared address space by detecting remote memory references (primarily at compile time) and sending data between processors with explicit messages.

Purely software solutions perform poorly for many programs, so many forms of shared-memory hardware have been proposed and built. Non-uniform access machines, such as the BBN Butterfly [36], physically partition memory among processors, which results in sharply higher costs for remote accesses. However, most shared-memory computers, even if memory is partitioned, use caches to keep copies of a memory location close to the processors that are actively accessing it. Caches reduce effective memory access time and communication network load since interprocessor communication occurs only when a block is brought into or removed from a cache. A major issue in these systems is keeping cached copies consistent as processors modify memory locations. Solutions to this cache coherent problem distinguishes several classes of parallel computers. Multis [6] are bus-based computers in which all processors watch memory accesses occurring on a shared bus and modify their caches appropriately. Directory-based computers—such as Stanford DASH [27] and MIT Alewife [1]—eliminate the non-scalable bus by having hardware—and sometimes software—maintain a directory that records which processors hold copies of a cache block. A cache-coherence protocol uses the directories to serialize conflicting updates and to invalidate copies at updates.

Cache-only (COMA) machines eliminate the memory underlying caches in cache-coherent machines and manage main memory as a cache. COMA hardware and protocols are complex since they must ensure that a copy of each location remains cached. It is not yet clear whether the model in this paper works for this type of machine.

In cache-coherent, shared-memory machines, a memory reference that requires interprocessor communication is more expensive than a reference satisfied by the local cache. For example, in the Stanford DASH computer, a read from the local cache can take a single cycle, while a remote memory read requires from 34 to 132 cycles [28]. Unfortunately, the behavior and associated cost of a memory reference is difficult to predict. On a parallel computer, the state of a processor's cache depends not only on a processor's previous memory references, but also on other processors' reference patterns. Without a technique for modeling a cache's externally-visible behavior (which locations it has cached), a programmer has few ways of understanding and improving a program's performance on this type of computer.

Programmers rarely analyze a program by simulating a particular computer. Instead, they rely on simplified models, which we call *programming models*, that abstract a machine's details. A programming model has a semantic component that describes how operations behave and a performance component that describes how rapidly operations execute. For most purposes, simple, qualitative programming models suffice. For example, in the sequential domain, asymptotic algorithm analysis assumes unit cost for operations and counts the number of times each operations executes. However, even simple programming performance models help in writing efficient programs. More sophisticated models, which may be more difficult and costly to apply, become necessary when the simpler models are not accurate enough.

Unfortunately, no practical and accurate programming performance model exists for shared memory. The most common model is *naive shared memory*, which assumes all memory references are equally cheap (or costly). Even on a simple, shared-bus multiprocessor, such as the Sequent Symmetry, this model does not accurately capture hardware behavior [21]. For example, on this computer, a memory reference that hits in the local cache is roughly 20 times faster than a reference to main memory. The discrepancy is even larger on more recent machines—for example, 100 times on Stanford DASH [27]—and is likely to continue increasing. Theoretical models, such as PRAMs [17, 16, 38], abstract too far from real machines to provide a practical programming model, and consequently do not model features of real machines, such as caches.

Programs written under an incorrect or inaccurate model often perform poorly and their performance does not scale because of bottlenecks and poor hardware utilization. Programmers who are unaware of true costs use hardware ineffectively because they unnecessarily invoke expensive operations and overuse resources, which causes performance-limiting bottlenecks. For example, in comparing shared-memory and non-shared-memory algorithms, Ngo and Snyder ran an LU factorization of a matrix on three shared-memory computers [32]. The naive shared-memory version of the program allowed any processor to update any portion of the matrix. Another version of the program was structured like a message-passing program. It partitioned the matrix among the processors so only one processor updated each portion. The latter program ran faster on all three shared-memory machines because data remained in processors' caches and the increased data reuse reduced interprocessor communication.

Shared-memory communication is the interprocessor message traffic caused by cache misses and invalidations. One part is the coherence traffic when processors exchange values or appear to exchange values because of false sharing [15]. The other part is the conflict and capacity misses caused by caches of finite size and associativity [23]. A programmer or compiler can reduce both aspects of shared-memory communication by modifying a program and its data structures to use caches more effectively. However, to make such a change, a programmer or compiler must understand why shared-memory communication occurs and be able to evaluate alternative program organizations.

This paper describes in detail the *Check-In, Check-Out (CICO)* model for reasoning about shared-memory communication in cache-coherence parallel computers. The model has been briefly mentioned elsewhere [22]. The model is less precise, but far easier to understand and employ than a detailed description of a particular cache-coherence protocol. CICO describes the shared-memory communication in a broad class of cache-coherent, shared-memory computers without attempting to precisely describe the exact behavior or cost of this communication on a particular machine.

Section 2 describes the CICO model. Section 3 shows how the model can be used to improve programs' performance. Section 4 briefly describes how hardware can also use the CICO annotations to improve programs' performance. Finally, section 5 describes related work.

1.1 Message-Passing Programming Model

The principal competitor to shared-memory programming is message passing. Other parallel programming models, such as data parallelism or actors, are mechanisms for specifying actions rather than accessing data, and are orthogonal to these two models. Message passing, unlike shared memory, combines a difficult semantic model with a simple performance model. It is worth contrasting briefly the advantages and disadvantages of the two models.

Message passing uses *send* and *receive* operations to communicate values between processors' distinct address spaces. These operations clearly identify the points at which communication

occurs, which facilitates calculating the communication cost. On the other hand, these operations also introduce new language semantics that sharply distinguish shared and local data. Shared data can only be read with the explicit cooperation of its producer and consumer, must be copied (and consequently renamed) to be shared, and must be carefully updated by the program to remain consistent. For programs with static communication patterns and flat data structures, such as arrays, these considerations may not be a serious impediment. In other programs, however, these factors greatly complicate message-passing programming.

By exposing the underlying hardware primitives, message passing operations identify interprocessor communication and facilitates performance modeling. A simple model, which is commonly used by compilers [19, 34], attributes a uniformly high cost to messages, independent of their size or destination. Although crude, this model identifies communication as a major bottleneck and suggests effective optimizations such as combining multiple sends into a single message and overlapping communication and computation. More complex performance models account for interconnection network topology [7].

2 CICO Shared-Memory Performance Model

Unlike message-passing programs, communication in shared-memory programs is not easily identifiable. In cache-coherent, shared-memory computers, interprocessor communication occurs when a memory reference misses in a cache and the hardware coherence protocol [2, 5] requests a copy of the referenced datum from another processor, which may cause outstanding copies to be invalidated. This mechanism offers several advantages: caches dynamically adapt to a program's reference pattern, the replicating data retains the same address as the original, and the hardware ensures a globally coherent view of data. However, these advantages become disadvantages when trying to understand a program's communication. The state of a location (i.e., where its copies are cached) depends on the memory-reference history of the processors and the details of the coherence protocol. The protocol's operation (and cost), in turn, depends on the memory's state.

An accurate, but slow and complex method of calculating the cost of a memory access is to simulate a particular machine in detail and record which statements cause interprocessor communication [11, 14, 35]. Simulation, because it is so time-consuming, is generally limited to studying short programs with small data sets.

For many programmers, a more attractive approach would trade accuracy for simplicity. The *CICO performance model* makes three approximations that permit reasoning about cache's externally-visible behavior in a cache-coherent parallel computer.

1. Communication occurs only at the points in a program demarcated by CICO annotations. A program's author adds annotations to the program to describe the movement of data in and out of the cache. The annotations are notations that *only* describe how data moves in and out of the cache. Since they are declarative, not imperative, they do *not* affect a program's semantics or a cache's operation. The programmer uses annotations to describe and reason about cache behavior, not to implement software cache coherence (cf. [8, 13, 31]). The advantage of summarizing cache behavior with these annotations is that the programmer only needs to study the communication at the annotations and need not reason about every memory reference.

   Moreover, the same annotations can also function as directives to a memory system designed to exploit them [22]. In this case, these annotations provide a mechanism by which

a program can inform the memory system of upcoming memory references, so the memory system can anticipate them to improve performance. To distinguish the two uses of annotations, we will call them *directives* when they pass information at run time.

2. Processor caches are fully associative. Uniprocessor cache misses occur because caches have finite size and associativity [23]. Unlike capacity misses due to finite size, conflict misses due to limited associativity are difficult to understand and predict because they depend on the relative location of objects in memory.

3. Communication at annotations can be attributed a cost with a simple, three-state model that ignores network and directory contention. Incorporating these issues in the CICO model is difficult, since the annotations collapse a sequence of memory references that occur over a period of time into a single event. The CICO cost model, however, computes a lower bound on communication cost, which provides a useful basis for reasoning about communication.

2.1 CICO Annotations

CICO relies on three annotations to delimit portions of a program in which a memory location resides in a processor's cache. The first annotation indicates the beginning of an interval in which a processor expects to use a block:

`check_out_X`	Expect exclusive access to block
`check_out_S`	Expect shared access to block

`check_out_X` asserts that the processor performing the check-out expects to be the only processor accessing the block until it is checked-in. `check_out_S` asserts that the processor is willing to share (read) access to the block. Operationally, the `check_out` annotations fetch a copy of the block into the processor's cache, as if the processor directly referenced the block.

The next annotation, `check_in`, marks the end of an interval in which a processor uses a block. An interval ends either because of a capacity miss in the cache or because another processor accesses the checked-out block.

`check_in`	Relinquish a block

Operationally, the `check_in` annotation flushes the block from the processor's cache, as if the processor had replaced it upon a cache conflict.

Communication cannot be eliminated from parallel programs. An important step to reduce the impact of communication is to overlap it with computation by prefetching data:

`prefetch_X`	Expect exclusive access to block in near future
`prefetch_S`	Expect shared access to block in near future

`prefetch_X` (`prefetch_S`) asserts that a processor performing a prefetch to a block is likely to access it exclusive (shared) in the near future. Operationally, this annotation brings the block into its cache while the processor continues the computation. This paper does not use this annotation.

To simplify the notation, we frequently apply an annotation to a range of memory locations, for example: `check_out_X A[1:N]`. This notation is a shorthand for performing a `check_out_X` on every cache block in the range of locations. The degenerate case, `check_out_X A[i]`, checks-out element i in array A, even if the value spans more than one cache block.

2.2 Adding Annotations

This section describes an approach for annotating cache-block race-free programs with CICO primitives. A program is *cache-block race-free* if it contains no data races and no unsynchronized false sharing [15]. Consequently, when two processors access the same cache block, they must execute a synchronization event between their accesses. These rules place CICO annotations in race-free programs so as to capture the interprocessor communication caused by a cache-coherence protocol. The rules can also be used, with a loss of accuracy, to annotate programs containing races.

Assume for the moment that processors have infinite, fully-associative caches so we need not worry about cache replacements. Partition a program into *epochs*, which are code segments that execute between two synchronization events. A program can be divided into epochs by splitting its control-flow graph at synchronization points. Each connected component of the graph is an epoch. Upon entry to an epoch, processor p imports data from other processors by checking-out cache blocks that it will access during the epoch. Processor p need not check-out data already in its cache. Upon leaving an epoch, a processor exports data to other processors by checking-in cache blocks that may be accessed subsequently by another processor. It also checks-in blocks that it will not access in subsequent epochs.

To apply these rules, a programmer may need to introduce several approximations. In some programs, a programmer cannot predict at the beginning of an epoch exactly which data a processor will access. In this case, the programmer can either check-out all data that might possibly be accessed or move check-outs closer to data references. The first solution causes the model to overestimate the communication cost. The second solution is more precise, but may be more difficult to apply and reason with, as in the degenerate case in which each memory reference is annotated. In addition, when a programmer cannot predict which processor will use a block after an epoch, the block must be conservatively checked-in. In all of these cases, improved knowledge of the program's sharing pattern increases the accuracy of the annotation and may identify program improvements that increase performance.

In an actual computer, a cache's size constrains the amount of data that a processor can check-out. To model a finite cache, a programmer must add check-ins that flush data, so as not to exceed the cache's capacity. These check-ins, in turn, may require additional check-outs to bring data back into the cache. These additional annotations illuminate the high cost of cache replacement, which requires interprocessor communication to retrieve data evicted from the cache.

These rules for inserting annotation do not enable a programmer to predict the cost of the cache-block races. The restriction to cache-block race-free programs permits the model to capture data movement in and out of caches. Cache block races cause timing-dependent communication as one processor steals a block from another processor. These races are undesirable, not only because they are difficult to model accurately, but also because they are expensive on a real computer.

2.3 CICO Cost Model

The *CICO cost model* described in this section computes a cost of shared-memory communication by attributing a cost to each CICO annotation. By analyzing a program to determine how many times an annotation executes, a programmer can determine the communication cost of the annotation. If the annotation accurately models the cache's behavior, the cost attributed to the annotation equals the communication cost of the memory references that the annotation

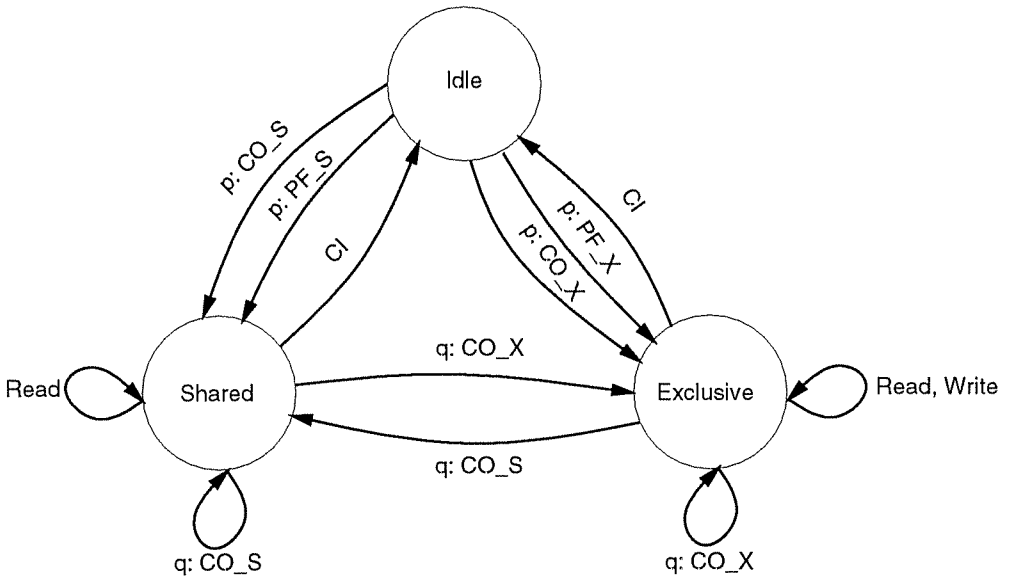

Figure 1: CICO performance model. A cache block can be in one of three states: *idle*, *shared*, or *exclusive*. Transitions between states occur at CICO annotations and are labeled with the annotation (CO is `check_out`, CI is `check_in`, PF is `prefetch`) and processor (p or q) that caused the transition. In the diagram, processor p obtains a block in the idle state and a distinct processor q operates on the block when it is not idle. A block becomes idle when the last shared or only exclusive copy is checked-in. The processor that causes a transition incurs a communication cost.

summarizes.

In the CICO cost model, the communication cost of CICO annotations is modeled with the aid of an automaton with three states: *idle*, *shared*, or *exclusive* (see Figure 1). Each cache block has its own automaton. Initially, every block is *idle*, which means that no cache has a copy. Transitions between states occur at CICO annotations. Edges in the automaton are labeled with the annotation that caused them. For example, if a block is *idle*, a `check_out_X` changes the block's state to *exclusive*. The processor causing a transition incurs the communication cost associated with an arc.

Communication costs can be modeled in three ways. The first uses values from an actual machine. The advantage of this approach is that the costs accurate model at least one machine. However, in many cases, these values are too machine- and configuration-specific. A more general approach uses values that are asymptotically correct for a large class of machines. Operations that execute asynchronously, such as prefetches or check-ins, are unit cost. Operations that require a synchronous message exchange, such as check-outs, require time proportional to a round-trip message time: in many machine, the cost of a message grows as $O(\lg P)$, where P is the number of processors. Finally, the transition $Shared \rightarrow Exclusive$ has worst-case cost proportional to $O(P)$ since all extant copies must be invalidated by explicit messages or a broadcast. The final model, which suffices for many purposes, attributes a unit cost to each transition that requires synchronous communication. Table 1 compares the three models.

Initial State	Action	Final State	Actual Cost	Asymp Cost	Unit Cost
Idle	prefetch_X	Exclusive	8	O(1)	0
	check_out_X		242	$O(\lg P)$	1
	prefetch_S	Shared	8	O(1)	0
	check_out_S		242	$O(\lg P)$	1
Exclusive	check_in	Idle	16	O(1)	0
	check_out_X	Exclusive	996	$O(\lg P)$	1
	check_out_S	Shared	996	$O(\lg P)$	1
Shared	check_in	Idle | Shared	8	O(1)	0
	check_out_X	Exclusive	1285	$O(P)$	1
	check_out_S	Shared	242	$O(\lg P)$	1

Table 1: Costs for transitions in three CICO performance models for directory-based shared-memory computers. The actual costs are for the Dir_1SW protocol with 64 processors and 100 cycle message latency. prefetch and check_in execute asynchronously and only incur the operation initiation cost. The cost of a check_out depends on the state of the cache block and can be thousands of cycles if a broadcast is necessary to invalidate outstanding copies. The asymptotic costs show how costs increase with the number of processors P in a machine. Finally, the unit cost model only counts the synchronous operations.

3 Performance Tuning With CICO

This section presents three examples of how the CICO model can be used to understand and improve shared-memory programs. In addition, it contains measurements that confirm the unit cost model's predictions and demonstrate the predicted performance improvement.

One thing that we do not provide is rules for transforming programs to improve their shared-memory performance. Our general approach is to reduce unnecessary communication by increasing a program's reuse of data already in its cache. The changes for a particular program depend on the program's structure and must account for issues orthogonal to this model such as load balancing and task granularity. CICO simply provides a framework for reasoning about a program's shared-memory communication and for comparing alternative strategies.

3.1 Matrix Multiplication

As a simple example to illustrate the CICO model, consider the well-known problem of multiply dense $N \times N$ row-major matrices. For simplicity, N is a multiple of P, the number of processors; matrices are stored in row-major order; and each processor p computes the N/P rows, L_p to U_p, of the product:

```
for i := L_p to U_p do
   for k := 1 to N do
      for j := 1 to N do
         A[i, j] := A[i, j] + B[i, k] * C[k, j];
      od
   od
od
```

The placement of the annotations depends, in part, on the relative size of the matrices and cache. In the best, but least interesting case, all three matrices fit in a cache and each processor only needs to check-out its portion before the loops and check it in after.

```
check_out_X A[L_p:U_p, 1:N];
check_out_S B[1:N, 1:N], C[1:N, 1:N];
for i := L_p to U_p do
   for k := 1 to N do
      for j := 1 to N do
         A[i, j] := A[i, j] + B[i, k] * C[k, j];
      od
   od
od
check_in A[L_p:U_p, 1:N]; B[1:N, 1:N], C[1:N, 1:N];
```

The amount of communication is $O(N^2)$. Matrices of this size do not require parallel computers (at least until caches become much larger).

A more interesting case, in which a cache can hold at least one row, but not N rows, we model by checking-out each row as early as possible:

```
for i := L_p to U_p do
   check_out_X A[i, 1:N];
   for k := 1 to N do
      check_out_S B[i, k], C[k, 1:N];
      for j := 1 to N do
         A[i, j] := A[i, j] + B[i, k] * C[k, j];
      od
      check_in B[i, k], C[k, 1:N];
   od
   check_in A[i, 1:N];
od
```

The first check-out executes N/P times on each processor and checks-out N/b cache blocks, where b is the number of double precision floating point values that fit in a cache block, for a total of $N^2/(bP)$ cache blocks. The latter two check-outs execute N^2/P times and check-out $1/b$ and N/b blocks, respectively. The P processors check out a total of $(N^3 + 2N^2)/b$ blocks. The annotations identify the data access patterns that effectively use the cache. Each element of A and B is reused N times, while an element of C is accessed only once each time it is brought into the cache, which contributes the $O(N^3)$ term.

In the worst case, in which the cache cannot hold a matrix row, the annotations look like:

```
for i := L_p to U_p do
   for k := 1 to N do
      check_out_S B[i, k];
      for j := 1 to N do
         check_out_X A[i, j];
         check_out_S C[k, j];
         A[i, j] := A[i, j] + B[i, k] * C[k, j];
```

```
        check_in A[i, j], C[k, 1:N];
      od
      check_in B[i, k]
    od
  od
```

The annotation for B executes N^2/P times, checking out $1/b$ blocks each time. The annotations for A and C execute N^3/P times, checking out $1/b$ blocks each time. The total number of checked-out blocks is $(2N^3 + N^2)/b$.

Blocking (also called tiling) is a well-known technique for reducing communication in this algorithm [25]. It breaks the problem into multiplications of smaller submatrices. Because the submatrices fit in the cache, array elements are repeatedly accessed locally rather than remotely fetched. Assume that the block factor T is chosen so a submatrix of size $T \times T$ and two rows of length T fit in the cache, the annotated blocked algorithm looks like:

```
for kk := 1 to N step T do
  for jj := 1 to N step T do
    U_k := kk + T - 1; U_j := jj + T - 1;
    check_out_S C[kk : U_k, jj : U_j];
    for i := L_p to U_p do
      check_out_X A[i, jj : U_j];
      check_out_S B[i, kk : U_k];
      for k := kk to U_k do
        for j := jj to U_j do
          A[i, j] = A[i, j] + B[i, k] * C[k, j];
        od
      od
      check_in A[i, jj : U_j]; B[i, kk : U_k];
    od
    check_in C[kk : U_k, jj : U_j];
  od
od
```

The three submatrices are checked-out before they are used and are checked-in immediately after. On each processor, the first check-out executes N^2/T^2 times and the other two check-outs execute $N^3/(PT^2)$ times. The first check-out accesses T^2/b blocks and the other two check-out's access T/b blocks. The program accesses a total of $(2N^3/T + PN^2)/b$ blocks. Blocking speeds matrix multiplication on real computers [25] by reducing the number of cache blocks accessed by a factor of $2/T$, in the limit.

3.1.1 Experimental Results

This section contains preliminary results from some experiments run to test the CICO model. We used CICO to analyze matrix multiplication and modified the code as described above. We then ran the original and modified programs on a detailed simulation of a 32 processor machine running the original Dir_1SW protocol [22]. The simulation ran on the Wisconsin Wind Tunnel, which is a detailed and accurate parallel architecture simulator that runs on a Thinking Machines CM-5 computer [35]. In both cases, the predicted performance improvements were observed and the predicted number of shared cache misses was close to the actual number of shared misses.

	Cache Misses (millions)	
	CICO Predicted	**Actual**
Unblocked	33.7	31.9
Blocked	4.3	2.7

Table 2: Predicted and actual caches misses for unblocked and blocked multiplication of 512×512 matrices. The numbers do not include 0.7 million cache misses in the initialization code.

The first program multiplies two 512×512 matrices containing double-precision floating-point value. In the original, unblocked code, each processor computed the product for N/P rows of the result matrix. To compute a row, a processor reads a complete matrix (more than 2 megabytes). In the modified, blocked code, each processor computes a 16×16 submatrix of the product. We simulated processors with 256K, 4-way set associative caches. Blocking decreased execution time from 784 million to 428 million cycles, a speedup of 1.8. Total cache misses (not including 0.7 million in initialization) fell from 31.9 million to 2.7 million, a 92% improvement.

Table 2 compares the cache misses predicted by CICO against the actual cache misses. In the blocked program, CICO underpredicts cache misses. We believe the discrepancy is due to unmodeled cache reuse of the blocks in the large cache. In the unblocked code, CICO's prediction is very close to the measured value and clearly demonstrate the importance of blocking this algorithm.

3.2 Water

Water is one of the Stanford Splash shared-memory benchmarks [37]. It is an N-body molecular dynamics code that calculates the forces and potentials among a collection of water molecules. Its principal data structure is a vector of N water molecules. Each of the P processors is statically assigned to compute forces and potentials for N/P contiguous molecules. Because inter-molecular forces are symmetric, each processor only computes the interaction between a molecule and $N/2$ other molecules. Process p's main loop is:

```
for s := 1 to time_steps do
  for m := L_p to L_p + N/P do
    for i := m + 1 to (m + 1 + N/2)  (mod N) do
      lock molecules m and i;
      compute forces on molecules m and i and update them;
      unlock molecules m and i;
    od
  od
  compute position of molecules L_p ... L_P + N/P
od
```

where L_p is the first molecule in the vector assigned to processor p.

To annotate these loops, we must make assumptions about the processors' caches. Again, the easy and uninteresting case is that all molecules fit in a cache. This assumption is unrealistic since the data for a molecule requires 672 bytes of storage (although only 168 bytes of it are accessed in the parallel loop discussed below). For now, assume that $N/2$ molecules cannot fit in the cache, but N/P molecules fit. Later, we will relax the second assumption to accommodate

larger N. The two assumptions imply that the parallel inner loop of *water*, as written, has little or no cache reuse of the molecules indexed by loop i:

```
for s := 1 to time_steps do
   for m := L_p to L_p + N/P do
      check_out_X molecule m;
      for i := m + 1 to (m + 1 + N/2)  (mod N) do
         lock molecules m and i;
         check_out_X molecule i;
         if i mod (N/P) = m then check_out_X molecule m; fi
         compute forces on molecules m and i and update them;
         check_in molecule i;
         unlock molecules m and i;
      od
      check_in molecule m;
   od
   compute position of molecules L_p ... L_P + N/P
od
```

The first check-out references N/P molecules per processor per time step. Each processor executes the second check-out N/P times per time step, so the $N/2$ molecules indexed by loop i are checked-out a total of $N^2/(2P)$ times. The conditional check-out in loop i is necessary because another processor accesses molecule m every N/P iterations of loop i (assuming all processors execute at roughly the same rate). Although the locks prevent data races, when another processor accesses molecule m, it leaves the first processor's cache and must be reobtained when the first processor acquires the lock. This communication occurs $P/2$ times per inner loop invocation or a total of $N/2$ times per time step, which does not significantly affect the communication cost.

Interchanging the two loops leads to a program that performs the same work in a more communication-efficient manner:

```
check_out_X molecules L_p ... L_p + N/P;
for s := 1 to time_steps do
   for i := L_p to (L_p + N/2 + N/P)  (mod N) do
      check_out_X molecule i;
      for m := L_p to L_p + N/P do
         if (i - m  (mod N)) ≤ N/2 then
            if i mod (N/P) = m then check_out_X molecule m; fi
            lock molecules m and i;
            compute forces on molecules m and i and update them;
            unlock molecules m and i;
         fi
      od
      check_in molecule i;
   od
   compute position of molecules L_p ... L_P + N/P
do
```

Each processor now check-outs molecule i $N/2 + N/P$ times per time step, so the total number of cache misses is $O(N)$ rather than the previous $O(N^2/P)$. The conflicting accesses to

molecules introduce $N/2 - N/P$ additional check-outs, which again do not significantly affect the communication cost.

Restructuring *water* reduces its communication cost without changing its time complexity. Moreover, a simple variant of the restructured program works even when N is large enough that N/P molecules do not fit in the cache. In that case, blocking loop i by a factor of T ensures $N/(PT)$ molecules fit in the cache.

3.2.1 Experimental Results

We modified this Splash benchmark in the manner described in Section 3.2, by interchanging the loops that iterated over the molecules, so as to increase cache reuse of the molecules. Improvements in running time were not as dramatic since the computation on a pair of molecules is far more costly than the communication to obtain the molecules. The program computed the interactions among 256 molecules through 10 time steps. Each simulated processor had a 32K, 4-way set associative cache (a 256K cache could hold all molecules). The unmodified program ran in 718 million cycles and the modified program took 690 million cycles (a 4% improvement). Cache misses, however, decreased from 2.4 million to 0.6 million (a 76% improvement). The CICO model in Section 3.2 predicted that cache misses should decrease from 2.2 million to 0.5 million, which are very close to the observed values.

The dramatic reduction in cache misses in these experiments is important in light of the current trend in which processor cycle times are decreasing much faster than network latencies. These experiments assumed a modest 100 cycle delay to obtain a remote cache block. Longer latencies in future machines will reduce performance even more unless cache misses can be reduced.

3.3 Mp3d

Mp3d is another Stanford Splash shared-memory benchmarks [37]. It performs a Monte Carlo simulation of a rarefied fluid flow. The simulation traces molecules through a three-dimensional space array of unit-sized cells and uses the accumulated information in each cell to compute the fluid flow. The original *mp3d* code statically partitioned molecules among processors, so each processor moves its collection of molecules through the space array. This approach produced considerable locality of reference for molecules (except after a collision, as in *water*), but little locality for the space array. Our changes concentrated on the latter data structure.

The code was restructured in two ways to exploit insights provided by the CICO model. The first modification updated a space cell's accumulated information when a molecule moves into the cell rather than waiting until the molecule leaves the cell in the following time step. The original code reduced computation at the expense of communication since it avoided multiple updates to a cell when a molecule was knocked from the cell before the end of a time step. The code operated as follows:

1] **if** molecule moves **then** update starting cell with velocity;
2] compute new position of molecule;
3] **if** collision **then** update molecule's position and velocity;

This organization requires a processor to check-out a space cell twice. The first check-out occurs in statement 2 or 3, when recording the molecule's presence in its new cell and checking for a collision. In the next time step, the processor updates the space cell's aggregate statistics

Version	Execution Time (millions of cycles)	Cache Misses (millions)	Cache Miss Time (millions of cycles)
Original	813.6	6.3	324.1
R1	653.8	5.5	254.4
R2	537.3	5.5	188.8

Table 3: Execution time and cache misses for the original *mp3d* and two modified versions. Version R1 updated a space cell while it was in a processor's cache instead of waiting until the following time step. Version R2 reduced the contention at space cell updates by locking them during a first pass over the data structure.

as the molecule leaves the cell in statement 1. The cache block for a space cell is unlikely to remain in a processors' cache between these accesses. A simple change is to perform the update from statement 1 after statement 3, while the space cell is still in the cache. This change requires minor corrections later in the computation to undo the update if the molecule is knocked from the space cell by a collision.

```
1] compute new position of molecule;
2] if collision then update molecule's position and velocity and
     undo accumulated statistics;
3] update starting cell with velocity;
```

The second change reduced the number of access conflicts when updating molecules' positions (in space cells). Originally, each processor iterated over its set of molecules, updating their position and the position of the molecules with which they collided. If several molecules reside in a cell, the cell data structure ping-ponged back and forth as the processors simultaneously checked it out exclusive. In the modified code, each space cell has an advisory lock. Space cells are updated in two steps. In the first step, no conflicts occur because each processor updates only the cells for which it obtained a lock. In the second step, each processor updates its remaining cells, regardless of the status of the lock. (We tried several iterations of the first steps, but the cost of the additional iterations exceeded the minor benefit of avoid a small number of conflicts.)

3.3.1 Experimental Results

Mp3d was modified as described above and run on WWT. Table 3 shows the effects of the modifications. The first change described above (updating the space cell immediately) reduced the number of cache misses by 13% and improved execution time by 20%. The second change barely affected the number of cache misses, but reduced the cost of processing these misses by 26% (remember that the cost of a cache miss depends on the state of the referenced block and is higher when a block is in use by other processors). These results are encouraging. *Mp3d* is a Monte Carlo code whose behavior is difficult to analyze precisely. Nevertheless, the insights provided for CICO were able to greatly improve its performance with minor modifications to the program.

4 CICO and Hardware

CICO annotations are not only useful for reasoning about cache behavior. They can be passed as *directives* to a memory system to improve program performance. For example, Dir_1SW [22]

is a minimal directory protocol that adds little complexity to the hardware of a message-passing machine, but efficiently supports programs written within the CICO model [40]. It uses a single pointer/counter field to either identify the processor holding a writable copy of a cache block or to count the number of outstanding readable copies. Simple hardware entirely handles programs conforming to the CICO model by updating the pointer/counter and forwarding data to a requesting processor. Programs not conforming to the model run correctly, but cause traps to software trap handlers that perform more complex operations, similar to MIT Alewife [2].

Dir_1SW performs better if programs flush unneeded blocks from caches with check-ins. If these blocks remain in a processor's cache, subsequent memory references would cause the protocol to trap to software to handle the exchange of messages necessary to invalidate copies of a block. By properly employing these annotations, a programmer can reduce the frequency at which blocks must be invalidated, which permits Dir_1SW to implement the invalidation mechanism in software rather than hardware. This, in turn, greatly simplifies Dir_1SW hardware. In addition, the `check_out_X` directive permits a program to reduce message traffic in the common case that it reads a location before modifying it. Ordinarily, the first read of a location would bring its block into the cache in a read-only state, which would must changed to exclusive at the first write. The `check_out_X` annotation brings the block into the cache in state exclusive, which avoids the need for an upgrade message.

5 Related Work

The work of Larry Snyder and students clearly identifies the need for a shared-memory performance model. Lin and Snyder reported an experiment in which programs written for a non-shared-memory (message-passing) machine ran faster on a shared-memory computer than shared-memory programs [30]. Ngo and Snyder reported similar results for more complex programs running on a wide variety of shared-memory computers [32]. The primary difference between the shared-memory and non-shared-memory programs was that the latter programs partitioned shared data so each processor used its cache more effectively and communicated with other processors less frequently than in the shared-memory programs. The approach in these papers—simulating message-passing on shared-memory computers—forfeits some advantages of shared-memory machines. The CICO model identifies a program's communication and encourages a programmer to reformulate the program in a similar manner, but still retains the underlying shared-memory paradigm.

LeBlanc and Markatos identified another advantage of shared memory in their comparison of shared-memory and message-passing programs [26]. They showed that programs with load imbalances are better formulated as shared-memory rather than message-passing programs because tasks and data are more easily moved dynamically in a shared address space. Their programs were written under a naive shared-memory model that was unconcerned with locality. Consequently, balanced programs, which could be partitioned statically, were better formulated as message-passing programs that performed less communication. CICO draws from the strengths of both paradigms. It, like message passing, focuses on optimizing communication and data reuse, but still retains the flexibility and load-balancing of shared memory.

Cheriton et al. describe the changes they made to MP3D [37], a three-dimensional particle simulator to improve its cache performance [9]. The changes eliminated false sharing by changing the data structures to increase their "processor affinity" and hence reduce communication. The changes increased execution speed by a factor of 3–4 on small-scale multiprocessors. Underlying the changes was an intuitive understanding of cache protocols, but no formal model like CICO

that would enable them to compare alternatives strategies. Instead, they spent six person-months running experiments to understand this 1,500 line program.

Alpern et al. describe a performance model, called the uniform memory hierarchy (UMH), for hierarchical uniprocessor memories [4]. This model is an abstract, but detailed description of a computer's memory hierarchy, starting at registers and continuing through virtual memory's paging storage. Unlike CICO, UMH models all levels in a uniprocessor's memory hierarchy and is concerned with the bandwidth between each level. Alpern et al. briefly describe an extension of UMH to shared memory. However, from the description, it is unclear whether UMH adequately describes cache-coherent shared memory and whether the details of the lower levels are necessary to reason about shared-memory references.

Hill and Larus describe a sequence of simple, qualitative models for cache-coherent, shared-bus computers (Multis) [21]. The sequence of models forms an increasingly precise description of the cache-coherence protocols used in Multis. The models enable a programmer to predict how the hardware performs for different patterns of memory references. The paper also presented a few rules of thumb (e.g, avoid false sharing) and explained their rationale in terms of the model.

Hill et al. described a preliminary version of CICO and showed that simple directory hardware could use the annotations to improve program performance [22, 40]. The earlier version of CICO was described in terms of Dir_1SW hardware, did not provide a cost model, and was used to reduce Dir_1SW traps rather than improve program performance.

Snyder argued that an abstract model of a class of computers, which he called a type architecture (and we call a programming model), should be fundamental bridge between programming languages and programmers and computers [39]. His principal requirement for a type architecture is that it should accurately reflect the cost of operations on real machines. In Snyder's terms, the combination of shared memory and CICO can be considered a type architecture for cache-coherent shared-memory computers. The extent to which CICO accurately captures shared-memory costs needs to be more fully explored, but the examples in Section 3 show that CICO is useful even with simple assumptions.

Many theoretical models of shared-memory computers exist. One of the most well-known is *parallel random access machine (PRAM)* model [17]. A PRAM has a single globally-addressable memory and n processors that operate in lock-step read, compute, write cycles. Although PRAM models handle simultaneous reads or writes differently [16, 38], they all assume that memory accesses are unit cost and that synchronization is unnecessary (because processors run in lock-step). These assumptions simplify analysis, but do not reflect real computers, particularly those with caches. For this reason, PRAM extensions model non-uniform memory and processor asynchrony [3, 10, 33, 20]. The new models are more descriptive and complex than traditional PRAM models, but still do not accurately describe cache-coherent parallel computers. Perhaps this paper will help inspire PRAM extensions that describe this important type of computers.

Culler et al. described another model of parallel computation, called LogP, that is closer to actual parallel computers [12]. Like CICO, LogP is an abstraction of real machines intended to provide programmers with insight into potential bottlenecks without falling into excessive detail. The LogP model is based on four parameters that describe the cost of communication in a parallel computer. LogP, however, differs from CICO in several ways. It assume that communications is explicit (though, not necessarily message passing) and provides tools for reasoning about the effect of this communication on an algorithm's running time. LogP also does not model machines with caches since it assumes that communication operations are identified. It might be possible to use LogP to model the costs of cache misses identified by CICO, instead of using the simplier models in Section 2.3. The principle difficulty is the one mentioned above: CICO aggregates communications that occurs over a period of time into a single annotation. One of LogP's

parameters is the time between consecutive messages, which is lost in the CICO annotation.

6 Conclusion

This paper describes the CICO programming performance model and shows how it can be employed to reason about and improve the performance of shared-memory programs running on cache-coherent parallel computers. The CICO model begins with annotations that a programmer adds a program to identify the points at which shared-memory communication occurs. A simple cost model calculates the cost of communication at these annotations and, therefore, the cost of the memory references described by the annotations. The CICO model abstracts from actual cache-coherent computers, which allows the model to be used to reason about interprocessor communication on many machines. In three experiments, the model accurately identified program modifications that greatly reduced the number of cache misses.

We are currently exploring a number of extensions to the model. One desirable addition would be synchronization. Synchronization, like shared-memory references, requires interprocessor communication, which should be measured by a communication model. On the other hand, synchronization is time dependent and requires an interprocessor rendezvous, which is difficult to model in this framework. Another extension could capture more than two levels of memory hierarchy, for example two levels of caches or virtual memory. We also plan to explore more accurate cost models for particular processors, to see when the increased accuracy is beneficial.

Acknowledgements

Mark D. Hill provided encouragement and many helpful comments on this paper. Tom Reps and Elizabeth Shriver provided many helpful comments on this paper. Singh et al. [37] performed an invaluable service by writing and distributing the SPLASH benchmarks. Michael Wolf provided the *mm* benchmark. Shubhendu S. Mukherjee restructured *mp3d* as described above.

References

[1] Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiatowicz. APRIL: A Processor Architecture for Multiprocessing. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 104–114, June 1990.

[2] Anant Agarwal, Richard Simoni, Mark Horowitz, and John Hennessy. An Evaluation of Directory Schemes for Cache Coherence. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 280–289, 1988.

[3] Alok Aggarwal, Ashok K. Chandra, and Marc Snir. On Communication Latency in PRAM Computations. In *Proceedings of the First ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 11–21, June 1989.

[4] Bowen Alpern, Larry Carter, Ephraim Feig, and Ted Selker. The Uniform Memory Hierarchy Model of Computation. *Submitted for publication*, 1992.

[5] J. Archibald and J.-L. Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions on Computer Systems*, 4(4):273–298, 1986.

[6] C. Gordon Bell. Multis: A New Class of Multiprocessor Computers. *Science*, 228:462–466, 1985.

[7] Mark Bromley, Steven Heller, Tim McNerney, and Guy L. Steele Jr. Fortran at Ten Gigaflops: The Connection Machine Convolution Compiler. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 145–156, June 1991.

[8] J. Cheong and A.V. Veidenbaum. A Cache Coherence Scheme With Fast Selective Invalidation. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 299–307, June 1988.

[9] David R. Cheriton, Hendrik A. Goosen, and Philip Machanick. Restructuring a Parallel Simulation to Improve Cache Behavior in a Shared-Memory Multiprocessor: A First Experience. In *International Symposium on Shared Memory Multiprocessing*, pages 109–118, April 1991.

[10] Richard Cole and Ofer Zajicek. The APRAM: Incorporating Asynchrony into the PRAM Model. In *Proceedings of the First ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 169–178, June 1989.

[11] R.C. Covington, S. Madala, V. Mehta, J.R. Jump, and J.B. Sinclair. The Rice Parallel Processing Testbed. In *Proceedings of the 1988 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 4–11, May 1988.

[12] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Rahesh Subramonian, and Thorsten von Eicken. LogP: Toward a Realistic Model of Parallel Computation. In *Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 1–12, May 1993.

[13] Ron Cytron, Steve Karlovsky, and Kevin P. McAuliffe. Automatic Management of Programmable Caches. In *Proceedings of the 1988 International Conference on Parallel Processing (Vol. II Software)*, pages 229–238, Aug 188.

[14] Helen Davis, Stephen R. Goldschmidt, and John Hennessy. Multiprocessor Simulation and Tracing Using Tango. In *Proceedings of the 1991 International Conference on Parallel Processing (Vol. II Software)*, pages II99–107, August 1991.

[15] Susan J. Eggers and Randy H. Katz. A Characterization of Sharing in Parallel Programs and its Application to Coherency Protocol Evaluation. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 373–382, 1988.

[16] Faith E. Fich and Prabhakar L Ragde. Relations Between Concurrent-Write Models of Parallel Computation. In *Proceedings of Principals of Distributed Computing*, pages 179–190, August 1984.

[17] Stephen Fortune and James Wyllie. Parallelism in Random Access Machines. In *Proceedings of the Tenth ACM Symposium on Theory of Computing*, pages 114–118, 1978.

[18] Hans Michael Gerndt. *Automatic Parallelization for Distributed-Memory Multiprocessor Systems*. PhD thesis, Rheinischen Friedrich-Wilhelms-Universität, 1989.

[19] Hans Michael Gerndt and Hans Peter Zima. Optimizing Communications in SUPERB. Technical Report ACPC/TR 90-3, ACPC - Austrian Center for Parallel Computation, University of Vienna, 1990.

[20] Phillip B. Gibbons. A More Practical PRAM Model. In *Proceedings of the First ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 158–168, June 1989.

[21] Mark D. Hill and James R. Larus. Cache Considerations for Programmers of Multiprocessors. *Communications of the ACM*, 33(8):97–102, August 1990.

[22] Mark D. Hill, James R. Larus, Steven K. Reinhardt, and David A. Wood. Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, pages 262–273, October 1992.

[23] Mark D. Hill and Alan Jay Smith. Evaluating Associativity in CPU Caches. *IEEE Transactions on Computers*, C-38(12):1612–1630, December 1989.

[24] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling Fortran D for MIMD Distributed-Memory Machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[25] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 63–74, April 1991.

[26] Thomas J. LeBlanc and Evangelos P. Markatos. Shared Memory Vs. Message Passing in Shared-Memory Multiprocessors. In *Fourth IEEE Symposium on Parallel and Distributed Processing*, page ?, Dallas, TX, December 1992.

[27] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica Lam. The Stanford DASH Multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[28] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. The DASH Prototype: Logic Overhead and Performance. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):41–61, January 1993.

[29] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[30] Calvin Lin and Lawrence Snyder. A Comparison of Programming Models for Shared Memory Multiprocessors. In *Proceedings of the 1990 International Conference on Parallel Processing (Vol. II Software)*, pages II–163–170, August 1990.

[31] Sang Lyul Min and Jean-Loup Baer. A Timestamp-based Cache Coherence Scheme. In *Proceedings of the 1989 International Conference on Parallel Processing (Vol. I Architecture)*, pages I–23–32, August 1989.

[32] Ton A. Ngo and Lawrence Snyder. On the Influence of Programming Models on Shared Memory Computer Performance. In *Scalable High Performance Computing Conference (SHPCC '92)*, page ?, April 1992.

[33] Naomi Nishimura. Asynchronous Shared Memory Parallel Computation. In *Proceedings of the Second ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 76–84, June 1990.

[34] Keshav Pingali and Anne Rogers. Compiling for Locality. In *Proceedings of the 1990 International Conference on Parallel Processing (Vol. II Software)*, pages II–142–146, August 1990.

[35] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David A. Wood. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. In *Proceedings of the 1993 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 48–60, May 1993.

[36] Randall Rettberg and Robert Thomas. Contention is no Obstacle to Shared-Memory Multiprocessing. *Communications of the ACM*, 29(12):1202–1212, December 1986.

[37] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared Memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[38] Marc Snir. On Parallel Search. In *Proceedings of Principals of Distributed Computing*, pages 242–253, August 1982.

[39] Lawrence Snyder. Type Architectures, Shared Memory, and the Corollary of Modest Potential. *Annual Review of Computer Science*, pages 289–317, 1986.

[40] David A. Wood, Satish Chandra, Babak Falsafi, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, Shubhendu S. Mukherjee, Subbarao Palacharla, and Steven K. Reinhardt. Mechanisms for Cooperative Shared Memory. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 156–168, May 1993.

[41] Hans Zima and Barbara Chapman. Compiling for Distributed-Memory Systems. *Proceedings of the IEEE*, 81(2):264–287, February 1993.