To appear: *Proceedings of the 1994 International Conference on Parallel Processing*, Aug. 1994.

Cachier: A Tool for Automatically Inserting CICO Annotations

Trishul M. Chilimbi and James R. Larus
Computer Sciences Department
University of Wisconsin–Madison
1210 West Dayton Street
Madison, WI 53706
[chilimbi, larus]@cs.wisc.edu

Abstract

Shared memory in a parallel computer provides programmers with the valuable abstraction of a shared address space—through which any part of a computation can access any datum. Although uniform access simplifies programming, it also hides communication, which can lead to inefficient programs. The check-in, check-out (CICO) performance model for cache-coherent, shared-memory parallel computers helps a programmer identify the communication underlying memory references and account for its cost. CICO consists of annotations that a programmer can use to elucidate communication and a model that attributes costs to these annotations. The annotations can also serve as directives to a memory system to improve program performance. Inserting CICO annotations requires reasoning about the dynamic cache behavior of a program, which is not always easy.

This paper describes *Cachier*, a tool that automatically inserts CICO annotations into shared-memory programs. A novel feature of this tool is its use of both dynamic information, obtained from a program execution trace, as well as static information, obtained from program analysis. We measured several benchmarks annotated by Cachier by running them on a simulation of the Dir_1SW cache coherence protocol [10], which supports these directives. The results show that programs annotated by Cachier perform significantly better than both programs without CICO annotations and programs that were annotated by hand.

Keywords: Shared-memory, parallel programming performance models, parallel programming tools, cache-coherence, directory protocols.

This work is supported in part by NSF NYI Award CCR-9357779, NSF Grants CCR-9101035 and MIP-9225097, and a Univ. of Wisconsin Graduate School Grant. 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.

1 Introduction

A programmer writing a parallel program can either write the program under a message-passing or a shared-memory model. Message passing requires the programmer to distribute data structures among processors and manage updates to data with messages. Explicitly managing communication complicates the already difficult process of writing correct parallel programs. Shared memory, on the other hand, offers a simpler programming model since shared data can be transparently accessed by any processor. Typically, scalable shared memory systems use a message-passing hardware base augmented by special hardware or software that implements a cache-coherence protocol—for example, Stanford DASH [14], MIT Alewife [3], or Wisconsin Dir_1SW [10][18]. A read to or write from a shared memory location will cause interprocessor communication in some cases, depending whether the referenced data was cached locally or is stored remotely. This communication can seriously impair a program's performance. Although it is often easier to write shared-memory programs, it may be more difficult to write a fast program.

To write efficient programs, a shared-memory programmer must be aware of the cost of memory references. The check-in, check-out (CICO) shared-memory programming performance model proposed by Larus *et al.* [13] is a first step in this direction. CICO exposes the communication underlying memory references in cache-coherent shared-memory computers. CICO consists of annotations that a programmer can use to capture the communication underlying shared-memory references and a cost model that attributes a cost to this communication.

The CICO annotations demarcate the point at which a program first reads or writes a shared location and the point at which the program finishes with the location. The model consists of five annotations—check-out exclusive, check-out shared, check-in, prefetch-exclusive, and prefetch-shared. Check-out annotations indicate the need for exclusive or shared access to the cache block containing a specified address. Prefetch annotations indicate the likelihood of an access to the cache block in the near future. The check-

in annotation relinquishes access to the specified cache block. The CICO cost model provides a measure of the communication incurred by non-local data references as well as the cache-coherence protocol overhead required to maintain consistency. The CICO annotations do not affect a program's semantics.

The CICO annotations can also be used as hardware *directives* to a memory system to improve program performance by reducing both communication latency and message traffic. The memory system can use prefetch annotations to reduce latency by overlapping communication with computation. The check-out exclusive annotation can reduce the message traffic caused by write faults when a shared location is first read and then written. The check-in annotation flushes shared data from the cache, thereby reducing the number of invalidate messages. Even in this role as memory system directives, the annotations do not affect a program's semantics.

Most parallel computers provide memory system directives similar to CICO directives. Perhaps the most common is a prefetch instruction. The Kendall Square KSR-1 [11] provides a post-store instruction that broadcasts read-only copies of a cache block to all other nodes that have it allocated but are in the invalid state. This operation is similar, though not identical, to a check-in. Even if a parallel computer does not support CICO directives, a programmer can always use the information from the annotations to restructure a program to improve its performance.

The first step in using the CICO model to compute a program's communication cost is to insert the annotations into a program. The accuracy of the cost calculation depends to a large extent on inserting the annotations so they capture the communication behavior of a program. Inserting CICO annotations requires reasoning about the dynamic behavior of a program and memory system. This is not easy, even for the simplest of programs.

This paper describes a tool—Cachier—that aids this reasoning by automatically inserting CICO annotations into shared-memory programs. Cachier uses a novel approach of combining information about the dynamic behavior of a program, obtained from its execution trace, with static information, obtained from program analysis. The dynamic information enables Cachier to annotate complicated programs that manipulate pointer-based data structures, for which static analysis is infeasible. Even for simpler programs, the dynamic information augments and refines the static information. Since CICO annotations need not be placed perfectly accurately, dynamic information from a single execution of the program is sufficient. The static analysis converts raw data addresses from the trace into references to program variables and presents the annotations in a readable form.

In addition to automatically inserting CICO annotations, Cachier also informs a programmer of potential data races and false sharing. These events are undesirable because their timing-dependent interprocessor communication can cause errors and complicates understanding a program's performance. A programmer can use the information from Cachier to eliminate some of these events.

We measured the usefulness of CICO annotations as memory-system directives by running several benchmarks on a simulator of the Dir_1SW memory system protocol [21]. The CICO annotations inserted by Cachier outperformed both the program without any annotations as well as a hand-annotated version. Interestingly, Cachier performed better on programs with complex, dynamic memory access, which caused programmers the greatest trouble.

The rest of this paper is organized as follows. Section 2 briefly describes the CICO model with an example. Section 3 describes the computing environment in which Cachier operates. Section 4 explains the techniques that Cachier uses to insert CICO annotations. Section 5 illustrates the use of the annotations by a programmer to reduce a program's communication cost. Section 6 evaluates the performance of several benchmarks that were annotated by Cachier. Finally, Section 7 discusses related work.

2 Programming Performance Model

A shared-memory programming performance model should aid a programmer in writing efficient programs by exposing the communication underlying shared-memory references and by providing a measure of the cost of this communication. This section briefly describes the CICO model [10] [13] with the help of an example.

2.1 CICO Model

The CICO model consists of check-out, check-in, and prefetch annotations that a programmer adds to a program and a cost model that uses these annotations to compute a program's shared-memory communication cost. To illustrate the CICO model, we consider a simple example of Jacobi relaxation code on a matrix of size *N x N*. This example is from Hill *et al*. [10]. Assume there are P^2 processors (where *N* is a multiple of *P*), each of which has been assigned a block of the matrix consisting of *N/P* rows, L_{ip} to U_{ip} and *N/P* columns, L_{jp} to U_{jp}. Assume the boundary columns and rows are first copied to local arrays and then the stencil computation is performed locally:

```
for each time step do
  copy boundary rows & columns to local arrays
  for j = Ljp to Ujp do
    for i = Lip to Uip do
      compute stencil on cols  & rows
```

The placement of the CICO annotations depends on the size of the matrix as well as the size of the cache. If the blocked matrix completely fits in the processors cache, the

CICO annotations appear as follows:

```
check_out_X U[L_ip: U_ip, L_jp: U_jp]
for each time step do
  check_out_shared Boundary rows & columns
  copy boundary rows & columns to local arrays
  check_in Boundary rows & columns
  for j = L_jp to U_jp do
    for i = L_ip to U_ip do
      compute stencil on columns and rows
check_in U[L_ip: U_ip, L_jp: U_jp]
```

The communication cost of the program can be found from the annotations. Assume that *b* matrix elements fit in a cache block and the matrix is stored in column major order. The check-out directives for the boundary columns check out *2N/bP* blocks per time step and those for the boundary rows check out *2N/P* blocks per time step, for a total of *2N(1 + b)/bP* blocks per time step. The check-out for the matrix results in N^2/bP^2 blocks being checked out. This is performed once. Thus if the program runs for *T* time steps, the P^2 processors check out a total of $(2NPT(1+b)/b + N^2/b)$ cache blocks.

If the block of the matrix assigned to a processor is too large to fit in the cache, but individual columns of the matrix fit, the annotations appear as follows:

```
for each time step do
  check_out_shared Boundary rows & columns
  copy boundary rows & columns to local arrays
  check_in Boundary rows & columns
  for j = L_jp to U_jp do
    check_out_exclusive U[L_ip: U_ip, j]
    for i = L_ip to U_ip do
      compute stencil on columns and rows
    check_in U[L_ip: U_ip, j]
```

In this case, the P^2 processors check out $(2NP(1+b)/b + N^2/b)$ cache blocks per time step, for a total of $(2NP(1+b)/b + N^2/b)T$ cache blocks, if the program runs for *T* time steps. If the processor cache is too small to hold even a single column, the check-out annotations would appear immediately before a reference and the check-in annotations would appear immediately after a reference. This would suggest blocking the loop to improve program performance [8][12]. In the first version of the program, each processor checked out a total of *N/bP* cache blocks per column of the matrix (ignoring the check-outs for the boundary elements which are anyway the same for both versions), while in the second version, each processor checked out a total of *NT/bP* cache blocks per column.

This example shows how the CICO annotations can be used to compute and understand a programs communication cost as well as suggest ways of restructuring it to reduce this cost.

3 Cachier Overview

Cachier is a tool that automatically inserts CICO annotations into shared-memory programs. Figure 1 shows the

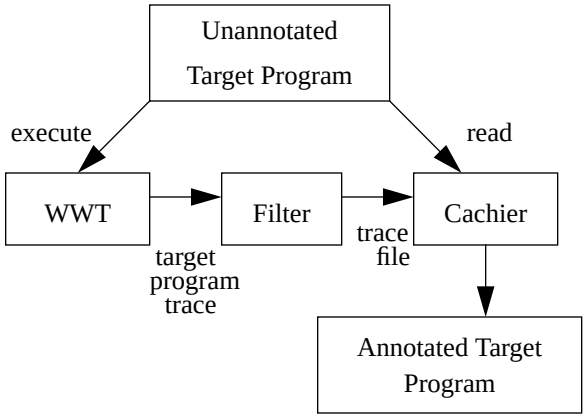

Fig 1. Overview of the Cachier Framework

process that Cachier uses to annotate shared-memory programs. Cachier uses both dynamic and static information in order to effectively insert CICO annotations. Section 4 explains the need for both types of program information, as well as Cachier's techniques for inserting the CICO annotations. This section describes the computing environment in which Cachier operates.

3.1 Target Program Model

We studied programs from the Stanford Splash shared-memory benchmark suite [19]. These programs use barrier synchronizations as their primary synchronization mechanism. The programs also use locks. However, a very small fraction of the program's total computation is performed within lock-unlock intervals, so we ignore locks and concentrate on the program model shown in Figure 2.

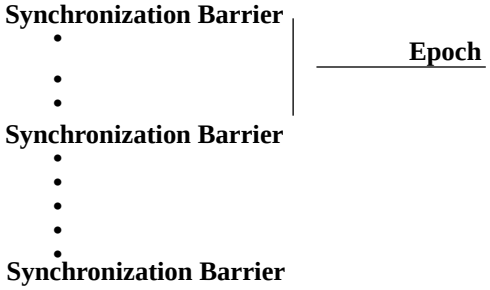

Fig. 2 Program Model

Epochs are code segments that execute between two synchronization events. Our program model consists of epochs demarcated by barrier synchronization points. This is a fairly general model as most parallel computers provide support for barriers. Also most programs using barriers typically perform the bulk of their computation in the intervals between barriers.

3.2 Wisconsin Wind Tunnel (WWT)

The Wisconsin Wind Tunnel (WWT) [18] is an accurate parallel architecture simulator that runs on a Thinking Machines CM-5 computer [20]. It uses a technique called virtual prototyping, by which it only simulates those features of the parallel architecture that are not present in the native hardware. We use it to simulate Dir_1SW [10], [21], which is a cache-coherence directory protocol that has support for programs written within the CICO model. We run the unannotated target program on WWT to generate its execution trace.

3.3 Dynamic Program Information

The dynamic information obtained from a program's execution trace enables Cachier to insert annotations into complicated programs that manipulate pointer-based data structures, which are difficult to analyze statically. Even for programs amenable to static analysis, dynamic information supplements the static information since static analysis alone can produce overly conservative estimates of sharing [1]. The trace file contains information about a cache miss, including its type, the address being accessed, the program counter at that point, the node making the access, and the epoch in which the access occurred (see Figure 3).

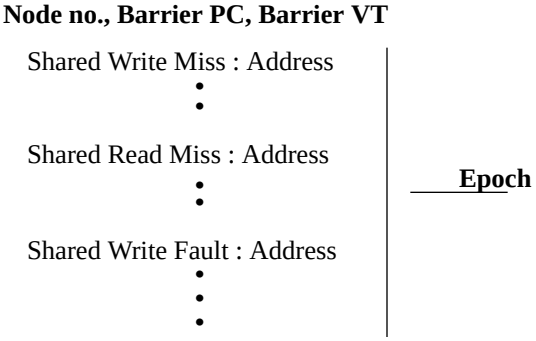

Fig. 3 Trace File Format

Each processor's shared data cache is flushed at every barrier synchronization to improve the quality of the trace data generated, as only accesses that miss in these caches show up in the trace. There is no time ordering of accesses within an epoch. However epochs are ordered by the barrier Virtual Times (VT's).

The information in the trace, such as program counters and addresses, are collected during program execution by WWT and stored in a hash table. At each synchronization barrier in the program, the processors' shared data caches are flushed and information in the hash table is written to the trace file. Collection of trace information may affect a program's behavior in two ways. First, it may affect the behavior of a program that has data races in the following manner. Suppose that in epoch *i*, two processors *X* and *Y* have a data race on a particular variable. Say processor *X* used that variable in the previous epoch. Since the shared data caches are flushed at every epoch boundary, processor *Y* may end up with the variable in its cache in the next epoch, rather than the other way around. This may cause the program to generate different results. Collection of trace information also slows the program's execution. On a simulator like WWT, time dilation does not affect the program's behavior.

3.4 Cachier

The input to Cachier consists of an unannotated target program and its trace file. Cachier parses the unannotated target program and constructs its abstract syntax tree and control flow graph. Cachier combines both the static and dynamic program information to determine which CICO annotations are to be inserted. It modifies the program's abstract syntax tree to include the annotations and produces an annotated target program by unparsing this modified abstract syntax tree. The annotated target program is the same as the unannotated target program, except for the CICO annotations inserted by Cachier.

4 Inserting CICO Annotations

This section describes the techniques used by Cachier to insert CICO annotations into shared-memory programs and illustrates them with an example. In order to insert annotations, three key questions have to be answered—what to CICO?, where to CICO?, and how to CICO? These questions are answered in Sections 4.1, 4.2 and 4.3 respectively. Section 4.4 provides an example to illustrate these techniques. Section 4.5 discusses a few issues related to the technique Cachier uses to insert the annotations.

Cachier operates in two distinct phases. In the first phase, Cachier processes and assimilates information about the epoch from the trace file and determines the annotations. Trace processing consists of removing addresses involved in shared write faults from the list of shared read misses, updating the list of shared write misses to include addresses involved in shared write faults, and storing labelling information contained in the trace to aid mapping addresses to program data structures. Cachier also determines locations involved in data races and false sharing. A potential data race exists if two or more processors access the same address within the same epoch and at least one access is a write. False sharing results from two or more processors accessing different addresses in the same cache block. Cachier next uses the equations described in Section 4.1 to compute addresses to be checked-out exclusive, checked-out shared and checked-in. Finally, Cachier uses static information from program analysis along with the labelling information in the trace to map addresses to pro-

gram data structures and program counters to lines in the program text.

In the second phase, Cachier uses this information to place these annotations in a readable form, as described in Sections 4.2 and 4.3.

4.1 Choosing CICO Annotations

CICO annotations serve two roles. They allow a programmer to reason about the communication in his program and also permit the memory system to improve program performance. To be useful for reasoning about communication, the annotations have to expose all communication. On the other hand, to improve program performance we want to optimize the annotations by removing unnecessary annotations wherever possible. To satisfy these conflicting goals, Cachier produces either Programmer or Performance CICO annotations.

For each epoch, Cachier determines the set of locations that should be checked-out, including their mode—shared or exclusive—and the set of locations to check-in. To find these sets for epoch *i*, Cachier uses the following set of equations.

$$co\_x\ [epoch\ i] = \overline{DRFS}\ \{\ SW_i - SW_{i-1}\ \} + DRFS\ \{\ SW_i\ \}$$
$$co\_s\ [epoch\ i] = \overline{FS}\ \{\ SR_i - SR_{i-1}\ \} + FS\ \{\ SR_i\ \}$$
$$ci\ [epoch\ i] = \overline{DRFS}\ \{\ S_i - S_{i+1}\} + DRFS\ \{\ S_i\ \}$$

where:

- *i-1* is the previous epoch and *i+1* is the next epoch,
- *co_x*, *co_s* and *ci* are the locations that should be checked-out exclusive, checked-out shared and checked-in respectively,
- SW_i is shared write misses$_i$ + shared write faults$_i$ (in *epoch i*),
- SR_i is shared read misses$_i$ - shared write faults$_i$ (in *epoch i*),
- $S_i = SW_i + SR_i$,
- *DRFS* is a function on a set of addresses that returns those addresses that are either involved in a data race or in false sharing. ($\overline{DRFS}$ is its complementary function)
- *FS* is a function on a set of addresses that returns a subset of those addresses that are involved in false sharing. ($\overline{FS}$ is its complementary function).

The basic idea behind these equations is that if there is either a data race or false sharing on a location's cache block, then a processor should check it out and check it back in immediately. The rationale is that since multiple processors are contending for this block, it will remain in a processor's cache only for a short time before another processor claims it. On the other hand, if a location is not involved in data races or false sharing, then a processor should check it out only if it was not checked out in the previous epoch by the same processor. Similarly, a processor should check-in a location only if it is not going to use it again in the next epoch. This annotation placement models caches and helps to eliminate many unnecessary check-in, check-out pairs at epoch boundaries. Using only a single epoch history simplifies the calculations. Moreover, since an epoch performs a large amount of computation, a variable left unused in the cache for multiple epochs is very likely to be replaced before it can be reused.

To find Performance CICO annotations for each epoch, Cachier uses these equations:

$$co\_x\ [epoch\ i] = \overline{DRFS}\ \{\ shared\ write\ fault_i - SW_{i-1}\ \} + DRFS\ \{\ shared\ write\ fault_i\ \}$$
$$co\_s[epoch\ i] = \{\}$$
$$ci\ [epoch\ i] = \overline{DRFS}\ \{\ SW_i - SW_{i+1}\} + \overline{DRFS}\ \{\ SR_i \cap SW_{i+1}\} + DRFS\ \{\ S_i\ \}$$

where the notation is the same as above.

The Dir$_1$SW protocol [10][21] that uses CICO annotations as memory directives performs an implicit check-out exclusive at each shared write miss and an implicit check-out shared at each shared read miss. Placing explicit check-out's for these cases reduces performance because of the overhead of the additional operation. However, many locations are read before being written, which results in their being in the cache read-only at the time of the write. An explicit check-out exclusive, before the read, can eliminate the extra message traffic to upgrade a shared to an exclusive copy. These are the only locations Cachier checks out (unless, of course, they were already checked out in the previous epoch).

The check-in annotations inserted by Cachier at the end of epochs has three parts. The first are shared locations, not involved in either data races or false sharing, that were written to in the current epoch, and are not going to be written by the same processor in the next epoch. The second are shared locations, again not involved in either data races or false sharing, that were read by some processor in the current epoch and which will be written by some processor in the next epoch. The last are shared locations that were involved in either a data race or in false sharing in the current epoch.

To make these ideas clearer, consider the example in Figure 4.

Using the equations for Programmer CICO, Cachier finds the following CICO annotations for epoch i: co_s(c), co_s(a) & ci(c), ci(d). The Performance CICO annotations for the same epoch is just ci(c). If epoch i-1 was the first epoch in the program, then the Programmer CICO for that epoch will be as follows: co_x(a), co_x(b), co_s(d) & ci(a). The Performance CICO for the same epoch will be just ci(a). The check-in for a is necessary as there is a potential data race on that variable (the trace does not maintain any ordering of accesses within an epoch).

4.2 Placement of CICO Annotations

The placement of CICO annotations depends on the role

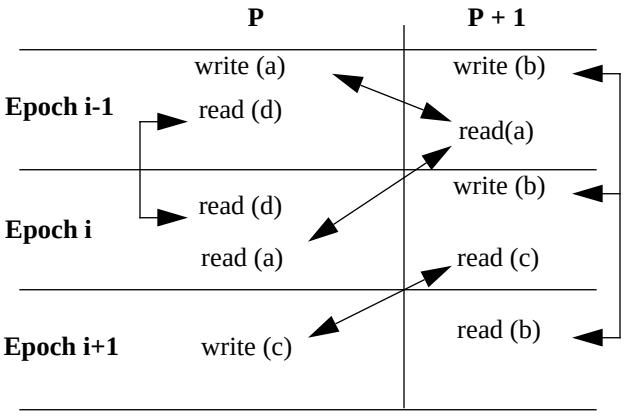

Fig. 4

they serve, whether the location is involved in either a data race or false sharing, and the relative sizes of the data set and the shared data cache. Static program information guides the decisions. Cachier models the finite capacity of a cache (but not its limited associativity) to improve its placement of CICO annotations.

In the case of Programmer CICO, Cachier tries to place check-out annotations as close to the beginning of an epoch and check-in annotations as close to the end of an epoch as possible under the cache size constraints. This placement facilitates use of these annotations by the programmer to reason about a program. Since an epoch can span multiple functions, Cachier uses static program information to place check-out annotations close to the beginning of the functions in which the locations are referenced and check-in annotations close to the end of these functions, again subject to cache size constraints.

In the case of Performance CICO, Cachier tries to place all annotations as close to the accesses as possible in an attempt to reduce interprocessor communication. Since a naive attempt to do this will result in code size explosion, it uses static information about the program, especially the loop structure to present the annotations in a readable form.

4.3 Presentation of CICO Annotations

For CICO annotations to be readable by a programmer, they must be presented in a compact, easily understandable form. To achieve this goal, Cachier uses static program information, obtained from its control flow graph and abstract syntax tree, as well as some information from the program's trace.

In the case of shared read misses, it may not be always possible to map an address to a program variable by examination of the line. For example, consider the following line:

```
C[i, j] = C[i, j] + A[i, k] * B[k, j]
```

To map a shared read miss on this line of code to a particular variable, further information is required. In such cases, Cachier uses another utility which allows labelled regions of memory to be mapped onto program data structures. The programmer uses a macro to label a continuous region of shared-memory with a name. To use Cachier, a programmer must label all important shared data structures.

Cachier uses the program's abstract syntax tree to analyze its loop structure. This information helps structure the CICO annotations in a form that makes it easy for the programmer to read the annotations. This process involves collapsing annotations, either by placing them inside program loops, or by generating new loops for them. To illustrate this step, consider the following piece of code. To its right is the result of naive insertion of CICO annotations followed by Cachier's more sophisticated insertion.

```
for i = 1 to N step 2 do
  A[i] = ....
od
 .....
for i = 1 to N do
  A[i] = ....
od
```

```
for i = 1 to N step 2 do
  check_out_X A[i]
  A[i] = ...
  check_in A[i]
od
for i = 1 to N do
  check_out_X A[i]
  A[i] = ...
  check_in A[i]
od
```

```
for i = 1 to N step 2 do
    check_out_X A[i]
    A[i] = ...

for i = 2 to N-1 step 2 do
    check_out_X A[i]

for i = 1 to N do
    A[i] = ...
    check_in A[i]
```

Moreover, since an epoch can be executed multiple times, Cachier ensures that the annotations are not duplicated. Cachier also flags data races and false sharing, to enable the programmer to use locks in the case of data races or pad the relevant data structures in the case of false sharing, to alleviate the problem.

4.4 Example Cachier Annotations

Consider the following example which performs matrix multiplication of two dense matrices, each of size N x N using an unconventional technique explained below. For simplicity, N is a multiple of P, the square root of the number of processors and each processor is assigned a block of rows, L_{kp} to U_{kp}, and columns, L_{jp} to U_{jp}, of the B matrix.

```
for i = 1 to N do
    for k = L_kp to U_kp do
        t = A[i, k]
        for j = L_jp to U_jp do
            C[i, j] = C[i, j] + t * B[k, j]
```

Figure 5 illustrates the technique used to multiply the matrices.

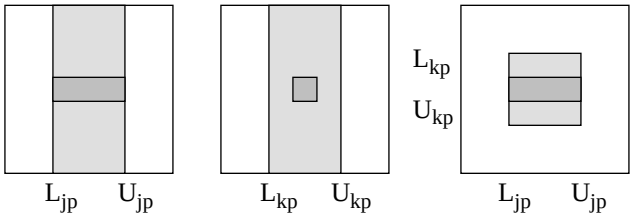

Fig. 5

Each processor is assigned a block of the B matrix which is not shared. The A matrix is read shared by the processors and the C matrix (result matrix) is read as well as write shared. This follows from the technique used to multiply the matrices in which each processor updates the result matrix with the values it computes.

In the case that the matrix size and the cache size are such that the entire matrix does not fit in the processor's cache but individual rows/columns do, Cachier inserts the following CICO annotations.

These are the Programmer CICO annotations inserted by Cashier

```
for i = 1 to N do
    for k = Lkp to Ukp do
        check_out_S A[i, k]
        t = A[i, k]
        check_out_S B[k, Ljp : Ujp]
        for j = Ljp to Ujp do
            check_out_X C[i, j]
            /***  Data Race on C[i, j]   ***/
            C[i, j] = C[i, j] + t * B[k, j]
            check_in C[i, j]
        check_in B[k, Ljp : Ujp]
        check_in A[i, k]
```

For the case of Programmer CICO, Cachier inserts annotations to check-out shared matrices A and B as they are only read. Matrix C which is read as well as written is checked-out exclusive. The data race on elements of matrix C is flagged and the check-out/ check-in annotations for these elements are placed as close to the reference as possible. On the other hand, since elements of matrices A & B are not involved in a data race, their corresponding check-out (check-in) annotations are placed as close to the beginning (end) of the epoch as is possible under cache size constraints. The notation $L_{jp} : U_{jp}$ indicates that the annotation is in a loop generated by Cachier.

The Performance CICO annotations inserted by Cachier look as follows

```
for i = 1 to N do
    for k = Lkp to Ukp do
        t = A[i, k]
        for j = Ljp to Ujp do
            check_out_X C[i, j]
            /***  Data Race on C[i, j]   ***/
            C[i, j] = C[i, j] + t * B[k, j]
            check_in C[i, j]
```

In this case the check-out shared annotations are absent as Dir_1SW performs an implicit check-out shared on each shared read miss. So an explicit check-out shared annotation would just result in an overhead due to address generation translation. However the check-out exclusive annotation for matrix C is still present because it incurs a shared write fault, which would have otherwise upgraded a shared copy of the block to be writable. The check-in annotation for matrix C is placed immediately after it is referenced, due to the presence of the data race. The check-in annotations for matrices A and B are omitted as they are not write shared.

4.5 Discussion

CICO annotations do not affect a program's semantics. Thus, even if the annotations are inserted at inappropriate points in the program, they only affect its performance. Also while it is conceivable that the instrumentation added to trace the program may substantially alter its memory access pattern causing Cachier to insert the annotations at inappropriate places, we have not observed such behavior in practice. Cachier can use dynamic information obtained from a single execution of the program to place annotations as the CICO annotations are not required to be perfectly accurate.

Cachier combines dynamic information obtained from a single execution with static analysis of the program. The alternative would have been to use a training set rather than a single input data set to obtain dynamic program information. However we found that the difference between executing a Cachier annotated program on the same input data set used to generate the dynamic information as opposed to executing the program on a different data set was small (< 2%) even for a dynamic application like *Barnes*. We believe this is due to two reasons. Firstly, Cachier does not rely solely on the dynamic information obtained, but combines this with a static analysis of the program source. Secondly, it appears that even dynamic applications are not all that dynamic as far as memory access patterns are concerned. Moreover, other measurements show that program behavior is typically independent of the input data set [7].

5 Restructuring with CICO

This section illustrates how the CICO annotations inserted by Cachier can be used to restructure a program. We do this using the same matrix multiply example from the previous section. The annotations inserted by Cachier indicate that the communication bottleneck is due to the

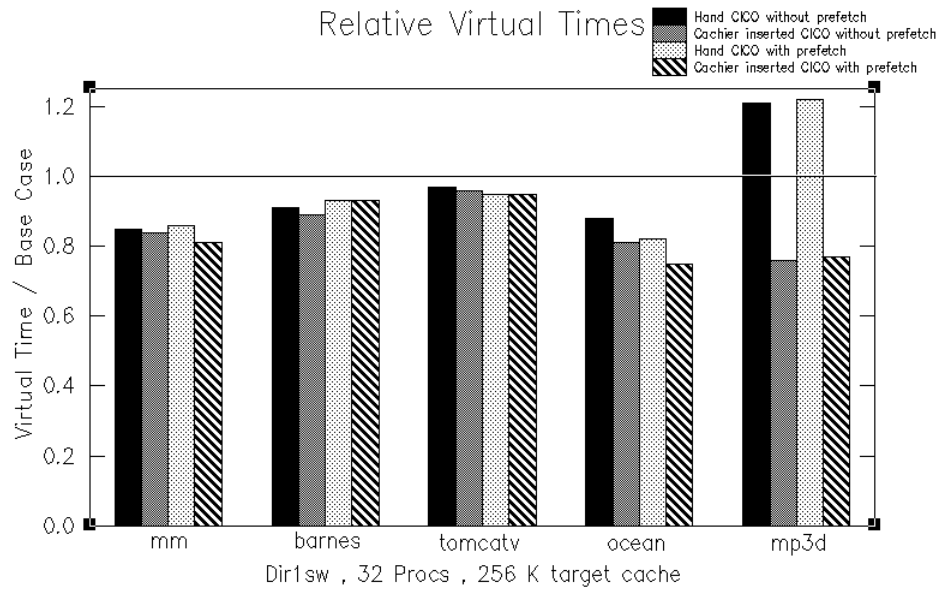

Fig. 6

cache block race on elements of the result matrix. Moreover, this race can cause an incorrect result due to multiple processors reading the same value of the C matrix at the same time, modifying it, and writing it back. This race is compounded by the fact that a single cache block contains multiple adjacent elements of the result matrix (in this case 4 elements). Since a cache block is the minimum granularity at which an element can be checked out, a solution would be to restructure the program as follows. First each processor copies the portion of the C matrix that it will be updating into a local array. Each processor then performs the computation on the C matrix locally, and finally copies back its local portion of the C matrix.

```
for i = 1 to N do
    for j = L_jp to U_jp step 4 do
        check_out_S C[i, j]
        C_p[i, j : j + 3] = C[i, j : j + 3]
        check_in C[i, j]
for i = 1 to N do
    for k = L_kp to U_kp do
        t = A[i, k]
        for j = L_jp to U_jp do
            C_p[i, j] = C_p[i, j] + t * B[k, j]
 for i = 1 to N do
    for j = L_jp to U_jp step 4 do
        lock C[i, j]
        check_out_X C[i, j]
        C[i, j : j + 3] = C[i, j : j + 3] + C_p[i, j : j + 3]
         check_in C[i, j]
         unlock C[i, j]
```

The original program had a total of N^3 (N * N/P * N/P * P^2) check-outs for elements of matrix C on which there is a cache block race. The restructured program only has $N^2P/2$ (2 * N * N/4P * P^2) check-outs for elements of matrix C out of which there is a cache block race on only $N^2P/4$ of them which is protected by a lock.

6 Performance of Automatic CICO

This section compares the performance of several unannotated shared-memory programs against hand-inserted CICO and Cachier-annotated CICO versions of the same program. The hand CICO was carefully done over a period of a few weeks with the aid of existing profiling tools by individuals with a detailed understanding of the problem and cache-coherence protocol. Cachier produced the automatic CICO version. All simulations were run on the Wisconsin Wind Tunnel (WWT) [18]. The simulated computer consists of 32 processor nodes, each containing a processor, shared-memory module, cache, and network interface. The cache is 256 KB, 4-way set-associative with a cache block size of 32 bytes. We used WWT to simulate a directory-based Dir_1SW cache-coherence protocol [10] [21].

For this evaluation we use five benchmarks: *Barnes*, *Ocean*, *Mp3d* (from the SPLASH Benchmark suite [19]), *Matrix Multiply,* and *Tomcatv* (a parallel version of the SPEC Benchmark). *Barnes* performs a gravitational N-body simulation using the Barnes-Hut algorithm. We simulated it for a data set of size 1024 bodies. *Ocean* performs a cuboidal ocean basin simulation using Gauss-Seidel with Successive Over Relaxation. We simulated it for a grid size of 98 x 98. *Mp3d* simulates rarefied fluid flow of idealized diatomic molecules in a three-dimensional active space. We simulated it for 50,000 molecules and 10 time steps. *Matrix Multiply* multiplies two matrices by dividing them into blocks. We simulated it for a matrix size of 256

x 256. We simulated *Tomcatv* for 10 iterations on a grid of size 1024 x 1024. The input data sets used to obtain the execution trace for Cachier were different than the data sets used in the performance comparison.

Figure 6 displays the execution times of these programs, normalized to the version without CICO annotations. For *Matrix Multiply*, the CICO annotations (without the prefetch annotation) inserted by Cachier show a 16% improvement in performance, as compared to the version without CICO annotations and a slight improvement over the hand-annotated version. In this program, one processor initializes the matrices with random values. Part of the improvement arises from checking-in these matrices after initialization. Also, the result matrix is read-write shared by the processors, so checking-out the required matrix elements exclusive eliminates upgrades of shared blocks to be writable. In addition, checking in the result values after a processor computes them reduces the number of invalidation messages that have to be sent. The small difference in performance between the hand-annotated and Cachier annotated versions is due to a few unnecessary annotations in the former. Using the prefetch annotation, Cachier improves the program performance by around 20%. In the hand-annotated version of the program, the prefetch annotations were inappropriately placed.

For *Barnes* as well, the version of the program annotated by Cachier outperforms the version without any annotations by around 11% and the hand-annotated version by 2%. In this case the hand-annotated version missed a few annotations. The prefetch annotations are not very successful in further improving performance due to the program's complicated pointer data structures.

For *Tomcatv*, the CICO annotations do not have a large effect on its performance as it performs little communication relative to its computation (around 90% of its execution time is spent in computation). For *Ocean*, the annotations inserted by Cachier improve program performance by around 20% without prefetch, and by 25% with prefetch. This is also a 7% improvement over the hand-annotated version in both cases. For *Mp3d*, the Cachier annotated version outperforms the unannotated version by 25% and the hand-annotated version by 45%. The hand-annotated version suffers from both checking-in cache blocks too early at certain places, (i.e., before a processor finished with the block) as well as neglecting to check-in blocks at other places.

These results emphasize the difficulty in hand-inserting CICO annotations, especially for programs with dynamic memory access patterns. They also show that Cachier's annotations are successful in improving program performance, even for complicated programs like *Barnes* and *Mp3d* that contain pointer-based data structures and dynamic memory access patterns. This performance improvement is due to a reduction in the time spent servicing shared data cache misses and write faults as well as a reduction in the number of these events. The greatest performance improvement is obtained for *Ocean* and *Mp3d*, both of which have the highest degree of sharing among the Splash benchmarks. In *Ocean*, 88% of loads read shared data and 68% of the stores write shared data, whereas for *Mp3d*, the corresponding numbers are 71% (shared reads) and 80% (shared writes) respectively [19]. On the other hand, in *Barnes*, where the performance improvement is not as large, the degree of sharing is much lower—25.5% of the loads are shared data reads and only 1.3% of the stores are shared data writes [19].

Moreover, Cachier-annotated versions of the programs consistently outperformed the hand-annotated versions, which shows that inserting annotations by hand is not an easy task, especially for programs with dynamic memory access patterns. In addition, since different input data sets were used to insert the annotations and to compare performance, the results show that Cachier's annotations are not overly specialized to a particular execution.

7 Related Work

Inserting CICO annotations appears similar to inserting primitives for software cache coherence [4][5][15]. The crucial difference is that coherence primitives must be conservatively inserted. To ensure correct execution, software cache coherence schemes must invalidate data along all possible program execution paths. Moreover, the schemes cannot use dynamic program information and rely solely on conservative static analysis. CICO annotations, on the other hand, do not affect a program's semantics and hence Cachier can aggressively insert annotations by combining dynamic program information with static program analysis.

Other work studied how to prefetch data so as to overlap communication with computation and reduce communication latency [2][9][16]. However, these schemes relied solely on static program analysis and were able to prefetch data only in scientific codes with fairly static memory access patterns that a compiler can analyze. Cachier uses dynamic information as well, which works in more circumstances. It also uses check-ins to flush data from a processor's cache. This results in a reduction in message traffic as well as communication latency.

Techniques for race detection in the context of debugging programs have either used dynamic information from a program's execution trace or static information from an analysis of the program text [17]. A few techniques have used dynamic information as well as static information [6]. However the static information supplements the dynamic information by ruling out races in certain parts of the program, thereby precluding the need to trace those parts. The

actual race detection uses dynamic information. The dynamic information used for race detection is similar to that used by Cachier except for a couple of key differences. The trace file used by Cachier does not contain all shared memory locations read and written, rather it contains only those that cause cache misses. Also while the trace has a relative ordering between synchronization events, there is no ordering maintained on other events (i.e. shared data cache misses) between two synchronization events (i.e. an epoch).

8 Conclusions

The CICO model is a practical shared-memory programming performance model. However, it requires a programmer to reason about a program's dynamic behavior and the memory system, which can be difficult. This paper describes Cachier, a tool for automatically inserting CICO annotations into shared-memory programs. It uses a novel approach of combining information about the dynamic behavior of a program, from its execution trace, with static information from an analysis of the program source. The resulting CICO annotations can be both read by a programmer to help in reasoning about communication in the program, as well as used by a memory system to improve the program's performance. In experiments on several benchmarks, CICO annotations inserted by Cachier outperformed both unannotated as well as hand-annotated versions of the programs.

Acknowledgements

We would like to thank Alvy Lebeck for the filter used to generate a program's execution trace. Also Babak Falsafi and Alvy Lebeck provided valuable help with the Wisconsin Wind Tunnel simulator. Satish Chandra, Babak Falsafi, Alvy Lebeck and Shubu Mukherjee provided the hand-annotated versions of the benchmarks.

References

[1] Sarita V. Adve, Vikram S. Adve, Mark D. Hill, and Mary K. Vernon, "Comparison of Hardware and Software Cache Coherence Schemes", *Proceedings of the 18th Annual International Symposium on Computer Architecture*, (June, 1991), pp. 298-308.

[2] David Callahan, Ken Kennedy, and Allan Porterfield, "Software Prefetching", *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, (April, 1991), pp. 40-52.

[3] David Chaiken, John Kubiatowics, and Anant Agarwal, "LimitLESS Directories: A Scalable Cache Coherence Scheme", *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, (April, 1991), pp. 224-234.

[4] J. Cheong, and A.V. Veidenbaum, "A Cache Coherence Scheme with Fast Selective Invalidation", *Proceedings of the 15th Annual International Symposium on Computer Architecture*, (June, 1988), pp. 299-307.

[5] Ron Cytron, Steve Karlovsky, and Kevin P. McAuliffe, "Automatic Management of Programmable Caches", *Proceedings of the 1988 International Conference on Parallel Processing* (Vol. 2 Software), (Aug., 1988), pp. 229-238.

[6] Perry A. Emrath, Sanjoy Ghosh, and David A. Padua, "Event Synchronization Analysis for Debugging Parallel Programs", *Supercomputing '89*, (Nov., 1989), pp. 580-588.

[7] J. A. Fisher, and S. M. Freudenberger, "Predicting Conditional Branch Directions from Previous Runs of a Program", *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, (Sept., 1992), pp 85-95.

[8] Dennis Gannon, William Jalby, and K. Gallivan, "Strategies for Cache and Local Memory Management by Global Program Transformation" *Journal of Parallel and Distributed Computing*, (Vol. 5, 1988), pp. 587-616.

[9] E. Gornish, E. Granston, and A. Veidenbaum, "Compiler Directed Data Prefetching in Multiprocessors with Memory Hierarchies", *International Conference on Supercomputing*, 1990.

[10] Mark D. Hill, James R. Larus, Steven R. Reinhardt, and David A. Wood, "Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors", *ACM Transactions on Computer Systems*, (Nov., 1993), pp. 300-318.

[11] Kendall Square Research, *Kendall Square Research Technical Summary*, 1992.

[12] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolfe "The Cache Performance and Optimizations of Blocked Algorithms", *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, (April, 1991), pp. 63-74.

[13] James R. Larus, Satish Chandra, and David A. Wood, "CICO: A Practical Shared-Memory Programming Performance Model", *Workshop on Portability and Performance for Parallel Processing*, (July, 1993), To appear: Ferrante & Hey eds., *Portability and Performance for Parallel Proces*sors.

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

[15] Sang Lyul Min, and Jean-Loup Baer, "A Timestamp-based Cache Coherence Scheme", *Proceedings of the 1989 International Conference on Parallel Processing* (Vol. 1 Architecture), (Aug., 1989), pp 23-32.

[16] Todd C. Mowry, Monica S. Lam, and Anoop Gupta, "Design and Evaluation of a Compiler Algorithm for Prefetching", *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, (Sept., 1992), pp. 62-73.

[17] Robert H. Netzer. Race Condition Detection for Debugging Shared-Memory Parallel Programs. Ph.D. thesis, University of Wisconsin-Madison, 1991.

[18] 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", *Proceedings of the 1993 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, (May 1993), pp. 48–60.

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

[20] Thinking Machines Corporation. *The Connection Machine CM-5 Technical Summary*, 1991.

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