Automatic Software Cache Coherence through Vectorization

Ervan Darnell
John M. Mellor-Crummey
Ken Kennedy

CRPC-TR-92197-S
January 1992

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

Appeared in Proceedings of the ACM 1992 International Conference on Supercomputing, Washington, DC, July 1992.

Automatic Software Cache Coherence through Vectorization [*][†]

Ervan Darnell John M. Mellor-Crummey Ken Kennedy

Computer Science Department
Rice University
Houston, TX 77251-1892

Abstract

Access latency in large-scale shared-memory multiprocessors is a concern since most (if not all) memory is one or more hops away through an interconnection network. Providing processors with one or more levels of cache is an accepted way to reduce the average access latency; however, in a multiprocessor, cached values must be kept coherent for the multiprocessor to support the abstraction of a shared global memory. There is no generally accepted hardware solution to provide cache coherence for large-scale shared-memory multiprocessors. Software coherence strategies offer scalability with current hardware. In this paper we examine a compiler-based software strategy for maintaining cache coherence that relies on dependence analysis and a vectorization algorithm to insert cache control directives. Experiments on the BBN TC2000 for a pair of numerical problems show that the run-time cost of coherence using our strategy is less than that for previously proposed compiler-based software methods and suggest that it should compare favorably with proposed hardware schemes.

1 Introduction

Shared-memory multiprocessors are composed of a collection of processors and memories linked by an interconnection network. In scalable machines, the network necessarily consists of multiple layers. With respect to any particular processor, most (if not all) memory is one or more hops away through the interconnect. For this reason, access latency is an important concern.

Approaches for coping with access latency in shared-memory multiprocessors include multithreading [2] and caching. In this paper we focus on using caching to reduce access latency. Caching can reduce the impact of network latency on average access latency by reducing the number of accesses that must traverse the interconnect. Reducing network traffic also reduces the likelihood of contention which can otherwise degrade performance.

Using caches in shared-memory multiprocessors introduces the need for cache coherence. To avoid changing the semantics of a program execution through the use of caches, memory must retain the appearance of *sequential consistency* [10]. By itself, caching hardware for uniprocessors is not sufficient for multiprocessors since uniprocessor caches have no knowledge of other processors' actions.

Most approaches to the cache coherence problem have focused on hardware mechanisms to maintain coherence. Unfortunately, the overhead of maintaining coherence in hardware can be high; scaling systems based on hardware coherence is a difficult problem [13]. Snoopy cache schemes, which monitor accesses broadcast on the processor-memory interconnect, are now in common use for small scale systems [14, 16]; however, snoopy schemes are problematic for large-scale machines because such machines cannot be based on a single, central broadcast medium for lack of sufficient bandwidth. Directory schemes [3, 11, 17], in which a directory entry associated with each memory location indicates which processors have cached values for that location, seem more promising for large-scale systems. However, directories can require large amounts of additional storage and directory maintenance operations may substantially increase network traffic. Others researchers suggest that caches include version number based support for coherence[4, 12]. Drawbacks to these schemes include dedication of precious cache real-estate to version numbers (decreasing the amount of useful data that the cache can hold), and the additional hardware complexity.

A promising alternative to hardware-based solutions for coherence is to use compilers to analyze programs and automatically augment them with calls to coherence operations where necessary. Coherence operations include UPDATEs, which write modified values from cache to main memory without evicting them, and INVALIDATEs, which remove values from cache forcing them to be fetched from main memory the next time they are accessed. Compiler-based coherence techniques require only minimal support from cache hardware. The hardware need only provide a mechanism to enable software control of INVALIDATEs and UPDATEs; currently available cache units such as the Motorola 88200 satisfy this requirement. With compiler-based approaches, caches need not have any global knowledge of other processors' actions: they can use caches otherwise intended for uniprocessor architectures. Furthermore, compiler-based techniques that use high-level information about the program, in particular the dependence structure (i.e., knowledge of how values flow and when storage is reused), can

[*]This work was supported in part by the National Science Foundation under Cooperative Agreement CCR-8809615 and ASC-8518578, and by IBM Corporation through its research contracts with the Department of Computer Science at Rice University.

[†]This paper appears in the Proceedings of the 1992 International Conference on Supercomputing

relax sequential consistency; they only need to maintain apparent coherence. Such techniques can defer achieving coherence until a more opportune time or determine that certain values are dead on a given processor and that their coherence need not be enforced. The limitation of such approaches is that cache control is based on compile-time decisions instead of run-time decisions; the imprecision of compile-time analysis may cause some potential run-time reuse in cache to be missed.

In this paper, we describe a compiler-based approach that maintains cache coherence in software for programs with fork-join parallelism expressed in the form of nested parallel loops. Our technique improves upon previous compiler-based approaches by carefully analyzing exactly which data requires cache control, aggregating cache control operations to amortize overhead, and adjusting the position of cache control operations in the code to facilitate run-time reuse of cached data.

Section 2 provides an overview of data dependence which is necessary for the exposition of our compiler-based software coherence strategy. Section 3 discusses a general framework for compiler-based software cache coherence and previous research in the area. Section 4 describes our approach and presents our vectorization-based algorithm for adding cache control operations to a program with loop-based parallelism. Using an example, we explain the behavior of our algorithm and differentiate its results from those of other compiler-based coherence strategies described in the literature. Section 5 presents some preliminary experimental results on the BBN TC2000 that compare the performance of two application kernels using our cache-coherence strategy to their performance using alternative approaches. Section 6 presents a summary and conclusions. In section 7 we outline several unresolved issues to be investigated in future work.

2 Data Dependence

A data dependence exists between two statements S_1 and S_2 if there is a path from S_1 to S_2 and both statements access the same location in memory. There are four types of data dependence [8, 9]:

True (flow) dependence S_1 writes a memory location that S_2 later reads.

Anti dependence S_1 reads a memory location that S_2 later writes.

Output dependence S_1 writes a memory location that S_2 later writes.

Input dependence S_1 reads a memory location that S_2 later reads.

Compile-time dependence analysis computes a conservative superset of a program's run-time dependences. A program's dependence graph contains a node for each program statement and an edge for each dependence.

A data dependence between statements S_1 and S_2 is *carried* by a loop if the execution of S_1 in loop iteration i can potentially access the same memory location as the execution of S_2 in loop iteration j, $i \neq j$. The nesting depth of the outermost loop which carries a dependence is said to be the *carrying level* of the dependence. For instance, in

```
DO I = 1, N
  DO J = 1, N
    A(I,J) = A(I,J-1)
  ENDDO
ENDDO
```

there is a true dependence from the write of `A(I,J)` to the read of `A(I,J-1)`. This dependence is carried by the `J` loop and the carrying level of the dependence is 2. Dependences that are not carried by loops are said to be *loop independent*.

For programs with parallel constructs, a dependence is called *processor crossing* [5] when the statement endpoints of a dependence may execute on different processors. Processor crossing dependences indicate when a value may be accessed by more than one processor. Compiler-based software coherence methods must consider processor crossing true, anti, and output dependences.

3 Software Cache Coherence

Compiler-based algorithms for software cache coherence are based on the following principle: processor-crossing true dependences must be augmented with coherence operations to ensure that values flow between processors. The WRITE in a processor-crossing true dependence must be followed by an UPDATE to send the new value to main memory; the READ must be preceded by an INVALIDATE if it is possible that the cache contains a stale value. To ensure values flow properly when a true dependence is present, previous work has focused on placing an INVALIDATE between the write of a value and its subsequent read. Here, we describe the notion of *access triples* as a more precise framework for compiler-based coherence problem.

For a value to be stale in a cache and not merely the object of a processor crossing dependence, the following sequence of operations must occur:

1. READ or WRITE on processor i
2. WRITE on processor j, $j \neq i$
3. READ on processor i

We call such a sequence an access triple. Assume the WRITE is followed by an update. With no other coherence operations, the READ in step 3 may see a stale value left in cache by the READ in step 1. Without step 1, the READ in step 3 would either find the correct value in cache or no value at all (and fault in the new value). All that is necessary to maintain coherence is to break each access triple on *either* dependence, the one to the WRITE or the one from the WRITE. Previous approaches add an INVALIDATE on processor i between steps 2 and 3 after the READ is scheduled to ensure that the stale value is cleared from cache. It would be equally correct to add an INVALIDATE on processor i between steps 1 and 2; purging the value from i's cache before the WRITE also ensures that a stale value is not present in cache after step 2.

The notion of access triples has a natural extension for accesses to arrays. The READ and WRITE operations in steps 1–3 of an access triple can represent access to arbitrary sections of an array. Assume that an update following step 2 ensures that all of the values modified by the WRITE are written to main memory. As before, to maintain coherence, an INVALIDATE on processor i must be added. However, the data to be invalidated need only subsume the intersection of the three sections that are touched by the accesses in the triple; it does not necessarily need to include all that is read nor all that is written.

Like previous approaches, we consider programs with loop-based parallelism that do not contain explicit synchronization. As in previous work, in this paper we focus on maintaining coherence based on processor crossing dependences and reserve a more general treatment of access triples (which requires more global program analysis) for future work. Thus, from this point on through section 5, we only consider placement of invalidates between the write and subsequent read in a true dependence. We assume that the run-time mapping of parallel loop iterations to processors is unknown at compile time. Furthermore, we assume that parallel loops do not carry any true, anti, or output dependences. Such dependences indicate that the data written by one loop iteration is not independent of the data accessed by other iterations of the loop (i.e., race conditions exist). For programs that satisfy this restriction, a processor crossing dependence must flow across the start or end of a parallel loop. In particular, a dependence crosses processors when it is carried by a serial loop outside of a parallel loop, or when it links a pair of statements with one of the statements nested inside a parallel loop and the other statement in a serial region or a different parallel loop.

For expository purposes, consider each serial section (a code section not enclosed by any parallel loop) to start with a dummy FORK and end with a dummy JOIN. With this assumption, the sequence of operations to ensure proper coherence for a processor-crossing true dependence is WRITE, UPDATE, JOIN, FORK, INVALIDATE, READ. Operations on unrelated data can be interleaved arbitrarily within this sequence. If complete knowledge were available, the best spot to place the INVALIDATE would be immediately after the FORK. Moving the INVALIDATE closer to the READ may remove an opportunity for reuse of a value since there may be some other reference that brings the value into cache before the INVALIDATE.

Moving the INVALIDATE closer to the FORK to facilitate reuse can make it difficult to issue a precise INVALIDATE, particularly if the INVALIDATE is moved to a point prior to calculation of subscripts used to determine the location of the READ. Such imprecision does not affect correctness as long as the region covered by the INVALIDATE subsumes the data accessed by the READ. This leads to a tradeoff affecting potential reuse of cached values.

Figure 1 illustrates the sort of tradeoffs that occur in the placement of INVALIDATEs. All program examples in this paper use Parallel Computing Forum (PCF) Fortran [15] syntax; PCF Fortran is an emerging standard for shared-memory parallel Fortran. The `PARALLEL` construct corresponds to a fork-join pair that specifies a block of code to be executed on every processor. `PDO` is a worksharing construct; iterations of a `PDO` are partitioned among the processors for execution. `PARALLEL DO` packs both these statements into one. If the INVALIDATE for reference 2 is placed immediately after the FORK (option 1), it must invalidate all of `A` since at that point it cannot be determined what part of `A` reference 2 will access. This invalidate will eliminate most reuse of `A` by reference 3 in the trailing serial loop. Alternatively, if the INVALIDATE for reference 2 immediately precedes the reference (option 2), `A(I)` is invalidated ten times, once for each iteration of the `J` loop, which is clearly unnecessary.

```
REAL A(10)
DO I=1,10
   A(I)= ...[reference 1]
ENDDO
PARALLEL
Inv A(1:10) [option 1]
PDO I=1,10
   DO J=1,10
      IF (MOD(J,2).EQ.1) THEN
         Inv A(I) [option 2]
         B(I) = A(I)  [reference 2]
      ENDIF
   ENDDO
END PDO
END PARALLEL
DO I=1,10
   ...=A(I)  [reference 3]
ENDDO
```

Figure 1: Placement of Invalidate

R_t = total reads in program
R_o = optimal number of main memory reads
R_a = actual number of main memory reads

W_t, W_o, and W_a are defined analogously for writes

$CRE = 1 - (R_a - R_o)/(R_t - R_o)$
$CWE = 1 - (W_a - W_o)/(W_t - W_o)$

Table 1: Definitions of coherence algorithm efficiency.

However, this placement of the INVALIDATE does not disturb reuse of `A` by reference 3 in the trailing serial loop since most of the INVALIDATEs will take place on a processor different from the one that executes the serial code.

When evaluating the effectiveness of a caching strategy, people most often focus on the hit ratio. The hit ratio is only half of the picture though. The hit ratio measures how effective reads are, but does not measure how effective writes are. The read miss ratio (one minus the hit ratio) is how often a read misses in cache causing lost performance. A complementary measure is the write miss ratio, the percentage of writes which are made to main memory (instead of just cache) relative to the total number of writes. Both the read and write miss ratios are necessarily greater than zero: there is some intrinsic number of reads from main memory and writes to main memory that a program requires. We are interested in how well coherence schemes perform relative to this standard. We define two efficiency measures, the Cache Read Efficiency, CRE, and the Cache Write Efficiency, CWE (table 1). These measures reflect the effectiveness of cache organization (e.g. associativity), but more importantly, how well a given coherence scheme does relative to an optimal scheme (for a given data set and number of processors). Optimal means the minimum number of main memory reads that a program execution would require given no evictions due to cache organization or size. Some of these reads are needed to bring values into cache initially. Others are needed because a value was changed on another processor and needs to cross processors.

The CRE is not the same as the 'hit ratio'. The CRE is how the performance of a given coherence algorithm and cache compares to the ideal for a particular program execution. Minor changes in the program under study (e.g., register allocation of some array references) might make large changes in the actual 'hit ratio' without changing the CRE. For some algorithms, $R_t = R_o$, in which case CRE is undefined. This is not just a mathematical triviality, but a reasonable interpretation. In this circumstance, no improvement is possible regardless of the coherence algorithm. Similar observations apply for the CWE.

3.1 Related Work

Here we examine two previously proposed software schemes in terms of the aforementioned trade-offs in the positioning of coherence operations. The first is Cheong and Veidenbaum's *fast selective invalidation* [4] (hereafter referred to as the FSI method). The second is a method proposed by Cytron, Karlovsky, and McAuliffe [5] (hereafter referred to as the CKM method). Both methods were developed for programs with fork-join parallelism expressed in the form of parallel loops. Evictions and cache line sizes greater than one word were not considered.

Both methods issue UPDATEs immediately after their corresponding WRITE. The FSI method does this implicitly by using write-thru caching. The CKM method uses copyback caching and thus requires explicit UPDATEs. However, the two methods treat INVALIDATEs in completely different ways. The FSI method moves all INVALIDATEs for a parallel region to immediately follow the FORK at region entry. FSI does not try to determine which locations need to be invalidated; it invalidates everything that is a shared writable object after every FORK, even if it is not referenced before the next JOIN. The CKM method goes to the opposite extreme and places every INVALIDATE immediately before its corresponding READ. To decide which READs need invalidates, the CKM method uses dependence analysis to determine which READs are involved in processor crossing dependences.

Both papers make part of the access triple observation and use it in their methods to the extent of not placing INVALIDATEs before READ references which follow only WRITE references. This is useful for FSI only if it applies to every reference in a loop. CKM can apply it on a reference by reference basis.

Cheong and Viedenbaum's simulations of the FSI method [4] show that it has a good CRE, approaching 100%, but its CWE is always 0%. There is no data on how the CKM method fairs but one would expect it to usually have a lower CRE and a CWE slightly above 0%.

Cytron, Karlovsky, and McAuliffe discuss replacing UPDATEs and INVALIDATEs with FLUSHes under certain circumstances. The treatment was apparently only for scalars and is not as general as it might be. They also suggest that coherence overhead could sometimes be reduced by moving INVALIDATEs to a pre-dominator and UPDATEs to a post-dominator. A precise algorithm is not given. INVALIDATEs can be moved to particular control flow branches so long as the other branches have an assignment to the variable. Without array kill information it is unlikely that these optimizations can be applied for subscripted variables. Scalars are not of interest since they are either local or read only; otherwise, the parallel loop would carry dependences thus violating one of the fundamental assumptions.

```
DO I=1,N
    DO J=1,N
        DO K=1,N
            DO L=1,N
                Inv A(I,F(J),K)
```

Figure 2: Level of aggregation

4 Coherence Through Vectorization

Conceptually, our approach starts with the solution generated using the CKM method and applies vectorization [1] to aggregate cache control operations and move them as far as possible, toward either the FORK or the JOIN. The resulting INVALIDATEs often resemble those created by the FSI method. We refer to our strategy as *Coherence Through Vectorization* (CTV). It maintains exactness in the sense that it never invalidates anything that does not need to be invalidated. And, of course, it never updates anything that does not need to be updated. It never pays the cost that the FSI method does of invalidating the wrong value. In the worst case CTV will pay the cost that the CKM method does of invalidating the same value too often, but it many cases redundant INVALIDATEs will be avoided.

CTV has the additional benefit of reducing run-time overhead by aggregating cache control operations and thus amortizing their initiation overhead. For the UPDATE case, this improves on both the CKM and FSI method. For the INVALIDATE case, CTV does better than the CKM method, which does not aggregate READs. But, CTV will usually not aggregate as much as the FSI method, which lumps every INVALIDATE for a given loop into a single global INVALIDATE.

4.1 Vectorization Background

We use vectorization here to mean a particular process of reconstructing a program from its dependence graph. Generating vector code is not the objective *per se*. Vectorization algorithms determine how many levels of aggregation a statement has and restructure computations so that the aggregation can be realized. Such algorithms work entirely from a program's dependence graph and deliberately ignore the program's original structure.

We use the term *aggregation* in a special sense. The level of aggregation is the number of serial loops out of which a statement can be hoisted. This might be possible due either to loop invariance or the presence of a discernable section. If we were actually generating vector code, this would be the number of dimensions of vector parallelism. In figure 2, the INVALIDATE has two levels of *aggregation*, K and L. The L level exists because the INVALIDATE is invariant with respect to it. The K level exists because the section can be analyzed. No section can be constructed for the J level because of the unanalyzable subscript. Unless the I and J loops can be interchanged, there is no aggregation at the I level because the J loop is nested inside of it. The two levels of aggregation could be realized by changing the INVALIDATE to Inv A(I,F(J),1:N) and hoisting it out of the K and L loops.

1. Analyze the program for dependence.
2. Determine which of the dependences are processor crossing.
3. For all references which have processor crossing dependences,
   (a) Create a coherence statment with the same reference.
   (b) Add dependences to put the coherence statement in the same `PDO` or serial region.
4. `CodeGen`, i.e. 'Vectorize'
   (a) Collapse `PDO`s and dependence cycles to single nodes, thus forming a DAG.
   (b) Generate code on this DAG in topological order.
   (c) For each collapsed node, recurse by calling `Gen` at the next deeper nesting level.
   (d) For single statements, produce the text of the statement with the proper amount of aggregation.

Figure 3: CTV algorithm

4.2 The Algorithm

The essence of the CTV algorithm is to add INVALIDATEs to the dependence graph in such a way that the INVALIDATE is constrained to occur after the FORK, before the READ that needs invalidation, and as soon as any subscripts needed to determine the memory location are known. Other dependences on the variable needing invalidation are *not* relevant to the placement of the INVALIDATE. The placement of the INVALIDATE is not specifically determined; it is implicit in the structure of the dependence graph. The vectorization algorithm then restructures the code so as to achieve as much aggregation as possible for the INVALIDATE. A similar process applies for WRITEs and UPDATEs. The algorithm is summarized in figure 3. The rest of this section discusses it. Consider all `PDO`s to be collapsed into `PARALLEL DO`s for purposes of discussion.

In a preprocessor step, all array references are rewritten using temporary variables in order to make the subscript references side effect free. Next, we apply conventional dependence analysis to construct a dependence graph for the program. The dependence graph serves as the program representation for the remaining steps of the algorithm.

The next step is to determine which of the dependences are processor crossing. This can be easily determined by comparing the depth of any enclosing `PARALLEL DO`s, the common nesting level of both ends of the dependence, and the carrying level of the dependence.

Next, INVALIDATEs are added. For each READ reference in the program, e.g. `A(t1,t2...)`, that is the sink of a processor crossing dependence (i.e. only true dependence is relevant), a statement to invalidate the same reference is added, e.g. `INV A(t1,t2...)`. A dependence from the INVALIDATE to the READ is added to the dependence graph. If the READ is in the scope of a `PARALLEL DO`, a dependence is added to place the INVALIDATE in the same `PARALLEL DO` as the READ. The INVALIDATE is constrained only to stay in the same parallel loop as the READ, not the serial loop(s) nested inside of the parallel loop. If the READ is in a serial section (not in the scope of a parallel loop), dependences are added to execute the INVALIDATE after any `PARALLEL DO`s which are the sources of processor crossing dependences reaching the READ (this can be trivially extended for nested `PARALLEL DO`s). This essentially means that invalidation can occur any time between processor assignment in the FORK and the actual reference, regardless of other program structure and dependences. Finally, any dependences that existed on the subscripts of the READ, `t1, t2` ..., are copied to references to those subscripts in the new INVALIDATE statement. This assures that any computations which are needed to determine subscript values have already been done. Other dependences on `A` itself are *not* copied. At this point, the placement of the INVALIDATE for the reference and the level of aggregation available is determined solely by the single reference.

UPDATEs for WRITE references which are the source of processor crossing true dependences are added next. This is done analogously to the adding of INVALIDATEs for READs. For anti and output dependences from WRITEs, an INVALIDATE is added instead (unless there is already an UPDATE). This is necessary because a dirty value that is no longer needed after a task might be evicted by the cache hardware at some later time when space is needed. This value could then overwrite a more recent value in main memory.

In the circumstance where a READ follows a WRITE in the same task and both access *some* of the same locations, the strategy as outlined above might produce the following sequence of events: WRITE, INVALIDATE, UPDATE, READ. When the INVALIDATE and UPDATE reference overlapping sections, a dirty value would be invalidated before it is updated. An *inversion dependence* is added in this step to prevent this problem. Adding coherence statements for all of the READs before any of the WRITEs makes it easy to determine when inversion dependences are necessary. This can be done is such a way as to not affect the overall time complexity of the algorithm or hinder any possible aggregation that would otherwise be available.

We refer to a dependence graph augmented with coherence operations as a coherence graph. A coherence graph contains all of the dependences which must be satisfied for a correct execution of the program on a shared memory multiprocessor without hardware coherence.

Code generation, CodeGen, proceeds by collapsing all cycles in the coherence graph and all `PARALLEL DO`s (but not serial `DO`s) to single nodes. The resulting DAG is then processed in topological order. Different node types are handled in different ways. `PARALLEL DO`s must be collapsed to single nodes to keep them from being distributed. This insures that INVALIDATEs will be done in the same task as their corresponding READs.

If a given node contains more than one statement, an appropriate type of `DO` statement is generated at this level and the contents of the cycle are processed by recursively calling CodeGen at one level deeper. In the recursive call, all statements and dependences outside of this node are ignored. Also, any dependences carried at the current level are ignored because they are satisfied by the `DO`. If the node is a single statement, code can be directly generated for it.

```
1    PARALLEL DO I=1,N
2       J2=0
3       DO J=1,N
4          J2=J2+1
5          DO K=1,N
9             C(I,J) = C(I,J2) + A(I,K)*B(K,J)
```

Figure 4: Matrix Multiply before coherence

The four cases are:

- Cycle: this indicates that the statements in the node originally came from a serial loop and they all depend on each other in some way including at least one loop carried dependence. This does not cause all of the statements originally in the DO to be generated, only those in the cycle.
- PARALLEL DO node: Generate a PARALLEL DO and then recurse.
- A single non-coherence statement: Generate it. If the current level is less than the original level of the statement then additional serial DOs will be necessary. This may occur because a statement is nested in a loop but has only loop independent dependence upon it. There are no cycles in this case, but the surrounding DOs are still necessary.
- A single coherence statement: Generate the coherence statement with a level of aggregration which is the difference between the current level of code generation and original nesting depth of the coherence statement. This is sufficient to ensure that coherence statements are generated at the outermost level possible, i.e. with as much aggregation as possible.

As a practical matter, cycles and single non-coherence statements on the same level will be fused together into one serial loop when doing so does not capture a coherence statement between them. This simply prevents loops from being distributed unnecessarily.

Non-unit cache line sizes cause aliasing of values. This problem must be addressed by any coherence scheme. If a parallel loop index does not appear in the fastest varying subscript of an array reference, it is not a problem. When it does, other compiler techniques such as changing the array layout or strip mining can be used to solve it. Failing that, either no-caching or write-thru caching must be used for that variable inside any parallel construct in which this situation occurs. If the architecture does not permit such an allocation, some of the parallelism will have to sacrificed for a completely automatic technique. This must be handled before the coherence graph is built.

A more detailed treatment of the algorithm can be found in [6].

4.3 Example

Consider a simple matrix multiply (figure 4) where the outer loop is parallel. The J2 assignment in statement 4 is added for the sake of example (assume that the compiler does not recognize auxiliary induction variables). References to elements in arrays A, B, and C must all be invalidated before they are used because their initial assignments may have occurred on a different processor. The new value of C must be updated from cache to main memory before the PARALLEL DO loop finishes.

```
1    PARALLEL DO I=1,N
2       J2=0
3       DO J=1,N
4          J2=J2+1
5          DO K=1,N
6             Inv A(I,K)
7             Inv B(K,J)
8             Inv C(I,J2)
9             C(I,J) = C(I,J2) + A(I,K)*B(K,J)
10            Upd C(I,J)
11         END DO
12      END DO
13   END PARALLEL DO
```

Figure 5: Matrix Multiply after CKM coherence

The FSI approach would solve this problem by invalidating the whole cache for each processor when the PARALLEL DO starts and using write-thru so that every assignment to C in statement 9 goes straight to main memory. The problem with this is that for any given I and J it writes C(I,J) N times (for each iteration of the K loop instead of just once).

The naïve CKM approach would solve the problem by specifically invalidating before every READ and updating after every WRITE (figure 5). This suffers from the same problem as the FSI method for both the WRITE and the READ of C(I,J). In this particular case, the INVALIDATE and the UPDATE for C contain a loop invariant expression and could be hoisted. The original CKM paper discusses this possiblity. But, that is still not good enough.

CTV conceptually starts with what the CKM approach produces (figure 5). Dependences are added from the PARALLEL DO I to the Inv A and from the Inv A to the reference to A in the assignment statement. Dependences for references to B and C are added similarly. Note that the dependences do not pin the coherence statements inside the inner K loop.

Next, CodeGen is invoked on the resulting coherence graph. The only node is the PARALLEL DO itself (other statements are collapsed into this node). That statement is generated. Then CodeGen recurses on the contents of the loop. The Inv A has no predecessors in the coherence graph so it can be generated, even though neither the J loop nor the K loop have been generated yet. The Inv A was originally at nesting level 3 and code is now being generated at level 1, so there are two levels of aggregation available. These are found and the final Inv A is generated. The B case is similar.

The Inv C cannot be generated yet because it depends on the assignment of J2. The J2=0 assignment has no dependence predecessors and can be generated. All of the other statements which reference J2 are in a cycle because of loop carried anti dependence. That cycle is handled by generat-

```
1    PARALLEL DO I=1,N
2       J2=0
6       Inv A(I,1:N)
7       Inv B(1:N,1:N)
3       DO J=1,N
4          J2=J2+1
8          Inv C(I,J2)
5          DO K=1,N
9             C(I,J) = C(I,J2) + A(I,K)*B(K,J)
11         END DO
12      END DO
10      Upd C(I,1:N)
13   END PARALLEL DO
```

Figure 6: Matrix Multiply after CTV

Method	CRE (%)	CWE (%)
FSI	100	0
CKM	0	0
CKM+	$\frac{n-1}{3n-p-2}$	100
CTV	100	100

Table 2: Cache Utilization of Different Methods on Matrix Multiply

ing the `DO J` loop then recursing on its contents. The `INV C` can then be generated after the `J2=J2+1` which it depends on. Thus, the `INV C` is hoisted out of the `K` loop, but not the `J`.

A rough comparison of the three methods can be made by counting the READs and WRITEs to main memory versus cache (table 2, where n = problem size, p = number of processors). CKM+ refers to the results of the CKM method after making the simple optimization of hoisting loop invariant coherence instructions.

The CTV and CKM method both suffer overheads that the FSI method does not which are not reflected in the table. For this problem, however, the CTV method has asympotically fewer operations than the FSI and must for some sufficiently large problem be faster. This will be approximately when $n*$ time for main memory write > the time to start a coherence operation + time for main memory write.

5 Experimental Results

The CTV method, the CKM method, and the FSI method were applied manually to two small, simple programs which were tested on a BBN TC2000, a distributed shared memory multiprocessor, to evaluate their effectiveness. The two programs were blocked LU decomposition and a heat flow relaxation.

The BBN TC2000 is a shared memory multiprocessor capable of supporting up to 512 nodes. Each processing node consists of a Motorola 88100 processor, an 88200 cache unit, and several megabytes of memory. Each processor can access its own memory directly, and can access the memory on any other node through a $\log_8$-depth interconnection network (organized as an indirect binary N-cube composed of 8x8 crossbar switching nodes). Virtual circuit connections through the switching network are used to perform remote accesses. Many non-interfering virtual circuit connections (i.e., the sources and sinks of all connections are unique systemwide) can be open across the switching network simultaneously. If collisions occur at a switch node, one transaction succeeds and all of the others are aborted, to be retried at a later time (in hardware) by the processors that initiated them. There is no hardware mechanism for ensuring coherence between cache units on different nodes. The 88200 cache unit allows any given memory segment to be configured with one of three caching policies: uncached, cached with write-thru, or cached with copy back. Transparent to the caching policy, the TC2000 architecture supports interleaved memory. In a page of interleaved memory, cache line size blocks are allocated in virtual address order across all processors in the machine in a round robin fastion. Interleaving can reduce potential contention for shared data.

The test programs all use the "Uniform System", a standard library of functions that support parallel programming on the TC2000. The tasking model supported by the Uniform System dedicates a set of processors to a program for its duration. Under the Uniform System, writable shared data must either be allocated uncached or the programmer must explicitly manage coherence with INVALIDATE and UPDATE calls.

The sample codes used in our experiments were not optimized by hand in order to avoid prejudicing the test results. The simple sequential code with appropriate DO loops changed to parallel was used. Interleaving was used for all shared data. The compiler does not allocate array references to registers across loop boundaries (such aggressive allocation techniques will tend to equalize coherence methods).

All caches were emptied before the start of each test. The times include the cost of faulting in the data initially, whatever evictions occur, and writing back the final results (except for the *no coherence* case, explained below).

The FSI method proposal calls for additional hardware to invalidate cache contents in constant time. The FSI method was run with software calls to invalidate all of cache where the method specifies and then run again with no invalidations (which produces wrong results). The former is referred to as FSI and the latter as OptFSI, Optimistic FSI. On the TC2000, the software call to invalidate all of cache takes constant time with a fairly large constant. The real effectiveness of FSI lies somewhere between the measured performance of FSI and OptFSI depending on just how efficient of a total INVALIDATE is available.

The CKM method has been slighted somewhat because it was never designed to deal with cache line sizes of other than one word. On the TC2000, two 'double's fit in one cache line. So, each fault in the CKM case causes 16 bytes instead of 8 to be read. None of the suggested CKM optimizations (section 3.1) were applicable to these test cases.

The CTV method as presented so far does not rely on being able to understand array sections in any way. The code actually produced has many duplicate invalidations of the same locations. If array section analysis were available, CTV could be changed so as to invalidate the smallest (describable in vector notation) section that subsumed all others for which complete subscript information is available.

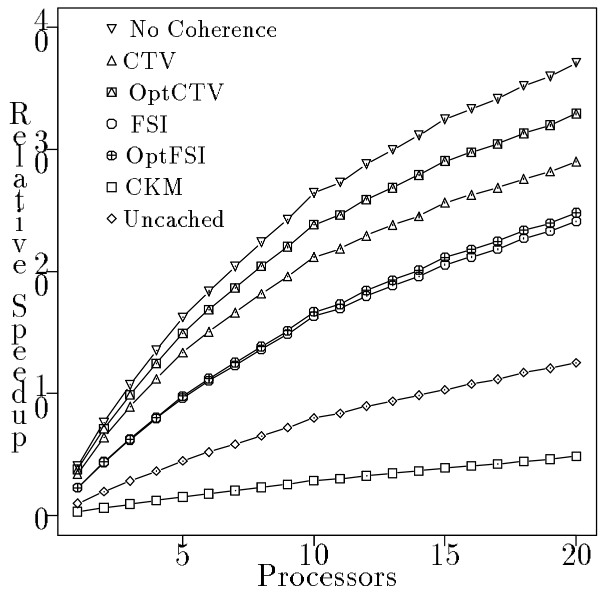

Figure 7: LU Decomposition, Interleaved, Size = 200

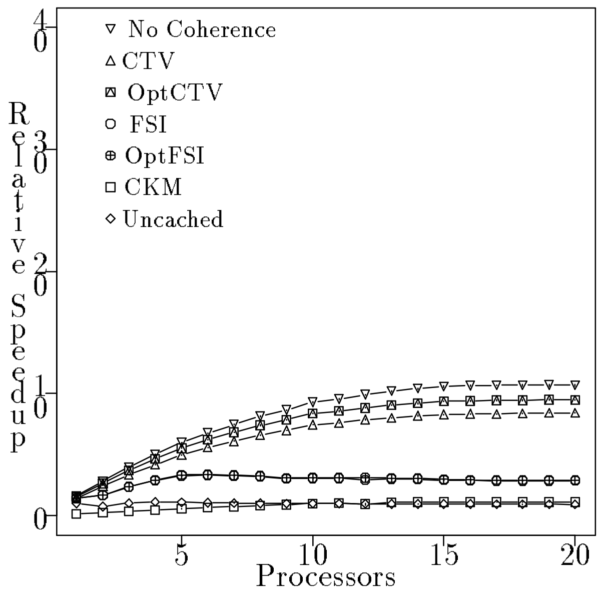

Figure 8: LU Decomposition, Not Interleaved, Size = 200

This approach is referred to as OptCTV, Optimistic CTV. This can over-invalidate and will not be better in all cases. What separates it from CTV is removing the overhead of unnecessary calls. For the tests cases in this paper, OptCTV has an invalidation (but not update) patten like FSI.

The three software methods are compared to the *no coherence* case. In this case, the program is run using a copy back caching strategy with no coherence provided in software. Read misses still occur to bring the data in the first time, but the final results are never written back and no values are communicated between processors (with the exception of values that are written out as a result of evictions and subsequently faulted in by another processor). The computed result is nonsense, but the performance in the no coherence case would seem to represent an upper bound on the performance that any coherence scheme, hardware or software, could hope to achieve. But for eviction anomalies, it is super-optimal because some coherence traffic is essential for the results to be correct.

The results are summarized in figures 7, 8, 9, and 10. Speedup for figures 7, 8, and 10 is versus the one processor uncached case. For figure 9, speedup is versus the uncached case for each problem size, i.e. the uncached case is defined to be a horizontal line. Each test point was run 30 times. The (sloping) lines are plotted on the averages. The vertical lines represent a 95% confidence interval, assuming normal distribution, for the measured performance of a test configuration. This is not a confidence interval for an arbitrary run because overall system loading conditions affect timing through the memory interconnect. The data accurately reflects the relation between any two methods though, since all were run under similar loading conditions. Confidence intervals are longer in the upper right of the graphs simply for the reason that reciprocals of small numbers (actual times) with constant absolute deviations are being graphed. These figures are not meant to show speedups over a well coded single processor version of the algorithm but only to show the relation of the different coherence methods.

The LU decomposition is a partial pivot blocked right-looking algorithm [7]. The blocking factor for all tests was 10. That is close to optimal for all measured test cases. The *no coherence* case produces numerous floating point exceptions due to both division by zero and overflow errors. Trapping these exceptions skews the results. To avoid this problem, all test cases for LU decomposition were run with a min operator in place of multiply and divide in at attempt to replace them with a nearly time equivalent operator. This makes all test cases run about 3% slower but does not affect the relative performance. In particular, it does not affect the memory reference pattern. The pivot decisions were fixed to be the same for all cases (though the search for the best pivot still occurs). This did not measurably affect the running time.

LU decomposition is an O(n^3) time algorithm working on an O(n^2) matrix. Therefore each matrix element is being written (on average) O(n) times. Most of these writes could stay in cache. CTV takes advantage of this in its creation of a section to update thereby raising the CWE. The performance of FSI is degraded by doing these unnecessary writes. CKM simply has too much overhead to achieve good performance. FSI performs better than CTV on small matrices because CTV is paying higher overhead for system calls. For larger matrices, CTV performs better (figure 9).

Another advantage of CTV is that interleaving is not critical. FSI's maximum speedup is small when running with the default allocation of all memory on one node because of high contention. CTV continues to perform well though (figure 8).

The heat flow algorithm gives the FSI method its best chance to do better than CTV. The computational kernel of the algorithm uses a simple four point iterative relaxation in which the value computed for each point is needed in the next time step to compute the new value of each neighboring point. Lacking any knowledge about processor scheduling, it is necessary to update to main memory every write that occurs in the inner loop of this algorithm. The CWE is undefined since $W_t = W_o$ and cannot be changed due to the nature of the algorithm. Still, the advantage of

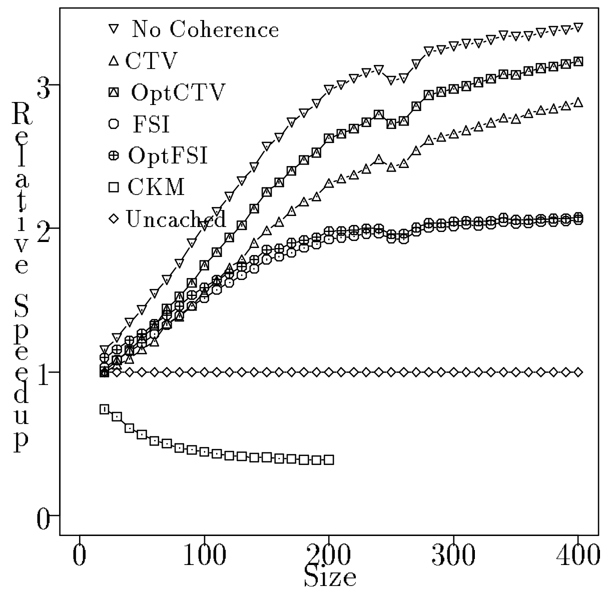

Figure 9: LU Decomposition, Interleaved, Processors = 20

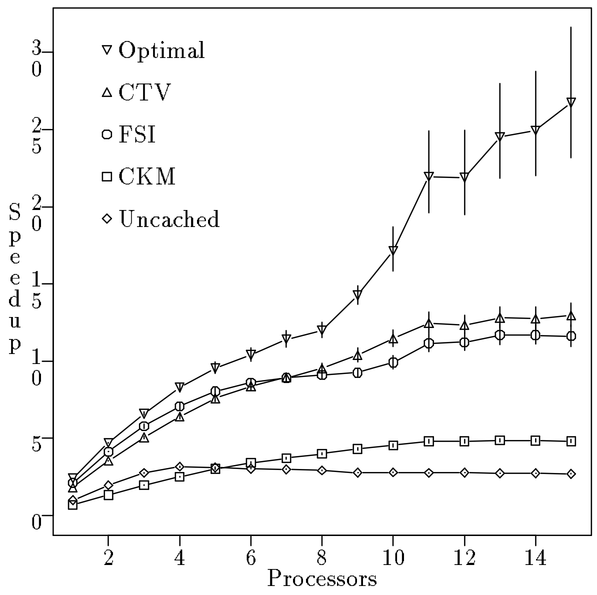

Figure 10: Heat Flow, Interleaved, Size = 100

handling a whole block at one time allowed CTV sometimes to do better than FSI and to be competitive in all cases. Without interleaved memory, CTV does noticeably but not dramatically better. Figure 10 shows the performance of the heat flow relaxation algorithm using each of the caching strategies.

We believe this data shows that the CTV method of aggregating coherence statements can reduce memory contention, amortize the inefficiences of numerous system calls, and avoid using write-thru caches. Even in those situations where the intrinsic communication costs are high, CTV does not add an unacceptable amount of overhead.

Unfortunately, at present we cannot directly compare the results in any of our test cases to the performance of a hardware scheme. However, the performance in the *no coherence* case also represents an upper bound on the performance of any hardware based solution. Extrapolating from our measurements, the performance of the CTV would appear to be comparable with any hardware-based scheme for each of the algorithms studied.

6 Summary and Conclusion

In this paper, we have presented a measure by which to gauge the effectiveness of software coherence methods(the CRE and CWE), a more general framework in which to describe their operation (access triples), and an algorithm for automatically adding coherence statements to a program so that it can be run on a shared memory multiprocessor without hardware cache coherence. This algorithm aggregates array references in coherence statements to avoid the overhead of handling each word independently (as, for instance, a write-thru cache would) and it restructures the program code so as to perform as few coherence operations as possible, within the precision of the analysis. It handles INVALIDATEs before READs and UPDATEs after WRITEs in the same manner. Thus, it addresses both the CRE and CWE simultaneously.

We have compared this algorithm to two previous software coherence schemes, and shown that it combines the best of both approaches. Preliminary data from a manual application of this algorithm to two small programs support this conclusion. Our work shows that software coherence techniques are efficient for some numerical codes. If further research shows that this approach is applicable to a large class of numerical codes, then it will become possible to build effective shared memory multiprocessors without (expensive) coherence hardware.

7 Future Work

With the model as presented so far, only fork-join parallelism and coherence actions being performed in the task that needs them, other coherence models, e.g. release consistency, might be sufficient to address the problem. Software schemes and CTV in particular have the advantage of working just on the sections of arrays that need coherence. This gives them the leverage to exploit different synchronization constructs to good advantage. For example, only the data actually used in a critical section needs to be made coherent instead of updating every dirty value in the cache when exiting a critical section. While the potential is there, an efficient algorithm has yet to be found.

Instead of changing the first assumption, that fork-join parallelism is used, one can change the second assumption, that coherence is performed in the same task as the relevant reference. This allows values to be left unaffected across synchronization points when it is known they will not be needed until subsequent tasks.

The CTV algorithm as given handles access triples by always invalidating in the same set of loops as the final READ thus catching already stale values. This solves the problem of finding the right processor by waiting until scheduling is done and it finds the right section implicitly by duplicating the reference of the final READ. However, it fails to take advantage of reuse between loops. It treats each loop separately. This is a consequence of considering the FORK,

```
PARALLEL DO I=1,N
    ...=A(I)
END DO
PARALLEL DO I=1,N
    Inv A(1:N)
    A(I)=...
    Upd A(I)
END DO
PARALLEL DO I=1,N
    ...=A(I)
END DO
PARALLEL DO I=1,N
    ...=A(I)
END DO
```

Figure 11: Avoiding stale values entirely

INVALIDATE, READ pattern as necessary. One relatively easy change that comes from considering access triples that could prove useful is to restrict the INVALIDATE before the READ to skip anything that was not written (on the joining dependence).

The more interesting question is how to handle multiple access triples at one time. One could instead invalidate before every WRITE the section that the whole loop writes thus removing values from cache before they become stale. For instance, the code fragment in figure 11 is correct. Not only is it correct but the third and fourth loops will have reuse for many probable schedules. The INVALIDATE before the WRITE must invalidate all of A though because it cannot be known for sure which processor will read a given element. How to best place invalidates for access triples remains an open question.

References

[1] R. Allen and K. Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, Oct. 1987.

[2] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera computer system. In *Proc. of the 1990 International Conference on Supercomputing/Computer Architecture News*, pages 1–6, Amsterdam, The Netherlands, June 1990. Proceedings published as ACM SIGARCH Computer Architecture News, 18 (3), Sept. 1990.

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

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

[5] R. Cytron, S. Karlovsky, and K. McAuliffe. Automatic management of programmable caches. In *Proc. of the 1988 International Conference on Parallel Processing*, pages 229–238, ?, Aug. 1988.

[6] E. Darnell, J. M. Mellor-Crummey, and K. Kennedy. Automatic software cache coherence through vectorization. Technical Report CRPC-TR92197, Computer Science Department, Rice University, Jan. 1992.

[7] J. J. Donagrra, I. S. Duff, D. C. Sorenson, and H. A. van der Vorst. *Solving Linear Systems on Vector and Shared Memory Computers.* Society for Industrial and Applied Mathematics, 1991.

[8] D. Kuck. *The Structure of Computers and Computations, Volume 1.* Wiley, New York, NY, 1978.

[9] D. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. J. Wolfe. Dependence graphs and compiler optimizations. In *Conference Record of the Eighth ACM Symposium on the Principles of Programming Languages*, Williamsburg, VA, Jan. 1981.

[10] L. Lamport. How to make a multiprocessor that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9), Sept. 1979.

[11] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the dash multiprocessor. *17th International Symposium on Computer Architecture/Computer Architecture News*, pages 148–159, May 1990. Special issue of Computer Architecture News, 18(2), June 1990.

[12] S. Min and J. Baer. A timestamp-based cache coherence scheme. In *Proc. of the 1989 International Conference on Parallel Processing*, volume 1, pages 23–32, Aug. 1989.

[13] S. Min, J. Baer, and H. Kim. An efficient caching support for critical sections in large-scale shared-memory multiprocessors. In *Proc. of the 1990 International Conference on Supercomputing/Computer Architecture News*, pages 4–47, June 1990. Special issue of Computer Architecture News, 18(3), Sept. 1990.

[14] A. Osterhaug, editor. *Guide to Parallel Programming on Sequent Computer Systems.* Sequent Technical Publications, San Diego, CA, 1989.

[15] Parallel Computing Forum. *PCF Fortran*, Mar. 1990. Working Draft.

[16] D. Schanin. The design and development of a very high speed system bus – the encore multimax nanobus. In *Proceedings of the Fall Joint Computer Conference*, pages 410–418, Nov. 1986.

[17] J. Willis, A. Sanderson, and C. Hill. Cache coherence in systems with parallel communication channels & many processors. In *Supercomputing '90*, pages 554–563, 1990.

A Vectorization Algorithm for Adding Coherence

Notation:

- $\exists$ - There Exists
- $|$ - such that

Notes on representation:

a) After dependence analysis all explicit program structure is lost.

b) Computation dependences are not used because statements are never broken, i.e. for the purpose of code generation a dependence to a reference is that same as a dependence to the statement the reference is in.

c) DOs are viewed as assignments to the corresponding upper bound variable, and remembered as something special.

d) Dependences run from the DO variable assignment to all of the statements (not references) that were in the loop.

e) A statement at the outer level is at level 0.

f) The level of a DO is the level of the DO itself and not the statements that it controls, e.g. a DO at the outer level is at level 0 for the DO node itself.

g) the level of a dependence is the level of the carrying loop + 1, i.e. the level of a statment immediately within the carrying loop.

h) loop independent dependences are at an infinite level.

Initial processing:

1. Introduce temporary variables so that all array references are of the form A[t1,t2,t3,...]. Note that this unrolls nested expressions such as A(B(I)) into two assignments and that it guarantees that no subscript expression will have side effects.

2. Perform dependence analysis. Be sure to mark all of the temporary variables introduced above as privatizable.

The Algorithm:

```
Program CTVCohere;

type
  PVariable = ^Variable;
  Variable = record
     – a description of a program variable
  end;

  PDependence = ^Dependence;
  Dependence = record
    src, snk :  PReference or PStatement or PNode;
    nlevel :  integer;    – carrying level of dependence, indepedent = ∞
    prc :  boolean;       – processor crossing
  end;

  PReference = ^Reference;
  Reference = record
    reftype :  (READ,WRITE);
    var :  PVariable;                – the variable itself
    refs :  list of PVariable;       – variables used to subscript var
    src, snk :  set of PDependence;  – deps with this ref as an endpoint
    stmt :  PStatement;              – statement this ref is in
    coh :  PStatement;               – coherence op for this ref (if any)
  end;

  PStatement = ^ Statement;
  Statement = record
```

```
    refs:  set of PReference;        – all refs in statement
    nlevel :  integer;               – loop nesting level of statement
    txt :  ...                       – Abstract Syntax Tree rep. of stmt
    par :  PNode;                    – parent node in collapsed graph
    enclosing_do_stmt :  PStatement;
    index_var:  PReference;           – defined only for PDO or DO statements
  end;

  PNode = ^Node;                      – used for reduced graph
  Node = record
    stmts :  set of PStatement;      – statements in the loop
    ntype :  (PDO,LOOP,SINGLETON);
    pdo :  PStatement;               – parallel loop itself, if ntype=PDO
  end;

Var
  Dependences:  set of PDependence; – all dependences
  References:  set of PReference;   – all references
  Statements:  set of PStatement;   – all statements
  d :  PDependence;                 – a temporary for scanning a set
  r,                                – a temporary for scanning a set
  w :  PReference;                  – a write reference in a set
  Inv :  PStmt;                     – The invalidate statement for a particular read reference

   – the loop nesting level for statements outside of any loop is -1
  NullStatement :  Statement = (nlevel :  -1);
  NullDoStmt :  PStatement = @NullStatement;

   – return the nesting level of the innermost loop that
   – encloses both references. if no common enclosing loop, return 0
  function enclosing_common_loop_nest_level (ref1, ref2:  PReference) :  integer;

   – return the nesting level of the nearest enclosing parallel loop
   – if no enclosing parallel loop , return 0
  function enclosing_parallel_loop_level (ref :  PReference) :  integer;

   — add the appropriate coherence operation for a particular reference
  procedure cohere (r :  PReference);

    var
      rsub,                       – subscript of r
      csub,                       – new subscript for coherence operation
      cref :  PReference;         – new Reference for coherence operation
      cstmt :  PStatement;        – new statement for coherence operation
      d :  PDependence;           – a temporary for enumerating dependences

       – create a new dependence with the specified fields and add it to
       – the set 'Dependences' and the sets src^.src and snk^.snk
       – (see note d)
      procedure NewDep (src, snk :  PReference; nlevel :  integer; prc :  boolean);

       – create a new loop independent dependence
      procedure NewLIdep (src, snk :  PReference);
      begin
        NewDep (src,snk,∞,false)
      end;


    begin
      new cref;
      cref^.reftype := READ;
      cref^.var := r^.var;

       – copy subscript expression and duplicate dependences therein
      for all rsub in r^.refs (in order)
```

```
        new csub;
        for all d in rsub^.snk
          NewDep (d^.src, csub, d^.nlevel, d^.prc);
        for all d in rsub^.src
          NewDep (csub, d^.snk, d^.nlevel, d^.prc);
        ListInsertAtTail(cref^.refs, csub)

       – build the actual invalidate or update statement
       – put it in the correct place in the dependence graph
      new cstmt;
      cstmt^.enclosing_do_stmt := r^.stmt^.enclosing_do_stmt;
      cstmt^.refs := {cref^.var} + cref^.refs;
      cstmt^.nlevel := r^.stmt^.nlevel;
      r^.coh := cstmt;
      cref^.stmt := cstmt;
      if r^.reftype = READ then
        cstmt^.txt := 'Invalidate' cref^.var '[' cref^.refs ']'
        NewLIdep (cref,r);
      else
        cstmt^.txt := 'Update' cref^.var '[' cref^.refs ']'
        NewLIdep (r,cref);

       – ensure that the coherence operation is represented in the loop
       – by adding a dependence from the loop index variable to the
       – coherence statement
      NewLIdep (cstmt^.enclosing_do_stmt^.index_var, cstmt);          – note (b)

    end;

 – Generate code for the given set of statements for the specified level
procedure Gen (Stmts :  Set of PStatement; nlevel :  integer);
  var
    Contents,                   – The statements in a cycle
    Stmt :  PStatement;         – the statement being operated on
    StmtSummary :  PNode;       – a new statement in the reduced graph
    d :  PDependence;           – a dependence in the passed graph
    r :  PReference;            – a temporary reference used to scan a set
    ProgDag :  set of PNode;    – the program reduced to a DAG

   – Add a statement or the contents of a previous summary node to the summary node, StmtSummary,
   – and fix up the pointers
  procedure AddStatement (Stmt :  PStatement);
  end;

begin
   – Reduce the graph by collapsing all dependence cycles & PDOs to single nodes
   – thus forming a DAG without distributing PDOs. note (a)
  For all Stmt in Stmts
    Stmt^.par := nil;
  RStat := {};
  While Stmts ≠ {}
    Stmt := any element of Stmts;
    new StmtSummary;
    if Stmt is a PDO          – PDOs at levels < nlevel will never be considered here
      StmtSummary^.ntype := PDO;
      StmtSummary^.pdo := Stmt;
       — Add all statements in the PDO loop to the node representing the PDO
      For all d in Stmt^.index_var^.src
        AddStatement (d^.snk^.stmt);
    else if there is path from Stmt to itself         – note (c)
      StmtSummary^.ntype := LOOP;
      for all Contents | ∃ a path from Stmt to Contents and from Contents to Stmt
                                 – including Stmt itself
        AddStatement (Contents)
    else
```

```
      StmtSummary^.ntype := SINGLETON;
      AddStatement (Stmt);
    ProgDag += {StmtSummary};
  end;

   – Regenerate the code
  while ProgDag <> {}
      – Pick a node which is not the sink of any dependence
    Pick an StmtSummary | ∄ d in Dependences | d^.snk^.stmt^.par = StmtSummary
    ProgDag - := {StmtSummary};

      — Delete all dependences from this summary node to a different node
      — and all dependences which are at the current level since they will be
      — satisifed before recursing
    for all Stmt in StmtSummary^.stmts     – note (e)
      for r in Stmt^.refs
        for all d in r^.src
          if (d^.snk^.stmt^.par <> StmtSummary) or (d^.nlevel = nlevel) then
            Dependences - := {d}
            d^.src^.refs - := {d};
            d^.snk^.refs - := {d};

    Stmt := any member of StmtSummary;
    if StmtSummary^.ntype = PDO then
      emit StmtSummary^.text;               – the PDO node itself
      Gen (StmtSummary^.stmts,nlevel+1);
      emit 'END PDO'
    else if StmtSummary is a serial DO ;   – do nothing, note (f)
    else if StmtSummary^.ntype=SINGLETON and
        (Stmt^.nlevel = nlevel or Stmt is a coherence statement)
      emit Stmt^.text with Stmt^.nlevel-nlevel levels of parallelism
    else   – loop
      emit Stmt^.enclosing_do_stmt^.text
      Gen (StmtSummary^.stmts,nlevel+1);
    emit 'END DO'
end;

 __ ***********
 — Start of Main program
begin
 – Calculate processor crossing flag
For all d in Dependences
  with d^ do
    prc = min( nlevel , enclosing_common_loop_nest_level(src,snk) )
        <= max( enclosing_parallel_loop_level(src), enclosing_parallel_loop_level(snk) );

 – Add coherence statements for all reads that have processor crossing dependences
For all r in References (ignoring additions made during this loop)
  if (r^.reftype = READ) and (∃ d in r^.snk | d^.prc) then
    Cohere (r);
 – Must be a separate loop so all 'Inv's are already added and inversion test
 – makes sense
For all w in Ref (ignoring additions made during this loop)
   – catches output- & anti- dependences and therefore handles evictions
  if (w^.reftype = WRITE) and (∃ d in w^.src | d^.prc) then
    Cohere (w);
      – The 'inversion' problem is handled here
      – For all reads that depend on this write, are in the same task,
      – and need coherence for other reasons, add inversion preventing dependences
    for all d in w^.src |  d^.prc and d^.snk^.coh <> nil
      Inv = d^.snk^.coh;         – The invalidate statement for the read
      if ∃ a path from w^.stmt to Inv
        NewLIdep (w^.coh,Inv)
      else
        NewLIdep (Inv,w)
```

– Rebuild the code

```
Gen (Statements,0);

end.
```

NOTES:

(a) PDO loop bodies must be preserved as single units. If they were not handled this way, they might get distributed. This would undo the logic that goes into adding the invalidates & updates. They could end up in separate PDOs from the references they were constructed for, if PDO structure weren't preserved.

(b) It isn't necessary to worry about building a dependence from an update to the join because the update will necessarily be part of same PDO loop as the reference it updates. It is necessary, however, to build a dependence from the fork node to the update so that the update will be recognized as being in the same PDO as the reference it updates. If the immediately surrounding DO loop is a serial DO instead of a PDO it is sufficient to put the statement in the serial DO because the serial DO must surely be within the PDO in question.

(c) This would not be a reasonable way to actually implement the algorithm. One would want to use Tarjan's algorithm for finding strongly connected components with appropriate modifications to handle the PDOs.

(d) What gets passed here might actually be a PStatement instead of a PReference. If that is the case, it is sufficient to simply ignore those operations for which a PStatement doesn't make sense.

(e) Remove all dependences for which this node is the source. But do not remove dependences that are contained entirely in this node because they will be passed to the next level. Also, remove all dependences at the level for which a DO node is about to be generated since these will be resolved by the DO itself and not relevant after recursion.

(f) Serial DO's are generated as needed for the statement they originally controlled based upon how much distribution actually occurs. The nodes representing the index variable assignment does not need to generate any code. It does need to be in the graph though.