

1-1-1993

# Compiler-Assisted Cache Coherence

Chiiwen Liou

*Purdue University School of Electrical Engineering*

Henry G. Dietz

*Purdue University School of Electrical Engineering*

Follow this and additional works at: <http://docs.lib.purdue.edu/ecetr>

---

Liou, Chiiwen and Dietz, Henry G., "Compiler-Assisted Cache Coherence" (1993). *ECE Technical Reports*. Paper 215.  
<http://docs.lib.purdue.edu/ecetr/215>

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact [epubs@purdue.edu](mailto:epubs@purdue.edu) for additional information.

# COMPILER-ASSISTED CACHE COHERENCE

CHIWEN LIOU  
HENRY G. DIETZ

TR-EE 93-4  
JANUARY 1993



SCHOOL OF ELECTRICAL ENGINEERING  
PURDUE UNIVERSITY  
WEST LAFAYETTE, INDIANA 47907-1285

---

# Compiler-Assisted Cache Coherence

*Chiiwen Liou and Henry G. Dietz*

Parallel Processing Laboratory  
School of Electrical Engineering  
**Purdue** University  
West Lafayette, IN 47907-1285  
**chiiwen@ecn.purdue.edu**  
**hankd@ecn.purdue.edu**

## **Abstract**

Although it is convenient to program multiprocessors as though all processors share access to the **same** memory, it is difficult to construct hardware directly implementing this model. Allowing each processor to have its own independent cache partially solves this problem; unfortunately, different caches might hold different values for the same main memory address. Making the **cache** hardware always be coherent — one value per address — makes the hardware **expensive** and slow. This paper proposes compiler analysis to generate explicit cache control **instructions** to ensure that all program memory accesses execute as *though* the caches were coherent. Compared to other approaches, the method given in this paper is superior because the analysis **more** accurately models interactions between tasks.

**Keywords:** Cache Coherence, Cache Policy, Compiler Optimization, Shared Memory, **MIMD**.

<sup>†</sup> This work was supported in part by the Office of Naval Research (ONR) under grant number **N00014-91-J-4013** and by the United States Air Force (USAF) Rome Laboratories under award number **F30602-92-C-0150**.

## Table of Contents

|  |           |
|--|-----------|
| <b>1. Introduction</b> .....   | <b>2</b>  |
| <b>2. Previous Work</b> .....  | <b>3</b>  |
| <b>2.1. Keeping Reads Coherent</b> .....                                   | <b>3</b>  |
| <b>2.2. Write-Through Vs. Write-Back</b> .....                             | <b>4</b>  |
| <b>2.3. System-Level Issues</b> .....                                      | <b>5</b>  |
| <b>3. Task Graph and Parallel Execution Model</b> .....                    | <b>5</b>  |
| <b>4. Compiler-assisted Reference-marking Scheme</b> .....                 | <b>7</b>  |
| <b>4.1. The Reference Marking Scheme</b> .....                             | <b>7</b>  |
| <b>4.2. Data-Flow Analysis</b> .....                                       | <b>8</b>  |
| <b>4.2.1. An Individual Statement</b> .....                                | <b>10</b> |
| <b>4.2.2. A Sequence of Statements</b> .....                               | <b>10</b> |
| <b>4.2.3. Conditional Constructs</b> .....                                 | <b>10</b> |
| <b>4.2.4. Looping Constructs</b> .....                                     | <b>11</b> |
| <b>4.3. Marking References</b> .....                                       | <b>11</b> |
| <b>5. An Example</b> .....   | <b>12</b> |
| <b>6. Generalization of Execution Model</b> .....                          | <b>16</b> |
| <b>6.1. Algorithm to Add Artificial Data Dependencies</b> .....            | <b>18</b> |
| <b>7. Implementation Issues of Compiler-assisted Cache Coherence</b> ..... | <b>18</b> |
| <b>7.1. False Sharing</b> .....  | <b>18</b> |
| <b>7.1.1. Software Solution</b> .....                                      | <b>18</b> |
| <b>7.1.2. Hardware Solution</b> .....                                      | <b>19</b> |
| <b>7.2. Validation of Cache Data</b> .....                                 | <b>19</b> |
| <b>8. Conclusions and Future Work</b> .....                                | <b>21</b> |

## 1. Introduction

Traditionally, the way to build a faster computer was primarily to employ faster circuit technologies in implementing the processor. However, recent progress in semiconductor fabrication technology has made high-end mass market microprocessors (Brooks's "Killer Micros" [Bro91] nearly as fast as the supercomputers built using the more exotic and expensive technologies. To build faster supercomputers, perhaps the most promising approach is to incorporate a large **number** of fast microprocessors in a machine so that portions of each program can be executed in parallel using a **MIMD** (Multiple Instruction stream, Multiple Data **stream**) execution model.

Programming these machines is much easier if the user can concentrate on **parallelizing** execution, without having to worry about where data items are placed. A **shared-memory** model (e.g. CREW [LaD90] — Concurrent Read but Exclusive Write) provides this abstraction, but hardware supporting this is generally slow when many processors share the same address space. Part of **the** delay (latency) in making memory references is due to the traversal time of the **interconnection** network, which can be extended by interference between accesses made simultaneously by multiple processors. However, as processor speeds have increased dramatically, memory speeds have risen only slightly; therefore, even without interprocessor communication, memory reference speed often limits system performance.

The classical solution to the memory latency problem in uniprocessor **machines** is to use a small, fast, cache memory to buffer values, either instructions or data, near the processor so that a "real" **memory** reference can often be avoided [Smi82]. Since it was first used in a commercial sequential computer in 1968 [Lip68], researchers have been analyzing uniprocessor caches and suggesting improvements [Chi89], [HiS89], [Puz85], [Smi82], [Smi85], [Smi87]. A.J. Smith [Smi82], [Smi86] has provided an excellent survey of uniprocessor cache analyses as well as a partial bibliography of cache research.

However, in multiprocessor systems, the cache organization is somewhat more complex. The lowest memory latency is obtained by associating a separate, independent, cache with each processor. Each of these "private" caches looks and acts much as the cache in a uniprocessor system, however, there is an important difference. Since a multiprocessor has multiple caches, some care must be taken to ensure that the values in the caches remain coherent. Censier and **Fautrier** [CeF78] defined a memory scheme is coherent if the value returned on a read is the value **given** by the latest store with the same address. Although most schemes currently in use satisfy this constraint by forcing all data to be coherent, the above definition **allows** incoherent data to **exist** in a coherent memory scheme as long as it is not fetched. Hence, the goal is to increase performance by taking advantage of the fact that data need not always be coherent.

The organization of this paper as follows: Section 2 discusses previous works. Section 3 discusses the parallel execution model used in this paper. Section 4 presents the algorithm for **marking** the reference as cache-read, cache-write, memory-read, and memory-write. Section 5 presents an example using algorithms developing in section 4. Section 6 generalizes the parallel execution model presented in section 3. Section 7 discusses some implementation issues of compiler-assisted cache coherence. Section 8 concludes the paper.

## 2. Previous Work

**Methods** for providing what appears to be a coherent distributed memory can be divided into two categories: cache coherence implemented by hardware [Ce78], [DuB82], [Egg89], [EgK89], [KEW85], [Tan76] or by a combination of simpler hardware and compile-time processing [ChV88], [CKM88], [CSB86], [Lee87], [Vei86]. We are most interested in the later, compiler-assisted cache coherence schemes; *i.e.*, those which involve the **compiler** generating different **code** for variable accesses that have different caching properties. **For** example, if a variable is **read** from main memory into the local caches of one or more **processors**, and then one processor writes a new value into that variable, there is a large amount of overhead involved in making all the locally cached copies coherent with the new value. These cached values can be made coherent by directing caches to invalidate their copy of the variable, thereby forcing any subsequent access to read the value directly from memory. Such action is not necessary if only one **cached** copy of the variable exists and it resides in the cache of the processor performing the write. **By** generating different code for this case, the hardware can avoid the overhead of needlessly communicating with other caches; the compiler can recognize such cases either by analysis or by the programmer's use of special directives. There are many possible variations on both the detection and treatment of such cache coherence problems.

### 2.1. Keeping Reads Coherent

The simplest approach to compiler-assisted cache coherence is to disallow the caching of shared **read/write** data for the duration of the entire program [WuB72]. This is accomplished by a compile-time marking of variables as either cacheable or non-cacheable; every shared variable that is writable is marked as **non-cacheable**. The mechanism is simple, but inefficient in the sense **that** every reference to non-cacheable data has to directly access main memory. Some variables marked as non-cacheable might be kept in cache without additional overhead for portions of the program's execution where those variables are either read-only or are accessed **exclusively** by one processor.

To conquer this performance penalty, Veidenbaum [Vei86] proposed a scheme that allows changing the cacheability marking of data when program execution crosses boundaries either

between parallel and serial sections of code or between two adjacent levels of nested parallel loops. **Caching** is prohibited inside loops with inter-iteration dependencies, and allowed otherwise. Although simulations [ChV87] predict good performance when the code between two boundaries provides sufficient temporal and spatial localities, there are two **potential** sources of **unnecessary** overhead [MiB89]. First, the blind invalidations of cache at the **boundaries** of loops will **sometimes** flush cache entries even though they hold up-to-date copies of data. Second, caching of read-only shared variables and variables that are exclusively accessed by only one **processor** inside **DoAcross** loops is possible, but is not done using this algorithm.

The first performance hit in Veidenbaum's scheme is reduced in the scheme proposed by Cheong and Veidenbaum [Che89] [ChV88]. Cheong proposed an updated version of Veidenbaum's scheme that incorporated fast selective invalidation using **data-flow** analysis to detect when caches may **be** incoherent. Each read reference is marked at compile time as either memory-read or cache-read. A read reference is marked as memory-read if the cache may potentially contain an out-of-date copy of the item to be read. A read reference will be marked as cache-read if it is guaranteed that the cache contains an up-to-date copy of the data. A small amount of additional hardware is needed for each cache to implement fast invalidation.

**The** second inefficiency of Veidenbaum's scheme was targeted by the techniques of McAuliffe [McA86] and Lee [Lee87]. In both of these techniques, cacheability of variables is determined **for** each region of code (epoch or computational unit). If a variable is potentially referenced by more than one processor in a given code region and at least one of the references is a write, the variable is marked as non-cacheable for that region. Otherwise, it is marked as **cacheable**.

Notice that all the above schemes apply equally well for both write-through and write-back handling of writes. In write-through, a processor writes directly into main memory; in **write-back**, the write first places data in the local cache and then propagates the data back to main memory. In either case, the cacheability marking of reads is simply **determined** by when the read **occurs** relative to the write.

## 2.2. Write-Through Vs. Write-Back

Although either **write-through** or write-back may be used, write-through requires less hardware; hence, Cheong suggested that write policy should be used. However, in terms of performance, write-back is generally more efficient than write-through [AnR89] [Goo83] [NoA82]. For example, in a shared-bus multiprocessor, write-back may result in less bus **traffic** since a sequence of writes to the same variable can result in a single bus access to **write** the last value into **main** memory. A similar benefit occurs in that for caches with multiple words per line, the line need not be written to main memory each time a word within the line is **written** into.

However, write-through can have better performance when only one or two words is written per **line**, since write-through would only transfer individual words to main memory, rather than the entire line. We suggest that this is a less likely scenario, as **evidenced** by the data of [AnR89] [Goo83] [NoA82], and hence will assume that write-back is used for all the caches **described** in this paper.

### 23. System-Level Issues

**Over** the last decade, cycle time has been decreasing much faster than main memory access time — **making** minimization of main memory references increasingly **important**. The average number of machine cycles per instruction has also been decreasing dramatically, **especially** for RISC **machine**. The two effects are multiplicative and results in a tremendous increase in miss cost. **For** example, a cache miss on a VAX 11/780 only costs 60% of the average insauction time. However, if a RISC machine like the **Stardent** Titan has a miss, the cost is ninefold the average instruction time.

Although a write-back policy, such as that of Lee and **McAuliffe** [Lee87], [McA86]. generally does well in reducing the number of main memory references. However, it does so only within **the** execution of a single task (process). The multiprocessor system must update the whole **cache** when a new task is initiated, and this requires memory accesses than would a **write-through** discipline like that proposed by **Cheong**. Thus, if task switching is common and **write-back** is to be used, it is important to observe that it is not always necessary to flush the entire **cache** at each task boundary.

### 3. Task Graph and Parallel Execution Model

**Since** the compiler will mark references with their cache-coherence properties before the program can be executed, it is not possible to base these markings on observations of the program in **execution**. Thus, we need some model of the program's execution **which** can be used by the compiler to predict how the program might behave. These predictions must be conservative. Some programs will only exhibit particular cache consistency problems if the program is run with certain inputs, or if it is executed by a certain number of processors, or if the runtime scheduling uses a particular heuristic; however, the compiler must detect and mark all places in the code in which these cache management problems could arise.

The model we use is based on the concept of a "task" — the smallest possible section of code that could be executed by a single processor. A purely sequential **program** consists of just one task; parallel programs are partitioned into multiple tasks related by a partial ordering [Pol86] [Tan89]. The program's execution may follow any complete ordering that meets the constraints of the partial ordering. Thus, the compiler's analysis of cache coherence could be

simplistically viewed as examining the task graph to determine:

- If an item is referenced only within one task, it cannot cause cache coherence problems. This follows from the assumption that each individual task must execute wholly within a single processor.
- If multiple tasks access the same item, exactly what cache coherence problems will arise. The type of coherence overhead can be determined by **observing** the partial ordering of the tasks; possibly concurrent tasks cause different problems from those which must **be** executed in series.

More precisely, we define a task graph,  $G = \{T, E, \psi, \psi^1\}$ , to **be** a directed graph.  $G$  is thus an **ordered** quadruple  $(T(G), E(G), \psi_G, \psi^1_G)$  consisting of a nonempty set  $T(G)$  of tasks, a set of  $E(G)$ , **disjoint** from  $T(G)$ , of arc, and incidence functions  $\psi_G$  and  $\psi^1_G$  that associate with each arc of  $G$  an ordered pair of (not necessarily distinct) task nodes of  $G$ . If  $a$  is an arc and  $x$  and  $y$  are task **nodes** such that  $\psi^1_G(a) = (x, y)$ , then  $a$  represents a control flow path from  $x$  to  $y$ . If  $a$  is an arc **and**  $u$  and  $v$  are task nodes such that  $\psi_G(a) = (u, v)$ , then  $a$  represents a data dependence relation **in** which data flows from  $u$  to  $v$ ; **i.e.**, some statements in  $v$  use data stored by statements in  $u$ .

We simplify the structure of this graph in two ways; one which never **harms** performance and a **second** which trades some performance for greatly simplifying the compiler analysis: respectively, task merging and level order scheduling.

Task merging attempts to make single tasks out of task sequences **which** are independent from other tasks but share data among themselves. Formally, given two nodes  $T_i$  and  $T_j$  connected by  $e_{ij}$ , they can **be** merged into one node if  $T_i$  has only one exit and  $T_j$  has only one entry. We **assume** all such task merges are performed.

Level order scheduling is a commonly used paradigm for implementing low-overhead parallel **schedules** without extending the critical path. Here, we use it to imply a complete order of **synchronization** points, which in turn allows our analysis to focus on what happens across all tasks in a level when the synchronization point ending that level is reached. Initially, we further restrict the tasks to appear in a level order schedule that is acyclic; **i.e.**, loops either are contained within **one** task or can **be** treated as defining edges across which our simple analysis algorithm will make conservative assumptions.

Formally, the level order schedule is derived by taking an acyclic task graph,  $G$ , and numbering each node with a label  $L_i$ , where  $i$  is the level of that node. All nodes that have no predecessors in  $G$  are assigned level 0 (denoted  $L_0$ ). From this set of start nodes, we use a greedy **scheduling** technique to obtain levels for the other tasks. Thus, for any node  $N \in G$ , the level of  $N$  is **max**(level of all predecessors of  $N$ ) + 1. This greedy schedule is then enforced

using a **barrier** synchronization to ensure that all tasks of level  $i$  have been executed before any task at level  $i + 1$  is initiated. Thus, there is no communication (*i.e.*, no sharing of writeable data) between tasks that can execute concurrently.

#### 4. Compiler-assisted Reference-marking Scheme

In general, compiler-assisted cache coherence is implemented by compile-time marking of references which might result in incoherent accesses. The reference marking process is carried out after partitioning a program into a task or tasks [Pol86], [Tan89]. Here we present a compiler algorithm that marks individual references as memory-read, cache-read, cache-write, or memory-write; thus, the compiler can generate code that efficiently manages the cache without coherence-control hardware and without violating the program's semantics.

##### 4.1. The Reference Marking Scheme

**Here** we assume that the input program already has been transformed into a set of acyclic graphs. Because different acyclic graphs do not write to the same memory address, there is no access to incoherent cache data among different acyclic graphs. Let us consider these acyclic graphs individually. Within a particular acyclic graph,  $G$ , the nodes have been assigned level numbers as described above. Because nodes assigned the same level number do not communicate, we need only consider interactions between nodes at different levels. Further, since the **execution** of nodes is strictly ordered by the barrier sequence, we need only **consider** the general case of **interactions** between the set of nodes at level  $L_i$  and those at level  $L_{i+1}$ .

By **examining** the read and write references in the source code, the **compiler** determines which read references will not access out-of-date cache copies. The read references that are known to access only up-to-date cache copies are cache-read accesses; the others are **memory-read** accesses. The write references that are known to be accessed by the next task level **are** memory-writes; the others are cache-writes.

The algorithm for marking read and write references begins with tasks in the top level,  $L_1$ , marks references within each task in that level, and then continues to the **next** level,  $L_2$ . This process **continues** for each level until all  $n$  levels have been marked. At the time level  $L_1$  is entered, the cache is assumed to be empty; thus, there is an imaginary level  $L_0$  that contains no **references**. Likewise, it is useful to imagine an empty final level,  $L_{n+1}$ .

The marking of read references is done according to the following rules:

1. A read reference to a read-only variable is marked cache-read.
2. A read reference within level  $L_i$  is marked memory-read if there is a task in level  $L_{i-1}$  that makes a write access to the same variable.

3. Any other type of read reference is marked as cache-read.

**Write references** are marked according to:

1. A write to a write-only variable is assumed to be a dead store and is removed from the program, **i.e.**, none will appear in the final code.
2. A write reference within level  $L_i$  is marked memory-write if there is a task in level  $L_{i+1}$  that make a read access to the same variable.
3. Any other type of write reference is marked as cache-write.

#### 4.2. Data-Flow Analysis

Before detailing the analysis, it is useful to define the terminology used. A write access to a variable is a definition (*def*) of the variable and a read access is a use. A *def* of a variable  $p$  is generated by a statement  $S$  when the statement contains a write reference to  $p$ . A variable  $d$  reaches a point  $p$  if there is a path from the point immediately following  $d$  to  $p$ . For a variable  $x$  and a point  $p$ , if the value of  $x$  at  $p$  could be used along some path in the flow graph starting at  $p$ , we say  $x$  is live at  $p$ ; otherwise  $x$  is *dead* at  $p$ .

The equations in Table 1 **are** an inductive, or syntax-directed, definition of the sets  $in(\mathbf{T})$ ,  $out(\mathbf{T})$ , and  $gen(\mathbf{T})$  for statements  $T$ . Note that these definitions differ slightly from those **normally** used in flow analysis.  $gen(\mathbf{T})$  is the set of variables definitely assigned values in the task  $T$  **prior** to any use of that variable in the task  $T$ . A gen set is computed for each task in the task graph; in addition, each task  $T$  has associated set values  $in(\mathbf{T})$  and  $out(\mathbf{T})$ . The  $in(\mathbf{T})$  set **contains** all variables whose values are written before entering  $T$  and read **within**  $T$ .  $out(\mathbf{T})$  is the set of variables written or read in task  $T$  whose values are available to be **read** by later tasks.

We define a portion of a task graph called a region to be a set of nodes  $N$  that includes a header, which dominates all other nodes in the region. All edges between nodes in  $N$  are in the region, **except** for loops that enters the header. The portion of a flow graph corresponding to a statement  $T$  is a region that obeys the further restriction that control can flow to just one outside block when it leaves the region.

For convenience we assume there **are** dummy nodes with no statements (indicated by small circles in Table 1) through which control flows just before it leaves the region. We say that the beginning points of the dummy blocks at the entry and exit of a statement's region are the beginning and end points, respectively, of the statement.

**Structured** control constructs have one property — a single entry and a single exit. Any structured control flow graph can be thought of as consisting of the composition of four basic **subgraph** patterns, shown in Table 1. Sections 4.2.1 through 4.2.4 give the analysis equations that are **used** to compute gen, in, and out for these constructs.

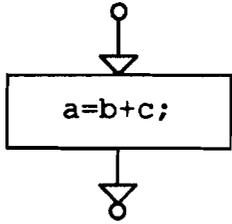
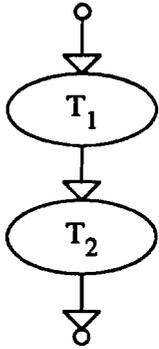
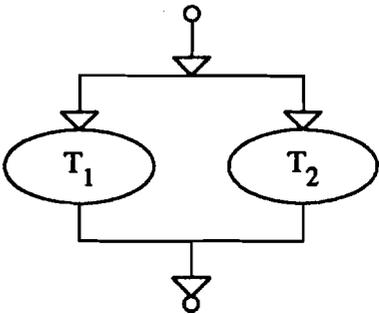
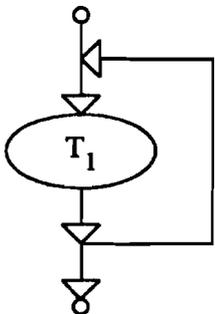
| Figure  | Equation   |
|---|--|
|    | $\begin{aligned} in(T) &= \{b, c\} \\ gen(T) &= \{a\} \\ out(T) &= in(T) \cup gen(T) = \{a, b, c\} \end{aligned}$  |
|    | $\begin{aligned} in(T) &= in(T_1) \cup \{in(T_2) - gen(T_1)\} \\ out(T_1) &= in(T_1) \cup gen(T_1) \\ out(T_2) &= in(T_2) \cup gen(T_2) \\ gen(T) &= gen(T_1) \cup gen(T_2) \\ out(T) &= out(T_2) \\ in(T_2) &= out(T_1) \end{aligned}$  |
|  | $\begin{aligned} in(T) &= in(T_1) \cup in(T_2) \\ out(T_1) &= in(T_1) \cup gen(T_1) \\ out(T_2) &= in(T_2) \cup gen(T_2) \\ out(T) &= out(T_1) \cup out(T_2) \\ gen(T) &= gen(T_1) \cup gen(T_2) \end{aligned}$  |
|  | $\begin{aligned} in(T) &= \bigcup_{\text{for all } n} \left\{ in(T_1^n) - \sum_{m=0}^{n-1} gen(T_1^m) \right\} \\ out(T_1^n) &= in(T_1^n) \cup gen(T_1^n) \\ gen(T) &= \bigcup_{\text{for all } n} gen(T_1^n) \\ out(T) &= \bigcup_{\text{for all } n} out(T_1^n) \end{aligned}$ |

Table 1 Data-flow equations

#### 4.2.1. An Individual Statement

Given a single assignment into the variable *a*, the equation for *gen* is:

$$\mathbf{gen(T) = \{a\}}$$

Suppose that the variables *b* and *c* are live before the assignment of variable *a*. Thus:

$$\mathbf{in(T) = \{b,c\}}$$

**Finally**, *out* can be computed as follows. First, if a variable is live at the beginning of *T*, then it **reaches** the end of *T*. If a variable is generated by *T*, it is reaching the end of *T*. Thus:

$$\mathbf{out(T)=in(T)\cup gen(T) = \{a, b, c\}}$$

#### 4.2.2. A Sequence of Statements

**The** equations for a cascade of statements, illustrated in Table 1, are more analytical. We start by **observing**  $\mathbf{in(T) = in(T_1)}$ . Then we can compute  $\mathbf{out(T_2)}$ , and this set equals to  $\mathbf{out(T)}$ .

Under what circumstances are variables live at the beginning of *T*? First of all, if it is live at the beginning of  $\mathbf{T_1}$ , then it surely is live at the beginning of *T*. If variables **are** live at the beginning of  $\mathbf{T_2}$ , they will be live at the beginning of *T*, provided each is not generated by  $\mathbf{T_1}$ . Thus:

$$\mathbf{in(T)=in(T_1)\cup \{ in(T_2)-gen(T_1) \}}$$

If a variable is generated by either  $\mathbf{T_2}$  or  $\mathbf{T_1}$ , it is generated by *T*. Thus:

$$\mathbf{gen(T)=gen(T_1)\cup gen(T_2)}$$

#### 4.2.3. Conditional Constructs

For an if-statement, illustrated in Table 1, we note that if variables are live at the beginning of either branch of the ‘if’, then they **are** live at the beginning of *T*. Thus:

$$\mathbf{in(T)=in(T_1)\cup in(T_2)}$$

**Similar** reasoning applies to the *out* and *gen*, so we have:

$$\mathbf{out(T)=out(T_1)\cup out(T_2)}$$

$$\mathbf{gen(T)=gen(T_1)\cup gen(T_2)}$$

#### 4.2.4. Looping Constructs

Lastly, consider the equation for loops, illustrated in Table 1. Unlike the other figures in Table 1, **we** can not simply use  $in(\mathbf{T})$  as  $in(\mathbf{T}_1)$ , because the variables reaching the end of  $\mathbf{T}_1$  will follow **the** arc back the beginning of  $\mathbf{T}_1$ . Some variables from  $out(\mathbf{T}_1)$  are part of the next iteration of  $in(\mathbf{T}_1)$ , but they are not part of  $in(\mathbf{T})$ .

We have to use an iterative approach for loop construct. Let us define  $in(\mathbf{T}_1^i)$  is the  $i$ -th iteration of  $in(\mathbf{T}_1)$  and the loop construct has  $n$  iteration. For every iteration,  $in(\mathbf{T}_1^i)$ ,  $gen(\mathbf{T}_1^i)$ , and  $out(\mathbf{T}_1^i)$  is computed as described in previous sections. The equation for  $gen(\mathbf{T})$  and  $out(\mathbf{T})$  are just the union of  $gen(\mathbf{T}_1^i)$ , and  $out(\mathbf{T}_1^i)$  respectively.  $in(\mathbf{T})$  is much more complicated. For every iteration,  $in(\mathbf{T}_1^n)$  subtract variables generated from previous iterations are part of  $in(\mathbf{T})$ . Thus:

$$in(\mathbf{T}) = \bigcup_{\text{for all } n} \left\{ in(\mathbf{T}_1^n) - \sum_{m=0}^{m=n-1} gen(\mathbf{T}_1^m) \right\} \quad (1)$$

$$out(\mathbf{T}_1^n) = in(\mathbf{T}_1^n) \cup gen(\mathbf{T}_1^n) \quad (2)$$

$$gen(\mathbf{T}) = \bigcup_{\text{for all } n} gen(\mathbf{T}_1^n) \quad (3)$$

$$out(\mathbf{T}) = \bigcup_{\text{for all } n} out(\mathbf{T}_1^n) \quad (4)$$

In equation (1)  $gen(\mathbf{T}_1^0)$  does not exist. We define:

$$gen(\mathbf{T}_1^0) = \phi \quad (5)$$

Similarly, if  $\mathbf{T}_1$  and  $\mathbf{T}_2$  are tasks instead of statements, the equations in Table 1 for a sequence of statement, conditional construct, and loop constructs can still apply to cascade tasks, **parallel/if-then-else** tasks, and **loop/recurrence** tasks respectively.

Similarly, if  $\mathbf{T}_1$  and  $\mathbf{T}_2$  are different graphs instead of statements, and  $\mathbf{T}_1$  and  $\mathbf{T}_2$  have different level, the equations in Table 1 for a sequence of statement, conditional construct, and loop constructs can still apply to cascade graphs, **parallel/if-then-else** graphs, and loop graphs respectively.

#### 4.3. Marking References

The previous section provides the basic equations to mark references (memory-read, **memory-write**, cache-read, and cache-write). Two sets of variables, memory-read and **write-back**, will be defined to represent memory-read variables, and memory-write variables. **write-back**( $\mathbf{T}$ ) is defined as the set of variables that have been updated in the task  $\mathbf{T}$  and will be used by successors of  $\mathbf{T}$  on the next task level. **memory-read**( $\mathbf{T}$ ) is the set of variables that might be

obsolete in cache and have to read up-to-date data from main memory in the current task **T**. **defunct(T)** is the set of variables that their values will not be used on successors of task **T**, therefore we do not have to update these variables back to main memory when reaching the end of task **T**. Let **T<sub>j</sub>** is a successor of **T** on the next level of the task **T**, **T<sub>k</sub>** is a predecessor of **T**, and **T<sub>l</sub>** is a successor of **T**.

**write-back(T)** is the set of variables that are generated in the task **T** and will be used by its successors on the next level:

$$\text{write-back}(\mathbf{T}) = \bigcup_{\text{for all } j} \left\{ \text{gen}(\mathbf{T}) \cap \text{in}(\mathbf{T}_j) \right\}$$

**memory-read(T)** is the set of variables that are generated by predecessors of **T** and are live in the task **T**:

$$\text{memory-read}(\mathbf{T}) = \bigcup_{\text{for all } k} \left\{ \text{gen}(\mathbf{T}_k) \cap \text{in}(\mathbf{T}) \right\}$$

**defunct(T)** is the set of variables whose values will not be used in successors of task **T**, therefore we do not have to update these variables back to main memory when task **T** is finished:

$$\text{defunct}(\mathbf{T}) = \text{out}(\mathbf{T}) - \bigcup_{\text{for all } l} \text{in}(\mathbf{T}_l)$$

The output of **write-back(T)** consists of variables that use memory-write instructions when variable!; need to be updated in the task **T**. Since our system has adopted a copy-back policy for cache update. In every task **T** the last write instruction whose variables belong to **write-back(T)** can use a memory-write instruction.

## 5. An Example

A sample code to show how to apply equations obtained from previous section to mark different memory access is:

```
for(j=1; j <= 9; j++)
    for(i=1; i <= 3; i++) {
        a(i, j) = a(i, j-1) + c(i, j) + a(i+1, j-1)
        b(i, j) = a(i, j) + c(i, j)
    }
```

Figure 1 Sample program

Let us define the loop body as  $B(i, j)$  and consider  $B(i, j)$  is a task unit. From data dependency analysis,  $B(1, i)$ ,  $B(2, i)$ , and  $B(3, i)$  do not have any data dependency among them. Therefore,  $B(1, i)$ ,  $B(2, i)$ , and  $B(3, i)$  can execute in parallel. The task graph can be shown in Figure 2. Then apply the equations shown in the previous section to calculate the in, gen, out, write-back, and *memory-read*. The partial equation output is shown in Table 2. We can find the pattern for in, gen, *out*, write-back, and memory-read, from Table 2. The complete equation output is shown in Table 3.

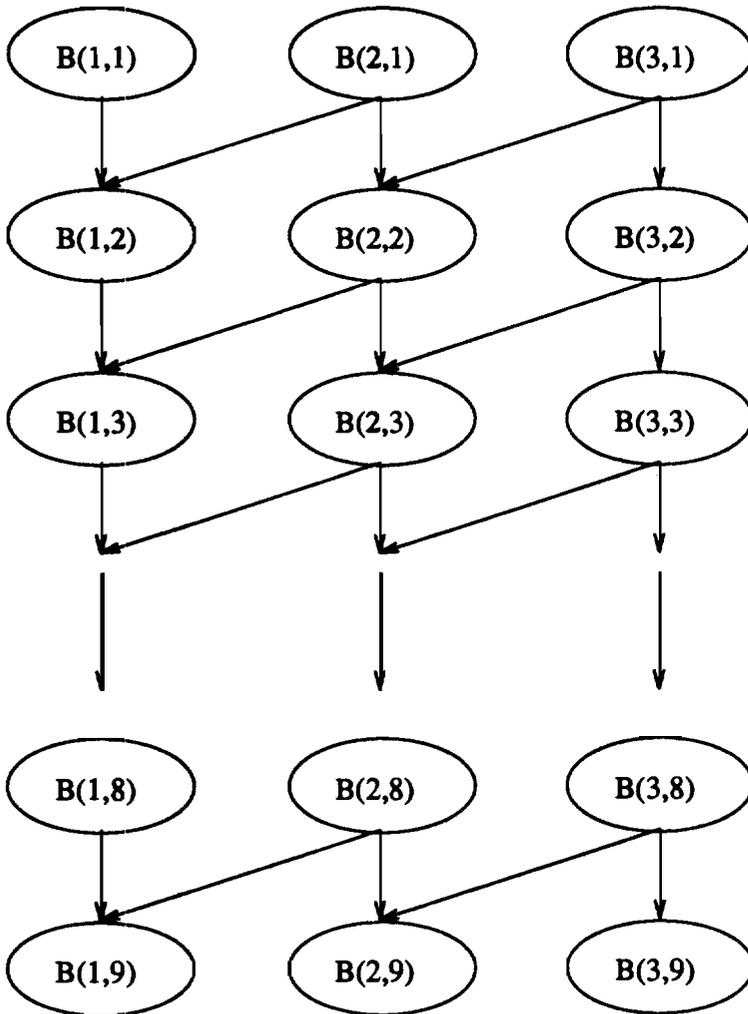


Figure 2 task graph

From Table 3,  $a(i, j-1)$  and  $a(i+1, j-1)$  are two variables that need memory-read instructions.  $a(i, j)$  needs a memory-write instruction. We can add comments that show what kind of instruction every variable needs to the sample program shown in Figure 3. Two comment lines correspond to two statements in the loop body.

| task level | loop body | <i>in</i>                    | gen               | <i>out</i>                                 | write-back | memory-read      |
|------------|-----------|------------------------------|-------------------|--|------------|------------------|
| 1          | B(1,1)    | a(1,0),<br>c(1,1),<br>a(2,0) | a(1,1),<br>b(1,1) | a(1,0),<br>a(2,0),a(1,1),<br>b(1,1),c(1,1) | a(1,1)     | $\phi$           |
| 1          | B(2,1)    | a(2,0),<br>c(2,1),<br>a(3,0) | a(2,1),<br>b(2,1) | a(2,0),<br>a(3,0),a(2,1),<br>b(2,1),c(2,1) | a(2,1)     | $\phi$           |
| 1          | B(3,1)    | a(3,0),<br>c(3,1),<br>a(4,0) | a(3,1),<br>b(3,1) | a(3,0),<br>a(4,0),a(3,1),<br>b(3,1),c(3,1) | a(3,1)     | $\phi$           |
| 2          | B(1,2)    | a(1,1),<br>c(1,2),<br>a(2,1) | a(1,2),<br>b(1,2) | a(1,1),<br>a(2,1),a(1,2),<br>b(1,2),c(1,2) | a(1,2)     | a(1,1)<br>a(2,1) |
| 2          | B(2,2)    | a(2,1),<br>c(2,2),<br>a(3,1) | a(2,2),<br>b(2,2) | a(2,1),<br>a(3,1),a(2,2),<br>b(2,2),c(2,2) | a(2,2)     | a(2,1)<br>a(3,1) |
| 2          | B(3,2)    | a(3,1),<br>c(3,2),<br>a(4,1) | a(3,2),<br>b(3,2) | a(3,1),<br>a(4,1),a(3,2),<br>b(3,2),c(3,2) | a(3,2)     | a(3,1)<br>a(4,1) |
| 3          | B(1,3)    | a(1,2),<br>c(1,3),<br>a(2,2) | a(1,3),<br>b(1,3) | a(1,2),<br>a(2,2),a(1,3),<br>b(1,3),c(1,3) | a(1,3)     | a(1,2)<br>a(2,2) |
| 3          | B(2,3)    | a(2,2),<br>c(2,3),<br>a(3,2) | a(2,3),<br>b(2,3) | a(2,2),<br>a(3,2),a(2,3),<br>b(2,3),c(2,3) | a(2,3)     | a(2,2)<br>a(3,2) |
| 3          | B(3,3)    | a(3,2),<br>c(3,3),<br>a(4,2) | a(3,3),<br>b(3,3) | a(3,2),<br>a(4,2),a(3,3),<br>b(3,3),c(3,3) | a(3,3)     | a(3,2)<br>a(4,2) |

Table 2 Partial equation output for Figure 1

| task level        | loop body                          | <i>in</i>                        | <i>gen</i>        | <i>out</i>                                     | <i>write-back</i> | <i>memory-read</i>      |
|-------------------|------------------------------------|----------------------------------|-------------------|--|-------------------|-------------------------|
| 1                 | <b>B(i,1)</b><br>$1 \leq i \leq 3$ | a(i,0),<br>c(i,1),<br>a(i+1,1)   | a(i,1),<br>b(i,1) | a(i,0),<br>a(i+1,1),a(i,1),<br>b(i,1),c(i,1)   | a(i,1)            | $\phi$                  |
| $2 \leq k \leq 9$ | <b>B(i,j)</b>                      | a(i,j-1),<br>c(i,j),<br>a(i+1,j) | a(i,j),<br>b(i,j) | a(i,j-1),<br>a(i+1,j),a(i,j),<br>b(i,j),c(i,j) | a(i,j)            | a(i,j-1)<br>a(i+1, j-1) |

Table 3 Complete equation output for Figure 1

The first comment maps to the first statement. A memory-write instruction is needed for a(i). Memory-read instructions are needed for a(i,j-1) and a(i+1,j-1). A cache-read instruction is needed for c(i, j).

It is similar for the second statement. A cache-store instruction is needed for b(i,j). A cache-read instruction are needed for c(i,j).

Given the above marking, we can calculate the improvement in cache access for this example. Assume that floating-point data take 4 bytes of memory space. Memory-,write instructions take twice as long as memory-read instructions. Memory-read instructions takes 4 times longer than cache reads. Cache block size is 12 bytes. We simplify this performance analysis by assuming that cache size is infinite so that cache replacement is never necessary. The cache data looks like Table 4. Compared to other software-based cache coherence methods [Che89], [Lee87], our result reduces memory access time by 38%. Even if we make cache size just 4 bytes, our method yields 26% less memory access time than the other methods.

```

for(j=1;j <= 9; j++)
  for(i=1;i <= 3; i++) {
    a(i, j) = a(i, j-1) + c(i, j) + a(i+1, j-1)
    /*memory-write, memory-read, cache-read, memory-read*/

    b(i, j) = a(i, j) + c(i, j)
    /*cache-store, cache-read, cache-read*/
  }

```

Figure 3 Sample program

|        |        |        |
|--------|--------|--------|
| a(1,1) | a(1,2) | a(1,3) |
| a(2,1) | a(2,2) | a(2,3) |
| a(3,1) | a(3,2) | a(3,3) |
| :      | :      | :      |
| b(1,1) | b(1,2) | b(1,3) |
| b(2,1) | b(2,2) | b(2,3) |
| b(3,1) | b(3,2) | b(3,3) |
| :      | :      | :      |
| c(1,1) | c(1,2) | c(1,3) |
| c(2,1) | c(2,2) | c(2,3) |
| c(3,1) | c(3,2) | c(3,3) |
| :      | :      | :      |

Table 4 Cache Organization

## 6. Generalization of Execution Model

In Section 3, we simplify the execution model by using level order **scheduling**. Synchronization is required at the end of every level. This restriction allows our analysis to focus on what happen **across** tasks in a level when synchronization points are reached. However, this restriction can **delay** execution by introducing otherwise unnecessary synchronization.

For example, consider the execution graph in Figure 6.1 given that the execution of  $A_{11}$ ,  $A_{12}$ , and  $A_{13}$  takes 10, 1, and 2 seconds respectively. Then  $A_{21}$ ,  $A_{22}$ , and  $A_{23}$  will execute after 10 seconds due to the synchronization at the end of  $A_{11}$ ,  $A_{12}$ , and  $A_{13}$ . However, if we remove that synchronization,  $A_{22}$  and  $A_{23}$  can execute after just 2 seconds.

**Assume** there is a data dependency between  $A_{11}$  and  $A_{32}$ , shown in Figure 6.2. According to our **reference** marking scheme, modified data that will be used in  $A_{32}$  only will not be updated at the encl of  $A_{11}$ . If we remove the synchronization point at the end of  $A_{11}$ ,  $A_{12}$ , and  $A_{13}$ , some modified data will not update in the main memory when  $A_{32}$  is executed. We have to add a data dependency between  $A_{21}$  and  $A_{32}$  to ensure the result is correct.

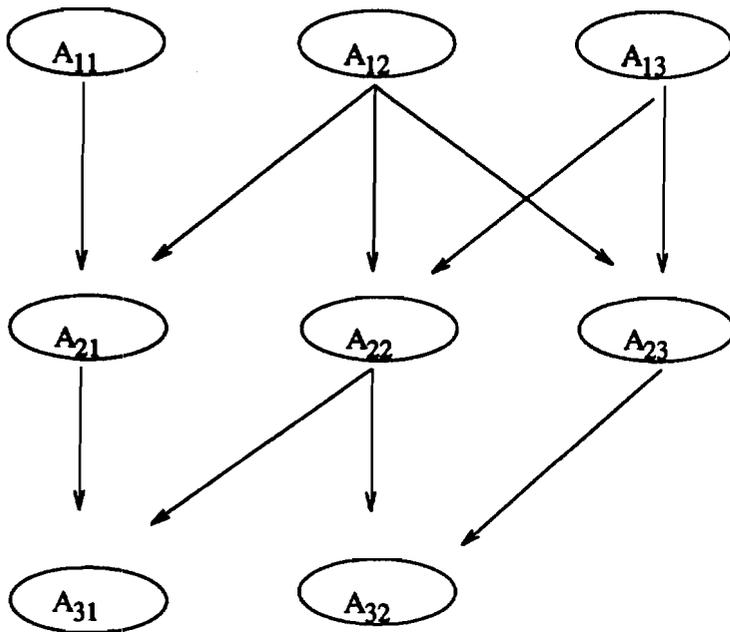


Figure 6.1 example execution graph

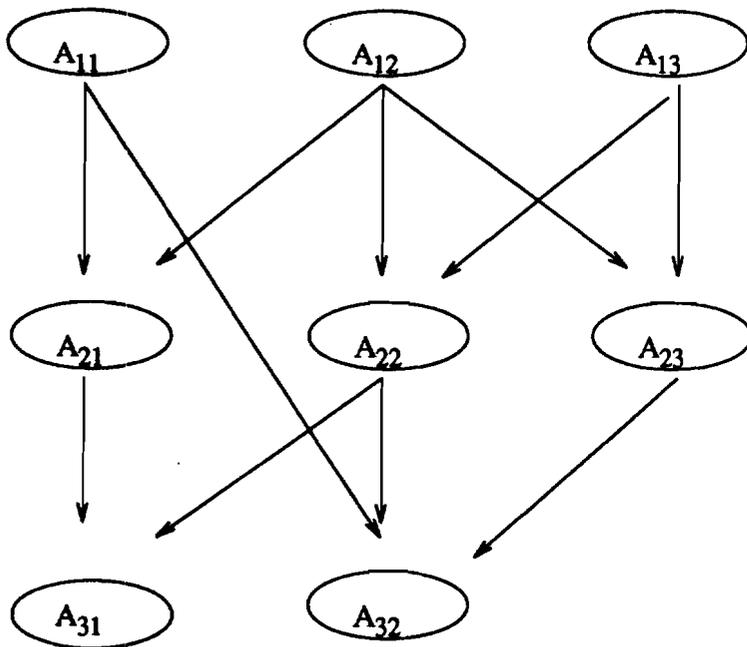


Figure 6.2 example execution graph

In this section, we will remove the synchronization requirement for every level and discuss when it is necessary to add artificial data dependencies between nodes to maintain the program's semantics.

### 6.1. Algorithm to Add Artificial Data Dependencies

Before adding artificial data dependencies, there are methods to reduce the number of artificial data dependencies that will be needed. For example, restructuring the program so that there is no data dependency between tasks across more than one task level ahead. If there are some data dependencies between tasks across more than one task level, we can force these variables to be written back to main memory when the end of the task is reached; of course, this method does not take advantage of the cache across task boundaries.

Our algorithm is:

1. **Restructure** the source program in the task such that all variables will be used by tasks in the next level, and every task takes about the same time.

2. For any task node T on the level j, if there exists a data dependency between task node T and task node K on the level n, where  $n - j > 1$ . Add artificial data dependency between task node m on the level  $n - 1$  to task node K if there exists a path from task node K to a task node m.

## 7. Implementation Issues of Compiler-assisted Cache Coherence

If we apply our compiler analysis to mark the reference in multiprocessor systems that adopt a write-back policy, there are two potential problems: false sharing and validation of cache data. In the next two sub-sections we will discuss these problems and how to solve them.

### 7.1. False Sharing

In multiprocessor systems with cache that has a multi-word cache block organization, variables a and b take one word each and are located in the same cache block. Processor 1 and Processor 2 both can access variables a and b; e.g., Processor 1 updates variable a and Processor 2 updates variable b. If these local updates occur simultaneously, a system using a write-back update policy cannot correctly update the variables a and b in main memory. This problem is called false sharing. This can be solved either by hardware approach or software approach, as described in the next two sections.

#### 7.1.1. Software Solution

One possible software solution is to allocate variables only on cache-block boundaries in memory. This will solve the false sharing problem caused by multiple-word cache blocks because each cache block contains only one variable. However, this solution does not make the maximum utilization of cache block space.

A refinement to the above scheme is to simply allocate variables such that no two variables that can be updated simultaneously on different processors reside in memory locations that

would map to the same cache line. Unfortunately, such data layout requires relatively complex analysis and data layout, similar to that proposed in [JuD92].

### 7.13. Hardware Solution

Setting the cache block size equal to word size is one possible hardware solution, but this is an inefficient way to organize a cache because it requires more cache address tags. Some systems also can move a blocks from memory to cache with higher throughput **if** the blocks are larger (**e.g.**, using block-mode memory transfers). However, perhaps the worst problem with word-sized cache lines is that *not all objects referenced in a program have the same word size*. **For** example, `char`, `float`, and `double` variables are usually different sizes — which should be the cache line size?

Cache sub-block structure can solve this problem by updating portions of cache entries separately. The cache memory is organized into block frames each of which **holds** a copy of a fixed-size block of memory, the base address in main memory from which the copy was made, and some: status bits. The block size is usually the most efficient unit of transfer between the cache and the main memory. If only part of such a block is modified, only a portion of the block is sent. The transfer unit size is called the sub-block size.

### 7.2. Validation of Cache Data

The lack of run-time information on task scheduling makes it difficult to determine which data on the cache are valid after processor execution reaching task boundary. A hardware organization is proposed to make the detection of obsolete entries easier and to maximize the utilization of up-to-date data in the cache.

The memory accesses (cache write, cache read, memory read, and memory write) within and across the task boundaries will effect whether cached data is up-to-date **or** obsolete. The state diagram in Figure 4 models the transition of reference on the different states. A cache word can be in one of five states. The edges denote state transitions and the labels on the edges denote the causes of the state transitions. The state diagram shown in Figure 4 **can** be tabulated as shown in the Table 5.

**For** any variable that executes a write operation in a task, it will go to either the **Cache-write** state or the **Memory-write** state. If a variable with a write operation is needed for the next **task** level., a memory-write instruction is executed instead of a cache-write one. A variable using memory-write will update its cache data across the task boundary if it does not perform cache replacement update within the task. A variable using cache-write will go to a memory-write state **when** the current task finish if it does not perform cache replacement update within the task.

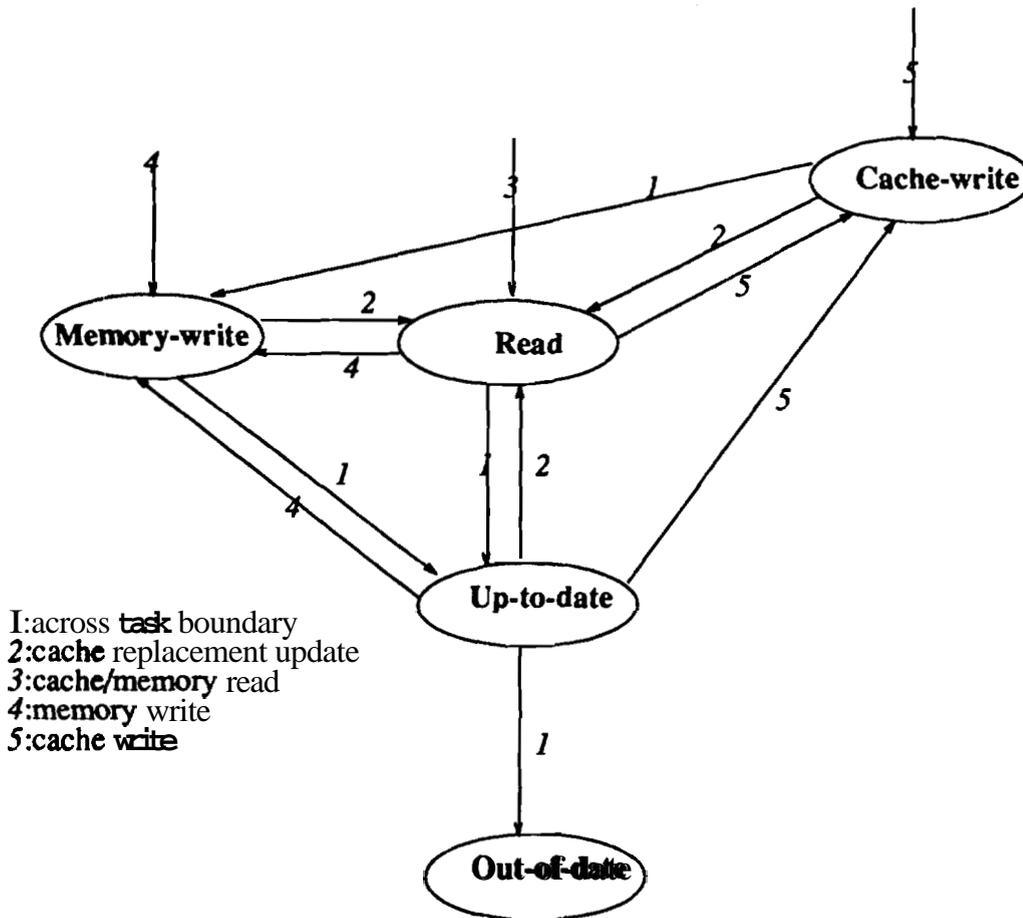


Figure 4 State and state transitions of a cache word

This means cache data must be updated in the next task level if they are not updated in the present task level. If a variable is either in Memory-write or in Cache-write state, after it performs cache replacement update, it enters into the Read state.

If an instruction is either cache-read or memory-read, the cache word **will** enter into the Read state. When the **current** task is finished, if variables are in the Read state, they will go to the **up-to-date** state. After finishing the current task level, the cache word has the up-to-date data if the cache word is in one of the two states: Read, and Memory-write. This means that after crossing a task boundary, if cache data are in the Memory-write state, or the Read state, then they are up-to-date.

If cache data are in the Up-to-date state and they do not go to one of the following states: Read, Memory-write or Cache-write states, then they enter the out-of-date state after finishing the **current** task level.

For memory-read instructions, check the status on the cache if the data is on the cache. If the **data is** in the cache, unless its status is out-of-date, memory-read instruction will fetch data from cache instead of fetching data from main memory.

| Operation                 | State before execution        | State after execution |
|---------------------------|-------------------------------|-----------------------|
| cache write               | any state except Memory-write | Cache-write           |
| cache write               | Memory-write                  | Memory-write          |
| memory write              | any state                     | Memory-write          |
| cache replacement update  | Cache-write                   | <b>Read</b>           |
| cache read or memory read | Any states                    | Read                  |
| Across task boundary      | Cache-write                   | <b>Read</b>           |
| Across task boundary      | Memory-write                  | <b>Read</b>           |
| Across task boundary      | Up-to-date                    | Out-of-date           |
| Across task boundary      | Cache-write                   | <b>Memory-write</b>   |
| Across task boundary      | Read                          | <b>Up-to-date</b>     |
| Across task boundary      | Memory-write                  | Up-to-date            |

Table 5 State and state transitions of a cache word

### 8. Conclusions and Future Work

This paper presents a compiler-assisted cache coherence scheme that can reduce the communication cost in cache coherence maintenance and maintain as much temporal locality as possible. Although other schemes have been proposed, the method presented **here** typically will perform better because it is based on analysis that goes across **tasks**, whereas other systems make conservative assumptions about updates needed at task boundaries. Although this paper is primarily a theoretical presentation and does not focus on detailed benchmarks, even a very simple benchmark (in section 5) yields 268-3896 faster memory access times than competing schemes.

Future work will be extended our compiler-assisted reference-marking scheme to multiprocessor systems that use a multi-level cache heirarchy [Lio93].

## References

- [AnR89] M., Annaratone, R. Ruhl, "Performance Measurements on a Commercial Multiprocessor Running Parallel Code." The 16th Annual International **Symposium** on Computer Architecture, 307-314, 1989
- [Bro91] E. Brooks, comments taken from a multitude of postings on `comp.arch` about "Killer Micros," 1991.
- [CeF78] L.M. Censier, and **Feautrier**, "A New Solution to Coherence Problems in Multicaches Systems," IEEE Transactions on computers, Vol. C-27, No. 12, December 1978
- [Che89] H. Cheong, Compiler-Directed Cache Coherence strategies for large-scale shared-*memory* multiprocessor systems , **Ph.D.** Thesis, University of Illinois at Urbana-Champaign, December 1989
- [Chi89] C.-H. Chi, Compiler-Driven Cache Management Using A State Level Transition Model, **Ph.D.** Thesis, **Purdue** University, May 1989
- [ChV87] H. Cheong, and A.V. Veidenbaum, "The performance of software-managed multiprocessor cache on parallel numerical programs," Proc. of 1st International Conference on Supercomputing , 316-337, June 1987
- [ChV88] H. Cheong, and A.V. Veidenbaum, "Stale Data Detection and Coherence Enforcement Using Flow Analysis," Proc. of International Conference on Parallel Processing, **138-145**, Vol. 1, August 1988
- [CKM88] R. Cytron, S. Karlovsky, and K.P. **McAuliffe**, "Automatic Management of Programmable Caches," Proc. of International Conference on Parallel Processing, 229-238, August 1988
- [CSB86] D.R. Cheriton, G.A. Slavenburg, and P.D. Boyle, "Software-Controlled Caches in the VMP Multiprocessor," The 13th Annual International Symposium on Computer Architecture, 366-374, 1986
- [DuB82] M. **Dubois**, and F.A. Briggs, "Effects of Cache Coherency in Multiprocessors," IEEE on Computers. Vol. C.31 No. **11.**, **1083-1099**, November 1982
- [Egg89] S.J. Eggers, Simulation Analysis of Data Sharing in Shared Memory Multiprocessors, **Ph.D** Thesis, University of California at Berkeley, March 1989
- [EgK89] S.J. Eggers, and R.H. Katz, "Evaluating the Performance of Four Snooping Cache Coherency Protocols," The 16th Annual International Symposium on Computer Architecture, 2-15, 1989
- [Goo83] J.R. Goodman, "Using Cache Memory to Reduce Processor-Memory Traffic," The **10th** Annual International Symposium on Computer Architecture, 124-131, 1983
- [HiS89] M.D. Hill, and A.J. Smith, "Evaluating Associativity in CPU Caches," IEEE Tans. on Computers, Vol. 38, No. 12, pp. 1612-1630, December 1989
- [JuD92] Y-J. Ju and H. G. Dietz, "Reduction of Cache Coherence Overhead by Compiler Data Layout and Loop Transformation," Languages and Compilers for Parallel Computing, edited by U. Banerjee, D. Gelernter, A. Nicolau, and D. **Padua**, Springer-Verlag, New York, New York, 1992, pp. 344-358.

- [KEW85] **R.H.Katz**, S.J. Eggers, D.A. Wood, C.L. **Perkins**, and R.G. Sheldon "Implementing a Cache Consistency Protocol," The 12th Annual International Symposium on Computer Architecture, 276-283, 1985
- [LaD90] S. **Lakshmivarahan** and S.K. Dhall, "Analysis and Design of **Parallel** Algorithms: Arithmetic and Matrix Problems," **McGraw-Hill** Publishing Company, 1990
- [Lee87] R.L. **Lee**, The *Effectiveness* of Caches and Data Prefetch *Buffers* in Large-Scale Memory Multiprocessor, **Ph.D.** Thesis, University of Illinois at Urbana-Champaign, May 1987
- [Lio93] C. **Liou**, Compiler-Assisted Cache Coherence, **Ph.D.** Thesis, Purdue University, 1993 in preparation.
- [Lip68] J.S. Liptay, "Structural Aspects of the **System/360** Model 85, Part **II**: The Cache", IBM Systems Journal, 7, 15-21, 1968
- [McA86] K.P. **McAuliffe** Analysis of Cache Memories in Highly Parallel Systems, **Ph.D** Thesis, New York University, 1986
- [MiB89] S.L. Min, and J.L. Baer, "A Timestamp-based Cache Coherence Scheme," Proc. of International Conference on Parallel Processing, 24-32, August, 1989
- [NoA82] R.L. Norton, J.A. Abraham, "Using Write Back Cache to Improve Performance of Multiuser Multiprocessors," Proc. of the International Conference on Parallel Processing, pp. 326-331, 1982
- [Pol86] C.D. Polychronopoulos, On Program Restructuring, Scheduling, and Communication fro Parallel Processor System ,**Ph.D.** Thesis, University of Illinois at Urbana-Champaign, August 1986
- [Puz85] T.R. **Puzak**, Analysis of Cache Replacement *Algorithms*, **Ph.D** Thesis, University of Massachusetts, 1985
- [Smi82] A.J. Smith, "Cache Memories." Computing Surveys, Volume 14, Number 3, pp. 473-530., September 1982
- [Smi85] A.J. Smith, "Cache Evaluation and The Impact of Workload Choice," The 12th Annual International Symposium on Computer Architecture, 64-73, 1985
- [Smi86] A.J. Smith, "Bibliography and Readings on CPU Cache Memories and Related Topics," Computer Architecture News, 22-42, January 1986
- [Smi87] A.J. Smith, "Line (Block) Size Choice for CPU Cache Memories," IEEE Transactions on Computers, Volume C-36, Number 9, pp. 1063-1075., September 1987
- [Tan76] C.K. Tang, "Cache System Design in the Tightly Coupled Multiprocessor System," AFIPS Proc., National Computer Conference, Vol. 45, pp. 749-753, 1976
- [Tan89] P. Tang, Self-Scheduling, Data Synchronization and Program *Transformation* for Multiprocessor System ,**Ph.D.** Thesis, University of Illinois at Urbana-Champaign, January 1989
- [WuB72] W.A. Wulf and C.G. Bell, "**C.mmp** - A multi-mini processor," Proc. of Fall Joint Computer Conference, Montvale, New Jersey, pp. 765- 777, December 1972
- [Vei86] A.V. Veidenbaum, "A Compiler-Assisted Cache Coherence Solution For Multiprocessors," Proc. of International Conference on Parallel Processing, 1029-1036, **August**, 1986