Delayed precise invalidation – a software cache coherence scheme

T.-S. Hwang
N.-P. Lu
C.-P. Chung

Indexing terms: Cache coherence, Compilers, Invalidation

Abstract: Software cache coherence schemes are very desirable in the design of scalable multiprocessors and massively parallel processors. The authors propose a software cache coherence scheme named 'delayed precise invalidation' (DPI). DPI is based on compiler-time markings of references and a hardware local invalidation of stale data in parallel and selectively. With a small amount of additional hardware and a small set of cache management instructions, this scheme provides more cacheability and allows invalidation of partial elements in an array, overcoming some inefficiencies and deficiencies of previous software cache coherence schemes.

1 Introduction

To enforce data coherence among multiple private caches in shared-memory multiprocessor systems, a great number of schemes have been proposed. These schemes may be classified as either hardware or software cache coherence schemes [1]. The hardware cache coherence schemes [1–3] use shared resources (bus or directory) to maintain cache coherence. The software cache coherence schemes [4–9] maintain cache coherence using information obtained in compiler-time analysis.

The advantages of software cache coherence schemes are two-fold: first, the scalability of the underlying network architecture is not constrained, and secondly, the runtime overhead is minimised. While bus-based cache coherence schemes [2] have scalability difficulties for large shared-memory multiprocessor systems, directory-based cache coherence schemes [3] also suffer from difficulty extending to large-scale shared-memory multiprocessor systems, since the centralised or distributed directory may become a serious performance bottleneck. On the other hand, by making each processor responsible for maintaining data coherence in its own cache via self-invalidation instructions, software cache coherence schemes are not subject to network topologies and further eliminate runtime interprocessor cache coherence traffic.

The performance of software cache coherence schemes relies on precise compiler-time analysis that can eliminate unnecessary invalidations and preserve more cache localities. In this paper, we propose the delayed precise invalidation (DPI) scheme to overcome some inefficiencies and deficiencies of previous schemes.

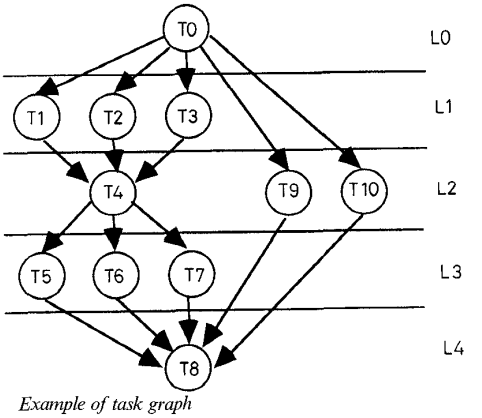

Fig. 1 *Example of task graph*

2 Background

In studying software cache coherence schemes the execution of a program can be modelled with a task graph, as illustrated in Fig. 1. A serial loop that cannot be parallelised is a task. For a DoAll or DoAcross loop, one iteration or a number of iterations can be considered as a task. A directed edge from one task to another represents all the dependencies between the two tasks. Tasks independent of each other can be scheduled for parallel execution. Dependent tasks will be executed in the order defined by program semantics. The execution order of dependent tasks is enforced through synchronisation. To maintain data coherence in the system, software cache coherence schemes use compiler-time analysis to determine which cache items may become stale, and insert special cache management instructions to invalidate the stale entries when processors cross task boundaries.

The simplest software cache coherence scheme is to use indiscriminate invalidation of the data caches to enforce coherence when processors cross task boundaries [4]. The indiscriminate invalidation scheme maintains cache coherence at the sacrifice of data

IEE Proceedings online no. 19960661

Paper first received 30th October 1995 and in revised form 20th May 1996

The authors are with the Department of Computer Science and Information Engineering, National Chiao Tung University, 1001 Ta Hsueh Road, Hsinchu, Taiwan 30050, Republic of China

cacheability. The fast selective invalidation scheme [5] enforces coherence at the point of a read reference so that the data cacheability can be improved. Every read reference to shared memory in a program will be classified and marked by the compiler as either memory-read or cache-read. A cache controller treats a cache-read as a read to a conventional uniprocessor cache. A memory-read implies that the data in the cache might be stale, and up-to-date data should be loaded from the main memory.

To exploit higher cacheability, two similar schemes, the timestamp-based scheme [7] and the version control scheme [6], have been proposed. Clock and timestamp in the former scheme correspond to the current version number (CVN) and birth version number (BVN) in the latter one, respectively. In the version control scheme, each processor maintains a CVN for each variable and a BVN for each cache line. The CVN represents the current version of the variable, while the BVN represents a particular version of the cache line. By comparing the CVN and BVN, a stale cache copy can be detected. The timestamp-based schemes use both compile-time markings of references and a local coherence detection mechanism to overcome the difficulty with limited information about the stale data at compiler time. In so doing, these two schemes are able to better preserve temporal locality across the boundaries of loops than previous schemes. However, these schemes require substantial additional hardware to maintain the time or version information. To reduce the hardware cost, one-bit time stamping (TS1) has been proposed in [9]. TS1 only requires a valid bit and an epoch bit per cache line, but it requires a more sophisticated invalidate instruction to maintain cache coherence.

```
                Dimension X(8)          (address encoding of X)
P_loop          DoAll i = 0 to 7        X(0)->10110000
                .....                   X(1)->10110001
S1                X(i) = ...            X(2)->10110010
                .....                   X(3)->10110011
                END DoAll               X(4)->10110100
Serial          .....                   X(5)->10110101
                .....                   X(6)->10110110
S2              X(3) = ...              X(7)->10110111
                .....
```

Fig.2 *Program example to illustrate PEI scheme*

Timestamp-based schemes operate on the whole array level: they invalidate all elements in an array even when only a small part of the array is modified, because an array is associated with only one CVN and one BVN. To correct this deficiency, the parallel explicit invalidation (PEI) scheme [8] allows parallel invalidation of all or partial array elements with the assistance of a well-structured memory allocation method and a store-and-invalidate instruction. We show how the PEI scheme works with the program example shown in Fig. 2. Assume that eight processors P_i (i = 0, ..., 7) are involved in executing the program and the processors are assigned such that the ith iteration of the DoALL loop is executed by P_i and processor P_0 executes the serial region. The addresses of $X(0)$, $X(1)$, ..., $X(7)$ are assigned 10110000, 10110001, ..., 10110111, respectively. Then, in the DoAll loop, P_i will use the store-and-invalidate instruction to modify $X(i)$ and invalidate all the cache copies of array X, except the $X(i)$, at the same time. In the serial region, P_0 will modify $X(5)$ while the other idle processors should also invalidate the copies of $X(5)$ in their caches, if they exist. The PEI scheme achieves cache coherence since anything written on a different processor will be invalidated by the local processor before moving to the next task.

```
                DoAcross i = 1 to 4
                .....
S1                X(i) = ...
                .....
S2              X(i + 1) = ...
                .....
                ... = X(i + 1)
                .....
                END DoAcross
```

Fig.3 *Example of cross-iteration output dependence*

However, a problem for DoAcross loops arises in the PEI scheme. The scheme may not flush the stale cache copies for a cross-iteration output dependence in a DoAcross loop. Consider the program segment in Fig. 3 and assume that four processors, P_1, P_2, P_3, P_4, are involved in executing the program. The processors are assigned such that ith iteration is executed on P_i. For a cross-iteration output dependence that exists in a DoAcross loop, the Sink-Write must overwrite the value produced by the Source-Write with proper synchronisation. In such a case, the cache copy produced by the Source-Write is stale, and the cache copy produced by the Sink-Write is up-to-date, in the subsequent task. In the PEI scheme, when processor P_1 executes iteration I_1 and statement S1 is reached, it modifies $X(1)$ and invalidates cache copies of $X(2)$, $X(3)$, and $X(4)$, if they exist. When statement S2 is reached, processor P_1 modifies $X(2)$ and invalidates cache copies of $X(1)$, $X(3)$, and $X(4)$, if they exist. The cache copy of $X(2)$ is not invalidated by processor P_1 and exists in the cache of processor P_1. Since statement S2 is the Source-Write in the cross-iteration output-dependent DoAcross loop, the cache copy of $X(2)$ is stale in the subsequent task and must be invalidated by processor P_1. However, the PEI scheme does not invalidate the cache copy of $X(2)$ in the cache of processor P_1. To solve this PEI scheme problem, the processors should execute the invalidate instruction before moving to the subsequent task, so that the stale data in the cache will not persist for a cross-iteration output-dependent DoAcross loop.

3 Delayed precise invalidation scheme

3.1 Motivation

In addition to the error in the cross-iteration output-dependent DoAcross loop, the PEI scheme has another drawback: its store-and-invalidate instruction will write the datum and invalidate the other cache data at the same time. Consequently it may invalidate the up-to-date data and cause unnecessary cache misses. To solve the deficiency of the PEI scheme and provide more cacheability, we propose the delayed precise invalidation (DPI) scheme. The DPI scheme delays the invalidate operations to only the necessary array elements on the processors entering the subsequent task: hence 'delayed'. Furthermore, the DPI scheme allocates shared data structures or scalars in a well structured form to support precise invalidation of the stale data in parallel and selectively: hence 'precise'. As a result, the DPI scheme avoids invalidating the up-to-date cache copies, and preserves the intertask or intratask temporal locality better than the PEI scheme does.

3.2 System model

In the DPI scheme, we assume a shared-memory multiprocessor system in which every processor has local data cache with write-through policy. Write-through policy is essential to the success of the scheme. A cache line is the basic cache coherence unit, and we choose the one-word line size in the scheme to avoid false sharing. (Multiword line size is possible in the scheme. To eliminate false sharing of multiword line size, allocating memory must take the line size into account carefully.) We also assume processor pre-emption and processor migration are not allowed in the scheme.

Numerical programs with DoAll-type or DoAcross-type loop-level parallelism are assumed to be executed in the system. All the programs are constructed by a parallelising compiler. The compiler inserts the necessary synchronisation instructions into the parallel codes. DoAll loops and DoAcross loops are called parallel loops. A parallel program can be viewed as a set of coherence segments, classified as parallel coherence segments and sequential coherence segments. A parallel coherence segment consists of a parallel loop of the DoAll or DoAcross type. A sequential coherence segment represents a serial region between parallel coherence segments that has to be executed by one processor.

3.3 Hardware mechanism

To support the DPI scheme, each private cache should contain the following components (as illustrated in Fig. 4):

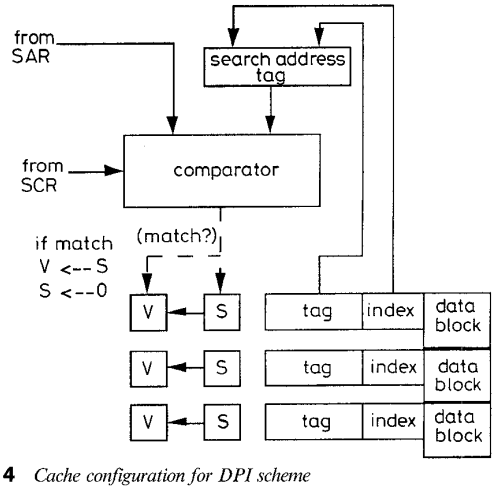

Fig.4 *Cache configuration for DPI scheme*

Search address register (SAR): The search address register is used to hold the address that is the operand of the invalidate instruction.

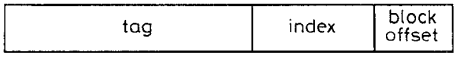

Fig.5 *Three fields of address to access cache*

Search address tag: The search address tag is used to be compared with the address operand of the invalidate instruction. Fig. 5 shows the three fields of an address to access the cache. The block-offset field is used to select the desired word from the block, the index field is used to select the set, and the tag field is used to determine whether the access is a hit or a miss. In the DPI scheme, the tag field combined with the index field forms the search address tag.

Selective compare register (SCR): The selective compare register is used to enable or disable the bit slices involved in the comparison between the search address register and the search address tag. With the SCR, the invalidation of a partial array can be precise.

Valid bit (V): This bit functions in the same way as the conventional valid bit in a cache line. The valid bit of a cache line is used to indicate whether the cache line has valid data.

Status bit (S): The status bit is used to indicate whether the cache line is the most up-to-date and should be preserved when the processor proceeds to the subsequent coherence segment. The status bit is set by the Read_Set_Status and Write_Set_Status instructions (defined subsequently). It is reset by the invalidate instruction (also defined later). A set status bit indicates that the cache line should be preserved when the processor proceeds to the subsequent coherence segment. A reset status bit indicates that the cache line should be invalidated when the processor proceeds to the subsequent coherence segment.

3.4 Cache management instructions

In the DPI scheme, the following cache management instructions need to be defined in the instruction set of the processor:

Read: The Read instruction is the same as a conventional memory read instruction

Read_Set_Status: The Read_Set_Status instruction is the same as the Read instruction except it sets the status bit S of the accessed cache line.

Write: The Write instruction is the same as a conventional memory write instruction.

Write_Set_Status: The Write_Set_Status instruction is the same as the Write instruction except it also sets the status bit S of the accessed cache line.

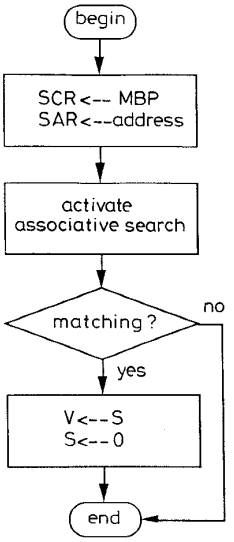

Fig.6 *Execution flow of invalidate instruction*
MBP mask bit pattern
SAR search address register
SCR selective compare register
V valid bit
S status bit

Invalidate: The purpose of the Invalidate instruction is to invalidate the cache lines which will become stale in the subsequent coherence segment. The Invalidate instruction is executed once by every processor at the

end of a coherence segment. Fig. 6 shows the execution flow of this instruction. The invalidate instruction shifts the values of the status bits into the corresponding valid bits of those cache lines searched by SAR and SCR. If the status bit of a matched cache line is reset, the valid bit V and the status bit S will both be reset, and the cache line is invalidated. If the status bit of a matched cache line is set, the valid bit will not be changed although the status bit will be reset.

3.5 Memory allocation and MBP generation

To explicitly invalidate the stale cache copies of the data in parallel and selectively, the DPI scheme allocates shared data structures or scalars in a well structured form, and generates a masking bit pattern (MBP) for each invalidate instruction. The purpose of the MBP is selective searching and invalidation of partial array. The DPI scheme uses the same memory allocation and MBP generation methods as the PEI scheme to allocate memory for array and scalar variables and generate an appropriate MBP [8]. The memory allocation and MBP generation methods are described in the following.

3.5.1 Memory allocation method: Assume that a parallel program consists of n shared one-dimensional arrays of sizes s_1, s_2, ..., s_n and shared scalar variables of a total size of s_0. Let S be a list of the sizes of the arrays and scalar variables, and let S' be an ordered list of S, $S' = \{s_0', s_1', s_2', ..., s_n'\}$, where $s_i' \leq s_j'$ if and only if $i \leq j\}$. A list B is derived from S' by replacing s_i' with b_i, where $b_i = \lceil \log_2 s_i' \rceil$, i.e. the number of bits required to encode s_i'. Huffman's optimal coding method is then applied to B, by considering b_i as the probability of the occurrence, to obtain a list $C = \{c_0, c_1, ..., c_n$, where c_i is the Huffman's code of $s_i'\}$. Let $|c_i|$ be the length of the code, and $w = \max(|c_i| + b_i)$, for all i, $0 \leq i \leq n$. Then, the base address of the array of size s_i' is constructed as $c_i 0^*$, where 0^* is a sequence of 0s whose length is $(w - |c_i|)$. Multidimensional arrays can also be encoded in the same way by treating them as multiples of one-dimensional arrays. The purpose of the application of Huffman's method to the memory allocation method is to uniquely identify an array using some of its most significant bits, i.e. c_i. An array of size s_i is allocated addresses from $c_i 0^*$ to $(c_i 0^* + s_i - 1)$. The addresses from $(c_i 0^* + s_i)$ up to the next base address $c_{i+1} 0^*$, are not used.

Consider an example of a shared memory of size 1 Kbyte and assume that the program accesses arrays X_1, X_2 and X_3, of sizes 25, 450 and 75 bytes, and a group of scalar variables Y of a total size of five bytes. According to the memory allocation method, $S = \{5, 25, 450, 75\}$, $S' = \{5, 25, 75, 450\}$, and $B = \{3, 5, 7, 9\}$. Now, if we apply Huffman's optimal coding method, we obtain $C = \{110, 111, 10, 0\}$. Then $w = \max(|c_i| + b_i) = 10$, and we append trailing 0s to make every code in C 10 bits long. As a result, $X_1 = 1110000000$, $X_2 = 0000000000$, $X_3 = 1000000000$, and $Y = 1100000000$ are the base addresses of the arrays and scalar variables. It is clear that arrays X_1, X_2, and X_3 are all addressed uniquely by the leading most significant bits.

3.5.2 MBP generation method: In the DPI scheme, each invalidate instruction requires an MBP to selectively invalidate the cache lines. Let MBP = $(p_1, p_2, ..., p_m)$. When $p_i = 0$, the corresponding bit slice of the address tag is disabled from matching. The MBP for scalar variables or array outside the parallel loop is (1, 1, ..., 1), that is, they are to be matched with full addresses. Let A be a set of addresses of n elements modified in a parallel loop $A = A_1, A_2, ..., A_n$, where A_j is represented as a binary number, that is, $A_j = a_{j1}\, a_{j2} \ldots a_{jm}$. MBPs of arrays inside the parallel loop are determined by the following equation:

$$p_i = (a_{1i} \wedge a_{2i} \wedge \ldots \wedge a_{ni}) \vee \overline{(a_{1i} \vee a_{2i} \vee \ldots \vee a_{ni})}$$
$$(\text{for } i = 1, \ldots, m)$$

This equation will find the address bit positions that have the same values for all A_j, $1 \leq j \leq n$.

3.6 Reference marking rules

In addition to the hardware support and cache management instructions, the DPI scheme must rely on a set of reference marking rules to maintain the cache coherence. Our reference marking rules are based on the data-dependence analysis of a parallel program. Data dependencies between read and write references are obtained in the compiler-time analysis.

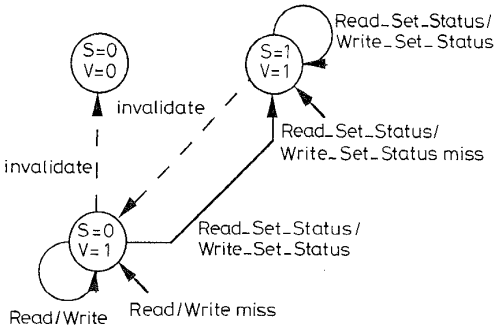

Fig. 7 *State transition diagram of cache line*
S: status bit
V: valid bit

Fig. 7 shows the state transition diagram of a cache line in the DPI scheme. If a cache copy becomes stale in the subsequent coherence segment, its status bit S should be reset. When the processor proceeds to the subsequent coherence segment it executes the invalidate instruction, and the cache line state is changed to ($S = 0$ and $V = 0$). If a cache copy will be an up-to-date copy in the subsequent coherence segment, the status bit S should be set. When the processor proceeds to the subsequent coherence segment, it executes the invalidate instruction, and the cache line state is changed to ($S = 0$ and $V = 1$). Therefore if the cache copy is stale and should be invalidated in the subsequent coherence segment, the associated read and write references should be marked Read and Write. If the cache copy is up-to-date in the subsequent coherence segment, the associated read and write references should be marked Read_Set_Status and Write_Set_Status.

The only situation that can cause cache copies of a variable to become stale is to have a write to the variable in another processor in a coherence segment. Therefore the reference marking method needs to identify a write to a variable, and insert an invalidate instruction before the subsequent coherence segment boundary. After the invalidate instructions are correctly inserted and the different types of references are identified and marked, it should be straightforward for the compiler backend to generate appropriate parallel code. In the following, we describe the reference marking rules to

properly insert the invalidate instructions and mark the different kinds of references.

3.6.1 Rule 1: If a DoAll loop or a serial portion of a program has a write reference to a variable, then the write reference should be marked Write_Set_Status. A read reference to the same variable after the write reference should be marked Read_Set_Status.

In a DoAll loop or a serial portion of a program, a write reference to a variable produces an up-to-date cache copy. Therefore the status bit S should be set when this cache copy is written, and such a write reference should be marked Write_Set_Status. A read reference to the same variable after the write reads an up-to-date cache copy produced by the write. Therefore the status bit S should be set when this cache copy is read, and such a read reference should be marked Read_Set_Status.

DoAcross loops incur data dependence across parallel loop iterations. The read and write references of the DoAcross loop that are either the source or the sink of a dependence arc across iterations are of concern. For a read (write) reference that is both a sink and a source, it is seen as a Source-Read (Source-Write) in the following rules. Before marking references, all dependencies of a DoAcross loop must be enforced by proper synchronisations between loop iterations.

3.6.2 Rule 2: If a DoAcross loop has a cross-iteration anti-dependence, then the Sink-Write should be marked Write_Set_Status and the Source-Read should be marked Read.

For an antidependence (write-after-read), the dependence is a Source-Read with a Sink-Write in a later iteration in the original DoAcross loop. The concerned array element of the cache copy referenced by the Source-Read will be written later by another processor in the same coherence segment. In such a case, the cache copy used by the Source-Read is made stale by this later write and should not be used beyond the current coherence segment. Therefore the status bit S should be reset when this cache copy is read, and it should be set when this cache copy is written. For this reason, such a read reference should be marked Read and such a write reference should be marked Write_Set_Status.

3.6.3 Rule 3: If a DoAcross loop incurs a cross-iteration flow-dependence, then the Sink-Read should be marked Read_Set_Status and the Source-Write should be marked Write_Set_Status. The required invalidate instruction should be inserted at the entrance point to the DoAcross loop to invalidate the Sink-Read operand.

For a flow-dependence (read-after-write), the dependence involves a Sink-Read and a Source-Write in a previous iteration in the original DoAcross loop. The Sink-Read must read a value produced by another processor in the same coherence segment, and this value can stay up-to-date beyond the current coherence segment boundary. However, because the cache copy used by the Sink-Read must be produced by another processor in the same coherence segment, the cache copy that existed at the beginning of the current coherence segment is stale. This necessitates that all existing cache copies of the same variable must be invalidated at the beginning of such a coherence segment. Therefore, invalidate instructions should be inserted at the entrance point to the current coherence segment to invalidate the existing cache copy of the same variable accessed by the Sink-Read.

With the extra invalidate instructions inserted at the entrance points to the current coherence segment, a Sink-Read to a variable is guaranteed to obtain the up-to-date value produced in the same coherence segment by another processor. The status bit S should be set when such a copy is read. For this reason, the Sink-Read should be marked Read_Set_Status. For the Source-Write, it produces the up-to-date copy in the cache. The status bit S should be set when such a copy is written. For this reason, the Source-Write should be marked Write_Set_Status.

3.6.4 Rule 4: If a DoAcross loop incurs a cross-iteration output-dependence, then the Sink-Write should be marked Write_Set_Status and the Source-Write should be marked Write.

For an output-dependence (write-after-write), the dependence involves a Source-Write and a Sink-Write in a later iteration in the original DoAcross loop. With proper synchronisation, the Sink-Write must overwrite the value produced by the Source-Write. In such a case, the cache copy produced by the Source-Write must not be used in the subsequent coherence segment, while the cache copy produced by the Sink-Write must be preserved beyond the current coherence segment boundary. Therefore, the status bit S should be reset when such a cache copy is produced by the Source-Write, while it should be set when such a cache copy is produced by the Sink-Write. For this reason, the Source-Write should be marked Write and the Sink-Write should be marked Write_Set_Status.

```
P_loop   DoAcross i = 0 to 6
                .....
S1              X(i) = ...          ; Write-Set-Status
                .....
S2              X(i + 1) = ...      ; Write
                .....
S3              .... = X(i + 1)     ; Read
                .....
         END DoAcross
         Invalidate X
Serial          .....
S4              X(0) = ...          ; Write-Set-Status
                .....
         Invalidate X(0)
```

(a)

X(0) S	X(0) V	X(1) S	X(1) V	X(2) S	X(2) V	Operations affecting status bits and valid bits	Row
0	1	0	1	0	1		1
0/1	1	0	1	0	1	Write_Set_Status(X(0))	2
1	1	0	1	0	1	Write(X(1))	3
1	1	0	1	0	1	Read(X(1))	4
1/0	1/1	0/0	1/0	0/0	1/0	Invalidate X	5
0	1	0	0	0	0		6
0/1	1	0	0	0	0	Write_Set_Status(X(0))	7
1/0	1/1	0	0	0	0	Invalidate X(0)	8

(b)

Fig. 8 *Example to illustrate DPI scheme*
a Program segment
b Cache status transitions of P_0

3.7 Example

We use Fig. 8 to show how the DPI scheme maintains cache coherence. According to the reference marking rules, statements S1, S2, S3, and S4 will be marked as

Write_Set_Status, Write, Read, and Write_Set_Status, respectively. Assume that eight processors P_0, P_1, ..., P_7 are involved in executing the program, and the processors are assigned such that ith iteration is executed by P_i in p_loop and P_0 executes the serial segment. According to the memory allocation method, the addresses of $X(0)$, $X(1)$, ..., $X(7)$ are 10110000, 10110001, ..., 10110111, respectively. Fig. 8*b* shows the status bits and valid bits of $X(0)$, $X(1)$ and $X(2)$ in P_0 cache. The initial states of $X(0)$, $X(1)$ and $X(2)$ in P_0 cache are shown in row 1 of Fig. 8*b*. The status bits and valid bits affected by the read/write operations are represented by two values separated by a slash (original value/new value), and the unaffected status bits and valid bits are represented by a single value.

Consider the program segment in Fig. 8*a*. In the DPI scheme, when processor P_0 executes iteration I_0 and statement S1 is reached, it modifies $X(0)$ and sets the status bit of $X(0)$ (row 2 of Fig. 8*b*) since statement S1 is a Write_Set_Status. When statement S2 is reached, processor P_0 modifies $X(1)$ and does not set its status bit (row 3 of Fig. 8*b*) since statement S2 is a Write. When statement S3 is reached, processor P_0 reads $X(1)$ and does not set its status bit (row 4 of Fig. 8*b*) since statement S3 is a Read. Because statement S2 is the Source-Write in the cross-iteration output dependence of the DoAcross loop, the cache copy of $X(1)$ will be stale in the subsequent coherence segment and must be invalidated when processor P_0 moves to the subsequent coherence segment. Moreover, the cache copies of $X(2)$, ..., $X(7)$ must also be invalidated when processor P_0 moves to the subsequent coherence segment since they are modified by other processors. Similar considerations are true for P_1, P_2, ..., P_7.

Next we explain how stale cache copies are invalidated. When P_0 moves to the subsequent coherence segment, it executes the invalidate instruction to invalidate those stale copies in parallel. By loading the selective compare register (SCR) with an MBP of 11111000, and loading the search address register (SAR) with the address of 10110000, P_0 activates the associative search to its local cache. Since only the bit slices whose corresponding bits in the SCR are set are examined in the search, the most significant five bits of 10110000 are compared. Therefore the cache copies whose most significant five address bits are 10110, i.e. all elements of array X are matched, and their valid bits are assigned the current status bit values and the status bits are reset (row 5 of Fig. 8*b*). In row 6 of Fig. 8*b*, one can see that the valid bits of $X(1)$ and $X(2)$ are reset since $X(1)$ and $X(2)$ will contain stale data in the subsequent coherence segment, while the valid bit of $X(0)$ is set since $X(0)$ is up-to-date in the subsequent coherence segment.

Finally, in the serial region, P_0 modifies $X(0)$ and sets it status bit while the other processors are idle (row 7 of Fig. 8*b*). Later, when all processors except P_0 move to the subsequent coherence segment, they need to execute the invalidate instruction to invalidate the copies of $X(0)$ in their caches, since these copies are no longer up-to-date. To invalidate these copies, all processors load the SCR with an MBP of 11111111, and the SAR with the address of $X(0)$, i.e. 10110000. The copies of $X(0)$ will be found in the associative search, and its valid bit will be assigned the value of its status bit and the status bit will be reset (row 8 of Fig. 8(*b*)).

4 Performance comparison

This Section compares the performance of the delayed precise invalidation (DPI) scheme and the parallel explicit invalidation (PEI) scheme.

4.1 Overhead of invalidate instruction

The PEI scheme explicitly invalidates the stale copies in the caches when a write instruction to sharable data is to be executed. In so doing, both the write and invalidate operations can be executed in parallel, but it causes extra cache misses. On the other hand, the DPI scheme executes the invalidate operations when the processors enter the subsequent coherence segment. Therefore the DPI scheme requires extra time and instruction to invalidate the stale copies in the cache, but it provides more cacheability than the PEI scheme. In the following, we analyse the miss ratio change and the overhead of invalidate instructions.

```
DoAll j = 1 to m
.....
... = X(j)
X(j) = ...
.....
END DoAll
```

Fig. 9 *Program segment to evaluate overhead of invalidate instruction*

Consider the program segment in Fig. 9, and assume that the cold start miss is not accounted for and n ($n \leq m$) processors are involved in executing the program. Assume also that the cache miss penalty is M_p and the invalidate instruction penalty is I_p. Since the invalidate instruction uses associative search to invalidate the cache copies, its penalty is much less than the cache miss penalty, i.e. $I_p << M_p$. For simplicity, we look only at processor P_1. Since in the DPI scheme the processors execute the invalidate instructions when they enter the subsequent coherence segment, there is no any extra cache miss. On the other hand, in the PEI scheme produce the inter-iteration misses, and the number of misses per processor is estimated to be $(m/n-1)$. Therefore the extra cache miss penalty per processor of the PEI scheme is $(m/n-1)^*M_p$, whereas the invalidate instruction penalty of the DPI scheme is I_p.

We use an example to show the invalidate instruction penalty and the cache miss penalty numerically. Assume that the value of I_p is 2 cycles, the number of loop iterations L is 64 and the value of M_p is 20 cycles. Fig. 10 shows the invalidate instruction penalty against the cache miss penalty. One can see that the invalidate instruction penalty is a small constant, whereas the cache miss penalty increases when the number of processors decreases. Furthermore, one can see that the total invalidate instruction penalty decreases and the total cache miss penalty increases when the number of processors decreases.

4.2 Simulation results

We also use trace-driven simulation to compare the performance of the DPI scheme and the PEI scheme quantitatively. Since we are only interested in the hit ratios of globally shared data, we assume that each processor has a data cache used exclusively for shared data and another local memory to accommodate instructions and local data. The coherence of data caches may be maintained by one of two software cache coherence schemes, the DPI and the PEI, and the local memories are free from cache coherence problem.

To eliminate conflict misses, every data cache is fully associative. The cache replacement policy is LRU (least recently used). To avoid false sharing, the cache line size is assumed to be one word.

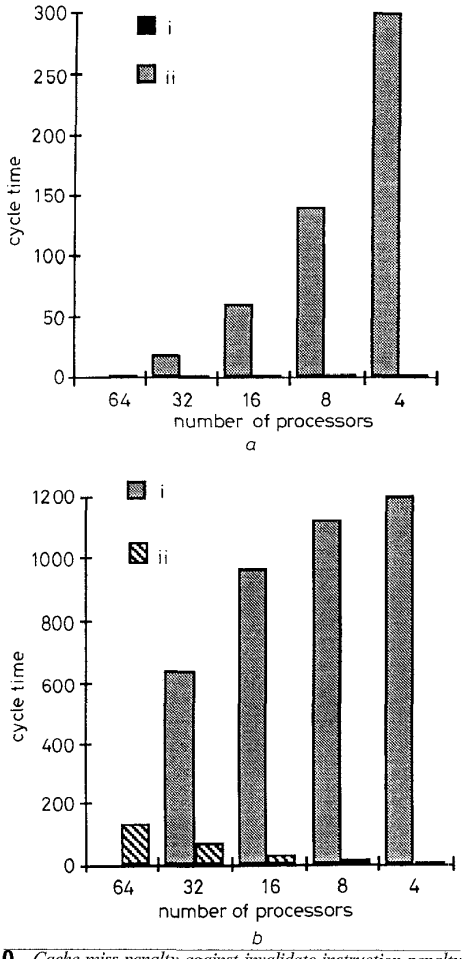

Fig. 10 *Cache miss penalty against invalidate instruction penalty*
a Penalties for each processor (i) invalidate instruction penalty, (ii) cache miss penalty
b Aggregate penalties (i) total cache miss penalty, (ii) total invalidate instruction penalty

In all simulation runs, the sequential code is always executed by processor P_0. When a parallel loop is encountered, each processor creates a parallel loop execution environment, obtains the value of the loop index, and executes the corresponding iterations independently. When one parallel loop finishes, another parallel loop will start or processor P_0 will resume its execution for the succeeding sequential code. To execute a parallel loop, a scheduling policy is needed to assign a processor to each iteration. Two different scheduling policies are used: prescheduling and self-scheduling. With the prescheduling policy, the ith iteration of a parallel loop is executed on processor $P_{(i \bmod n)}$, where n is the number of processors available. With the self-scheduling policy, whenever there is an iteration of a parallel loop for execution, a free processor is randomly selected to execute it.

Two numerical FORTRAN programs are used to generate the traces. The first program, Gauss, uses the Gaussian elimination technique to solve a linear system of equations. It does not contain any DoAcross loop.

The second program, Adi, uses the alternating direction implicit method to solve ordinary differential equations. It contains many dynamic occurrences of partial array modifications.

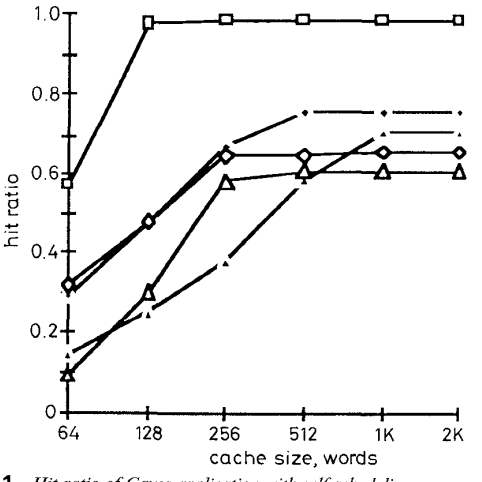

Fig. 11 *Hit ratio of Gauss application with self-scheduling*
—■— DPI scheme, P = 64
—□— PEI scheme, P = 64
—◆— DPI scheme, P = 32
—◇— PEI scheme, P = 32
—▲— DPI scheme, P = 16
—△— PEI scheme, P = 16

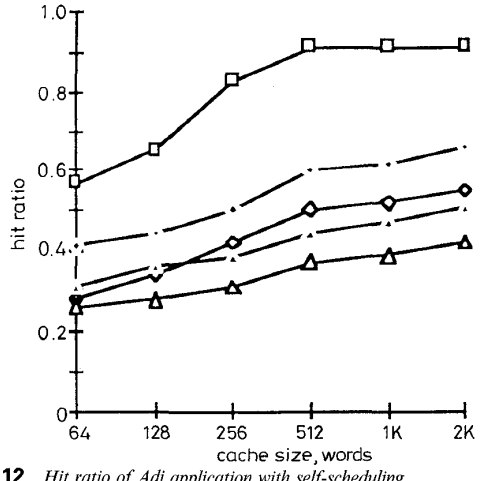

Fig. 12 *Hit ratio of Adi application with self-scheduling*
—■— DPI scheme, P = 32
—□— PEI scheme, P = 32
—◆— DPI scheme, P = 16
—◇— PEI scheme, P = 16
—▲— DPI scheme, P = 8
—△— PEI scheme, P = 8

Figs. 11–14 show the hit ratios of the two schemes for various cache sizes with two different scheduling policies, the prescheduling and self-scheduling, respectively. It is obvious that the DPI scheme outperforms the PEI scheme in the simulations as predicated. Given that one iteration is executed by one processor, the DPI scheme performs at least as well as the PEI scheme. This is because the PEI scheme does not cause unnecessary interiteration misses and intercoherence-segment misses. When a processor executes more than one iteration, the DPI scheme is definitely superior to the PEI scheme. This is because the PEI scheme explicitly invalidates the stale copies in the caches when a write operation to sharable data is to be

executed, whereas the DPI scheme executes the invalidate operations only when the processors are entering the subsequent coherence segments. From the simulation results, it is also found that the hit ratios of the prescheduling policy are higher than those of the self-scheduling policy. This is the virtue of the prescheduling policy that favours the reuse of cache contents across different coherence segments.

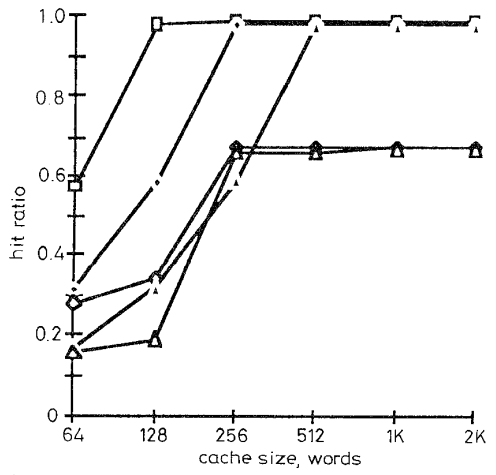

Fig. 13 *Hit ratio of Gauss application with prescheduling*
—■— DPI scheme, P = 64
—□— PEI scheme, P = 64
—◆— DPI scheme, P = 32
—◇— PEI scheme, P = 32
—▲— DPI scheme, P = 16
—△— PEI scheme, P = 16

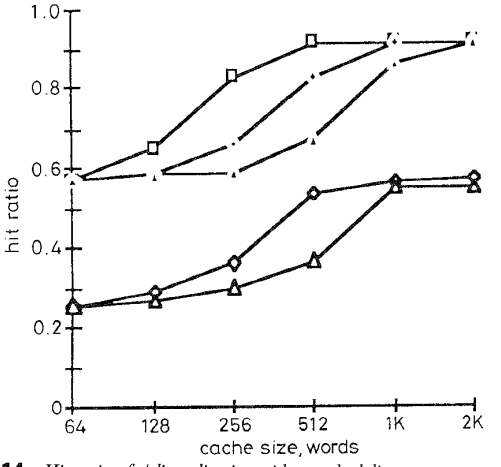

Fig. 14 *Hit ratio of Adi application with prescheduling*
—■— DPI scheme, P = 32
—□— PEI scheme, P = 32
—◆— DPI scheme, P = 16
—◇— PEI scheme, P = 16
—▲— DPI scheme, P = 8
—△— PEI scheme, P = 8

5 Conclusion

Software cache coherence schemes are very desirable in the design of scalable high-performance multiprocessors. This is because they are not constrained by network topologies, and they can eliminate runtime coherence traffic. However, there are some inefficiencies and deficiencies in the previous schemes. The version control scheme [6] unnecessarily invalidates all elements in an array even when only a part of the array is modified. Though the parallel explicit invalidation (PEI) scheme [8] solves this problem, it incurs incorrect result for a crossiteration output dependence in a DoAcross loop.

In this paper, we thus propose the delayed precise invalidation (DPI) scheme to remedy the problems of the previous schemes. Based on compiler-time markings of references and local explicit invalidations of stale data, the DPI scheme offers more cacheability by supporting delayed invalidation and selective invalidation of partial array. Simulation results show that the DPL scheme preserves more cacheability than the PEI scheme. In the simulation, we also found that it is more efficient to schedule an iteration on a particular processor than randomly, since reuse of data can improve cache performance. This fact suggests that processor/cache affinity should be considered in designing software cache coherence schemes. Good scheduling policies that consider both processor/cache affinity and cache coherence penalty will increase the effectiveness of caches and balance the loads of processors.

6 References

1 STENSTROM, P.: 'A survey of cache coherence schemes for multiprocessors', *Computer*, 1990, **23**, (6), pp. 12–24
2 ARCHIBALD, J., and BAER, J.-L.: 'Cache coherence protocols: evaluation using a multiprocessor simulation model', *ACM Trans. Comput. Syst.*, 1986, **4**, (4), pp. 273–298
3 CENSIER, L.M., and FEAUTRIER, P.: 'A new solution to coherence problems in multicache systems', *IEEE Trans. Comput.*, 1978, **C–27**, (12), pp. 1112–1118
4 VEIDENBAUM, A.V.: 'A compiler-assisted cache coherence solution for multiprocessors'. Proceedings of the 1986 international conference on *Parallel processing*, August 1986, pp. 1029–1036
5 CHEONG, H., and VEIDENBAUM, A.V.: 'A cache coherence scheme with fast selective invalidation'. Proceedings of the 15th annual international symposium on *Computer architecture*, June 1988, pp. 299–307
6 CHEONG, H., and VEIDENBAUM, A.V.: 'A version control approach to cache coherence'. Proceedings of the 1989 international conference on *Supercomputing*, June 1989, pp. 322–330
7 MIN, S.L., and BAER, J.-L.: 'Design and analysis of a scalable cache coherence scheme based on clocks and time stamps', *IEEE Trans. Parallel Distrib. Syst.*, 1992, **3**, (1), pp. 25–44
8 LOURI, A., and SUNG, H.: 'A compiler directed cache coherence scheme with fast and parallel explicit invalidation'. Proceedings of the 1992 international conference on *Parallel processing*, August 1992, Vol. 1, pp. 2–9
9 DARNELL, E., and KENNEDY, K.: 'Cache coherence using local knowledge'. Proceedings of *Supercomputing 1993*, 1993, pp. 720–729