Exact Distributed Invalidation

R.W. Ford[1], M.F.P. O'Boyle[2], and E.A. Stöhr[1]

[1] Department of Computer Science, The University, Manchester M13 9PL, U.K.
[2] Department of Computer Science, The University of Edinburgh, Mayfield Rd., Edinburgh EH9 3JZ, U.K.

Abstract. This paper develops and proves an exact distributed invalidation algorithm for programs with compile time decidable control-flow. We present an efficient constructive algorithm that globally combines locally gathered information to insert coherence calls in such a manner that eliminates all invalidation traffic without loss of locality and places the minimal number of coherence calls. Experimental results show that it outperforms existing compiler directed coherence techniques and hardware based memory consistency.

1 Introduction

The main goal of any distributed shared memory system is to support a shared memory programming model across distributed resources as efficiently as possible. More specifically, we would like to minimise the system overhead associated in maintaining memory coherence. This paper focuses on reducing the amount of coherence traffic associated with maintaining consistency. In particular we are interested in entirely eliminating invalidation traffic in a write-invalidation based protocol using compiler directed distributed invalidation. Given certain preconditions, we can provably eliminate all invalidation traffic thereby reducing latency. Furthermore, we can easily expand this approach to tackle the general case without adversely increasing memory traffic.

In invalidation protocols, attempts to write a new value may be delayed until all remote copies are invalidated. Performance can be degraded both by the delay on the writing node, and by the resulting network traffic. One approach to improving performance is to reduce the overhead of invalidation traffic within a write-invalidate based protocol by using distributed invalidation (DI). DI [10] transfers the responsibility of invalidation from the writing processor to the processors with the remote copies. The writing processor does not incur a write miss and can proceed without stalling as the invalidation of copies is done locally. The DI scheme also has the advantage of reducing network invalidation traffic by removing invalidation and acknowledgement messages.

Early work invalidated all cached data at each parallel region or epoch. More recent schemes have used tags or timestamps to maintain cached data across epochs [2, 3, 5]. Some schemes use a compiler controlled directory to help in run-time dependence analysis, whilst others remove the need for a directory altogether [2, 11, 6, 12, 15, 17]. In [4], a more sophisticated form of analysis based on reads after RDS (relaxed determining sequence) is described. In [7], vectorisation is used to minimise the overhead of redundant invalidation when using RDS analysis. Array data-flow analysis was used in [5] to detect and eliminate stale data references. However, the greater accuracy of this method was not exploited in determining the coherence action of the entire program.

In our previous work [13] we presented an algorithm based on coherence equations which allowed the exact modelling of coherence actions. However, this approach, in general, is undecidable and relies on a heuristic technique. This paper describes an exact algorithm for distributed invalidation for static control-flow programs and makes the following contributions:

- it presents an exact compiler algorithm that provably eliminates all invalidation traffic in static control-flow programs.
- it eliminates unnecessary invalidations which cause loss of temporal locality and provably places the minimal number of self-invalidations to maintain consistency.

2 Approach

In DI if each processor invalidates its stale read copies and marks as exclusive the data it will write then memory consistency is maintained without incurring any invalidation traffic by inserting local invalidate calls (LI) and local exclusive calls (LEx) (see [13]). The goal of any compiler directed technique is to determine exactly those memory elements that require coherence actions. Under estimation will partly rely on the systems coherence mechanism, over estimation leads to over invalidation and loss of temporal locality.

Given a certain sequence of memory accesses, we can exactly determine the coherence actions required, based on array section analysis and static scheduling information. The key feature of our algorithm is that relatively short sequences can be summarised locally before combining the results to determine the coherence actions throughout the program. In figure 1, starting at the lowest loop nest depth, the statements are examined to see if there exists a cross-processor anti-dependence. If there is a dependence, only those read and write actions occurring within the loop at that level are considered. Once coherence calls have been inserted, it is necessary to determine those upwardly exposed writes that may form the sink of anti-dependences causing coherence traffic. Similarly we need to determine those reads that are not covered by coherence calls and may be the source of anti-dependence causing later coherence traffic. The loop nest can now be considered as a single statement with a modified read and write set. This approach not only guarantees that all anti-dependences are exactly determined, it also means that coherence calls are always placed at the highest lexical level - removing the overhead of repeatedly making redundant coherence calls.

2.1 Example

Column 1 of figure 2 shows a parallelised program fragment based loosely on the Eispack routine Tred2, where `lo` and `hi` refer to the local upper and lower

1. Forall statements, determine enclosing static if conditions
2. For each loop nest L
   (a) For each loop with the nest deepest first *(section 5)*
      i. Determine coherence actions required
      ii. Insert coherence code
      iii. Determine exposed writes and live reads
      iv. Remove anti-dependences and consider each loop as a statement
3. For a basic block *(section 4)*
   (a) Determine coherence actions required
   (b) Insert coherence code

Fig. 1. The coherence algorithm

loop bounds. Column 2 shows a series of 4 boxes denoting the particular memory actions at various stages in the program to array z, in each of the four processors, while column 3 presents a summary of the state of memory at various points - critical to our algorithm. In column 2, reading or writing data that is already in exclusive state on the local processor does not affect it's memory state and is denoted by the colour grey. Similarly, reads to local data already in read-only state remain in read-only state denoted by a box containing a wave pattern. Reading remote data requires data to be in read state, once again denoted by a wave pattern. When a processor wishes to write data previously in read state, it must first mark the data to be written exclusive, i.e. Ro->Ex, with a call to LEx, this is denoted by the area of memory marked in black. Remote read copies must also be invalidated, i.e. Ro->I, with a call to LI, denoted by the cross [1].

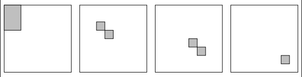
We first of all consider those deepest nested loops containing statements S2 and S3 and only consider those read and write actions within the immediate enclosing loop. As these are parallel loops there are no cross-processor anti-dependences and we may simply summarise the read and write actions as an array section for later use. Moving up syntactic levels, we must also consider statement S1 then S4 and cross-processor dependences within L1.

In the case of the write in statement S3, the cross-processor anti-dependence from S2 to S3 is from one iteration to the next. Hence, in S3 the read copies invalidated corresponding to those of the previous iteration. Once the coherence actions have been determined within loop L1, they must be summarised before the whole fragment must be considered. In particular we must consider those read copies that may still be sources of anti-dependences and those write accesses that may be sinks. If we examine column 3, we notice that there are read copies of the final column after executing L1. This uncovered read has a cross-processor dependence with S4, hence the coherence calls in the final entry of column 2.

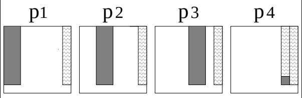
[1] Coherence calls are only shown for one case due to space restrictions.

Parallel Program	Reads/Writes/ Coherence Actions	Exposed Writes and Reads

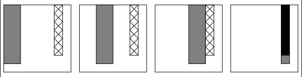
```
   if (n.ne.1) then
L1:Do i=2,n
    if (lo<=i-1<=hi)then
S1:   z(i-1,i-1)=1.0
    endif
    Do j=lo,min(hi,i-1)
     Do k=1,i-1
S2:   gg(j)=gg(j)+z(k,i)
                 *z(k,j)
     Enddo
    Enddo

<Barrier+Coherence code>

    Do j=lo,min(hi,i-1)
     Do k=1,i-1
S3:   z(k,j)=z(k,j)-
              gg(j)*d(k)
     Enddo
    Enddo
   Enddo


   call mp_barrier()
   if (lo<=n<=hi) then
    LEx(z(1,n))
   else
    LI(z(1,n))
   endif
   endif

   Do i=lo,hi
S4: z(1,i)=0.0
   Enddo
```



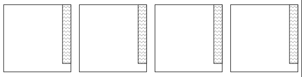
L1: Exposed write, assumed to be Ex.



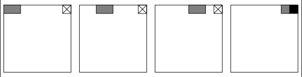
S2,i=n: Read Non-Local (Ex->Ro) (wave), Read Local (Ex) (grey), Read Local (Ro) (wave).



S3,i=n: just Write (Ex) (grey), Invalidate (Ro->I) (X), Mark Ex. and Write (Ro->Ex) (black).



L1: Uncovered reads in Ro.



S4: just Write (Ex) (grey), Invalidate (Ro->I) (X), Mark Ex. and Write (Ro->Ex) (black).

Fig. 2. Illustrative Example

3 Coherence Equations

This section summarises equations describing coherence based on [13]. Let an action i be a read or write operation occurring after action $i-1$ and before $i+1$. Let W_i and R_i be the set of all the pages (global data) written and read respectively in action i and let $\underline{W}_i$ and $\underline{R}_i$ be the corresponding local pages. Let $\underline{\mathcal{E}x}_i$ and $\underline{\mathcal{R}o}_i$ be the set of local pages in exclusive and read state respectively after action i. A superscript is used, if necessary, to distinguish between actions occurring on different processors, e.g. $\underline{W}_i^1$ and $\underline{W}_i^2$ refer to the local data written on processors 1 and 2 respectively.

We use an owner-computes rule and static scheduling. To eliminate multiple writer false sharing, arrays are padded and partitioned along page boundaries where necessary [12].

After a write, the local pages in exclusive/read state will be modified as follows:

$$\underline{\mathcal{E}x}_i = \underline{\mathcal{E}x}_{i-1} \cup \underline{W}_i \tag{1}$$
$$\underline{\mathcal{R}o}_i = \underline{\mathcal{R}o}_{i-1} - (\underline{\mathcal{R}o}_{i-1} \cap W_i). \tag{2}$$

Let p be the number of processors and let z be the processor id of the local processor. After a read action, the local pages in read and exclusive state will be modified as follows:

$$\underline{\mathcal{R}o}_i^z = \underline{\mathcal{R}o}_{i-1}^z \cup ((\bigcup_{k\in\{1,\ldots,p\}}^{k\neq z} \underline{R}_i^k) \cap \underline{\mathcal{E}x}_{i-1}^z) \cup (\underline{R}_i^z - \underline{\mathcal{E}x}_{i-1}^z) = \underline{\mathcal{R}o}_{i-1}^z \cup \hat{R}_i^z \tag{3}$$
$$\underline{\mathcal{E}x}_i^z = \underline{\mathcal{E}x}_{i-1}^z - (\underline{\mathcal{E}x}_{i-1}^z \cap (\bigcup_{k\in\{1,\ldots p\}}^{k\neq z} \underline{R}_i^k)). \tag{4}$$

Let $\underline{LEx}_i$ and $\underline{LI}_i$ be the local pages to be set to exclusive and invalid state respectively due to action i. The local pages to be made exclusive are those which will be written locally and are currently in read state:

$$\underline{LEx}_i = \underline{W}_i \cap \underline{\mathcal{R}o}_{i-1}. \tag{5}$$

The local pages to be invalidated are those formerly in read state if written to by remote processors:

$$\underline{LI}_i = (W_i - \underline{W}_i) \cap \underline{\mathcal{R}o}_{i-1}. \tag{6}$$

The above equations (5) and (6), if honoured by the compiler, will eliminate invalidation traffic and unnecessary misses [2].

[2] apart from those due to read/write false-sharing [14].

3.1 Compiler Implementation

Static control-flow is a well defined form of program containing statements, loops, procedures and if statements, where we restrict the form of conditionals to affine functions of compile-time known values and constant run-time parameters. We apply if-conversion throughout the program with the generated guards modelled as additional constraints. Those pages to be invalidated have been defined in terms of set algebra. To be amenable to compiler analysis, they have to be expressed as array sections. In our implementation we make use of Omega Library [9] for set manipulation.

4 Basic Blocks

Although the equations in section 3 precisely define those array elements to mark exclusive/invalid, they are in general, undecidable, as they are recursively defined in terms of previous states. We derive a non-recursive formulation of the coherence equations for basic blocks which can be used as the basis for a constructive algorithm and translation to Presburger formulae.

A basic block program or fragment of code B is an ordered set of statements and memory actions. Adjacent read actions within a statement in B can be merged into one read action. Therefore, a general basic block program or code fragment B can be represented as $B = \{R_1, W_1, ..., R_n, W_n\}$.

We make the initial assumption that all previous actions prior to entering the basic block have already been dealt with, i.e. $\underline{\mathcal{R}o}_0 = \emptyset$.

Given the restraint on $\underline{\mathcal{R}o}_o$ and our restriction to basic blocks we may recursively enumerate equation (3) to a point $s-1$ and rearrange the resulting equation using set manipulation to get:

$$\underline{\mathcal{R}o}_{s-1} = \bigcup_{t=1}^{s-1} (\underline{\hat{R}}_t - \bigcup_{u=t}^{s-1} W_t) \cup \underline{\hat{R}}_s. \tag{7}$$

The value of $\underline{\mathcal{R}o}$ is then substituted in equations (5) and (6).

Theorem 1. *Equations (7), (5) and (6) exactly define the coherence actions required before a write statement W_s [8].*

Due to space constraints the proofs of all theorems have been omitted and can be found in [8].

Based on the above formulation we have the following efficient algorithm to insert coherence code in basic blocks:

1. Find the first sink of cross processor anti-dependence S_k.
2. Find the union of local cross processor anti-dependence reads $\underline{\hat{D}}_t$ at each source of cross processor anti-dependence with a sink is S_k:
   $\underline{\mathcal{D}}_{k-1} = \cup_{t=1}^{k-1} \underline{\hat{D}}_t$.

3. Determine which coherence units should be made exclusive and which should be invalidated before S_k:
$\underline{LEx}_k^z = \underline{W}_k \cap (\bigcup_{k \in \{1,\ldots,p\}}^{k \neq z} \underline{\mathcal{D}}_{k-1}^z)$
$\underline{LIn}_k^z = (W_k - \underline{W}_k) \cap \underline{\mathcal{D}}_{k-1}^z$.
4. Insert coherence calls between statements S_{k-1} and S_k.
5. Reduce the local cross processor anti-dependence reads at each source of cross processor anti-dependence with a sink in S_k by what has been written:
$\underline{\hat{D}}_t = \underline{\hat{D}}_t - W_k$.
6. Delete all anti-dependences with sinks in S_k and repeat steps 1-6 till the end of the basic block is reached.

Theorem 2. *The Basic Block algorithm eliminates all invalidation coherence and guarantees memory consistency [8].*

Theorem 3. *The algorithm inserts the minimum number of invalidation calls [8].*

5 Loops

Consider a loop with $2n$ statements and N iterations. Let $\underline{\hat{R}}_t(i)$ be a read access within iteration i in statement t which causes a cross-processor anti-dependence. Similarly, $\underline{W}_t(i)$ denotes a local write at iteration i within statement t. Before we can express the equations denoting the read state, we first introduce a term Q to allow a more succinct presentation of the read state equation:

$$Q(j,i,t,s) = \underline{\hat{R}}_t(j) - \bigcup_{u=t}^{n} W_u(j) - \bigcup_{k=j+1}^{i-1} \bigcup_{t=1}^{n} W_t(k) - \bigcup_{u=1}^{s-1} W_u(i), \qquad (8)$$

if $j < i$.

This equation summarises all the reaching reads from statement S_t in iteration j to statement S_s at iteration i. It takes into consideration all the intervening writes which reduce the amount of data in read state. In those cases where we are considering statements within the same iteration, we can simplify Q as follows:

$$Q(i,i,t,s) = \underline{\hat{R}}_t(i) - \bigcup_{u=t}^{s-1} W_u(i),$$

if $i = j$, $t < s$. Finally, $Q(i,i,s,s) = \underline{\hat{R}}_s(i)$. We now have two cases:

No cross iteration dependences: This case is very similar to that of the basic block - except that we have an additional parameter - namely the iterator. This can be expressed as follows:

$$\underline{\mathcal{R}o}_s(i) = \bigcup_{t=1}^{s} Q(i,i,t,s).$$

General Case: For the general case where there may be cross-iteration data dependence, we have the following expression:

$$\underline{\mathcal{R}o}_s(i) = \bigcup_{j=1}^{i-1} \bigcup_{t=1}^{n} Q(j,i,t,s) \bigcup_{t=1}^{s} Q(i,i,t,s). \tag{9}$$

Theorem 4. *Equations (9), (5) and (6) exactly determine the necessary coherence actions in a loop before a write in a statement $W_s(i)$ [8].*

5.1 Nested Loops and Summarising

Once coherence actions have been determined for this loop level, we determine the array section of those writes that are upwardly exposed to possible anti-dependences at a higher loop level or earlier statement, i.e. $\bigcup_{i=1}^{N} \bigcup_{s=1}^{n} (\underline{W}_s(i) - \underline{LEx}_s(i) \cup \underline{LI}_s(i))$.

Similarly the read set associated will be those reads that may form the sources of anti-dependences is simply the read state after the last iteration, i.e.$\underline{\mathcal{R}o}_n(N)$. This leads to the overall algorithm described in figure 1.

6 Experiments

We prototyped the above algorithm in our compiler MARS [1] and applied it to two benchmarks, `Power`, an iterative eigenvalue solver and `Cholsky`, a routine from the Spec92 benchmarks. Our scheme was compared with hardware based sequential consistency (SC) which is guaranteed not to over-invalidate data and a compiler based scheme which uses lazy distributed invalidation based on relaxed determined sequences and time stamps [4]. The resulting programs were run on our DELTA simulator and execution times and memory statistics were gathered and presented below.

In the `power` program, both RDS and DI are capable of entirely eliminating the invalidation traffic required by sequential consistency shown by (Total remote cache line invalidates). This was achieved in the DI case by simply inserting local invalidates (Total local invalidates), however, additional write-backs were also needed by the RDS scheme. As we assume that write-backs are relatively cheap and the cost of invalidation traffic relatively small, both schemes give a modest improvement over sequential consistency. For larger systems, the improvement would be more dramatic. We further assume that special time-reads and the checking of the entire cache for data to flush, required by RDS, have no runtime cost. In practice, this could be a significant overhead and RDS would perform much worse than the DI scheme.

In the `cholsky` program, both sequential consistency and DI give the same performance as there are no runtime cross-processor dependences. In the RDS case, however, conservative compiler analysis has inserted excessive invalidation calls and write-backs leading to extremely poor execution time.

In both cases, DI has the best execution time.

Power (SC)	32 processors
Cycles (*10^3)	23,536
Total wait for invalidate	54,976
Total remote cache line invalidates	48,608
Write backs to main memory	0
Power (DI)	
Cycles (*10^3)	21,341
Total wait for	0
Total remote cache line invalidates	0
Total Local invalidates	48,608
Write backs to main memory	0
Power (RDS)	
Cycles (*10^3) 21,346	283,666
Cache Flush checks	18
Time-reads (*10^3)	1,606
Total remote cache line invalidates	0
Total local invalidates	50,624
Write backs	3,136

Cholsky (SC)	32 processors
Cycles (*10^3)	975
Total wait for invalidates (*10^3)	0
Total remote cache line invalidates	0
Write backs to main memory	0
Cholsky (DI)	
Cycles (*10^3) 975	1,874
Total wait for invalidates	0
Total remote cache line invalidates	0
Total Local invalidates	0
Write backs to main memory	0
Cholsky (RDS)	
Cycles (*10^3)	187,934
Cache Flush checks	64
Time-reads (*10^3)	381
Total remote cache line invalidates	0
Total local invalidates	34,422
Write backs to main memory	70,912

7 Conclusion

This paper has presented, for the first time, an exact compiler based distributed invalidation algorithm. Assuming a static control-flow program we provably insert the minimal number of coherence calls to guarantee consistency, eliminating all coherence traffic and reducing network contention without destroying temporal re-use due to over-invalidation. Furthermore, we can outperform existing hardware and compiler-based techniques. Future work will combine this exact technique with our previous work on general control-flow, based on a hybrid coherence based scheme.

References

1. Bodin F., O'Boyle M.F.P., **A Compiler Strategy for SVM**, Proc. of Workshop on Lang., Compilers and Runtime Sys. for Scalable Comp., May 1995.
2. Cheong H., Veidenbaum A.V., **Compiler Directed Cache Management in Multiprocessors**, IEEE Computer, 23(6):39-48, June 1990.
3. Cheong H., **Life-Span Strategy - A Compiler-Based Approach to Cache Coherence**, Proc. of Int. Conf. on Supercomp., July 1992.
4. Choi L., Yew P-C., **A Compiler-Directed Cache Coherence Scheme with Improves Intertask Locality**, Proc. of Supercomp.'94, Nov. 1994.
5. Choi L., Yew P-C., **Compiler analysis for cache coherence: Interprocedural array data-flow analysis and its impacts on cache performance**, Tech. Report, University of Illinois, Sep. 1996.
6. Darnell E., Kennedy K., **Cache Coherence Using Local Knowledge**, Proc. of Supercomp.'93, Nov. 1993.
7. Darnell E., Mellor-Crumney J.M., Kennedy K., **Automatic Software Cache Coherence through Vectorisation**, Proc. of Int. Conf. on SuperComp., July 1992.
8. Ford R.W., O'Boyle M.F.P., Stöhr E.A., **Exact Distributed Invalidation**, Tech. Report, Dept. of Computer Science, Univ. of Manchester, 2000.
9. Kelly W., Maslov V., Pugh W., Rosser E., Shpeisman T, and Wonnacott D., **The Omega Library Interface Guide**, Tech. Report, Dept. of Computer Science, Univ. of Maryland, 1996.
10. Lebeck A.R., Wood D.A., **Dynamic Self Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors**, Proc. of Inter. Symp. on Comp. Arch., 1995.
11. Louri A., Sung H., **A Compiler Directed Cache Coherence Scheme with Fast and Parallel Explicit Invalidation**, Proc. of Inter. Conf. on Parallel Processing, August 1992.
12. Mounes-Toussi F., Lilja D.J., Li Z., **An Evaluation of a Compiler Optimization for Improving the Performance of a Coherence Directory**, Proc. of Inter. Conf. on Super., July 1994.
13. O'Boyle M.F.P, Nisbet A.P., Ford R.W., **A Compiler Algorithm to Reduce Invalidation Latency in Virtual Shared Memory Systems**, PACT'96, October 1996.
14. O'Boyle M.F.P., Ford R.W., Nisbet A.P., **Compiler Reduction of Invalidation Traffic in Virtual Shared Memory Systems**, EuroPar'96.

15. Skeppstedt J., Stenstrom P., **Simple compiler algorithms to reduce ownership overhead in cache coherence protocols**, ASPLOS, 1999.
16. Skeppstedt J., Stenstrom P., **A Compiler Algorithm that Reduces Latency in Ownership-Based Cache Coherence**, Proc. of Parallel Arch. and Compiler Tech. 95, June 1995.
17. Skeppstedt J., Dahlgren F. and Stenstrom P., **Evaluation of Compiler-Controlled Updating to Reduce Coherence-Miss Penalties in Shared-Memory Multiprocessors**, JPDC, vol 56, 1999.