Compiler Reduction of Invalidation Traffic in Virtual Shared Memory Systems

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

[1] Department of Computation, UMIST, Manchester, UK
[2] Centre for Novel Computing, Dept. of Computer Science, University of Manchester, UK

Abstract. This paper presents new compiler analysis for the elimination of invalidation traffic in virtual shared memory, using a hybrid distributed invalidation coherence scheme. The invalidation and acknowledgement messages are removed; this reduces both network invalidation traffic and the latency of a write fault. It aggressively exploits the SPMD execution model and uses array section analysis to accurately determine only those instances when invalidation is necessary, thus avoiding the additional read misses of previous schemes. Equations determining precisely what data should be invalidated are presented and translated into a form amenable to compiler analysis. Preliminary experimental results on a 30 node prototype architecture demonstrate the performance attainable using this scheme.

1 Introduction

Virtual shared memory (VSM) systems provide a shared address space on distributed memory architectures. A shared memory programming model is attractive because it is simple to program, thus speeding the implementation and porting of parallel programs, and enabling the parallelisation of complex adaptive programs which may be difficult to implement in message-passing. However, for program segments with static communication patterns, sequentially consistent VSM fares significantly worse as, although it provides coherence mechanisms to support general access patterns, it seldom provides facilities to allow exploitation of compile-time knowledge. Scalable VSM systems use a directory structure (which typically records the location and the read/write access permissions of copies of data) and invalidation/update based protocols to maintain consistency. In the sequential consistency model with an invalidation protocol, an attempt to write a new value is 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.

This paper presents new compiler analysis to exploit a hybrid Distributed Invalidation (DI) scheme in the context of invalidation based sequential consistency. The DI technique [5, 7] removes the responsibility of invalidation from the writing processor and distributes it to the processors with the remote copies. The writing processor can now proceed without stalling and the invalidation of copies is done locally. The invalidation and acknowledgement messages are removed, reducing both network invalidation traffic and the latency of a write fault. In the case of compile-time unknowns, the directory structure guarantees correctness. Thus, the compiler may be aggressive in its analysis as it is does not have to make the conservative assumptions of compiler directed schemes where the compiler is entirely responsible for coherence.

We have implemented [5] a call to our VSM system *LocalExclusive(addr)* which sets local access for the location *addr* of a variable to exclusive and, by default, deletes the directory pointers that refer to copies of data. This allows a write to proceed without stalling (as in weak coherency) and removes the associated invalidation traffic. The corresponding call *LocalInvalidate(addr)* sets a read-only copy of the *addr* to invalid. Notice that a write can now occur before the remote copies have been invalidated, however compiler placement of barriers ensures that no read to incorrect data can take place. An important feature of these calls is that after they have completed, the system is left in a coherent state. Thus in the absence of further compiler optimisation, the directory will continue to maintain sequential consistency.

Previous Work All existing analysis for compiler directed coherence assumes a fork/join model. Parallel regions are separated into *epochs* by a synchronisation, even though there may be no cross-processor dependence between regions. Scheduling within an epoch is considered to be dynamic so it is impossible to determine the location of accesses between epochs. This has the effect of making worst case assumptions about the location of previous reads and writes, generating unnecessary invalidation or misses.

Compiler directed coherence places the entire burden of maintaining cache coherence on the compiler. Some schemes use a compiler controlled directory to help in run-time dependence analysis, whilst others remove the need for a directory altogether [2, 4]. Early work invalidated all cached data at each epoch. More recent schemes have used tags or timestamps [3] to maintain cached data across epochs. This method relies on the compiler analysis for correctness and is limited in the amount of inter-epoch locality that can be exploited by the size of the tag and epoch counters, as well as by the conservative nature of the compiler analysis. Other approaches include a directory based method, Dynamic Self Invalidation [7] and hybrid techniques, [6, 9] each of which is developed for CC-NUMAs. In [9] they note the critical role of (but do not develop) compiler analysis.

We use an SPMD execution model, exploiting data parallelism with owner-computes scheduling. Owner-computes guarantees that the write to any page will be performed by the same processor, and hence two or more write actions cannot make data stale on the processor that issued the first write. Using data layout and alignment we have implemented a scheme that determines whether two accesses can cause a cross-processor dependence. We have used this to optimise barrier placement, drastically reducing synchronisation overhead. This same analysis can be used to detect if an anti-dependence is cross-processor, thereby requiring coherence action. By using array section analysis rather than

<table>
<tr><th>Original Program</th><th>Local Program</th></tr>
<tr><td><pre>
Do s = 1,n
1: b(s) = s
Enddo
Do i = 2,n-1
  Do j = 1,i
2:    a(i) = a(i) + b(j) + b(i)
  Enddo
Enddo
Do t = 4,n-5
3: b(t) = t
Enddo
</pre></td><td><pre>
Do s = max(1,lo),min(hi,n)
1: b(s) = s
Enddo
barrier()
Do i = max(2,lo),min(hi,n-1)
  Do j = 1,i
2:    a(i) = a(i) + b(j) + b(i)
  Enddo
Enddo
barrier()
Do t = max(4,lo),min(hi,n-5)
3: b(t) = t
Enddo
</pre></td></tr>
</table>

Figure 1. Example to show coherence state

variable names we can accurately determine the data to invalidate, whereas previous schemes require the invalidation of entire arrays, or an additional bit mask per write instruction [8].

2 Analysis

A processor must invalidate all its copies of pages that will be written by others and ensure exclusive access to pages it will write. This section is concerned with precisely determining *which* pages should be modified, when using owner-computes scheduling. It develops a set of equations that determine the local actions each processor must take in order to maintain coherence, without incurring invalidation traffic.

2.1 Coherence Equations

Before deriving equations to mark the data invalid and exclusive, it is necessary to define a number of primitives.

- Let an action i be a read or write operation occurring after action $i-1$ and before $i+1$. This ordering must be ensured by program synchronisations or local sequential ordering.
- Let W_i, R_i be the set of a pages (global array data) written/read in action i.
- Let $\underline{W}_i, \underline{R}_i$ be the set of local pages written/read in action i.
- Let $\underline{\mathcal{E}x}_i, \underline{\mathcal{R}o}_i$ be the set of local pages in exclusive/read state after action i.

The state of a page after a particular action is dependent on its previous state, and on the action. After a write, the local pages in exclusive mode will be increased by any pages written locally that were not previously marked as exclusive:

$$\underline{\mathcal{E}x}_i = \underline{\mathcal{E}x}_{i-1} \cup \underline{W}_i. \tag{1}$$

The set of local pages in read state will be decreased by writes occurring across the processors if they refer to local pages previously in read state:

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

Let p be the number of processors in the target machine, and z the processor id of the local processor where $z \in \{1, ..., p\}$. After a read action, the set of local pages in read state will be increased by other processors reading processor z's local pages previously in exclusive state, and by the remote pages read by the local processor. As it is necessary to distinguish between the local pages of distinct processors, the processor id will be used as a super-script.

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

Note that, from this equation, reads by a processor to its own local data do not change its state from exclusive to read state, only reads by other processors will do this. The local pages in exclusive state are decreased by reads from other processors to local pages previously in exclusive state.

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

Once again the pages in exclusive state are unaffected by local reads to local data. We now introduce the equations which are used to define coherence state transition. Let $\underline{LE}_i, \underline{LI}_i$ be the local pages to be set to exclusive/invalid state due to action i. The local pages to be made exclusive are those which will be written locally but are not already exclusive and those local pages which are in a read state should be invalidated if written to by remote processors.

$$\underline{LE}_i = \underline{W}_i - (\underline{\mathcal{E}x}_{i-1} \cap \underline{W}_i) \tag{5}$$

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

2.2 Sets $\mapsto$ Constraints

So far pages that need to be invalidated, or made exclusive, have been defined in terms of set algebra. To be amenable to compiler analysis, they have to be expressed as array sections. We restrict our attention to programs where the set based description may be interpreted in terms of array sections using affine constraints and access functions.

To denote the elements or array section accessed within a loop nest of an array v we use the (iteration space, access) pair:

$$[AJ \leq b], [v(f(J))] \tag{7}$$

where $AJ \leq b$ denote the constraints on the iterators J from the bounds in the enclosing loop nest and $v(f(J))$ represents the access to variable v which is an affine function f of the iterators. For example, the elements of `a` accessed in the second loop of Figure 1 are:

$$[(2 \leq i \leq n-1) \wedge (1 \leq j \leq i)], [a(i)] \tag{8}$$

Let X be the *set* of elements accessed:

$$X = [A_x J_x \leq b_x], [x(f_x(J_x)] \tag{9}$$

In order to derive the relation between the set expressions of Section 2.1 and the compiler's constraint representation, we need to define the effect of set operators. The intersection of two sets is equivalent to the conjunction of the two systems of constraints from the iteration space and the requirement that the access functions are equal.

$$X \cap Y \ : \ [(A_x J_x \leq b_x) \wedge (A_y J_y \leq b_y) \wedge (f_x(J_x) = f_y(J_y))], [x(f_x(J_x))] \tag{10}$$

The addition of two non-intersecting sets corresponds simply to two systems of constraints.

$$X + Y \ : \ ([A_x J_x \leq b_x], [x(f_x(J_x))]) + ([A_y J_y \leq b_y], [y(f_y(J_y))]) \tag{11}$$

The difference of two sets, given one is the subset of the other is similar to that of intersection except that one set of constraints is negated.

$$X - Y : [(A_x J_x \leq b_x) \wedge (\neg A_y J_y \leq b_y) \wedge (x(f_x(J_x)) = y(f_y(J_y))], [x(f_x(J_x))] \tag{12}$$

The main difficulty with the constraint equivalences of set algebra is that negation is not directly defined for polyhedra as it can lead to non-convexity. This is overcome by distributing the negation through each set of inequalities using DeMorgan's Law to give a new set of inequalities connected by $\vee$ (OR) rather than $\wedge$ (AND). This is returned to a set of inequalities connected by $\wedge$ by using the following identity.

Let x, y be inequalities then

$$(x \vee y) \Leftrightarrow (\neg x \wedge y) + (x \wedge \neg y) + (x \wedge y) \tag{13}$$

Fortunately negation is defined for a single integer inequality:

$$\neg(a \leq b) \Leftrightarrow (a > b) \Leftrightarrow (b \leq a - 1). \tag{14}$$

Thus negation and hence union can be expressed. Unfortunately expanding $\vee$ may lead to an explosion of terms. For two constraints there are only three terms, but this grows exponentially as $2^C - 1$ where C is the number of constraints. However, many of the new terms can be eliminated as they are contradictory (or more strictly their solution space is empty) which is readily determined using Fourier-Motzkin elimination. The final terms to consider are unions. The simple union of two sets can be expressed in terms of previously defined operators

$$X \cup Y = X + Y - (X \cap Y). \tag{15}$$

The indexed union, as used in equations (10) and (11) cause more difficulties. We first trivially split it:

$$\bigcup_{k \in \{1,\ldots,p\}}^{k \neq z} X^k = (\bigcup_{k \in \{1,\ldots,z-1\}} X^k) \cup (\bigcup_{k \in \{z+1,\ldots,p\}} X^k) \tag{16}$$

and then store each union as a parameterised set of inequalities

$$\bigcup_k X^k \Leftrightarrow [A^k J^k \leq b^k \; \forall k], [x(f_x(J_x))]. \tag{17}$$

2.3 Example

When the iterations and data are mapped to several processors, each processor will have its own local iteration and data spaces. For example, in Figure 1 the second column shows the local program generated by the MARS compiler [1] after partitioning; lo, hi are the introduced lower and upper bounds of the local data space. In MARS each dependence is marked as being either an internal or a cross-processor dependence. The dependences `b(s)` δ^f `b(i)` and `b(i)` δ^a `b(t)` are within a processor while `b(s)` δ^f `b(j)` and `b(j)` δ^a `b(t)` are cross-processor dependences requiring synchronisation. In this section we will derive the data regions of array `b` to be invalidated for the program shown in Figure 1. Before calculating the regions to be marked as exclusive and invalid, a number of terms $\underline{\mathcal{R}o}_1, \underline{\mathcal{E}x}_1, \underline{\mathcal{R}o}_2, \underline{\mathcal{E}x}_2$ must be derived. For the sake of illustration, assume that there are local read copies of `b` (`b[2..n-1]`), before the execution of the first loop, so $\underline{\mathcal{R}o}_0 : [2 \leq i_0 \leq n-1], [b(i_0)]$. The write access in the first loop is $W_1 : [1 \leq s \leq n], [b(s)]$ and we wish to calculate the read state $\underline{\mathcal{R}o}_1$ after this write, based on equation (2). First, we must determine $\underline{\mathcal{R}o}_0 \cap W_1$:

$$[(2 \leq i_0 \leq n-1) \wedge (1 \leq s \leq n) \wedge (i_0 = s)], [b(s)] \tag{18}$$

i_0 is redundant as it does not occur in the access function. Removing i_0 by Fourier-Motzkin elimination gives the constraints on s:
$[2 \leq s \leq n-1], [b(s)]$. We now calculate $\underline{\mathcal{R}o}_0 - (\underline{\mathcal{R}o}_0 \cap W_1)$:
$[(2 \leq i_0 \leq n-1) \wedge \neg(2 \leq s \leq n-1) \wedge (i_0 = s)], [b(s)]$ giving

$$[(2 \leq i_0 \leq n-1) \wedge ((2 > s) \vee (s > n-1)) \wedge (i_0 = s)], [b(s)] \tag{19}$$

On expanding the $\vee$ each summand is found to be inconsistent; that is there does not exist a solution and hence $\underline{\mathcal{R}o}_1 = \emptyset$. We will now derive the remaining terms. As all of array `b` is written in the first loop, all local data is in exclusive state and there are no local copies of remote data, $\underline{\mathcal{E}x}_1 = \underline{W}_1 : [(\max(1, lo) \leq s \leq \min(n, hi))], [b(s)]$.

Determining $\underline{\mathcal{R}o}_2$ based on equation (3) is more complex and will be dealt with piecewise. One of the simpler terms is $(\underline{R}_2 - (\underline{\mathcal{E}x}_1 \cap \underline{R}_2))$, which reduces to $[1 \leq j \leq lo-1], [b(j)]$. The indexed union term in equation (3) can be expressed as:

$$\left(\bigcup_{k \in \{1,\ldots,p\}}^{k \neq z} \underline{R}_2^k \right) \cap \underline{\mathcal{E}x}_1 : [(1 \leq j \leq \min(hi^k, n-1)) \wedge \\ ((\max(1, lo^z) \leq s \leq \min(n, hi^z)) \wedge (s = j),\ \forall k, k \neq z], [b(j)] \tag{20}$$

We have introduced superscripts to distinguish the local constraints. On eliminating j and splitting the range of k we have:

$$[\max(1, lo^z) \leq s \leq \min(hi^z, hi^k, n-1)\ \forall k \in \{1, ..., z-1\}], [b(s)] \tag{21}$$

and

$$[\max(1, lo^z) \leq s \leq \min(hi^z, hi^k, n-1)\ \forall k \in \{z+1, ..., p\}], [b(s)] \tag{22}$$

. Equation (21) is inconsistent for all values of k as $lo^z > hi^{z-1}$ while equation (22) reduces to

$$[\max(1, lo^z) \leq s \leq hi^z], [b(s)] \tag{23}$$

Combining the subexpressions gives

$$\underline{\mathcal{R}o}_2 : [(1 \leq j \leq hi)], [b(j)] \tag{24}$$

The final term can easily be found to be $\underline{\mathcal{E}x}_2 = \emptyset$. Given $W_3 : [4 \leq t \leq n{-}5], [b(t)]$ and $\underline{W}_3 : [\max(4, lo) \leq t \leq \min(hi, n-5)], [b(t)]$ it is possible to define the data to be marked invalid and exclusive before the write W_3.

$$\underline{LE}_i : [\max(4, lo) \leq t \leq \min(n-5, hi)], [b(t)] \tag{25}$$

$$\underline{LI}_i : ([4 \leq t \leq \min(\max(4, lo-1), hi)], [b(t)])\ + \\ ([\max(1, \min(n-5, hi+1)) \leq t \leq \min(n-5, hi)], [b(t)]) \tag{26}$$

3 Experiments

In this section we apply the above techniques to two applications, an in house version of the NAS Parallel benchmark CG and the core communication structure of Shallow on a 30 node EDS prototype [12] supporting a prototype VSM implementation. For further details of the implementation see [11].

Figures 2 and 3 present counts of read/write misses and invalidation traffic with and without distributed invalidation for the two applications. Figures 4 (a)

Nodes	2	4	8	16
Read Misses	100	600	2800	12000
Write Misses	100	200	400	800
Invalidations	100	600	2800	12000

Nodes	2	4	8	16
Read Misses	100	600	2800	12000
Write Misses	0	0	0	0
Invalidations	0	0	0	0

Figure 2. Counts of misses for in-house CG (a) standard (b) with distributed invalidation.

Nodes	2	4	8	16
Read Misses	100	300	700	1500
Write Misses	100	300	700	1500
Invalidations	100	300	700	1500

Nodes	2	4	8	16
Read Misses	100	300	700	1500
Write Misses	0	0	0	0
Invalidations	0	0	0	0

Figure 3. Counts of misses for core communication of Shallow (a) standard, (b) with distributed invalidation.

and (b) show the temporal performance where the *Naive Ideal* line assumes linear speedup from the standard sequential version. The *Optimised* line presents performance improvement from the use of distributed invalidation. Note that counts and timings are taken from the second to last iteration, thereby eliminating the effect of initial (cold start) faults from the results. Our approach gives a dramatic reduction in invalidation messages without the potential increase in unnecessary misses of previous compiler-controlled schemes.

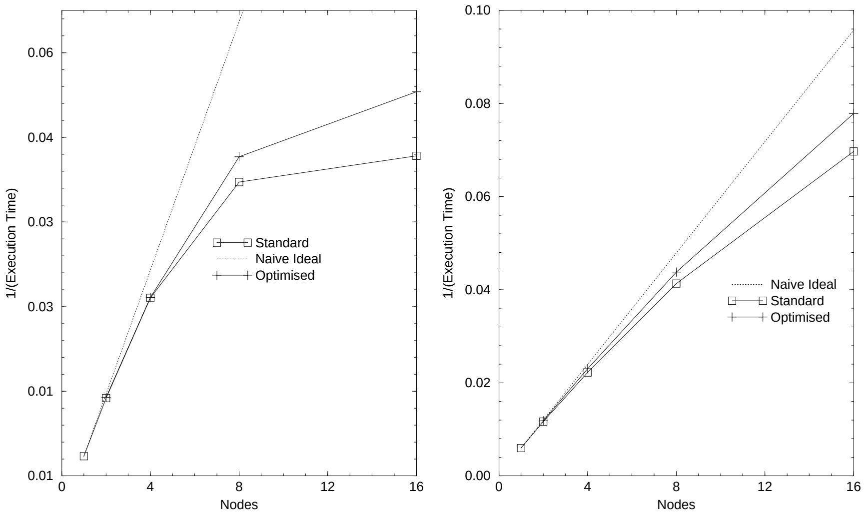

Figure 4. Temporal Performance of (a) in-house CG and (b) core communication of Shallow.

4 Conclusions and Further Work

This paper has described compiler analysis for a hybrid distributed invalidation scheme making explicit use of compile-time knowledge of scheduling and data layout. Future work will focus on completing implementation of the algorithm in our compiler and investigating the trade-off between invalidation traffic and unnecessary misses. We intend to investigate compiler analysis for other VSM optimisations such as latency-hiding post-storing [10] and to develop a unifying approach to compiling for VSM.

Acknowledgements This work was partly funded by the ESPRIT SODA project (9124). We would like to thank Thomas Fahringer and Mark Bull for their helpful comments on drafts of this paper.

References

1. Bodin F., O'Boyle M.F.P. **A Compiler Strategy for SVM**, Third Workshop on Languages, Compilers and Runtime Systems for Scalable Computing, New York, 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**, Proceedings of ICS, July 1992.
4. Darnell E., Kennedy K., **Cache Coherence Using Local Knowledge**, Proceedings of Supercomputing 1993, pages 720-729, Nov 1993.
5. Ford R.W., Nisbet A.N., Bull J.M. **User-level VSM Optimisation and its Application** Proceedings of PARA95, Lyngby, Denmark, LNCS, 1995.
6. Hill M., Larus J., Reinhardt S.,**Cooperative Shared Memory**, ACM Trans. on Computer Systems 11(4): 300-318, Nov 1993.
7. Lebeck A.R., Wood D.A. **Dynamic Self Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors**, Proceedings of ISCA '95, 1995.
8. Louri A. and Sung H., **A Compiler Directed Cache Coherence Scheme with Fast and Parallel Explicit Invalidation**, Proceedings of ICPP, August 1992.
9. Mounes-Toussi F., Lilja D.J., Li Z. **An Evaluation of a Compiler Optimization for Improving the Performance of a Coherence Directory**, Proceedings of ICS '94, July 1994.
10. Nisbet A.N., Ford R.W., **Spinning on Coherency: A New VSM Optimisation for Write-Invalidate**, Proceedings of HPCN Europe, April 1996.
11. O'Boyle M.F.P.,Ford R.W., Nisbet A.N., **Compiler Reduction of Invalidation Traffic in Virtual Shared Memory Systems**, CNC Tech. Rep. June 1996.
12. Skelton, C.J. et al., **EDS a Parallel Computer System for Advanced Information Processing**, Parallel Architectures and Languages Europe, PARLE 92, July 1992.