A Compiler Algorithm to Reduce Invalidation Latency in Virtual Shared Memory Systems

M.F.P. O'Boyle
Department of Computation
UMIST, UK

A.P. Nisbet, R.W. Ford
Department of Computer Science
University of Manchester, UK

Abstract

This paper presents a new compiler algorithm to eliminate invalidation traffic in virtual shared memory using a hybrid distributed invalidation scheme. It aggressively exploits static scheduling and data layout 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 followed by the derivation of approximations aminable to compiler manipulation. Compiler-directed invalidation in the presence of arbitrary control-flow is described before the definition of a compiler algorithm is presented. Preliminary experimental results on three programs show that this analysis can drastically reduce the amount of invalidation traffic and write misses.

1. Introduction

Scalable Virtual Shared Memory (VSM) systems often 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 latency. In an update model, all remote copies are sent the new data values. Although this helps reduce read misses, it increase the amount of write traffic, often increases read-miss latency and further delays the writing node[19].

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 (also called self invalidation [13]) removes the responsibility of invalidation from the writing processor and distributes it to the processors with the remote copies. The writing processor does not incur a write miss and can proceed without stalling and 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. This paper presents a new compiler algorithm to exploit a hybrid DI scheme in the context of invalidation based sequential consistency. This eliminates invalidation traffic without incurring the read misses of previous schemes. The compiler is 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. In the case of compile-time unknowns, the directory structure guarantees correctness.

Existing analyses [4],[2], [6] for compiler directed coherence assume 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 read and writes, generating unnecessary invalidation or misses.

Our compiler [1] exploits data parallelism with owner-computes scheduling. Using compile-time knowledge of data layout and alignment we have implemented a scheme that determines whether two access can cause a cross-processor dependence [17]. We have used this to optimise barrier placement and the same analysis can be used to detect dependences requiring invalidation.

By using array section analysis [8] [18] we can accurately determine the data to invalidate, without the need of additional bit mask per write instruction [14] or invalidating the entire array as required in previous schemes. Although using *local* information [6] our scheme approaches the accuracy of *global* schemes as we have accurate array section analysis to determine what to invalidate without the runtime overhead of invalidation traffic.

This paper makes the following contributions:

- It introduces a new compiler directed distributed invalidation algorithm to eliminate invalidation traffic in a hybrid write-invalidate based VSM system.

- It precisely defines what data should be locally invalidated when using an SPMD execution model and static-scheduling to eliminate unnecessary invalidations.
- It presents results illustrating the performance benefit in terms of invalidation traffic reduction and actual execution times on an existing system.

In Section 2 we describe our scheme, discuss related work and present an illustrative example. Section 3 defines what data is to be invalidated when using this scheme. Section 4 describes the impact of control-flow on our scheme and in Section 5 the compiler algorithm is developed. Section 6 details the experiments and results and Section 7 concludes the paper.

2. Distributed Invalidation

This section describes the hybrid scheme and compares our approach with the widely used write-invalidate sequential consistency model and previous approaches to distributed invalidation.

2.1. Hybrid Scheme

We assume sequential consistency and an invalidation based protocol with three states; exclusive ($\mathcal{E}x$), read-only ($\mathcal{R}o$) and invalid ($\mathcal{I}$). Our approach is a hybrid one in that we use compiler analysis to implement DI where possible, however, when compiler analysis is difficult or impossible the directory will ensure correctness and maintain a reasonable level of performance.

Invalidation is required when a processor writes to a location which has valid copies on other processors. If these remote copies are not invalidated then a remote processor may read an 'old' or stale data value. In traditional sequential consistency the directory will point (either directly or indirectly) to all remote copies. Before writing to a location a processor will send invalidation messages to all processors pointed to by the directory. Only when all acknowledgements have been received will exclusive access be granted and the new value written. If, however, each processor invalidates its stale read copies, and marks as exclusive the data it will write memory consistency is maintained without incurring any invalidation traffic.

We have implemented [9] 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 [16]. This both 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 invalid 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.

2.2. Example

To illustrate the compiler approach developed in this paper, consider the Tred2 fragment (from the EISPACK suite[1]) shown in column 1 of Figure 1. The second column shows the local node program generated by the MARS compiler [1] after column partitioning; $lo1, hi1$ are the introduced lower and upper bounds of the local data space. The barriers are introduced by the compiler due to the detected cross-processor data dependences.

The compiler uses a SPMD execution model, with static data layout and owner-computes scheduling and detects that the only cross-processor data dependences (δ^f) are the flow-dependence on `e` from (s2,s5 and s6) to s7 and the anti-dependence (δ^a) on `d` from (s1,s2, s4, s7) to s8 and s9 and on `z` from s2 to s7. Placing a barrier between s6, s7 and s7, s8 covers these cross-processor dependences. The owner-computes rule guarantees that all output dependences are within a processor; for further details on barrier placement optimisation see [17]. With static data layout and owner-computes scheduling, invalidation is required only where there exits a **cross-processor anti-dependence**. Arrays `d` and `z` must both be invalidated; the code for the invalidation of array `d`, for example, is shown in column 3 of Figure 1.

2.3. Related Work

The majority of research related to DI has been targeted at scalable cache coherent non-uniform memory access architectures (CC-NUMA). However the coherence issues for CC-NUMAs and VSM architectures are very similar.

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], [6]. Early work invalidated all cached data at each epoch. More recent schemes have used tags or timestamps 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.

[1] after induction variable substitution etc.

Original Program

```
Do j = 1, n+1-ii
s1: z(j,n+2-ii) = d(j)
EndDo

Do j = 1, n+1-ii
  Do k = j+1 ,n+1-ii
s2: e(k) = e(k)+z(k,j)*d(j)
  EndDo
EndDo

Do j = 1, n+1-ii
s3: gg(j) = e(j) +z(j,j)*d(j)
EndDo

Do j = 1, n+1-ii
  Do k = j+1 ,n+1-ii
s4: gg(j) = gg(j) +z(k,j)*d(k)
  EndDo
EndDo

Do j = 1, n+1-ii
s5: e(j) = gg(j)
EndDo

Do j = 1,n+1-ii
s6: e(j) = e(j)/h
EndDo

Do j = 1,n+1-ii
  Do k = j,n+1-ii
s7: z(k,j) = z(k,j)-d(j)*e(k)
           -e(j)*d(k)
  EndDo
EndDo

Do j = 1,n+1-ii
s8: d(j) = z(n+1-ii,j)
EndDo

Do k = 1, n+1-ii
s9: d(k) = d(k)/scale
EndDo
```

Local Program

```
if (n+2-ii .LE. hi1)
  if (lo1 .LE. n+2-ii)
    Do j = 1, n+1-ii
      z(j,n+2-ii) = d(j)
    EndDo
  endif
endif

Do j = 1, n+1-ii
  Do k=max(j+1,lo1),min(hi1,n+1-ii)
    e(k) = e(k)+z(k,j)*d(j)
  EndDo
EndDo

Do j = max(1,lo1),min(hi1,n+1-ii)
  gg(j) = e(j) +z(j,j)*d(j)
EndDo

Do j = max(1,lo1),min(hi1,n+1-ii)
  Do k = j+1 ,n+1-ii
    gg(j) = gg(j) +z(k,j)*d(k)
  EndDo
EndDo

Do j = max(1,lo1),min(hi1,n+1-ii)
  e(j) = gg(j)
EndDo

Do j = max(1,lo1),min(hi1,n+1-ii)
  e(j) = e(j)/h
EndDo

call barrier1()

Do j = max(1,lo1),min(hi1,n+1-ii)
  Do k = j,n+1-ii
    z(k,j) = z(k,j)-d(j)*e(k)
           -e(j)*d(k)
  EndDo
EndDo

call barrier2()

Do j = max(1,lo1),min(hi1,n+1-ii)
  d(j) = z(n+1-ii,j)
EndDo

Do k = max(1,lo1),min(hi1,n+1-ii)
  d(k) = d(k)/scale
EndDo
```

Local Invalidation of d

```
  .
  .
call barrier2()


Do j = 1, min(max(1,lo1)-1,n+1-ii),s
 call local_invalidate(d(j))
EndDo

Do j = min(hi1,n+1-ii)+1,n+1-ii,s
 call local_invalidate(d(j))

EndDo

Do j = max(1,lo1),min(hi1,n+1-ii),s
 call local_exclusive(d(j))
EndDo

Do j = max(1,lo1),min(hi1,n+1-ii)
  d(j) = z(n+1-ii,j)
EndDo

Do k = max(1,lo1),min(hi1,n+1-ii)
  d(k) = d(k)/scale
EndDo
```

Figure 1. Compiler directed invalidation example

Fast selective invalidation (FSI) [2] will invalidate all the data, read and write, accessed within an epoch while time stamping (TS) strategies [2] will invalidate all write data after the parallel loop, if in the next epoch the data is read. In the example in Figure 1, FSI will invalidate the entire contents of different arrays 21 times while TS reduces this to 8. The life span [3] strategy will only invalidate read data at the next write in a different epoch. In our example, this restricts the invalidation to the entire arrays after the epochs surrounding s5, s7 and s8. A more sophisticated form of analysis [4] based on reads after RDS (relaxed determining sequence)[2] examines each epoch for the first occurrence of an upwardly exposed read which could be preceded by an RDS. Such reads will require a special cache read to determine their validity. In [7], vectorisation is used to the overhead of redundant invalidation when using RDS analysis. In our example the data will be considered invalid on the read access to `gg` in s5, `e` in s6, `e` in s7, `z` in s8 and `d` in s9 . Thus previous compiler based schemes invalidate or consider stale between 3 and 21 of the array access, while our scheme accurately determines that there is just 2. In fact this is a generous analysis. When using KAP [12] an autoparallelising compiler it parallelises the `k` loop of s2, so the inner rather than the outer loop defines the epoch boundary. As `e` is both read and written in this smaller epoch, the above schemes would all increase the amount of invalidation performed by an order of magnitude.

A directory based method, Dynamic Self Invalidation [13], developed for CC-NUMAs, adds extra state to tag copies which are to be invalidated. The invalidation is part of the directory protocol so no compiler or human intervention is needed. A hybrid technique, CICO [10], lets the programmer, or a profile based tool annotate the program with *checkin* and *checkout* directives to inform the directory that it should invalidate data. In contrast to compiler directed coherence these directives are only performance hints; the directory hardware is still responsible for correctness. The use of hybrid techniques where the compiler is used to eliminate unnecessary coherence actions rather than relying on it to maintain coherence is also investigated by Mounes-Toussi et al. [15] for CC-NUMAs. They note the critical role of (but do not develop) compiler analysis and show that if static scheduling and accurate dependence analysis were available then their coherence scheme would show marked improvement across all the programs examined they examined from the Perfect Benchmark suite. The analysis described in this paper will be of use in their scheme. Our hybrid scheme[9] allows exploitation of the SPMD execution model; however, if a fork/join model is preferred, the above techniques may also be used.

[2] Simplistically this is a read or write followed by a write in different epochs.

3. What to Invalidate

A processor must invalidate all its copies of pages that will be written by others and ensure exclusive access to the page it will write. This section is concerned in precisely determining *what* pages should be modified, when using owner-computes scheduling, and develops a set of equations which are used to determine the local action each processor must take with respect to local data to maintain coherence without incurring invalidation traffic. This analysis is related to that used in communication optimisation for distributed memory machines [11]. We assume in the remainder of this paper that cross-processor dependence analysis and barrier placement have already been undertaken and that discussion is restricted to data involved in cross-processor anti-dependences.

3.1. Equations

Before the data marked invalid and exclusive can be derived 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.

Definition 1 *Let W_i, R_i be the set of all the pages (global data) written/read in action i*

Definition 2 *Let $\underline{W}_i, \underline{R}_i$ be the set of local pages written/read in action i*

Definition 3 *Let $\underline{\mathcal{E}x}_i, \underline{\mathcal{R}o}_i$ be the set of local pages in exclusive/read state after action i*

A superscript is used, if necessary to distinguish between actions occurring on different processors e.g. $\underline{W}_i^1, \underline{W}_i^2$ refer to the 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 [15]. Therefore each processor writes to the same data throughout the life time of the program and no two processors write to the same pages

We also assume no read/write false sharing when invalidating copies and making local data exclusive. 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 state will be increased by any pages written locally that were not previously marked as exclusive.

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

Remote processors will not write to local data previously in exclusive state. Thus remote writes do not invalidate or effect local data in exclusive state.

The number 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 i.e.

$$\underline{\mathcal{R}o_i} = \underline{\mathcal{R}o_{i-1}} - (\underline{\mathcal{R}o_{i-1}} \cap W_i)$$

Let p be the number of processors and z be the processor id of the local processor. After a read action, the number of local pages in read state will be increased by other processors reading local pages previously in exclusive state and by the remote pages read by the local processor.

$$\underline{\mathcal{R}o_i} = \underline{\mathcal{R}o_{i-1}} \cup ((\bigcup_{k \in 1,\ldots,p}^{k \neq z} \underline{R_i^k}) \cap \underline{\mathcal{E}x_{i-1}}) \cup (\underline{R_i^z} - (\underline{\mathcal{E}x_{i-1}} \cap \underline{R_i^z}))$$

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 processor will do this. The directory structure guarantees that remote reads are not made to invalid copies and therefore reads to previous invalid data need not be considered. The local pages in exclusive state are decreased by reads from other processors to local pages which were previously in exclusive state.

$$\underline{\mathcal{E}x_i} = \underline{\mathcal{E}x_{i-1}} - (\underline{\mathcal{E}x_{i-1}} \cap (\bigcup_{k \in 1..p}^{k \neq z} \underline{R_i^k}))$$

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 and will be used by the compiler algorithm to perform distributed invalidation.

Definition 4 *Let $\underline{LEx_i}, \underline{LI_i}$ 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.

$$\underline{LEx_i} = \underline{W_i} - (\underline{\mathcal{E}x_{i-1}} \cap \underline{W_i}) \quad (1)$$

The local pages to be invalidated are those that will no longer in either a read or exclusive state.

$$\underline{LI_i} = \underline{\mathcal{R}o_{i-1}} - (\underline{\mathcal{R}o_i} \cap \underline{\mathcal{R}o_{i-1}}) - (\underline{\mathcal{E}x_i} \cap \underline{\mathcal{R}o_{i-1}}) \quad (2)$$

By set manipulation (8) can be reexpressed more usefully as equation (9) where those local pages which are in a read state should be invalidated if written to by remote processors.

$$\underline{LI_i} = (W_i - \underline{W_i}) \cap \underline{\mathcal{R}o_{i-1}} \quad (3)$$

The above equations (1) and (3) if honoured by the compiler, will eliminate invalidation traffic and unnecessary misses [3].

[3] apart from those due to compile-time unknown read/write false-sharing [18]

3.2. Properties

Sequentially consistency write-invalidate protocols have the following property

Property 1 *Any page x that is written to will only be in exclusive mode on that processor z and all read copies on other processors will be invalidated*

It can be shown that the equations (1) and (3) also satisfy this property. However as we use a hybrid scheme it is not necessary for those invalidations inserted by the compiler to satisfy this property to maintain memory consistency. The following weaker property is all that need be satisfied

Property 2 *Any page x that is marked exclusive on one processor is marked as invalid on all other processors containing that page.*

If the compiler overestimates the region to mark exclusive, there will be excess invalidation and later unnecessary read misses. If it underestimates, then there will be invalidation traffic. As long as this property holds, however, the memory will be sequentially consistent at the barrier immediately following the write. This property is used in section 5 when presenting an approximate scheme.

3.3. Example

The previous sub-section defines the data to be invalidated in terms of set algebra which is not suitable for compiler manipulation. However, the elements belonging to these sets are derived from the array sections generated from the iteration spaces and access functions. In [18] we described how the set based expressions may be translated into affine constraints (polyhedra) and access functions. We will use this representation in deriving the data regions, of array d to be invalidated for the program shown in Figure 1. To denote the elements or array section accessed, we use the iteration space, access pair; e.g. the elements of d accessed in s2: are

$$[(1 \leq j \leq n+1-ii)], [d(j)]$$

which are also the local copies of d in read state before s7 ($\underline{\mathcal{R}o_6}$).We assume for the purposes of this example that the remaining elements of d are in exclusive state i.e

$$\underline{\mathcal{E}x_6} : [max(lo1, n+2-ii) \leq j \leq min(hi1, n)], [d(j)]$$

The data to be written in s8 is determined by its local iteration space i.e.

$$\underline{W_8} : [max(lo1, 1) \leq j \leq min(hi1, n+1-ii)], [d(j)]$$

Substituting these values into equations (1) and (3) gives

$$\underline{LEx_7} : [max(lo1, 1) \leq j \leq min(hi1, n+1-ii)], [d(j)]$$

Original Program

```
Do j = 1,n
  Do i = max(1,lo1),min(n,hi1)
1:   x(i) = x(i) +a(j)
  Enddo
Enddo
Do t = 1,nt
  call barrier()
  Do i = max(1,lo1),min(n,hi1)
2:  a(i) = i + t
  Enddo
  if (P) then
    Do i = max(1,lo1),min(n,hi1)
3:   c(i) = i - t
4:   x(i) = b(n-i+1)
    Enddo
  else
    Do i = max(1,lo1),min(t,hi1)
6:   x(i) = b(i+1)
7:   y(i) = c(i+1)
    Enddo
  endif
  call barrier()
  Do i = max(1,lo1),min(t,hi1)
8:  z(i) = a(i) +a(i-1)
9:  b(i) = i + t
  Enddo
Enddo
```

SSA for a

```
Do j = 1,n
  Do i = max(1,lo1),min(n,hi1)
1:  x(i) = x(i) +a1(j) {,}
  Enddo
Enddo
a1_read = eta(n>=max(1,lo1),a1,0)
a1_write = eta(n>=max(1,lo1),0,0)
Do t = 1,nt
  a2_read = mu(t>1 ,a1_read,a4_read,)
  a2_write = mu(t>1 ,a1_write,a4_write)

  Do i = max(1,lo1),min(n,hi1)
2:  a3(i) = i + t {a2_write,a2_read}
  Enddo
  a3_read = eta(n>=max(1,lo1),a2_read,a2_read)
  a3_write = eta(n>=max(1,lo1),a3,a2_write)

  Do i = max(1,lo1),min(t,hi1)
8:  z(i) = a(i) +a4(i-1) {a3_write,a3_read}
  Enddo
  a4_read = eta(t>=max(1,lo1),a4,a3_read)
  a4_write = a3_write

Enddo
```

SSA for b

```
if (P) then
  Do i = max(1,lo1),min(n,hi1)
4:  x(i) = b1(n-i+1)
  Enddo
  b1_read = eta(n>max(1,lo1),0,b1)
  b1_write = eta(n>max(1,lo1),0,0)
else
  Do i = max(1,lo1),min(t,hi1)
6:  x(i) = b2(i+1)
  Enddo
  b2_read = eta(t>max(1,lo1),0,b2)
  b2_write = eta(t>max(1,lo1),0,0)
endif
b3_read =gamma(P,b1_read,b2_read)
b3_write =gamma(P,b1_write,b2_write)
Do i = max(1,lo1),min(t,hi1)
9:  b3(i) = i + t {b3_read,b3_write}
Enddo
```

Figure 2.

$\underline{LI}_i : [1 \leq j \leq min(max(lo1,1)-1, n+1-ii)], [d(j)] +$
$[max(min(hi1, n+1-ii)+1, 1) \leq j \leq n+1-ii], [d(j)]$

Thus there is one convex region to mark exclusive and two separate convex ones to mark invalid. The code for this is shown in column 3 of Figure 1.

Note there is a step s, which denotes page size; invalidation need only be done a page rather element-wise basis. The impact of page size on code generation is discussed in [18]. Enumerating all previous reads and writes even for simple programs would be very expensive. Section 5 develops approximations to equations (1) and (3) aminable to compiler manipulation.

4. Control-flow

The coherence state is defined recursively in terms of previous states. For general programs it is therefore necessary to have a systematic method to determine previous actions in programs with possibly complex control-flow. To determine previous read and write actions, we use a modified version of the gated Static Single Assignment (SSA) form described in [20]. Due to space restrictions only a very brief overview of this approach is given. At certain points in the program pseudo-functions μ, γ, η' functions are inserted allowing the tracing back of reaching dependences.

- μ function: merges values at the head of a loop. On the first iteration, values before the loop body are selected; afterwards the last value from the previous iteration
- γ: merges values after a conditional selection
- η': selects the last value at the end of a loop if non-empty

Construction of SSA-forms of programs is well documented [5]; the benefit of gated SSA is that the conditionals are preserved. For our application, we are not interested in the values of reaching dependences, but the access regions of data involved in cross-processor anti-dependences, which is determined by their enclosing local iteration space and access function after partitioning. In [20], an elegant method to determine previous actions, by demand-driven back propagation is described

Consider the local node program in Figure 2 and the the modified gated SSA form of the program for the access to a in the second column (where mu = μ,gamma= γ and eta = η'). For each access the previous write and read accesses are available in the pair {prev_write,prev_read} For instance, the previous access to array a before the write to a3 is given by {a2_write,a2_read}. This can be expanded by substitution, (given $n > max(1, lo1)$)

$$\begin{aligned}
\{a2_write\} &= \mu((t=1,nt), a1_write, a4_write) \\
&= \mu((t=1,nt), \\
&\quad (\eta(n >= max(1,lo1), 0, 0), a3_write)) \\
&= \mu((t=1,nt), 0, a3)
\end{aligned}$$

Thus there was either no previous write $t = 1$ or it was a3 on a previous iteration $t > 1$. Translating into a system of inequalities [18] gives:

$$\begin{aligned}\mu((t = 1, nt), 0, a3) &= [(max(1, lo1) \le i \le min(n, hi1)) \\ &\wedge \ (t \ge 2)], [a3(i)]\end{aligned}$$

The previous read action can be found similarly to be $\mu((t = 1, nt), a1, a4)$ or

$$[(max(1, lo1) \le j \le min(n, hi1)) \wedge (t = 1)], [a1(j)]$$

$$+[(max(1, lo1) \le i \le min(t - 1, hi1)) \wedge (t \ge 2)], [a4(i)]$$

Note that the value of t is decremented by the number of previous iterations considered. If we wish to traverse further back, previous actions may form a recurrence [20]. Reaching read actions in the presence of if constructs are handled in a similar manner. The previous read action to the write to array `b` in statement s9:(as shown in column 3 of Figure 2) depends on the conditional P i.e.

$$[(max(1, lo) \le i \le min(n, hi)) \wedge P], [b(n - i + 1)]$$

$$+[(max(1, lo) \le i \le min(t, hi)) \wedge \neg P], [b(i + 1)]$$

Determining previous accesses within nestings of loops and conditionals, is more complex. If condition P is invariant and statement s3 is reached, and a write action to `c` takes place then there are no previous reads as they may only occur in statement s3 when P must be false. Difficulties arise when P is not loop invariant. Determining the last iteration of t when P was false would require an array of values of P $P[1..t]$ to be maintained. The previous read would be defined as

$$\begin{gathered}[(max(1, lo) \le i \le min(k, hi)) \\ \wedge \neg P[k] \wedge P[k + 1..t]], [c(i + 1)]\end{gathered}$$

More complex control-flow structures make this impractical and as our compiler is, at present, restricted to systems of affine constraints involving iterators and parameters, it does not consider general predicates. However, as we have a hybrid scheme, approximations can be made while guaranteeing that correctness is preserved.

5. Algorithm

Determining all previous read and write actions using gated SSA will lead to an explosion of terms and therefore a number of approximations are necessary for a practical algorithm. This section describes a tractable technique to determine which pages to invalidate and a compiler algorithm to perform these tasks.

1. if $\exists$ a write $w_i \in W$ of each shared variable V with a non-linear access remove V
2. $\forall$ write access $w_i \in W$ of each remaining shared variable V
   (a) determine all dominating cross-processor reads R after all possible previous writes, w_{i-1}
   (b) $\forall$ read access $r_j \in R$
      i. $prevread_i := prevreadsi \cup r_j$
      ii. if write w_i does not *cover* read r_j then
         A. $notcover_i := notcover_i \cup r_j$
3. $\forall$ write access $w_i \in W$ of each shared variable V, if $notcover_i \neq \emptyset$ then
   (a) if $\exists$ w_t that strictly postdominates w_i then
      i. $\forall$ read access $r_j \in notcover_i$
         A. if write w_t *cover* read r_j then
         B. * $prevread_t := prevread_t \cup r_j$
         C. * $notcover_i := notcover_i - r_j$
4. $\forall$ write access $w_i \in W$ of each shared variable V determine
   (a) $\forall$ read access $r_j \in prevread_i$
      i. $\underline{L\hat{I}}_i = (w_i - \underline{w_i}) \cap r_j$
      ii. $\underline{L\hat{E}x}_i = \underline{w_i} \cap r_j$
5. $\forall$ write access $w_i \in W$ of each shared variable V
   (a) Insert conditional invalidation before the write access w_i

Figure 3. Algorithm

5.1. Approximate Read and Exclusive State

The major problem of generating a correct but efficient implementation is in determining precisely the coherence state of memory before a write. The equations in section 3 define precisely what data is to be invalidated in terms of all previous read and write actions. Unfortunately, in the general case, enumerating all previous actions is impractical, leading to an explosion in the number of terms and hence approximations must be made. In this section we start at the point where a write will take place and restrict our attention to those reads that have occurred since the last output dependence or write.

Let the previous write action be considered as taking place at $i = 1$. Thus there are $2..i - 1$ read actions between write action 1 and the write action i, under consideration.

Let $\overline{R_i}$ denote the read actions since the last write i.e

$$\overline{R_i} = \bigcup_{\ell=2..i-1} ((\bigcup_{k\in 1,\dots,p}^{k\neq z} \underline{R}_\ell^k) \cap \underline{\mathcal{E}x}_1) \cup (\underline{R}_\ell^z - (\underline{\mathcal{E}x}_1 \cap \underline{R}_\ell^z))) \quad (4)$$

Write-Invalidate	2	4	8	16
Read Misses	1000	6000	28000	120000
Write Misses	2026	3032	5040	9056
Invalidations	1000	6000	28000	120000
Compiler Invalidate				
Read Misses	1000	6000	28000	120000
Write Misses	1026	1032	1040	1056
Invalidations	0	0	0	0

Figure 4. Mvm

Write-Invalidate	2	4	8	16
Read Misses	1000	3000	7000	15000
Write Misses	2024	4024	8024	16024
Invalidations	1000	3000	7000	15000
Compiler Invalidate				
Read Misses	1000	3000	7000	15000
Write Misses	1024	1024	1024	1024
Invalidations	0	0	0	0

Figure 5. Jacobi

as there are only read accesses, it can be shown that

$$\underline{\mathcal{R}o}_i = \underline{\mathcal{R}o}_1 \cup \overline{R_i}$$

The data to be invalidated defined in equation (2) can therefore be rewritten as:

$$\underline{LI}_i = (W_i - \underline{W}_i) \cap \underline{\mathcal{R}o}_0 \cup (W_i - \underline{W}_i) \cap \overline{R_i}$$

It can be shown that the data to mark exclusive can also be reexpressed as:

$$\underline{LEx}_i = \underline{W}_i - (\underline{\mathcal{E}x}_0 \cap \underline{W}_i) + (\underline{W}_i \cap \overline{R_i})$$

If we assume that all data accessed in action $2..i-1$ was in exclusive state when $i = 1$ i.e.

$$(\underline{W}_i - (\underline{\mathcal{E}x}_0 \cap \underline{W}_i)) = \emptyset, \underline{\mathcal{R}o}_0 = \emptyset$$

then equations (2) and (2) can be rewritten as:

$$\underline{L\hat{E}x}_i = W_i \cap \overline{R_i}$$

$$\underline{L\hat{I}}_i = (W_i - \underline{W}_i) \cap \overline{R_i}$$

where $\underline{L\hat{E}x}_i$, $\underline{L\hat{I}}_i$ denote approximations to the actual data to be invalidated under the assumption that data was previously in exclusive state. These definitions satisfy the property 2 described in section 3 and maintain memory coherence.

If we assume that all reads are covered by the first write post-dominating them, then at each write action, only reads involved in a cross-processor anti-dependence after the preceding write need be handled. From this assumption after each write, all the data is in exclusive mode and it is relatively straightforward to determine the data to be marked exclusive/invalid.

Clearly this assumption is not true in general and when a write does not cover a read, we make the decision that it be explicitly considered by the immediately post-dominating write. If there does not exist a unique post-dominator or if it too does not cover the read, we can either ignore it, possibly incurring later invalidation traffic or kill the dependence by invalidating all read copies even though the write access does not cover it, at the possible expense of additional read misses. In our algorithm we choose not to incur unnecessary reads misses at the potential cost of invalidation traffic. Thus we guarantee that we incur no more read misses than a sequential consistent scheme and should significantly reduce (but not entirely eliminate) invalidation traffic. Later work will consider a more sophisticated approach

5.2. Description

The algorithm in Figure 3 is based on the approximations described above and makes simplifying assumptions in the presence of non-linear array access regions and compile-time unknown control-flow. In the first step if a variable has a non-linear array access regions we resort to sequential consistency for that variable; this is a major benefit of having a strong hybrid model in that there is always a reasonably efficient fallback position that maintains memory consistency. The problem of considering all previous array accesses is solved by starting at the point where the write will take place and considering only previous read actions after the most recent write on all possible reaching control paths. This is achieved by associating a set *prev-reads* (which corresponds to $\overline{R_i}$ in equation 4) with each write access . Each set contains only those reads occurring after the previous write. As a write access may not cover all of the reads in *prev-read* a second set *notcover* is generated and is used in the next step. Step 3 considers those accesses that are not covered by the write access and determines if subsequent write accesses will cover the dependence. To prevent an expensive search of all future write actions, only write accesses that strictly postdominate are considered. Those read accesses remaining in *notcover* after step 3 may, however, cause invalidation traffic if later write accesses refer to them. Step 4 generates the invalidation sets based on equations (5.1),(5.1) which is then inserted into the program in step 5. Once the data accesses to be considered for invalidation have been determined we place coherence calls immediately before the appropriate write. As this follows the barrier nearest the the

Write-Invalidate	2	4	8
Read Misses	71147	192401	465473
Write Misses	72177	109671	135315
Invalidations	71141	192365	465269
Compiler Invalidate			
Read Misses	71147	192401	465473
Write Misses	1546	1558	1582
Invalidations	510	1530	3570

Figure 6. Tred2

sink of the dependence, there cannot be any remaining read requests.

This algorithm is conservative in the presence of non-affine terms and does not consider procedures or subroutines. They can be ignored at present as the hybrid scheme guarantees correctness, but later work will need to consider inter-procedural analysis.

6. Experiments

The compiler algorithm has been partly implemented in MARS [1] - an autoparallelising compiler based on an extended linear algebraic representation of Fortran programs. Cross-processor, cover, and kill dependences have been implemented, as have dominator and post-dominator determination. Gated SSA has not been implemented within MARS, so this information was added by hand. The target architecture for the experiments was a 30 node EDS prototype with a supervisor level process known as the *mapper* which provides a fixed-distributed scheme for the management of VSM consistency on a 4KByte page basis through the manipulation of access permissions in MMU Page Table Entries.

6.1. Results

We present performance results for three applications, an iterative solver Mvm (N=1024), Jacobi (N=512) and Tred2 (N=512) a benchmark from the EISPACK suite. These are executed on 1 to 16 processors, except for the case of Tred2, which was not executed on 16 processors due to hardware problems with our EDS prototype.

Invalidation Figures 4, 5 and 6 present counts of read misses,write misses and invalidation traffic with and without distributed invalidation for the three programs. The invalidation traffic for Mvm and Jacobi is entirely eliminated without incurring any additional read misses. The number of write misses is severely reduced, those remaining being unavoidable cold start misses. For Tred2, the compiler reduces the amount of invalidation and write misses by over 97 %. On closer inspection of the misses associated with each array, the number of invalidations is unaltered for one of the arrays 4. This is due to the algorithm assuming that previous writes had covered all previous read accesses. If the algorithm is altered for this dependence, such that all dependences are explicitly killed by over invalidating at each write, the write misses (apart from coldstarts) and invalidations are entirely eliminated. In this case the number of read misses is not increased, but if such a decision were used throughout the program for all arrays, there would be an increase in read misses.

Execution Time Although the primary purpose of these experiments is to demonstrate how the algorithm can reduce invalidation in a hybrid write-invalidation scheme, it is interesting see its impact on actual execution time when this scheme is implemented on a real system. Figure 7 shows the execution time for standard write-invalidate and distributed invalidation (dist inv.) for Mvm, Jacobi and Tred2. Compiler-inserted invalidation enables a finer-grained synchronisation optimisation[16] to be used in Mvm and Jacobi and is labelled optimised in Figure 7. The performance improvement for Mvm and Jacobi (not including initialisation) varies from 0.26% to 46.78% across 1 to 16 processors. The communication within Jacobi is restricted to border elements between processors, while in Mvm an entire vector is read (and later invalidated) by all processors. For 16 processors, there is over 10 times as much invalidation traffic for Mvm as for Jacobi and thus Mvm benefits to a larger extent with its removal.

Although the compiler reduces the execution time for Tred2 by between 24% and 41% this is not sufficient to prevent the slow down of this application. This slow down is due to the large number of read misses and the current inefficient implementation of the mapper. Nevertheless, near elimination of invalidations has improved execution time and more efficient implementations should show a corresponding improvement.

7. Conclusions and Further Work

This paper has described compiler analysis for a hybrid distributed invalidation scheme making explicitly use of compile-time knowledge of scheduling and data layout. We applied this algorithm to some sample programs and showed a dramatic reduction in invalidation messages without the increase in unnecessary misses of previous compiler-controlled schemes. The method is applicable to any cross-processor anti-dependence in the absence of false sharing.

Further experimentation on a larger set of problems is needed to validate our algorithm. Future work will focus on

Mvm	1	2	4	8	16
Standard	1364.3	704.5	384.2	251.7	233.4
Dist Inv.	-	700.4	383.2	234.1	198.5
% reduction	0	0.58	0.26	6.99	14.95
Optimised	-	688.2	362.3	191.6	124.2
% reduction	0	2.31	5.70	23.87	46.78
Jacobi					
Standard	1670.8	856.1	450.5	242.1	143.4
Dist Inv	-	844.4	434.9	228.2	128.4
% reduction	0	1.37	3.46	5.74	10.46
Optimised	-	839.7	421.0	219.2	116.6
% reduction	0	1.92	6.55	9.46	18.68
Tred2					
Normal	4319.8	15100.4	15462.6	19485.3	
Dist Inv	-	8907.0	11695.9	14690.4	
% reduction	0	41.01	24.36	24.61	

Figure 7. Execution Times

implementing gated SSA in our compiler, investigating the trade-off between invalidation traffic and unnecessary miss optimisation and extending our work to deal fully with false sharing. We intend to also consider write-update protocols and investigate compiler analysis for other VSM optimisations such as latency-hiding post-storing and develop a unifying approach to compiling for VSM.

References

[1] Bodin F., O'Boyle M.F.P. **A Compiler Strategy for SVM**, the Third Workshop on Languages, Compilers and Runtime Systems for Scalable Computing, *Kluwer Press*, 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** Proceeding of International Conference on Supercomputing, July 1992

[4] Choi L., Yew P-C., **A Compiler-Directed Cache Coherence Scheme with Improves Intertask Locality**, Proceedings of Supercomputing '94, Washington *IEEE Press*, Nov. 1994.

[5] Cytron R., Ferrante J., Rosen B.K., Wegman M.N., Zadeck F.K.,**Efficiently Computing Static Single Assignment Form and the Control Dependence Graph**, ACM Transactions on Programming Languages and Systems 13 (4), October 1991.

[6] Darnell E., Kennedy K., **Cache Coherence Using Local Knowledge**, Proceedings of Supercomputing 1993, pages 720-729, Nov 1993.

[7] Darnell E., Mellor-Crumney J.M., Kennedy K., **Automatic Software Cache Coherence through Vectorisation**, Proceedings of ICS, pages 129-138, July 1992.

[8] Feautrier P., **Data flow Analysis of Array and Scalar References**, Int Journal of Parallel Programming, vol 20 no 1, 1991.

[9] Ford R.W., Nisbet A.N., Bull J.M., **User-Level VSM Optimisation and its Application**, Proceedings of PARA '95, LNCS, Lyngby Demark.

[10] Hill M. Larus J. Reinhardt S.,**Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors** ACM Trans. on Computer Systems 11(4) 300-318 Nov 1993.

[11] Hiranandi S., Kennedy K., Tseng C-W.,**Evaluation of Compiler Optimisations for Fortran D on MIMD Distributed-Memory Machines**, Proceedings of ICS'92, July 1992.

[12] Kuck and Associates Inc. **KAP User's Guide**, Champaign, Illinois, 1988.

[13] Lebeck A.R., Wood D.A. **Dynamic Self Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors**, Proceedings of the 22nd Annual International Symposium on Computer Architecture, pages 48-59, 1995.

[14] Louri A.,Sung H., **A Compiler Directed Cache Coherence Scheme with Fast and Parallel Expicit Invalidation** Proceedings of the International Conference on Parallel Processing, pages 2-9,August 1992.

[15] Mounes-Toussi F., Lilja D.J., Li Z. **An Evaluation of a Compiler Optimization for Improving the Performance of a Coherence Directory** Proceedings of the International Conference on Supercomputing, pages 75-84, Manchester U.K., July 1994.

[16] Nisbet A.N., Ford R.W., **Spinning on Coherency: A New VSM Optimisation for Write-Invalidate**, To Appear in Proceedings of High Performance Computing and Networking Europe, Brussels, April 1996.

[17] O'Boyle M.F.P., Kervella L.,Bodin F., **Synchronisation Minimisation in a SPMD Execution Model**, Journal of Parallel and Distributed Computing Vol 29. 196-210, *Academic Press Inc.*, September 1995.

[18] O'Boyle M.F.P., Ford R.W., Nisbet A.N. **Compiler Reduction of Invalidation Traffic in Virtual Shared Memory Systems**, CNC Tech. Rep. February 1996. *a shorter version appears at EuroPar '96*

[19] Skeppstedt J., Stenstrom P.,**A Compiler Algorithm that Reduces Latency in Ownership-Based Cache Coherence** Proceedings of PACT 95, ACM Press, Cyprus. June 1995.

[20] Tu P., Padua D., **Gated SSA-Based Demand-Driven Symbolic Analysis for Parallelizing Compilers**, Proceedings of the International Conference on Supercomputing, pages 414-423, Barcelona Spain., July 1995