A scalable cache coherence mechanism using a selectively clearable cache memory

D.A. Abramson
Division of Information Technology, CSIRO,
c/- Department of Communication and Electronic Engineering, Royal Melbourne Institute of Technology
K. Ramamohanarao
Department of Computer Science, University of Melbourne.
M. Ross
Department of Computer Science, Royal Melbourne Institute of Technology

In this paper we propose a scalable solution to the cache coherence problem for write through caches which uses advice from software on when to enforce coherence. The solution removes shared structures from the cache efficiently, and thus forces the subsequent cache misses to proceed to main memory. Shared structures are removed by a novel selectively clearable memory chip. The chip is an extension of already available clearable devices. The shared structures are removed after mutual exclusion has been obtained. This guarantees that the cache cannot hold stale copies of the data. A number of optimisations are described which improve the performance of the scheme. These optimisations allow a write back cache as well as write through. They also increase the selectivity of the clear operation for caches with a large number of sets. Some simulation studies indicate expected performance of the system.

Keywords and Phrases: Cache coherence, caches, memories, shared memories.
CR Categories: b3.

Manuscript received July 1988; revised February 1989.

1. INTRODUCTION

Cache memories have been used for many years to improve the memory access times on uniprocessor systems. They are even more important in multiprocessor systems as they not only reduce memory access times but also reduce memory and network contention. Figure 1 illustrates a shared memory multiprocessor system with a cache dedicated to each processor.

Unfortunately it is not possible to add caches freely to such a multiprocessor because the data in one cache may become inconsistent with the data in the other caches. The problem can be illustrated by considering two processes, A and B, and two processors C and D. A cache inconsistency can occur under two main conditions. First, consider process A executing on processor C. The local variables of A will be loaded into the C cache. If process A is then migrated to processor D, the variables will then be loaded into the D cache, and may become inconsistent with the values in the C cache. If process A is then returned to the C processor, it will use stale values, causing incorrect operation of the process. This situation is usually solved by clearing the processor cache when a process is migrated. The second condition arises if processes A and B share a variable. If process A executes on processor C, and process B executes on processor D, their local variables will be loaded into the respective caches as they are accessed. However, when they each access the shared variable, it will also be loaded into the processor cache. If they update the shared variable independently, then each will only see the value held in its cache, and will not see the changes made by the other process.

This problem has been studied widely, and there are a number of available solutions (Censier and Feautrier, 1978; Archibald and Baer, 1984; Dubois and Briggs, 1982 and Scheurich and Dubois, 1987). Most of the proposed solutions involve some additional hardware which forces the caches to become consistent. Such hardware allows communication between processors and external caches, as well as to the shared memory. This hardware also involves, during the enforcement of coherence, extra bus or network transactions.

The major advantage of such schemes is that they require no, or very little, advice from the high level software. Consequently, a multiprocess program which executes correctly on a uniprocessor will also behave cor-

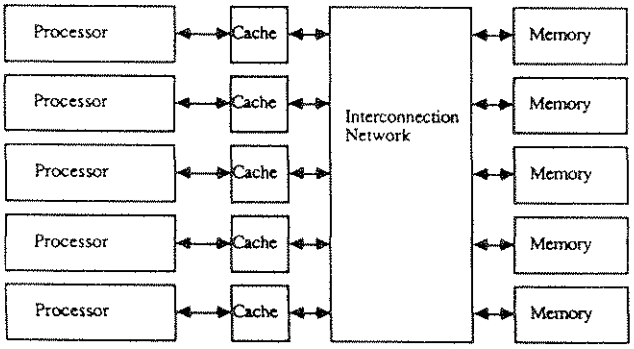

Figure 1. Shared Memory Multiprocessor.

rectly on a multiprocessor system. In other words, the synchronisation mechanisms needed to ensure correct operation on the uniprocessor are the same as those required on a multiprocessor. The hardware based schemes simplify the task of process migration in the multiprocessor, because all caches are consistent.

This paper is concerned with two main criticisms of hardware based schemes. First, the solutions may not scale up well when the number of processors is increased (Stone, 1987). For example, almost all large scale multiprocessors cannot use a shared bus for the processor-memory communication mechanism because the bus cannot support the required bandwidth. Consequently, communication networks such as multilevel switches are used because they offer scalable performance. Such networks rule out the use of simple bus monitoring mechanisms because it is not possible for a processor to watch all of the processor-memory traffic. The more complex hardware schemes which require a broadcast message to all caches (or some selected set of caches) can increase the network traffic significantly, and thus severely reduce the network bandwidth (Stone, 1987). Notably, the RP3 multiprocessor (Pfister et al., 1985) does not attempt to provide cache coherence for shared structures, but tags such data as non-cacheable.

Second, the hardware based schemes add complexity to the processor design (or the associated bus interface), and thus increase the **cost** and **size** of multiprocessors. If such hardware was not present then more space and money could be devoted to other areas, such as increasing the size or associativity of the cache, or adding an extra pipeline stage,. etc, which may offer a better performance improvement than the original cache coherence hardware. Such tradeoffs are particularly relevant when caches are placed on the processor chip itself, such as in the MC68030 microprocessor (Motorola).

A much ignored technique for solving the cache coherence problem involves using advice from the software on exactly *when* coherence is required, rather than attempting to provide it invisibly at all times. Such schemes significantly reduce the amount of hardware required (and thus the cost and size) and are also scalable to a large number of processors. In this paper we examine existing cache coherence software solutions and propose a new type of cache memory which when combined with software directives solves the coherence problem for a large number of processors connected by any communication mechanism. The proposed algorithm is simulated on a number of large memory reference traces and compared to other software based schemes. The additional hardware required for the new type of cache memory over and above a standard cache (non-coherent) is negligible.

2. SOFTWARE SOLUTIONS

The simplest software cache coherence algorithm allows only non-sharable or ready-only data to be placed in a processor's local cache memory. Such a solution is effective because incoherence can only occur when multiple processors are allowed to modify data. The performance of this scheme degrades when a processor modifies a shared data structure and then makes multiple read requests for the data. Whilst these access patterns are unlikely to occur in a general time sharing system, they are quite likely to occur in many tightly coupled multi-process applications. This approach is used in the RP3.

Another software based solution allows shared variables to migrate into the cache, but purges the entire cache when coherence is required. This scheme was suggested by Veidenbaum (1986). Such purging is requested by explicit code which demands cache coherence. In this scheme all processors use a write through memory update protocol, thus if a processor clears its cache then all subsequent reads will obtain the correct data. Unfortunately this scheme not only removes the inconsistent data from the cache, but also all of the other data, and possibly program instructions when a unified cache is used. As would be expected, clearing the entire cache often significantly reduces the hit rate. Therefore such a scheme appears to have little practical value in systems where coherence requirements occur frequently. Later in this paper we show the effect of the above policy on hit rates for a variety of cache configurations. The performance of a cache operating under such a policy is similar to a conventional cache experiencing a high rate of context switches.

A disadvantage of the software based schemes is that the only solution to incoherence, which is caused by process migration, is clearing the cache when a process is removed from a processor. The effect of this cache purge has been widely studied (Smith, 1982), and does not cause a serious performance degradation.

Software based solutions are attractive because they require no extra hardware and use advice on the way data is to be manipulated. However, it is clear that some hardware assistance is required in order to increase the performance of software coherence schemes.

3. A HYBRID SOLUTION

The software based scheme, described above, only demands coherence when it is actually required, but at this point it requires coherence on the complete cache. A program does not generally require total cache coherence at any instant, but coherence on a particular structure or structure component. Thus, it is possible to enforce coherence on a structure component when it is actually required. This approach poses two problems. First, how does the system know which structures are shared. Second, how does the system know when to request coherence. Detecting which structures or locations are shared is easy, because the programmer must make some explicit statement that a structure is shared so that the operating system knows how to load the program into memory. This declaration often takes the form of a modifier to the base *type* of the object. For example, an integer variable may be declared as type **shared integer**. In the case of a structure which is allocated statically, the compiler can easily determine that cache coherence code is required. Equally, in the case of a dynamically allocated structure, the *type* of the object allows the compiler to determine that cache coherence code is required. The knowledge of when to **insert** a request for coherence can be deduced from the

explicit synchronisation primitives which the programmer or compiler must insert in order to provide correct execution of the program. The sample code shown in Program 1 illustrates a critical region augmented with an instruction to the cache memory to enforce coherence on a structure.

```
        ~
        ~
lock(variable_name.lock)
Make_coherent(variable_name.data)
        ~
        ~
unlock(variable_name.lock)
        ~
        ~
```

Program 1 *Critical Region Psuedo Code*

The **unlock** and **lock** primitives may operate on a semaphore which controls the synchronisation of operations on variables. Cache coherence can be achieved by removing the shared data from the cache. Thus the **Make_coherent** instruction *selectively* purges the variable, structure or part structure from the cache. This solution relies on using a *write through* cache update policy for all data, however, later in the paper, after describing the selectively clearable memory we extend this scheme to handle a write back strategy for non-shared structures.

It should be noted that the program usually only requires part of a complex structure to be coherent. This can be done by naming not only the structure address, but a range of addresses which must be made coherent. This method only removes the structure or address range specified from the cache, therefore enforcing coherence on the minimal size. It is applicable to both large and small structures, and differs from the software approaches in that only the structure is removed from the cache, not the complete cache contents. This general approach has been suggested by Smith (1982) and Alexander et al. (1986) previously. The problem with conventional cache memories is that it is not possible to selectively clear a range of addresses without enormous overheads. The next section describes a new cache design which allows selective clearing.

4. A SELECTIVELY CLEARABLE CACHE MEMORY

The above solution requires a cache memory in which a selected range of addresses can be purged. In general this would require a complete or partial cache search, which would be too expensive. The solution described in this section allows a program to request coherence on a range of addresses which are then effectively invalidated from the cache, rather than purging the entire cache.

An effective way of purging a range of addresses from a cache is to invalidate all of the cache sets that the address range covers. It is necessary to clear an entire set because the data can appear in any element of the set. The disadvantage of this approach is that in addition to removing the desired structure from the cache it also removes any other variables (synonyms) which share the same set. Our simulation studies, presented later in the paper, show that such non selective clearing only causes a very small performance degradation.

Cache memories can be considered as three separate memories: the tag memory, the data memory and the valid bit memory. The tag memory is searched pseudo-associatively and is used to hold all of the keys for data in the data memory. The valid bit memory, whilst logically an extension of the tag memory is used to indicate which locations in the cache hold a valid tag and data set. The valid bit memory can be separated so that it can be cleared when the cache is purged. In this way it can be reset by special hardware. This arrangement is shown in Figure 2. Thus, in order to invalidate a set in the cache it is only necessary to clear the appropriate valid bit. It is usually not possible to wait a long time for the valid bits to be reset when the cache is purged. Consequently, two fast clearing techniques have traditionally been used, shadow clearing and more recently directly clearable memories.

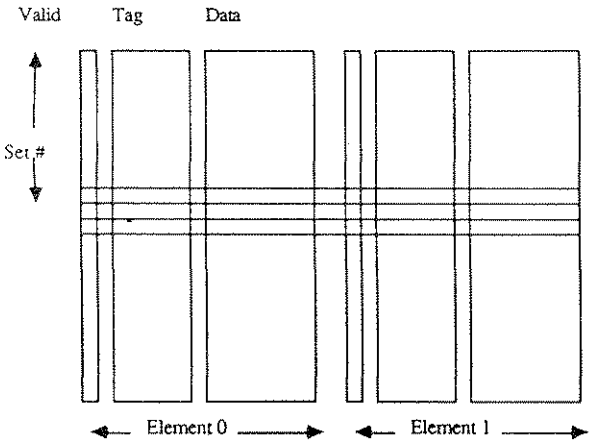

Figure 2. Cache Memory Organisation.

In shadow clearing the processor holds two sets of valid bits, either of which can be connected to the tag match logic. When the cache must be purged a clean set is switched in and the current dirty set is switched out. Before the next cache purge the dirty set is cleared by sequentially scanning the memory. It may then be switched in again when required. Providing the time between cache purges is larger than the time taken to clear the valid bit memory the processor never has to wait for a cache purge. This scheme has been used successfully in the Amdahl 470V/7 for clearing the translation lookaside buffer (Smith, 1982). This solution, however, is inappropriate for the proposed coherence algorithm because a processor only declares its need for coherence on a structure before the structure is accessed. Thus the valid bit memory (or a selected range of locations) cannot be cleared without a delay.

The second technique relies on special random access memory devices which provides a clear signal. When the clear signal is asserted each cell of the memory is simultaneously reset. Such devices are commercially available from Advanced Micro Devices and others. Figure 3 shows the possible internal layout of such a memory device. The

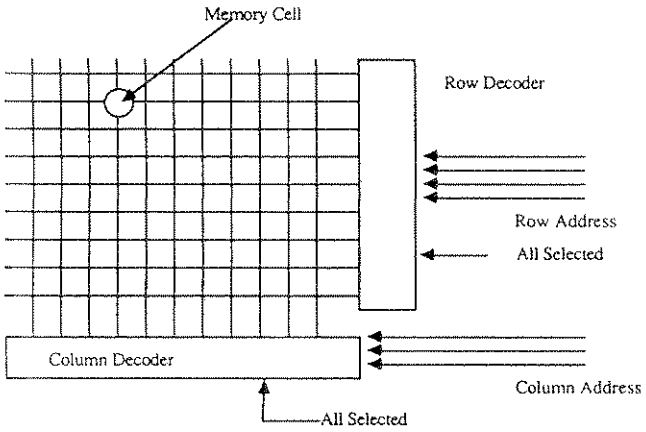

Figure 3. A Clearable Memory.

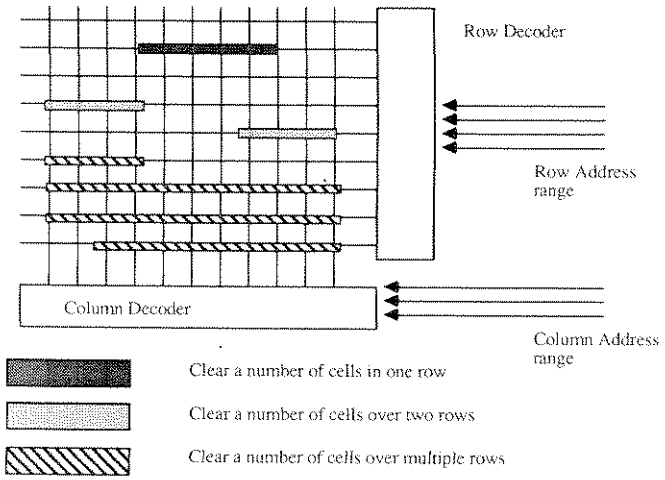

Figure 4. Clearing Selected Rows and Columns.

storage cells are held in an array which is addressed by a row select signal and a column select signal. When an individual bit is accessed only one row and one column signal are asserted. When all row and column signals are asserted all locations are referenced, making it possible to clear all cells in one cycle.

The proposed solution requires a selectively clearable valid bit memory in which only a portion of the memory cells are purged when the clear signal is asserted. This type of device can be implemented as a modification of the clearable memory, by selectively controlling which row and column enable signals to the array are enabled. The normal row and column decoders are replaced with ones which decode the values between a range of addresses. Such decoders can be constructed from priority decoders and some simple combinational logic. There are four different useful cases:

1. When only one row and one column are selected.
2. When one row and multiple columns are selected.
3. When multiple rows and multiple columns are selected.
4. When all rows and all columns are selected.

Case 1 corresponds to a normal memory write operation and can be used for resetting one valid bit (i.e. one set). Case 2 allows a number of bits to be reset, thus it is possible to clear a range of bits from a starting address to an end address within one row of the memory. Case 3 allows a number of bits to be reset when the start address and end addresses span more than one row and Case 4 clears the entire memory.

Given these operations it is possible to clear a number of bits in one cycle if the address range is contained in one row, two cycles if the range spans two rows and three cycles if the range spans many rows. The entire memory can be cleared in one cycle. These are shown in Figure 4 below.

In order to implement this solution a new chip is required. However, it is also possible to implement the valid bit memory from more than one of the commercially available clearable memories. The choice of how many memory chips are used determines the granularity of selective clearing. This arrangement has been used in the MONADS-PC processor to allow expansion to a multiprocessor network (Rosenberg and Abramson, 1985). In this machine eight valid bit memories were provided allowing up to one eighth of the cache to be cleared. It is not clear whether finer granularity is required until experimental results have been obtained.

5. OPTIMISATIONS

5.1 Prefetching the Critical Region

The performance of the selective cache clearance scheme can be enhanced by the use of prefetching. Since the selective cache clearance method relies on software directives, a priori knowledge of shared data references can be used for prefetching this shared data. This prefetching would then enforce coherence. The prefetch can be issued without a preceding cache clear operation, however, subsequent references need to wait for the prefetch operation to complete. If however a preceding clear operation is performed before the prefetch then the processor can continue executing code without waiting.

The introduction of prefetching into the selective cache clearance scheme gives rise to a number of subtle points. First, while it is shown that prefetching does lower the miss rate (see section 7) the actual programs may not run any faster. Consider the example shown in Program 2, where the first access (*Access1*) to the coherent data is imme-

```
        ~
        ~
lock(Variable.lock)
Make_coherent(variable_name.data)
Prefetch(variable_name.data)
Access1 variable_name.data
        ~
        ~
Access2 variable_name.data
unlock(Variable.lock)
        ~
        ~
```

Program 2 *Critical Region with Coherence and prefetching*

diately after the prefetch. Due to the latency of the prefetch the actual speed of execution will not be improved. If however *Access2* had been the only access to the data then the prefetch latency may not have been significant. Second, consider the case where *Access2* is the only access to the data in the critical region. It is possible in such a case to increase the bus traffic by the following sequence of memory references:

1. Prefetch data {*Memory Access 1*}.
2. A synonym is accessed and prefetched data is displaced {*Memory Access 2*}.
3. The data is accessed {*Memory Access 3*}.

In this case *Memory Access 1* and *Memory Access 3* are identical, with *Memory Access 1* being redundant and hence the memory traffic has increased. The likelihood of a synonym displacing the prefetched data of course decreases as the associativity of the cache is increased. The results of the prefetching simulations are presented in section 7 and show an increased hit rate.

5.2 Improving the Selectivity of the Clear Operation

When the associativity of the cache exceeds one (i.e. set association) the clear operation on any datum will not only remove that datum from the cache but the entire set. This is more information than necessary in order to enforce coherence. As the degree of associativity increases, the number of synonyms held in each set increases, and hence the expected miss rate increases. In fact, a larger degree of associativity causes the clear operation to become less selective. Optimal selectivity is obtained for a direct mapped cache. In contrast, increasing the associativity in a conventional cache memory decreases the miss rate. This contrary behaviour is only evident when coherence is required on the cache.

The problem can be avoided by improving the selectivity of the clear operation. This can be done by constructing a set associative cache which behaves like a direct mapped cache for shared data but a set associative cache for non-shared data. Thus, all accesses except those in shared data are treated as normal references to a set associative cache. Write accesses to the cache for shared data (either from a write or a read miss which causes a cache write) are treated as references to a direct mapped cache by using extra bits from the address to map the shared data to a specific element in the set. In a normal set associative cache the data can be loaded in any element of the set, and is usually selected by a random or least recently used (LRU) policy. The advantage of the combined cache is that it is not necessary to clear the entire set, as the placement of any shared data within the set is deduced directly from the address. The major disadvantage of this cache is that the compiler must emit special load and store operations on shared data. However, the compiler has sufficient information available to determine which type of instructions to use. In section 7 we illustrate the performance of this cache design.

5.3 Write-back Versus Write-through

A potential disadvantage of clearing entries in a cache to resolve coherence is that it does not allow a write back memory update policy. In a write back cache a modified cache cell must first be written to memory before the cell can be reused, thus it is not possible to simply reset the valid bit of the cell. A solution to this problem involves conditional clearing of the valid bits of the selected elements. If a cell has been modified then the clear operation is over-ridden. To enforce coherence it is necessary to use write-through for shared structures, but all other data may be write-back. Because shared data in the cache can never have the dirty bit set (it is always written through) the old data will always be invalidated by the clear operation. Data which is not invalidated because it has been modified will be replaced by the shared data when the cache is loaded and thus will be written back to memory correctly. It is possible to use a write-back policy for both shared and private data, but then a flush must be performed on the shared data when the unlock primitive is issued. In either case a flush must be issued for all modified cells when a process is migrated from a particular processing element to a different processing element.

In order to implement a memory which can perform the conditional clearing of valid bits, the line-dirty status bit must be held in the same device as the valid bit. If the dirty bit is set, then the clear is inhibited on the corresponding valid bit. Such a memory is not much more complex than the type described in section 4 and requires an extra row or column enable signal to differentiate between the valid bit and the dirty bit.

5.4 Process Migration

When a process is removed from a processor the cache must be cleared so that stale data cannot affect the process if it returns to the same processor. This cache purge should not degrade performance significantly because a new process usually requires different data in the cache. Thus, all processes experience a cold-cache when they are started on a new processor. Because the clear operation can be done very quickly there is no waiting period before a new process can be started. This scheme only works for a write through cache protocol.

The previous section has shown that a write back strategy can be used for all non-shared data, which should improve the performance significantly. Unfortunately, the use of write-back demands that the dirty cache locations are flushed when the process is removed from a processor before the cache is cleared. Such flush operations may cause significant delays between unloading a process and starting the next one. One solution to this problem is to overlap the flush operation with the startup of the next process. The overlap can be implemented by a separate *flush* machine which copies out all dirty lines and resets the dirty bit for the line. The data is left valid. Thus, when a process is removed from a processor the *flush* machine is started. All non-dirty lines in the cache are then cleared using the mechanism described in the previous section. This action guarantees that when a process returns to a processor there is no stale data in the cache. It should be noted that the remaining dirty lines cannot affect the new process because only non-shared data can be dirty, thus all dirty lines constitute private data of a previous process.

Should the new process wish to reuse one of these dirty lines, then a normal cache write back operation will occur. If the *flush* machine has already written the line back to memory then the new process will simply overwrite the line. Since hardware must already be present for flushing a line from the cache, the only additional hardware required is a counter and a small state machine for co-ordinating the activities.

Using this technique a processor is not left idle whilst the cache is cleared. It is necessary to synchronise this flush operation so that the process being removed is not started on another processor before the flush has been completed. The solution may require some additional cache bandwidth so that the cache can support both the processor traffic as well as the flushing traffic, although it is unlikely that any processor uses all of the available cache bandwidth.

5.5 Semaphores

The selectively clearable memory relies on the use of sempahores for locking sections of shared data. Semaphores themselves are shared variables and thus must be consistent at all times. It would be possible to apply a selective clear operation on a semaphore before its value is read, in order to enforce coherence. However, this operation also removes synonyms from the cache. Further, a read-modify-write cycle is still required at the memory so that the sempahore update is indivisible. Alternatively, it may be simpler to stop semaphores from migrating into the cache by marking them as non-cacheable. This decision is unlikely to affect the system performance because semaphrores are only accessed infrequently.

6. SIMULATION RESULTS

The clear operation used in the coherence algorithm not only removes the shared structure from the cache but all synonyms as well. A number of simulations were performed using program address traces to determine whether this would have an adverse effect on the behaviour of the cache. Three software based coherence schemes were simulated on the address traces. These were the total cache clear as suggested by Veidenbaum (1986), the set clear as described in section 3 and the improved set clear scheme as described in section 5.2. We call the final scheme the direct map clear scheme. This scheme is also simulated with prefetch. These simulations provide a relative performance measure between our suggested strategies. In addition to these simulations we also present simulations of a standard cache with no flushing, on the memory reference strings as a base comparison point.

Since the authors did not have access to programs which manipulate shared structures between multiple processes the data was taken from some conventional programs, namely the UNIX text formatter NROFF, the UNIX C compiler and a Prolog interpreter (NU-PROLOG). The trace data is summarised in Table 1. The memory reference strings were collected on a VAX 750 computer using a program simulator. This simulator collects all memory references made by a program in the VAX user address space (i.e. P0 and P1 region), but does not collect references made in the S0 region by the Operating System. The effect of clearing a shared structure from the cache was created by randomly choosing a block of the cache and clearing all of the sets in the range of addresses. This technique only indicates the effect of removing synonyms from the cache. It does not model the effect of cache invalidations due to multiprocess activity. However, it does indicate whether removing a section of the cache periodically will affect the overall cache performance due to loss of synonyms.

Table 1. Programs used in simulations.

Program	Code Size	Data Size Static + Dynamic	# million references
NROFF	40960	15360 + 586244	17.1
CC	91136	215044 + 205892	7.4
PROLOG	88064	27648 + 1968956	19.2

The graphs presented show the effect of clearing a fixed block at regular intervals from the cache. The notation used in the legends on the graphs is as follows:

ss *n* **ns** *m* [tcl | dcl *b* | scl *b* | std] [+pf *b*]

where *n* is the number of elements per set
m is the number of sets
b is the number of bytes cleared/prefetched
tcl indicates a total cache clear
scl is a set clear as described in section 3
dcl is a direct map clear as described in section 5.2
std is a standard cache with no flushing
+pf is a prefetch as described in section 6.1

For example, a cache with four element sets, 256 sets, cleared with the direct clear protocol clearing 80 bytes would appear as:

ss4 ns256 dc180

Figures 5a, 5b and 5c show for each trace a 16 kbyte cache being totally cleared. The results indicate that such a scheme does not become effective until the clearing interval is very large.

Figures 6a through 6f show cache performance for the three program traces with a 16 kbyte cache for dcl and scl simulating an 80 byte shared structure. They indicate that a selectively clearable cache memory would provide acceptable performance even when cleared frequently. As discussed in section 5.2 these figures illustrate that a high degree of associativity (large set size) can have poor performance because the granularity of the clear is degraded. This anomaly is apparent in Figures 6a, 6c and 6e when the frequency of clearing is high. Figure 6b, 6d and 6f illustrate that dcl eliminates this problem. Furthermore, the miss rates for dcl are lower than the corresponding scl miss rates. The simulations all use a line size of eight bytes. In varying the line size we found that the behaviour of a flushed cache was the same as that of a normal cache except that the standard cache had a slightly lower miss rate.This is shown in Figures 7a, 7b and 7c. In general, a larger line size will improve the selectivity of the clear. However, for line sizes much larger than the shared structure size the selectivity may decrease, thus increasing the miss rate.

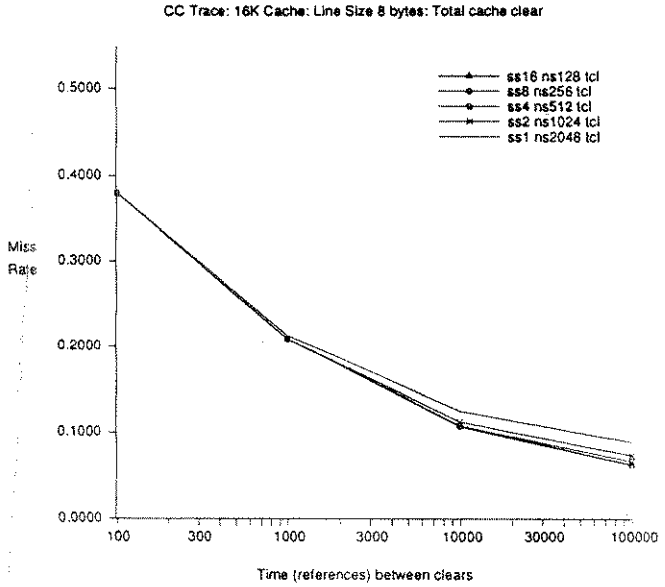

Figure 5a.

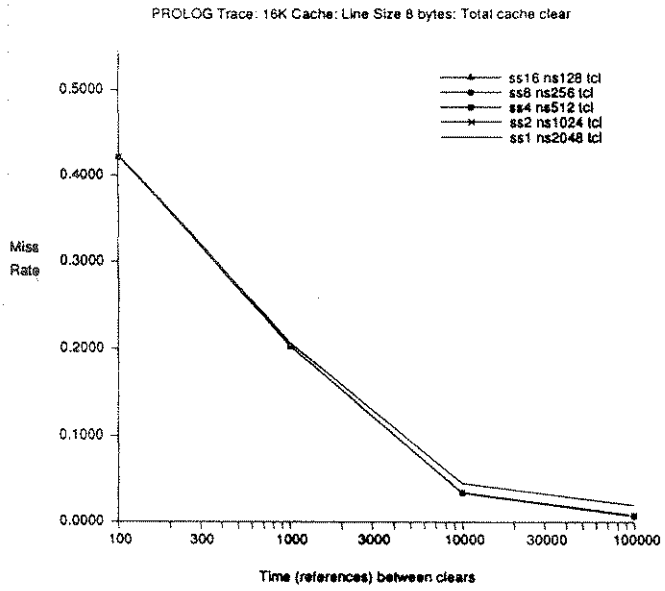

Figure 5b.

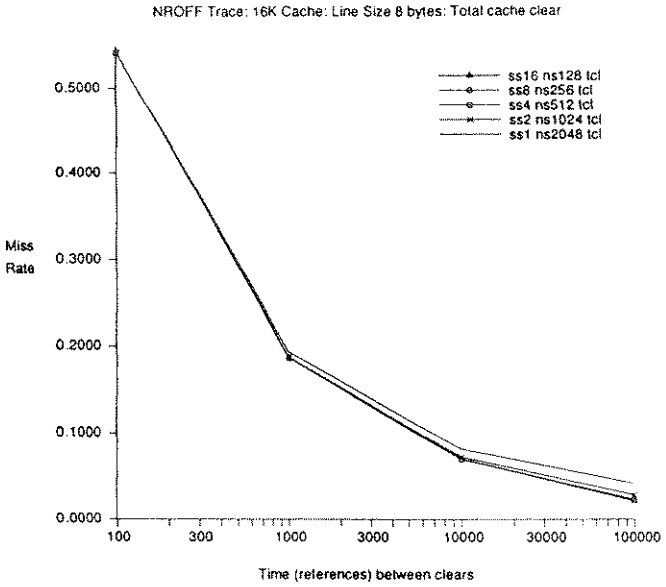

Figure 5c.

Figures 8a, 8b and 8c indicate upper and lower bound miss rates when the block size is varied. The top curve shows a total cache clear and the bottom curve shows an 80 byte clear. Miss rates for other block sizes will lie in between the two curves as illustrated by the 1k block size.

Figure 9 shows the effect of prefetching shared structures as described in section 6.1. As indicated, prefetching does lower the miss rate. However, the simulations do not show some of the other effects of prefetch, such as a possible increase in bus traffic due to prefetching more data than required. Also, the program may not actually run faster due to latency introduced by the prefetch.

The simulation results discussed so far choose a block of memory randomly. This technique does not model any *temporal* locality because it assumes that the addresses being cleared are independent of those currently being accessed. In order to account for some temporal locality the simulations were modified to use the current trace address as the block address rather than choosing a random address. This should model data accesses more closely because the access patterns which arise from sequential execution of code should match any sequential accesses to data. These simulations confirm that even with frequent clearing the cache performance is still acceptable. The two curves converge for less frequent clearing rates. These results are shown in Figure 10. It should be noted that regardless of the cache coherence algorithm chosen the cache miss rate will rise when many processors are actively modifying a shared region. This behaviour has been illustrated in Dubois (1987).

7. AN ALTERNATIVE INVALIDATION SCHEME

A different proposal was suggested by Smith (1985). This scheme uses a 'one-time-identifier' in combination with software advice to achieve cache coherence. When a page translation entry is entered into a translation lookaside buffer (TLB) for a particular processor an additional field associated with the translation entry is filled with a unique bit string, called a one-time identifier. When a line of the data cache is filled the one time identifier is copied to another field in the cache line. If a cache hit is recorded the one-time identifier in the cache is compared with the identifier in the lookaside buffer. The hit is ignored if they are different and the access proceeds to main memory. This scheme has the effect of removing all entries belonging to a page from the cache.

There are a number of problems with the one-time identifiers. First, this scheme cannot use the optimisations we have suggested for a write back strategy because it is not possible to clear only the clean data from the cache.

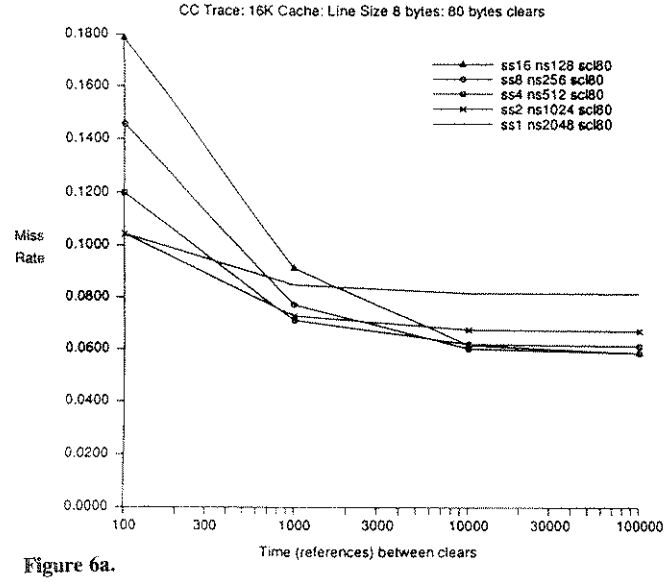

Figure 6a.

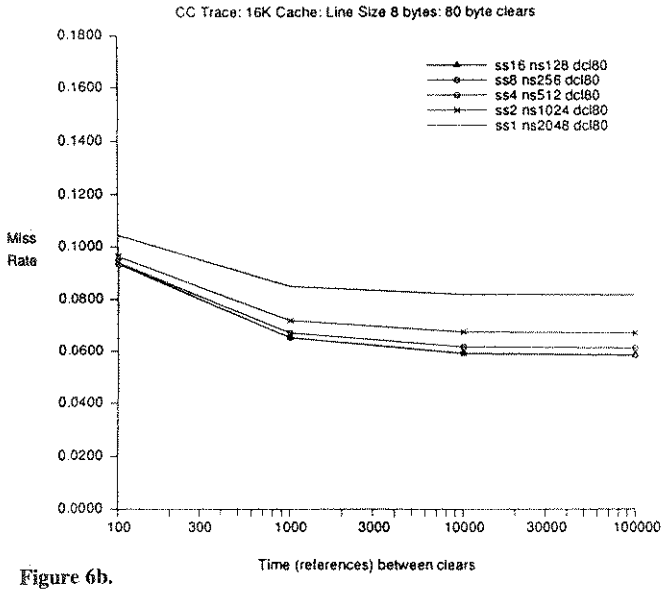

Figure 6b.

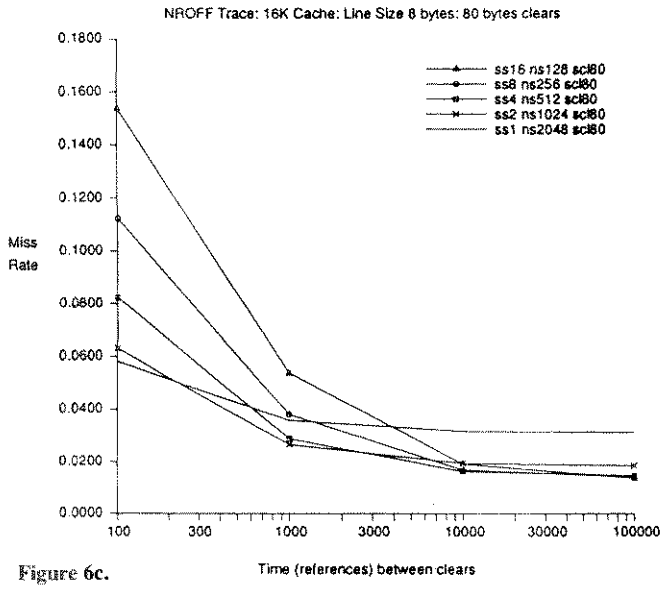

Figure 6c.

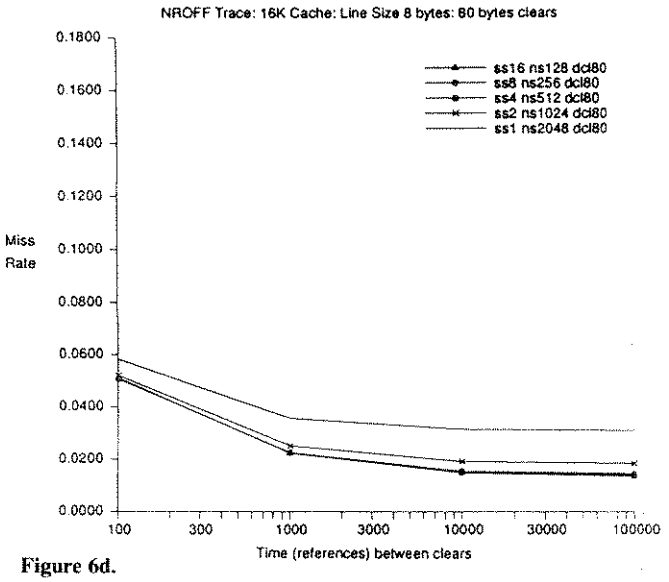

Figure 6d.

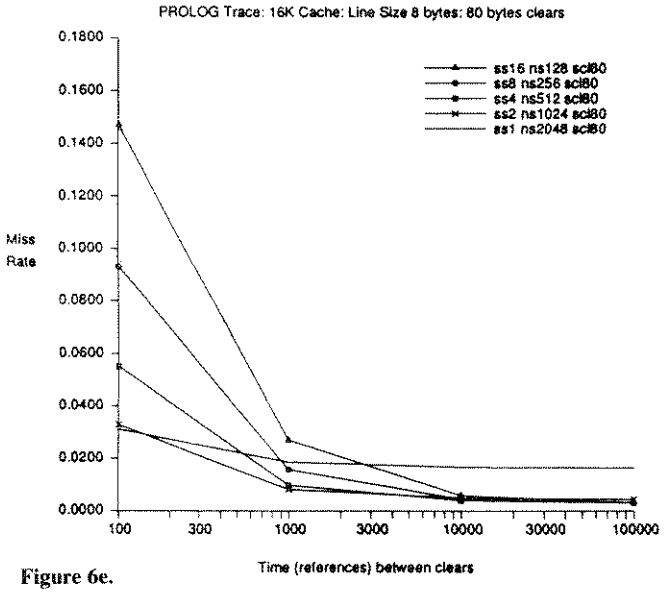

Figure 6e.

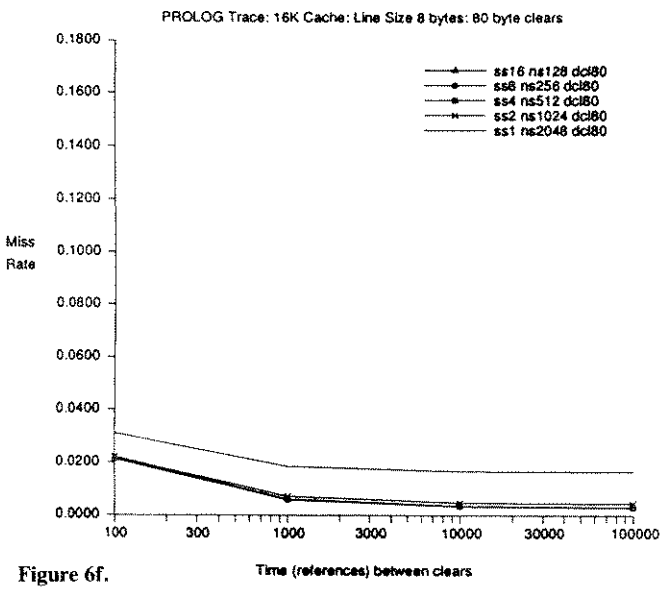

Figure 6f.

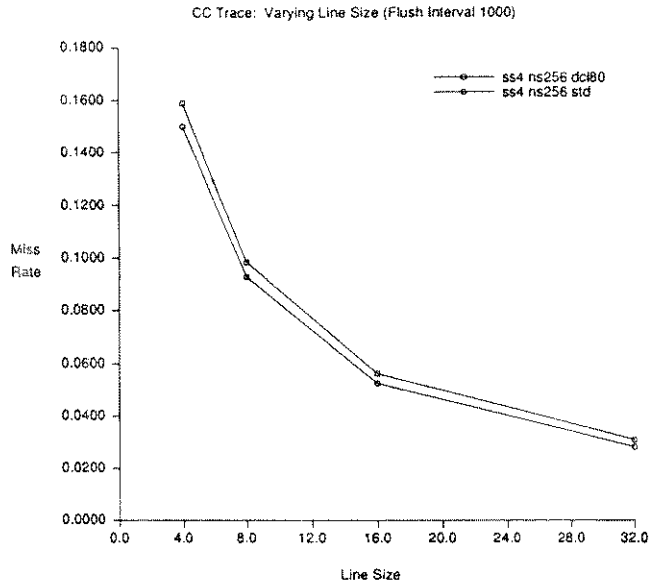

Figure 7a.

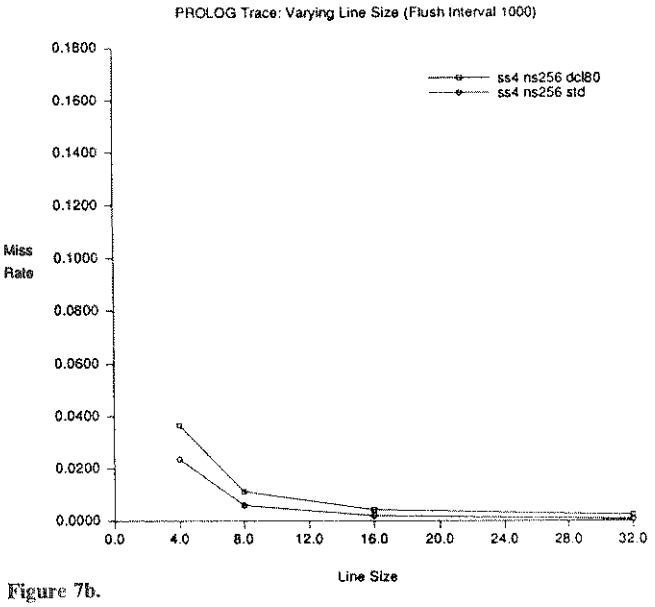

Figure 7b.

Thus it is necessary to wait for the flush operation to complete before the cache can be cleared. This flush delays process migration so severely that a write through strategy would almost certainly be required. Second, an identifier must be associated with each virtual page which increases the page table size. Third, the TLB width must be increased to accommodate the one-time identifier. Fourth, the scheme requires the cache to be physically close to the TLB because the one-time identifier is required for each cache access. This is not the case when virtual addresses are passed to the memory modules for translation as is done in Pose et al. (1986). It is possible to add hardware similar to a conventional TLB which only holds the one-time identifiers for associated virtual pages. However, this solution increases hardware cost. Finally, the one-time identifier must be large enough to allow many selective invalidations. When the one-time identifiers are exhausted

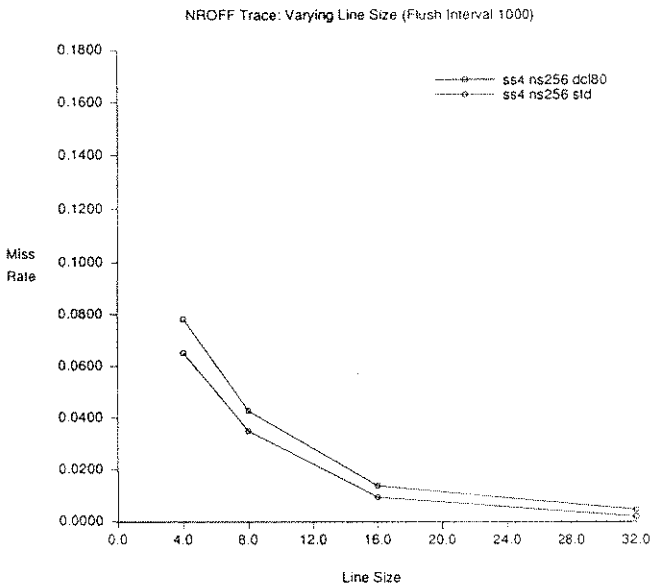

Figure 7c.

the cache contents must be flushed back to memory. In spite of these objections, this scheme should be seen as an alternative method for enforcing cache coherence through software control.

8. CONCLUSION

In this paper we have presented a solution to the cache coherence problem for write through caches which requires minimal extra hardware. It relies on a slightly unusual implementation of a conventional cache valid bit memory, which would be incorporated in the memory device itself. The scheme is scalable to a very large number of processors because it makes no assumptions about the system size, memory bandwidth or interconnection structure. The method relies on a selectively clearable cache using a special memory device for the tag valid bits. The method does not introduce any extra delays over conventional cache designs. The additional hardware required is minimal, and can be incorporated easily into an existing computer system. Simulation results indicate that the performance of the cache is not degraded significantly when coherence is maintained. Some optimisations have been presented which further improve the performance.

The optimisations are quite simple to implement. The simulations would indicate that the direct map clear solution proposed in section 5.2 (dcl) provides the best performance. This scheme improves the selectivity of the cache clear significantly when a set associative memory is used. If a write back policy is required then the optimisations suggested in section 5.3 should be applied.

9. ACKNOWLEDGEMENTS

Much of this work has been performed under the CSIRO/RMIT Parallel Systems Architecture Project. The project is being supported jointly by the Royal Melbourne Institute of Technology and the CSIRO, Division of Information Technology. The authors wish to acknowledge the referees, who have assisted in increasing the clarity of the paper.

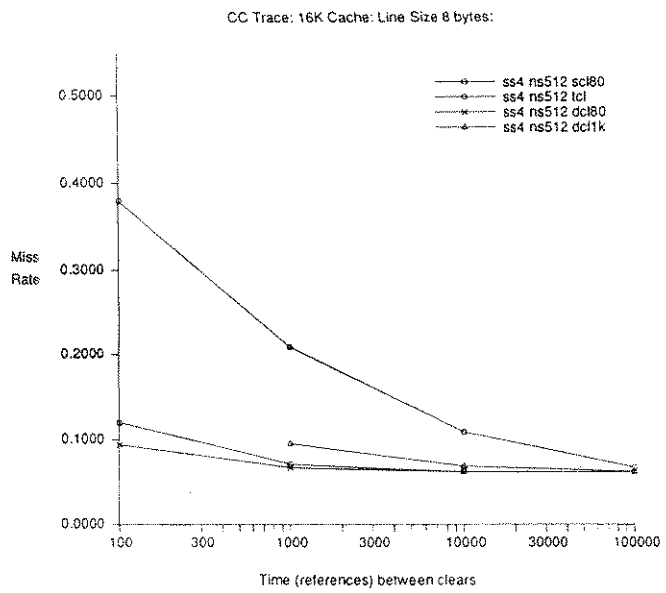

Figure 8a.

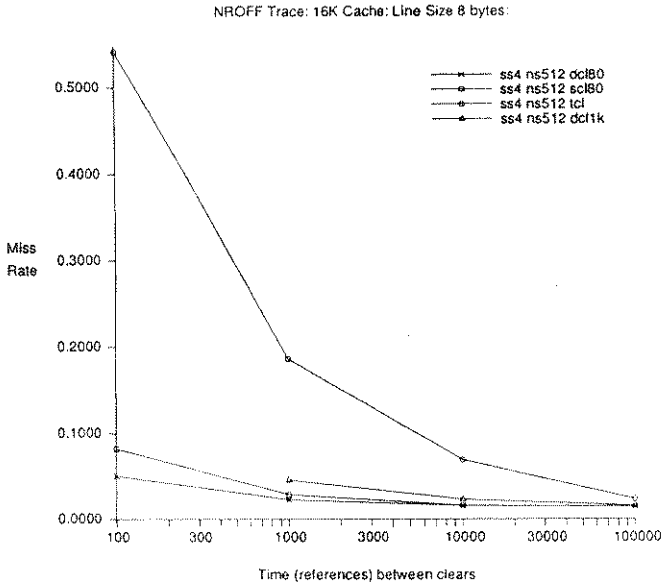

Figure 8b.

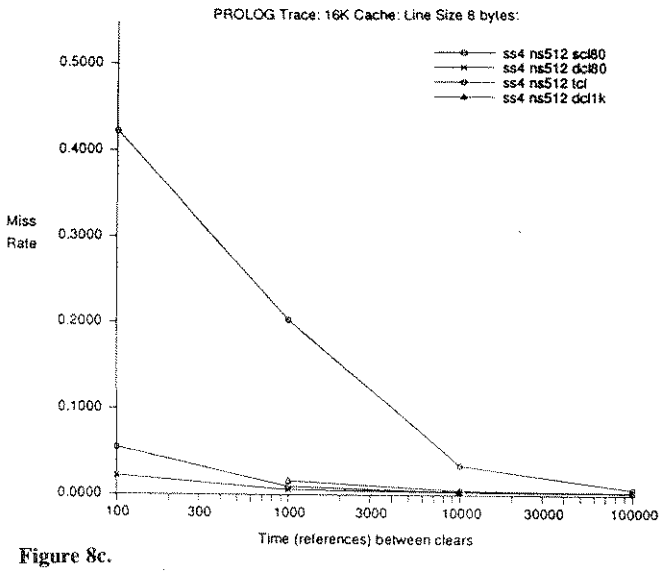

Figure 8c.

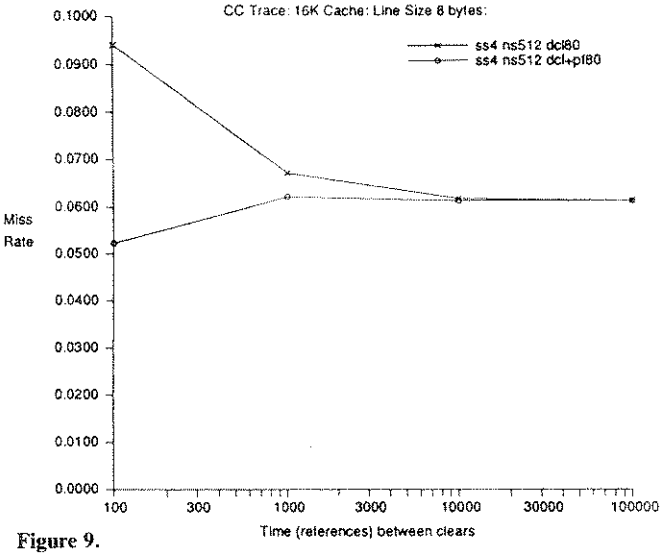

Figure 9.

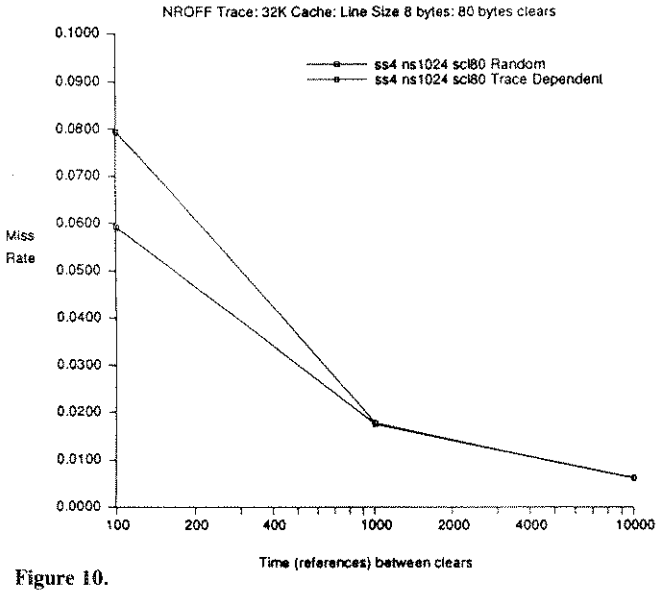

Figure 10.

10. REFERENCES

ADVANCED MICRO DEVICES, *Am 9150 High Speed Memory,* AMD Data Sheet.

ALEXANDER, C., KESHLEAR, W., COOPER, F. and BRIGGS, F. *Cache Memory performance in a UNIX environment,* Computer Architecture News, V 14, N 3, June 1986, pp. 41-70.

ARCHIBALD, J. and BAER, J.-L. (1984): *An economical solution to the cache coherence problem,* Proceedings of the Eleventh Annual International Symposium on Computer Architecture, Ann Arbor, Michigan, June, pp. 355-362.

CENSIER, L and FEAUTRIER, P. (1978): *A new solution to coherence problems in multicache systems,* IEEE Transactions on Computers,- December, pp. 1112-1118.

DUBOIS, M. and BRIGGS, F.A. (1982): *Effects of cache coherency in multiprocessors,* IEEE Transactions on Computers, November, pp. 1083-1099.

DUBOIS, M. (1987): *Effect of Invalidations on the Hit Ratio of Cache-Based Multiprocessors,* Proceedings of the 1987 International Conference on Parallel Processing, August, pp. 255-257.

MOTOROLA, MC68030 User's Manual.
NU-PROLOG REFERENCE MANUAL TR 86/10, Department of Computer Science, University of Melbourne, Australia.
PFISTER, G. et al. (1985): *The IBM Research Parallel Prototype (RP3): Introduction and architecture*, Proceedings of the 1985 International Conference on Parallel Processing, IEEE Cat. No. 85CH2140-2, August, pp. 764-771.
POSE, R., ANDERSON, M. and WALLACE, C.S. (1986): *Aspects of a Multiprocessor Architecture*, Proceedings of the Workshop on Future Directions in Computer Architecture and Software, May 5-7.
ROSENBERG, J. and ABRAMSON, D. (1985): *MONADS-PC — A Capability-Based Workstation to Support Software Engineering*, Proceedings of the Eighteenth Annual Hawaii International Conference on System Sciences, pp. 222-231.
SCHEURICH, C. and DUBOIS, M. (1987): *Correct memory operation of cache-based multiprocessors*, Proceedings of the Fourteenth Annual International Symposium on Computer Architecture, Pittsburgh, Pennsylvania, June, pp. 234-243.
SMITH, A.J. (1982): *Cache Memories*, ACM Computing Surveys, V 14, N 3, September.
SMITH, A.J. (1985): *Cache Consistency with Software Support and Using One-time Identifiers*, Proceedings of Pacific Comp. Commun. Symp., October.
STONE, H.S. (1987): *High-performance Computer Architecture*, Addison Weseley Publishing Company, pp. 328.
VIEDENBAUM, A. (1986): *A Compiler-assisted cache coherence solution for multiprocessors*, Proceedings of 1986 International Conference on Parallel Processing, August, pp. 1029-1036.

BIOGRAPHICAL NOTES

Dr David Abramson is a Senior Research Scientist with the Division of Information Technology, CSIRO. His major research interests include computer architecture, computer hardware, parallel systems, parallel algorithms and operating systems. He is currently in charge of the CSIRO staff working on the Parallel Systems Architecture Project at RMIT. Prior to joining CSIRO in 1986 he was a lecturer at Monash University and was involved in the Monads project. He is a member of the IEEE and the ACM. Abramson received a BSc (Hons) in 1978, and a PhD in 1982 from Monash University.

Dr Kotigiri Ramamohanarao is a reader in the Department of Computer Science at the University of Melbourne. His research interests include computer architecture, Logic Programming and deductive databases. He is currently leading the machine intelligence project at Melbourne University, which is funded by the Australian Research Council. Ramamohanarao received a BE in 1972, ME in 1974 in India and a PhD from Monash Unversity in 1980.

Mark Ross is a senior lecturer in the Department of Computer Science at the Royal Melbourne Institute of Technology. His research interests include computer architecture, prolog and operating systems. Ross received a BSc(Hons) from the University of Melbourne and is completing a PhD in computer science at the University of Melbourne.