Cache Inclusion and Processor Sampling in Multiprocessor Simulations

Jacqueline Chame and Michel Dubois

CENG Technical Report 92-13

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213)740-4475

CACHE INCLUSION AND PROCESSOR SAMPLING IN MULTIPROCESSOR SIMULATIONS

Jacqueline Chame and Michel Dubois

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, CA 90089-2562
jchame@pollux.usc.edu
dubois@paris.usc.edu
(213) 740-9130

September 1992

CACHE INCLUSION AND PROCESSOR SAMPLING IN MULTIPROCESSOR SIMULATIONS

Abstract

The evaluation of cache-based systems demands careful simulations of entire benchmarks. Simulation efficiency is essential to realistic evaluations. For systems with large caches and large number of processors, simulation is often too slow to be practical. In particular, the optimized design of a cache for a multiprocessor is very complex with current techniques.

This paper addresses these problems. First we introduce necessary and sufficient conditions for cache inclusion in uniprocessors and in multiprocessors with and without invalidations. Second, under cache inclusion, we show that an accurate trace for a given processor or for a cluster of processors can be extracted from a multiprocessor trace. With this methodology, possible cache architectures for a processor or for a cluster of processors are evaluated independently of the rest of the system, resulting in a drastic reduction of the trace length and simulation complexity. Moreover, many important system-wide metrics can be estimated with good accuracy by extracting the traces of a set of randomly selected processors, an approach we call *processor sampling*. We demonstrate the accuracy and efficiency of these techniques by applying them to three 64-processor traces.

1. INTRODUCTION

As larger multiprocessor systems are built and as the speed of individual processors keeps increasing, the performance of private caches and associated cache coherence protocols becomes more and more critical. A detailed performance analysis of these very complex memory systems is key to the design of high performance multiprocessors.

Trace driven simulation is the most popular method for the performance analysis of multiprocessor memories. Traces of memory references from different processes are generated through simulations or measurements, merged into a single string of references and then used in the simulations of many different systems. Trace-driven simulations of multiprocessor systems have some validity problems [3] [12]. Program-driven simulations may be more accurate [5] but are often more compute-intensive and less efficient. Short of building the target machine, trace-driven simulations are generally considered accurate enough to yield useful measures of parallel program performance.

The running time and memory requirement of trace-driven simulations are prohibitive even for moderate system sizes. When the number of processor increases, the benchmark data set must be scaled up, and the trace length grows to the point where the simulation approach is impractical or even unfeasible. Even for systems with 16 processors, the caches must be made artificially small to keep simulation times reasonable.

There are techniques to reduce the time and memory requirements of trace driven simulations; many rely on *cache inclusion,* originally introduced by Mattson *et al* [14]. Cache inclusion relates two caches such that the content of one is always a subset of the content of the other. Mattson showed that cache inclusion holds for uniprocessor caches in many cases provided the replacement policy is a *stack algorithm*. *Stack simulation* is possible under cache inclusion. Stack simulation is a very efficient cache simulation technique in which the miss rates for several caches with different associativities are computed in a single pass through the trace. *Trace reduction* techniques include *trace stripping* and *set sampling*. Under cache inclusion, references that hit in a small filter cache can be *stripped* from a trace. Set-sampling consists of measuring the miss rate in a few randomly selected sets instead of the whole cache. In contrast to trace stripping, which yields accurate miss rates, set sampling yields statistical estimates of miss rates [16].

In this paper we focus on trace reduction for efficient simulation and storage of multiprocessor traces. First we introduce new necessary and sufficient conditions for cache inclusion in uniprocessor and multiprocessor systems with and without invalidations. Based on these conditions, we then show how to generate a trace of references for a group of processors and how to use it to simulate the activity of the processors in isolation. Applications of this technique are the efficient designs of processor nodes (especially their cache) and processor clusters [9] with a simulation efficiency independent of the number of processors. Finally, we show how to estimate system-wide metrics such as number of misses, number of write backs and number of cache coherence events, by measuring them in a set of randomly selected processors. We call this approach *processor sampling* because the idea is very similar to set sampling. Processor sampling yields reliable estimates in the case of parallel applications following the SPMD (Single Program Multiple Data) programming paradigm[1], because of their symmetry and homogeneity. We have not tried processor sampling in the case of heterogeneous computations in which every processor executes a different program.

Current trace reduction techniques are reviewed in Section 2. In Section 3 we introduce a new set of necessary and sufficient conditions for inclusion in cache-based systems; these conditions are needed to understand when and how to extract the trace for a group of processors, a technique presented in Section 4. By selecting the group of processors at random, system-wide metrics can be estimated; we explain this approach and apply it to three traces in Sections 5 and 6. Finally, we conclude in Section 7.

2. TRACE REDUCTION

Trace stripping and set sampling were first proposed by Puzak [15]. The idea behind trace stripping comes from the observation that all references which hit in a direct-mapped cache (called the *filter cache*) also hit in caches with the same number of sets and larger associativities and do not affect their total miss count; therefore they can be removed from their trace. Puzak also showed that the stripped trace produces the same number of misses in the simulation of caches with a larger number of sets than the filter cache and for set mapping functions defined by bit selection. The reduction factor depends on the miss rate of the filter cache: for example, a miss rate of 10% in the filter cache yields a reduction factor of 10. The reliability of the miss rate esti-

1. This paradigm leads to homogeneous computations in which each processor executes the same code on its share of the data. Arguably, because of the complexity of debugging hundreds of heterogeneous cooperating processes, SPMD is the only practical approach for the programming of large multiprocessor systems.

mates obtained by randomly sampling several sets increases with the number of sets; in practice choosing one tenth of all sets gives acceptable confidence intervals [16] and cut the trace length by an additional factor of 10.

Trace stripping was originally aimed at miss rates only. Wang [18] [19], following Thompson's work [17], extended the technique to generate exact miss and write back counts. In his method, he keeps not only the references that miss in the filter cache but also all writes on clean blocks. He also shows how to generate a multiblock trace, by collecting the superset of misses and writes on clean blocks in simulations of filter caches with different block sizes. Wang's approach is applicable to multiprocessor traces as well and yields accurate counts of misses, write backs and invalidations.

The above reduction techniques are still not sufficient to enable simulations of large-scale multiprocessors. Moreover, it is almost unthinkable with current techniques to explore systematically the possible designs of a processor cache in a multiprocessor configuration as was done for uniprocessors [22]. To make this possible we must extract a valid trace of references affecting a processor or group of processors.

Extracting the trace of a given processor from a multiprocessor trace is not a new idea. Lee [13] took this approach to evaluate multiprocessor cache designs for the case where shared writable data are non-cacheable under compiler control. He first derived the trace of one processor (without reducing it), and then, assuming a fixed miss latency, he used it to obtain the processor execution time for caches with different sizes and different block sizes. Because coherence is solved by restricting cacheability, no reference by other processors affect the contents of the local cache and therefore references of other processors are not retained in the trace. More recently Gharachorloo *et al.* [11] also isolated the trace of a processor for a given cache to optimize memory access scheduling in the processor; the multiprocessor in their study maintains coherence through hardware and the cache architecture does not vary. In this paper we exploit similar ideas and extract a trace for a given processor or group of processors in a multiprocessor system with hardware cache coherence so that the trace is valid for caches with various organizations, capacities and block sizes. We also estimate system-wide metrics by sampling a set of processors.

It is important first to identify the conditions under which cache inclusion holds in order to understand when and how the extraction of an accurate trace is possible.

3. CACHE INCLUSION

In general, cache C_2 includes cache C_1 if, after any series of references, any block present and valid in cache C_1 is also present and valid in cache C_2. The conditions for cache inclusion in multiprocessor systems are different for write-broadcast and write-invalidate protocols. In a write broadcast protocol such as the Dragon or the Firefly protocols [2], the state of a cache is not affected by other processors' write accesses; consistency is maintained by broadcasting the value written to the caches sharing a copy of the block. Therefore the conditions for inclusion in multiprocessors with write-broadcast coherence protocols are the same as for uniprocessor caches. Independently of the protocol, cache invalidations are often needed by operating system activity in both uniprocessors and multiprocessors. In virtual-address caches, for example, invalidations are issued by the processor to deal with synonyms and other problems. In general, we must distinguish between systems with and without invalidations.

3.1. Systems with no invalidation

It is known since [14] that, in systems with no invalidation, inclusion holds for caches that have the same set-mapping function, have the same block size, do not prefetch and use a *stack replacement algorithm*. Any replacement algorithm inducing a total order on all previously referenced blocks and victimizing the block with the lowest priority is a stack algorithm. The total order is represented at any time t by the following priority list on all the blocks accessed by the processor up to time t:

$P_t = p_t[x_1], p_t[x_2],..., p_t[x_k]$, where $p_t[x_i]$ is the priority of block x_i at time t.

This priority list must be independent of the content of the cache. Useful and common stack replacement algorithms, such as LFU (Least Frequently Used), LRU (Least Recently Used), and Random fit the priority list model. A fixed priority algorithm in which the priority level of a given block is constant for the whole trace and each block has a different priority level is also a stack algorithm.

More recently it was shown that, in systems with no invalidation, inclusion also holds for caches that have the same block size, do not prefetch and use the LRU replacement algorithm; the set-mapping function is arbitrary provided that a larger cache refines[2] a smaller one [10].

In the following we prove a necessary and sufficient condition for cache inclusion under the following assumptions:

Assumption 1. Caches C_1 and C_2 are the caches of the same processor in two systems in which blocks cannot be invalidated. In the case of a multiprocessor, the coherence protocol is the same in both systems.

Assumption 2. The only difference between the two caches is the number of sets in the caches of each system. Otherwise, the caches have the same block size, have the same associativity N, do not prefetch, and the set-mapping function f_2 of C_2 refines the set-mapping function f_1 of C_1.

Assumption 3. The replacement algorithm is the same in all caches and generates a priority list at each time t defining a total order on all blocks referenced up to time t and independent of the caches' contents at time t.

Theorem 1 (Necessary and sufficient condition). Under assumptions 1, 2 and 3, cache C_2 includes cache C_1 iff, at every time t and for every set, the total order induced by the replacement algorithm is such that a full set always contains the N-1 highest priority blocks mapping into it.

Proof:

Sufficient condition: Assume that, at every time t, the total order induced by the replacement algorithm is such that the N-1 highest priority blocks mapping to any full set are always in cache. Suppose that set S of C_1 is refined by sets $S_1,..., S_m$ of C_2. Initially both caches are empty and inclusion holds. While no set of C_2 is full, inclusion holds. Inclusion could be violated for the first

2. Set-mapping function f_2 *refines* set-mapping function f_1 if $f_2(x) = f_2(y)$ implies $f_1(x) = f_1(y)$, for all blocks x and y. Cache C_2 is said to *refine* cache C_1 if $f2$ refines $f1$.

time when at least one of the sets S_i of C_2 is filled up. From that time on, if a block x is displaced from set S_i in C_2, is present in set S of C_1 but is not displaced from S, inclusion is violated. However, this cannot happen for the following reason. If, at time t, a valid block x is displaced from set S_i in C_2, x must have the lowest priority among the blocks resident in set S_i. Thus there must exist $N-1$ blocks resident in set S_i with higher priority than x. If block x is not displaced from set S at time t, there must exist at least one block y resident in set S such that $p_{t-1}[y] < p_{t-1}[x]$. Following the hypothesis, set S would have to contain at least block x, block y and $N-1$ blocks with higher priority than x, a total of $N+1$ blocks. This is not possible and therefore y could not be present in S in the first place and block x must have been replaced in S.

Necessary condition: We need to show that we could refine the sets of cache C_1 so that a block could be replaced in cache C_2 and still remain in C_1, violating inclusion, if the $N-1$ highest priority blocks mapping into a set of associativity N are not kept in cache at every time t. Assume that, at some time t, at least two blocks, x and y, are present in S and are not among the $N-1^{th}$ highest priority blocks mapping into set S. Without loss of generality, assume that at least $N+1$ different blocks have been referenced up to time t and that y has lower priority than x at time t. Since the priority list is independent of the caches' contents, the algorithm must guarantee inclusion for all possible set-refinements. Suppose a set mapping function that refines set S by sets $S_1, S_2, \ldots, S_m$ of C_2 such that

1. x and the $N-1$ highest priority blocks that map into set S map into S_1,

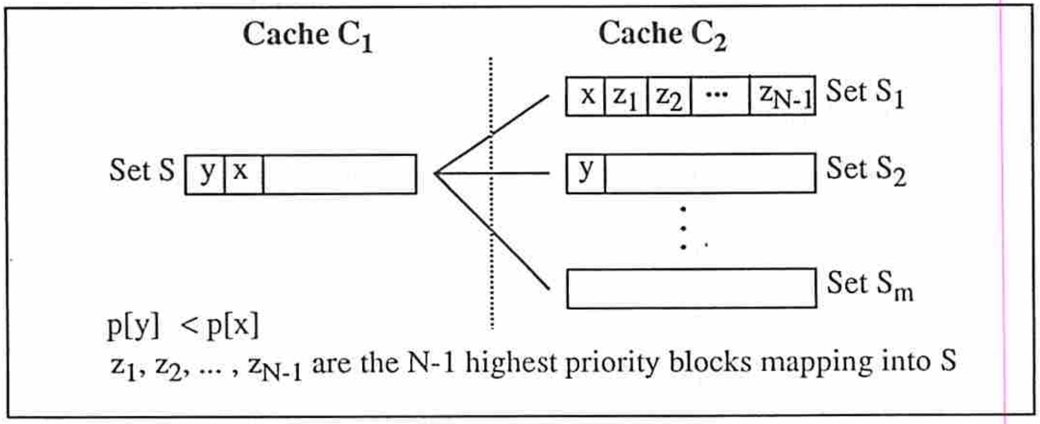
2. y maps into S_2, and

3. all other blocks map into S_j, $j \neq 1,2$.

Inclusion can then be violated if a miss occurs in S_1, causing x to be replaced in S_1 and y to be replaced in S. ❑

Figure 1 shows the situation leading to a violation of inclusion in Theorem 1.

Figure 1. Contents of C_1 and C_2 for Theorem 1


Cache C_1
Cache C_2
x | z_1 | z_2 | ... | z_{N-1}
Set S_1
Set S
y | x
y
Set S_2
Set S_m
$p[y] < p[x]$
$z_1, z_2, \ldots, z_{N-1}$ are the N-1 highest priority blocks mapping into S

The following is a sufficient condition for cache inclusion derived from Theorem 1.

Condition 1 (Sufficient condition). Under assumptions 1, 2 and 3, cache C_2 includes cache C_1 if no block can change priority level in the priority list from time t to time $t+1$, except for the referenced block at time t: if the block referenced at t was never referenced before t, then it can take any priority level; if the block was referenced before t then its priority level must either increase or remain constant at $t+1$.

Proof:

We only need to prove that the replacement algorithm is such that the $N-1$ highest priority blocks are always kept in every full set. The claim is true at time t_0 when a set S is filled for the first time, since at time t_0 the N blocks resident in S are the N blocks that have been referenced up to time t_0. Assume now that, at time t, a set contains the $N-1$ highest priority blocks mapping into it. With no loss of generality, we may assume that an access is made to a block mapping to S at time $t+1$. There are two possibilities.

1. It is the first reference to the block. Since the new block replaces the block in the set with the lowest priority, i.e., the one which was not among the $N-1$ highest priority blocks, the claim is satisfied, independently of the priority level of the new block.

2. The block was referenced before. Its priority either increases or remains the same. If the block was already resident in the set, the set still contains the $N-1$ highest priority blocks after the reference. If the block was not present, the argument is the same as for case 1. ❑

LRU, LFU (Least Frequently Used) and any other stack algorithms such that a block is less likely to be replaced when it is referenced satisfy Condition 1. Another replacement algorithm satisfying Condition 1 is a *fixed-priority* replacement algorithm. Random replacement violates the conditions of both Theorem 1 and Condition 1.

3.1.1. Remark on Theorem 1

Cache inclusion holds for caches with same set-mapping function, increasing associativities and stack replacement algorithms [14]. Because inclusion is a transitive property, the condition of Theorem 1 is also sufficient for caches with same block size, no prefetching, set-refinement and increasing associativities. The condition of Theorem 1 is also necessary for caches with increasing associativities because the priority list imposed by the replacement algorithm must be independent of the contents of any cache.

3.2. Systems with invalidations

Stack replacement algorithms as defined by Mattson do not deal with invalid blocks. In order to avoid rederiving results already established for stack algorithms, we need to extend these algorithms and include invalidations.

3.2.1. Stack algorithms with invalidations

There are two significant differences between an invalid and a valid block. First, an invalid block is never accessed. Second, it is impossible to differentiate between two invalid blocks. For the purpose of cache inclusion, the only events relevant to a given cache are processor references and invalidations to its blocks. Therefore, for the purpose of cache inclusion, it is sufficient to abstract the behavior of the system with respect to a given cache by a trace in which each record $r(t)$ is either a reference or an invalidation.

In order to model a stack algorithm with invalidations we need to build a priority list for each reference in the trace. The two requirements are:

1. to associate a priority level to the memory block when it is referenced for the first time after its invalidation, and

2. to associate an entry in the priority list to an invalid block right after the invalidation.

Requirement 1 is invariably the same in any implementation of a replacement algorithm with invalidations. Right after an invalidation, there is no valid copy of the block in any set of any cache that could be compared. Since realistic replacement algorithms are such that statistics are kept on blocks present in caches only, we must consider that the first access to a block after its invalidation is equivalent to an access to a new, never accessed block. Requirement 2 depends on the implementation of the algorithm: the priority level given to invalidated blocks must reflect the policy adopted by the replacement algorithm with respect to them. There are many possibilities and it would be impossible to cover all of them. In this paper, we restrict ourselves to two major classes of policies.

The first class of policies, which we call *oblivious* replacement policies, treats invalid blocks as if they were valid blocks. At the time when a block is invalidated, its priority level is unchanged. The replacement algorithm does not take advantage of the fact that the invalid block will never be referenced again to replace it sooner than a valid block. The rationale for oblivious replacement algorithms is that the hardware implementation of the algorithm is simpler since the block valid bit does not affect the way statistics are kept in each cache set. Of course, oblivious policies are not optimum with respect to the hit rate but they may be acceptable if invalidations are rare.

The second class of policies, which we call *non-oblivious* replacement policies, distinguishes between valid and invalid blocks in the sense that an invalid block is always victimized before any valid block in the set; if there are more than one invalid blocks in the set, one of them is selected arbitrarily; if there is no invalid block, one valid block is selected for replacement according to the policy in the absence of invalidations. Right after its invalidation, a block is inserted at the bottom of the priority list. In general, non oblivious policies are more complex to implement in hardware than oblivious ones but they may also yield better hit ratios.

3.2.2. Priority list model for oblivious replacement policies

Oblivious replacement policies treat invalid blocks in the same way as if they were valid. At every time t after the invalidation, an invalid block has the same priority as its corresponding valid block would have if it had not been invalidated and were never referenced again. Thus the priority list for an oblivious replacement policy is constructed as follows.

```
1. Start with the trace of references and invalidations $r(t)$
2. Rename every block with address $x$ in the trace by $x0$.
3. Initialize the set of referenced blocks to empty.
4. Scan the trace from start to finish:
    4.1. For every record $r(t)$ corresponding to a reference,
        - add the block to the set of referenced blocks if it is a first reference.
        - assign a priority to all blocks in the set of referenced blocks according to
```

the replacement algorithm in the absence of invalidation.

4.2. For every record *r(t)* corresponding to an invalidation, say the i^{th} invalidation for the block with address *x(i-1)* in the priority list and in the trace,
- rename all future references to *x(i-1)* by *xi*.
- remove the invalidation from the trace.

The renaming is necessary to distinguish between invalid copies of a block and the block itself. Copy *x(i-1)* is associated with the invalid copy generated by the i^{th} invalidation to block *x*. *xi* represents the valid block between the i^{th} invalidation to *x* and the $i+1^{th}$ invalidation to *x*.

Therefore an oblivious replacement policy is derived from a known replacement algorithm with no invalidations by renaming the blocks after each invalidation so that invalid blocks are treated like valid ones and by applying the replacement algorithm to the new trace.

Figure 2 illustrates the trace transformation procedure and the definition of the priority list at every time for oblivious LRU. References are represented by the block addresses *(a, b, ...)* and invalidations are represented by *Ix*, where *x* is the address of the block. In the priority list, invalid blocks are in boldface letters. The blocks are ordered by priority in the priority lists, with the highest priority block at the top.

Figure 2. Trace transformation for oblivious LRU

time	1	2	3	4	5	6	7	8	9	10
trace	a	b	c	Ib	b	c	a	Ia	d	a
renamed trace	a0	b0	c0		b1	c0	a0		d0	a1
priority list	a0	b0	c0	c0	b1	c0	a0	**a0**	d0	a1
		a0	b0	**b0**	c0	b1	c0	c0	**a0**	d0
			a0	a0	**b0**	**b0**	b1	b1	c0	**a0**
					a0	a0	**b0**	**b0**	b1	c0
									b0	b1
										b0

3.2.3. Priority list model for non oblivious replacement policies

Non oblivious replacement policies always victimize an invalid block before any valid block. Invalid blocks must have lower priority than any valid block in the cache and must be inserted at the bottom of the priority list. Thus the priority list for a non oblivious replacement policy is constructed as follows.

1. Start with the trace of references and invalidations *r(t)*
2. Rename every block with address *x* in the trace by *x0*.
3. Initialize the set of referenced blocks to *empty*.
4. Initialize the set of invalidated blocks to *empty*.
5. Scan the trace from start to finish:
   5.1. For every record *r(t)* corresponding to a reference,
   - add the block to the set of referenced blocks if it is a first reference.

- assign a priority to all blocks in the set of referenced blocks according to the replacement algorithm in the absence of invalidations.
- assign lowest priority to all invalidated blocks (breaking ties by the order in which the invalidations occurred.)

5.2.For every record *r(t)* corresponding to an invalidation, say the i^{th} invalidation for the block with address *x(i-1)* in the priority list and in the trace,

- rename all future references to *x(i-1)* by *xi.*
- add the block to the set of invalidated blocks
- remove the invalidation from the trace.

Therefore, a non oblivious replacement policy is derived from a known replacement algorithm with no invalidation by first renaming the blocks after each invalidation to distinguish between the valid block and its invalid copy, by assigning the lowest priority levels to invalid blocks and by applying the replacement algorithm to the remaining valid blocks. Figure 3 illustrates the trace transformation procedure for non oblivious LRU.

Figure 3. Trace transformation for non oblivious LRU

time	1	2	3	4	5	6	7	8	9	10
trace	a	b	c	Ib	b	c	a	Ia	d	a
renamed trace	a0	b0	c0		b1	c0	a0		d0	a1
priority list	a0	b0 a0	c0 b0 a0	c0 a0 **b0**	b1 c0 a0 **b0**	c0 b1 a0 **b0**	a0 c0 b1 **b0**	c0 b1 **a0** **b0**	d0 c0 b1 **a0** **b0**	a1 d0 c0 b1 **a0** **b0**

3.3. Cache inclusion in the presence of invalidations

Since we have defined a priority list and a total order on blocks at any time, replacement algorithms with invalidations can be treated as stack algorithms and known sufficient conditions for cache inclusion do not need to be rederived.

3.3.1. Caches with same set-mapping function

Oblivious and non oblivious replacement algorithms with a priority list on all referenced blocks are stack replacement policies after the trace transformations described in Section 3.2 and, following Mattson's result [14], inclusion holds in two systems with same set-mapping function, same block size and increasing associativity, under oblivious and non oblivious stack policies.

3.3.2. Caches with arbitrary set-mapping functions

Cache inclusion does not hold, in general, for caches with arbitrary set-mapping functions. This Section introduces necessary and sufficient conditions for inclusion under oblivious and non oblivious replacement policies. In this Section, we replace assumption 1 by assumption 1':

Assumption 1'. Caches C_1 and C_2 are the caches of the same processor in two systems in which blocks can be invalidated. In the case of a multiprocessor, the coherence protocol is the same in the two systems.

Case 1: Oblivious replacement policies

Theorem 2 (Necessary and sufficient condition). Under assumptions 1', 2 and 3, and an oblivious replacement algorithm, cache C_2 includes cache C_1 iff, at every time t and for every set, the total order induced by the replacement algorithm is such that a full set always contains the N-1 highest priority blocks mapping into it.

Proof:

The model of Section 3.2 for oblivious replacement policies transforms the trace with invalidations into a trace without invalidations. The proof of Theorem 1 can be directly adapted to apply to the transformed trace (a minor modification in the proof is to specify that block x used in the proofs cannot be invalid.) ❑

Case 2: Non oblivious replacement policies

The conditions of Theorem 1 seem to indicate that inclusion cannot hold for non oblivious replacement policies because there may be more than one invalidated block in a set and invalid blocks have lower priority than any other blocks.

However, the proof of Theorem 1 assumes that all blocks are valid, and consequently the replacement of any block in C_2 could lead to a violation of cache inclusion. Theorem 1 could be adapted to oblivious policies because the relative priority of the blocks does not change at the time of invalidations and therefore the fact that other blocks in cache may be invalid does not affect the proof. On the other hand, under non oblivious policies, not only the block being displaced from a given set must be a valid block, but also all other blocks in the same set must be valid. Therefore, Theorem 1 does not apply to non oblivious policies and new results have to be derived.

Theorem 3 (Necessary and sufficient condition). Under assumptions 1', 2 and 3, and a non oblivious replacement algorithm, cache C_2 includes cache C_1 iff, at every time t and for every set, the total order induced by the replacement algorithm is such that a full set (i.e. a set with N *valid* blocks) always contains the N highest priority blocks mapping into it.

Proof:

Sufficient condition: Assume that, at every time t and for every set, the total order induced by the replacement algorithm is such that the N highest priority blocks mapping to any full set are always in cache. Initially both caches are empty and inclusion holds. Suppose that set S of C_1 is refined by sets $S_1, S_2, \ldots, S_m$ of C_2. Inclusion can be violated for the first time when a valid block is replaced in one of the sets S_i. If a valid block y is replaced in one of the sets S_i, y must be the N^{th} highest priority block in S_i, and there cannot be any invalid blocks in S_i. If y is also in S, S must contain the same N highest priority blocks as S_i because all blocks mapping into S_i also map into S. Therefore y must also be replaced in S.

Necessary condition: Assume a trace such that at time *t*, at least $N+1$ blocks have been referenced and such there has been no invalidation. Assume that a block x is present in S but is not one of the N highest priority blocks mapping into S. Since the priority list induced by the replacement algorithm is independent of the cache contents, the replacement algorithm must guarantee inclusion for any possible set-refinement. If set S is refined by sets S_1, S_2,..., S_m of C_2, such that

1. x maps to S_1,

2. one of the blocks in S, z, maps into S_2,

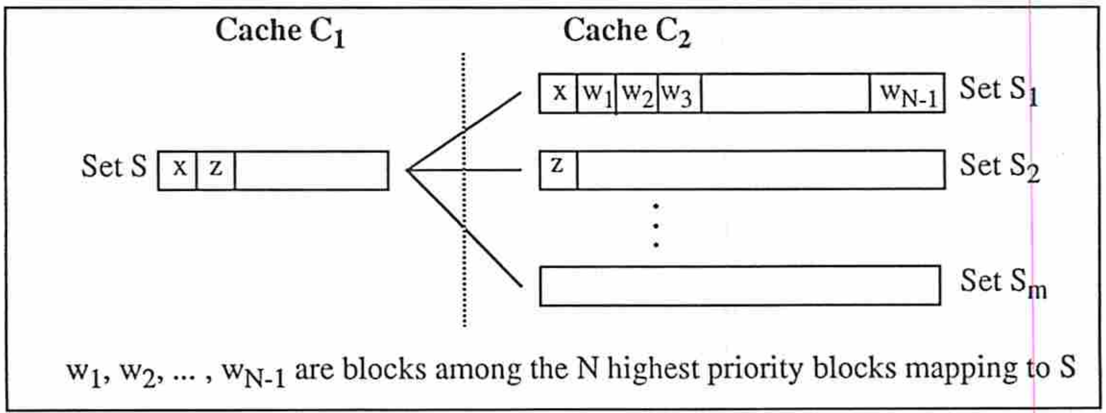
2. N-1 blocks among the N highest priority blocks mapping into S are in S_1 (if z is one of the N highest priority blocks mapping into S, choose the N-1 other than z among the N highest priority blocks mapping into S), and

4. all other blocks map into S_j, $j \neq 1,2$.

Now, suppose that z is invalidated. Set S_1 contains N valid blocks, and set S contains one invalid block. A miss to S_1 causes x to be replaced in S_1 and the invalid block to be replaced in S, violating inclusion. ❑

Figure 4 shows the situation leading to a violation of cache inclusion in Theorem 3.

Figure 4 Contents of C_1 and C_2 for Theorem 3.


Cache C_1
Cache C_2
Set S
x
z
x
w_1
w_2
w_3
w_{N-1}
Set S_1
z
Set S_2
Set S_m
w_1, w_2, ... , w_{N-1} are blocks among the N highest priority blocks mapping to S

Non oblivious replacement algorithms guaranteeing inclusion for caches with different set-mapping functions are such that a block becomes one of the N highest priority blocks mapping to its set when referenced. Therefore, neither LFU nor fixed priority algorithms are in this class. A replacement algorithm that satisfies the above requirement is LRU: when a block is referenced, it becomes the highest priority block mapping into any set.

Remark on Theorem 3

Since inclusion is transitive and holds for caches with same set-mapping function and increasing associativities, the condition of Theorem 3 is also sufficient for caches with same block size, no prefetching, set-refinement and increasing associativities, under a non oblivious replacement algorithm. The condition of Theorem 3 is also necessary for caches with increasing associativities since the priority list imposed by the replacement algorithm must be independent of the contents of any cache.

3.4. Special cases

3.4.1. Direct-mapped caches

It follows from Theorems 1, 2 and 3 that cache inclusion also holds in all cases for direct-mapped caches C_1 and C_2 such that C_2 refines C_1 and have the same block size because direct-mapped caches are a special case of set-associative caches under LRU and LRU satisfies the requirements of all three Theorems.

3.4.2. Infinite C_2

Another special case of inclusion is the case where C_2 has infinite size. Since a block can never be replaced in an infinite cache and since invalidations affect both caches the same way, the only conditions for inclusion are that both caches have same block size and do not prefetch.

4. EXTRACTING THE TRACE FOR A GROUP OF PROCESSORS

To evaluate different cache architectures for a group of processors in a multiprocessor we extract a reduced trace of references affecting their cache states. These references are issued by the group of processors and by processors outside the group. The processors in a group may be randomly selected or may correspond to a subsystem such as a cluster in an hierarchical machine.

Following Wang and Baer's idea [19] we filter the multiprocessor trace by simulating the multiprocessor with direct-mapped caches; the references issued by the processors in the group and causing misses or writes on clean blocks are retained in the trace. Additional references must be retained to make sure that *the caches in the group of processors receive the same invalidations and other coherence messages as in a simulation with the full trace*. A superset of all cache coherence events affecting any cache configurations to evaluate for the group of processors must also be recorded in the reduced trace. It is easy to show [20] that the coherence events affecting a multiprocessor system with a given cache size are a superset of the coherence events affecting a multiprocessor system with smaller caches provided inclusion holds[3]. Therefore the superset of all coherence events are added to the reduced trace by simulating a multiprocessor with caches that include all possible caches to evaluate.

4.1. Extraction of the reduced trace

In order to extract the trace of a group of processors to evaluate caches C_1, C_2,..., C_{N-1}, we first identify a direct-mapped cache C_0 such that C_0 is as large as possible and is included in all C_i's, $i=1,...,N-1$. We then identify a cache C_N which includes all C_i's, $i= 0,...,N-1$. If C_N is the infinite cache with the same block size as all C_i's, the reduced trace is valid for the simulation of any cache that includes C_0. However, in cases where there is not enough memory available to simulate infinite caches, the replacement policy and cache organization of all C_i's, $i=1,...,N$ must comply to the appropriate conditions of Section 3.

The algorithm for extracting the reduced trace of a group of processors is as follows:

Step 1 - Using the full trace (T_0) as input, simulate the multiprocessor with caches C_0. Retain the

3. The interested reader should consult the Appendix for more details.

references causing misses and writes on clean blocks to generate a first reduced trace (T_1).

Step 2 - Using T_1 as input, simulate the multiprocessor with caches C_N. Retain the references issued by the processors in the group, plus all references issued by the processors outside the group and affecting the state of any cache in the group to generate the final reduced trace (T_2).

T_1 is the trace obtained by Wang and Baer's technique [19]. A simulation with T_2 of the group of processors with any cache that includes C_0 and is included by C_N triggers the same changes of state of the caches of all processors in the group as in the simulation of the whole multiprocessor with T_0 or T_1. Therefore, the simulation with T_2 generates the same cache states, the same misses, the same write-backs and the same coherence events in the processors in the group as a simulation of the whole system.

4.2. Restrictions on the protocol

Our methodology works for all cache coherence protocols such that the next state of a cached block does not depend on its state in other caches. This is true for the Basic [7], the Write-Once, the Berkeley and the Synapse protocols [2]. We have not yet found a way to deal with the Illinois protocol [2], in which a read miss returns a block as shared-unmodified or exclusive-unmodified depending on whether the block is cached remote or not. The reason is that the contents of the caches outside the group are not available.

4.3. Multiblock trace reduction

A trace reduced with a given block size is not valid for a different block size because cache inclusion does not hold between caches with different block sizes. We can however obtain a "universal" T_1 trace, valid for the simulation of caches with different block sizes, by filtering the trace through multiple simulations with different block sizes and by retaining the superset of references causing misses or writes on clean blocks. Multiblock traces for five different block sizes were derived in [19]. They were only 40 to 48% longer than traces for a single block size because misses are often common to caches with different block sizes.

The multiblock trace for a group of processors is generated by extracting their traces in simulations of systems with different block sizes and by collecting a superset of the references. We need to identify a cache $C_{0,B}$ and $C_{N,B}$ for each block size B. Caches $C_{0,B}$ must have the same number of sets and caches $C_{N,B}$ must have the same number of sets and the same set-associativity for all values of B. The algorithm for extracting a reduced multiblock trace is as follows:

Step 1. Using the full trace (T_0) as input, simulate multiprocessors with caches $C_{0,B}$ for all values of B. Retain the superset of references causing misses and writes on clean blocks in all simulations to generate a first reduced trace (T_1).

Step 2. Using T_1 as input, simulate multiprocessors with caches $C_{N,B}$. Retain the superset of the references issued by the processors in the group, plus the superset of all references issued by processors outside the group and affecting the state of any cache in the group in all simulations to generate the final reduced trace (T_2).

T_1 is the trace obtained by Wang and Baer's technique [19]. T_2 is a valid trace to obtain the correct cache states and cache events at any time for the group of processors with any cache that includes $C_{0,B}$ and is included in $C_{N,B}$, for every value of B.

5. PROCESSOR SAMPLING

Different processors do not generate the same number of events in multiprocessor simulations even for a Single Program Multiple Data (SPMD) application. In fact there can be wide variations among individual processors as our simulations have shown. Reliable estimates of average system-wide metrics must be based on average measures for a group of *randomly* selected processors. We call this approach *processor sampling*. The confidence intervals for these estimates narrow when more processors are sampled.

To estimate the system *average miss rate*, *average write back rate*, *average rate of invalidations sent* and *average rate of invalidations received*, we measure their equivalent in the simulation of the sampled processors with the extracted trace. The definition of estimates for metrics involving the processors removed from the trace is more subtle. For example, consider the *average number of copies invalidated per invalidation sent*, a very important measure for directory schemes in large-scale systems [8]. Since the contents of the caches outside the group of sampled processors are unknown, it would seem that this metric cannot be estimated with our methodology. However, by invoking statistical arguments, we can obtain a reliable estimate for the average number of copies invalidated per invalidation sent by measuring the ratio between the numbers of invalidations received and of invalidations sent by the sampled processors.

6. SIMULATION RESULTS

We have run simulations to check the estimates for system-wide metrics based on simulations of sampled processors and to evaluate the reduction factors afforded by the methodology.

6.1. Traces

The three traces used in the simulations were given to us by Anant Agarwal's group at MIT [1][4].[4] They come from FORTRAN applications compiled for 64 processors and are called WEATHER, SIMPLE and FFT. WEATHER models the atmosphere as a three-dimensional grid and solves a set of differential equations by the finite-difference method. SIMPLE solves equations for hydrodynamic behavior also by a finite difference method. FFT is a radix-2 fast Fourier transform program. The main characteristics of the traces are displayed in Table 1.

Table 1: Trace Characteristics (number of references in 1000's)

trace	instruction references	data references	private data references	shared data references
WEATHER	13,638	15,627	13,110 (15.7% writes)	2,518 (19.2% writes)
SIMPLE	11,594	14,010	9,936 (34.7% writes)	4,074 (10.9% writes)
FFT	3,124	4,312	3,279 (26.6% writes)	1,034 (49.9% writes)

4. These traces are in the public domain and can be obtained upon request for the purpose of reproducing the results in this Section.

There are four types of references in the traces: accesses to instructions, accesses to private data, accesses to shared data and accesses to synchronization variables. In all of our experiments, we removed all instructions from the traces, because their hit rate is very high, which would inevitably yield very high reduction factors and would obscure the comparison between trace lengths and simulation times of different techniques. This should be kept in mind when observing trace reduction factors.

6.2. Simulation methodology

Each multiprocessor trace was filtered to produce two reduced traces according to the methodology of Section 4.1. The first trace is the one obtained with the reduction technique proposed by Wang and Baer. The second trace is a trace extracted for a group of eight processors randomly selected among the 64. We refer to these two traces simply as T_1 and T_2, respectively. The full trace is referred to as T_0.

Reduced traces are derived from simulations of the entire multiprocessor system with the basic cache coherence protocol described in the Appendix. The simulation method consists of representing each cache as a table. References are processed one by one in the order of the trace. In the simulations of the entire system, the simulator fetches the next trace record, updates the cache of the processor issuing the reference, performs any necessary coherence actions, and updates the metrics. In the simulations with the extracted trace the caches of the processors outside the sample are not simulated. We have performed three sets of simulations with different C_0 and C_N caches and some results are reported in Sections 6.3, 6.4, and 6.5.

6.3 First set of simulations

For the results reported in this Section we chose a 16K byte direct-mapped cache for C_0. Ideally C_N should be chosen infinite. However the data set sizes of the benchmarks are huge. Our SPARC station 2 with 40 Mbytes of main memory could not cope with the simulation of infinite caches. To make the simulation feasible on our machine, we chose C_N to be a 512K byte, 4-way associative cache with non oblivious LRU replacement. Both filter caches have a 16 bytes block size. The reduced trace is applicable to systems with caches containing 1K, 2K, 4K and 8K sets, set associativities of 1, 2 or 4 and non oblivious LRU replacement in each set.

Table 2 shows the lengths of the full and reduced traces. The ratio between the lengths of

Table 2: Trace Lengths (in bytes)

trace	WEATHER	SIMPLE	FFT
length of the original trace (T_0) (without instruction references)	93,763,218	84,059,514	25,874,664
length after first reduction (T_1) filter cache C_0: 16Kbytes, 1-way	7,480,302 (7.9% of T_0)	14,852,574 (17.3% of T_0)	6,356,730 (24.5% of T_0)
length after second reduction (T_2) filter cache C_N: 512K, 4-way # of processors sampled: 8	1,252,908 (1.3% of T_0) (16.7% of T_1)	2,299,200 (2.7% of T_0) (15.5% of T_1)	1,122,576 (4.3% of T_0) (17.5% of T_1)

T_1 and T_0 is very much related to the miss rate since all misses are kept in T_1. The average data miss rates for the WEATHER, SIMPLE and FFT, for 16K direct-mapped caches are 6.9%, 16.9% and 23.6%, respectively. Besides misses, writes on clean blocks also contribute to the size of T_1.

The reduction factor, from T_1 to T_2, is mainly dependent on the fraction of processors sampled because the number of references issued by all processors in the system is roughly the same. The amount of write sharing also affects the final trace length because it determines the amount of coherence interference.

Table 3 shows the simulation times for a 128Kbyte direct-mapped cache.The reduction of the simulation times between T_0 and T_1 is not as large as the corresponding reduction of the trace length (this was previously noted by Wang and Baer in their simulations) because the simulator processes read hits and write hits on dirty blocks much faster than misses or writes on clean blocks. From T_1 to T_2 the simulation time is reduced by a factor greater than the trace length mostly because a smaller number of caches are simulated.

Table 3: Simulation execution times (in seconds)

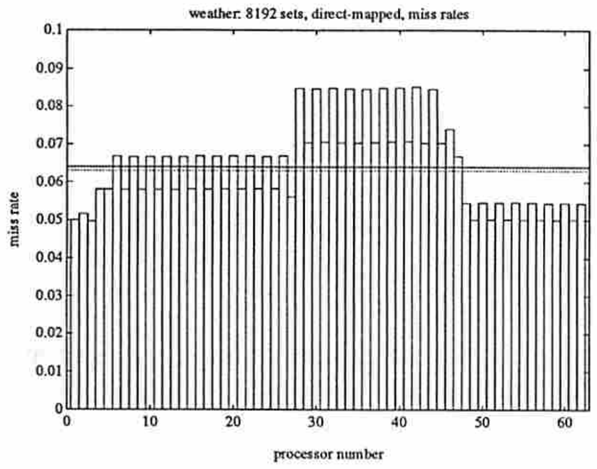
trace	WEATHER	SIMPLE	FFT
T_0	579.09	697.84	183.7
T_1	183.6 (31.7% of T_0)	375.8 (56.0% of T_0)	88.9 (48.4% of T_0)
T_2	6.7 (1.16% of T_0) (3.6% of T_1)	12.6 (1.81% of T_0) (3.4% of T_1)	5.2 (2.83% of T_0) (5.8% of T_1)

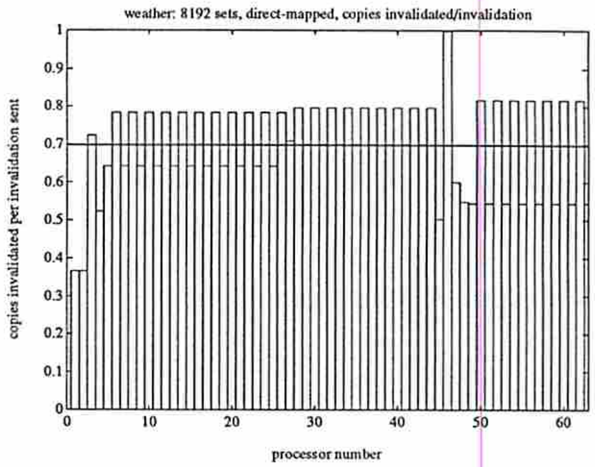
Figure 5 corresponds to a system with 128Kbyte direct-mapped caches and 16 bytes block. It shows the exact miss rates for each processor and the exact average miss rate (these miss rates were derived with T_1) as well as the average miss rate estimated with T_2 by measuring the miss rate in eight randomly sampled processors. Figure 5 also displays the exact average number of copies invalidated per invalidation for each of the 64 processors as well as the average and estimated average numbers for the whole system.

Figure 5 demonstrates that the number of cache events of each type can vary widely in different processors. The number of processors needed for accurate estimates depends on the variance of the processors' metrics. Parallel applications exhibiting high homogeneity and symmetry tend to have a small variance among the processors' metrics. Intense block sharing among processors is a good indicator of larger variations in the number and the pattern of invalidations in different processors. The FFT trace exhibits a large amount of write sharing, most of it due to false sharing [6]. As seen in Figure 5.c, FFT shows a high variance in the miss rates and the number of copies invalidated per invalidation sent. The WEATHER and SIMPLE traces have much less block sharing and therefore show less variability. To increase the accuracy of the estimates, the metrics for each sampled processor and the average metrics could be collected during the trace reduction process; if significant differences are observed between the metrics in the sampled processors and the average metrics, the reduction should be performed again either with a new selection of processors or with a larger number of sampled processors.

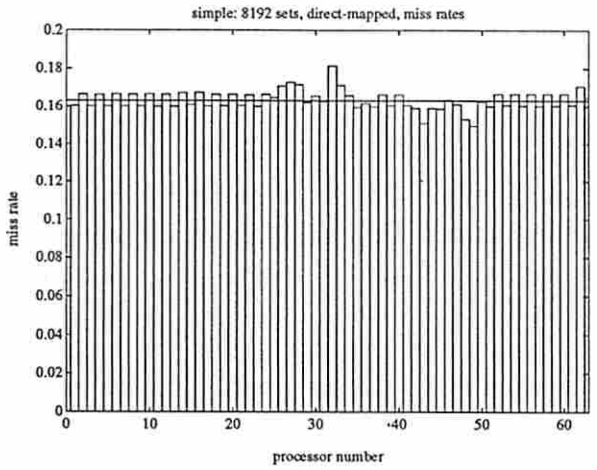
Figure 5. Miss rates and average number of invalidated copies
Cache: 128KB, 1-way; Block size 16B; C_N: 512KB, 4-way; C_0: 16KB, 1-way.
—— exact average .... estimated average

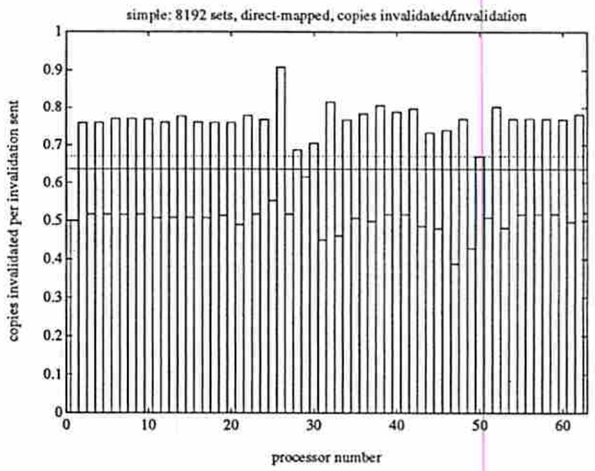
Miss rates **Copies invalidated/invalidation**



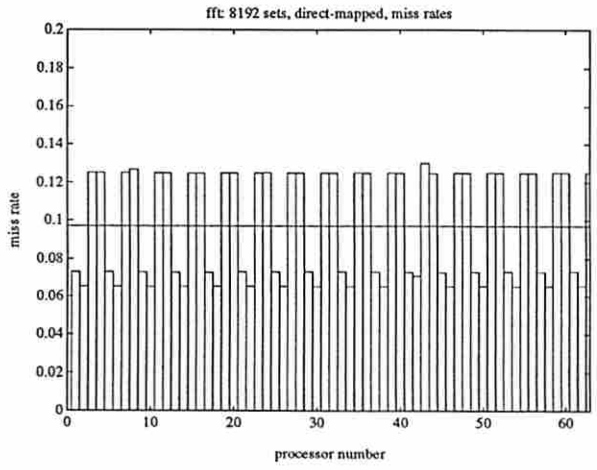


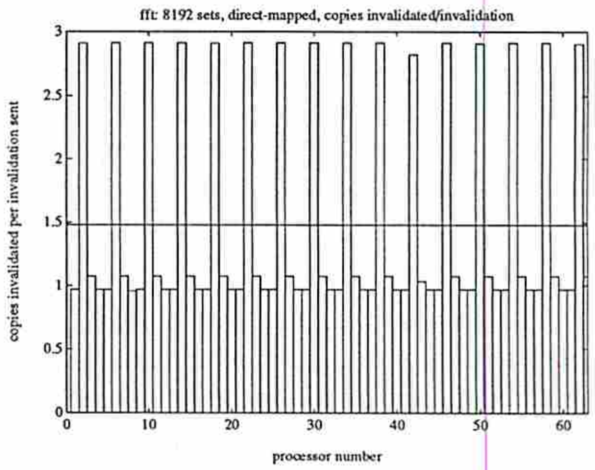
(a) WEATHER





(b) SIMPLE





(c) FFT

6.4. Second set of simulations

The reduced trace used in this Section was obtained with a 4Kbyte direct-mapped (C_0) and a 64Kbytes, 4-way-associative (C_N) filter caches. All caches have a 16 byte block size and their replacement algorithm is non oblivious LRU.

Tables 4, 5, and 6 show the average and estimated average miss rates as well as the minimum and maximum miss rates experienced by each processor with different cache configurations for WEATHER, SIMPLE and FFT, respectively. The average, minimum, and maximum miss rates were obtained by simulating the 64 processors with the full trace. The estimated average is the average miss rate of eight randomly selected processors. The estimates are highly reliable even when the variations among processors are high.

6.5. Third set of simulations

The length of the trace extracted for each processor in a sample depends on the number of references issued by the processor (i.e., on the size of C_0) and on the amount of coherence events affecting the processor's cache (i.e., on the size of C_N). Figure 6 shows the effect of the size of C_N on the length of the reduced trace (C_0 is a 4Kbytes direct-mapped cache with a 16 byte block size). All C_N caches are direct-mapped caches with number of sets varying from 256 to 16384. We note that the total number of references in the reduced trace grows very slowly with the size of C_N. We expect the trace length to converge to the length of the trace for an infinite C_N, once the size of C_N is large enough. Unfortunately, we could not simulate the case of infinite C_N for reasons given previously.

In the case of FFT, the coherence references contribute to 34% of all references in T_2 for the 256Kbytes C_N and to 21% for the 4Kbyte C_N. Of the 3 traces, FFT exhibits the highest percentage of coherence references in the trace due to its large amount of sharing. The fraction of coherence references in the reduced trace is 22.8% for WEATHER and 14.6% for SIMPLE, for the 256Kbytes C_N. It is expected that these ratios will be even lower when a group of processors is sampled because some coherence references affect more than one processor in the sample. Thus, the size of C_N does not affect significantly the length of the reduced trace of the sampled processors.

7. CONCLUSIONS

There are three main contributions in this paper. The first contribution is a set of conditions for cache inclusion and the extension of stack replacement algorithms to caches with invalid blocks. The second contribution is the proposed methodology of extracting a valid trace for a processor or a cluster of processors in order to explore efficiently different cache architectures. The third contribution is to show that processor sampling can yield reliable estimates for system-wide metrics.

The proposed techniques yield dramatic reductions in both trace length and simulation time. The reduction of the simulation time is even larger than that of the trace length because less caches are simulated. With this new approach we can probably obtain reliable performance measures for systems with hundreds of processors. Our simulations have also shown that the size of the extracted trace for one processor is only marginally affected by the coherence interferences. Thorough performance studies of the effect of the cache architecture on the performance of each

Table 1: Miss rates for WEATHER

cache size \ miss rate	minimum	maximum	average	estimated average
4K, 256 sets, 1-way	9.57%	13.71%	11.28%	11.21%
8K, 256 sets, 2-way	5.24%	8.85%	6.67%	6.60%
8K, 512 sets, 1-way	6.14%	10.12%	7.76%	7.67%
16K, 256 sets, 4-way	4.92%	8.42%	6.31%	6.23%
16K, 512 sets, 2-way	4.95%	8.47%	6.34%	6.26%
16K, 1024 sets, 1-way	5.43%	9.27%	6.93%	6.84%
32K, 512 sets, 4-way	4.92%	8.41%	6.30%	6.21%
32K, 1024 sets, 2-way	4.92%	8.42%	6.30%	6.22%
64K, 1024 sets, 4-way	4.91%	8.37%	6.27%	6.19%

Table 2: Miss rates for SIMPLE

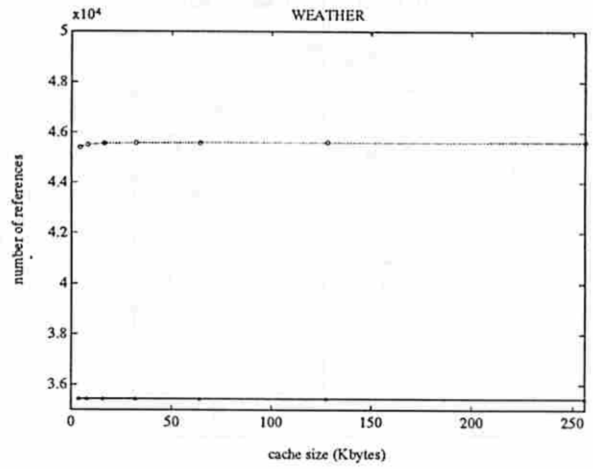
cache size \ miss rate	minimum	maximum	average	estimated average
4K, 256 sets, 1-way	17.40%	20.26%	18.83%	18.78%
8K, 256 sets, 2-way	12.82%	16.63%	14.56%	14.40%
8K, 512 sets, 1-way	16.54%	19.40%	17.94%	17.89%
16K, 256 sets, 4-way	10.04%	13.08%	11.56%	11.39%
16K, 512 sets, 2-way	12.67%	16.44%	14.39%	14.23%
16K, 1024 sets, 1-way	15.52%	18.74%	16.91%	16.84%
32K, 512 sets, 4-way	9.95%	12.92%	11.44%	11.27%
32K, 1024 sets, 2-way	12.61%	16.34%	14.30%	14.13%
64K, 1024 sets, 4-way	9.90%	12.82%	11.36%	11.20%

Table 3: Miss rates for FFT

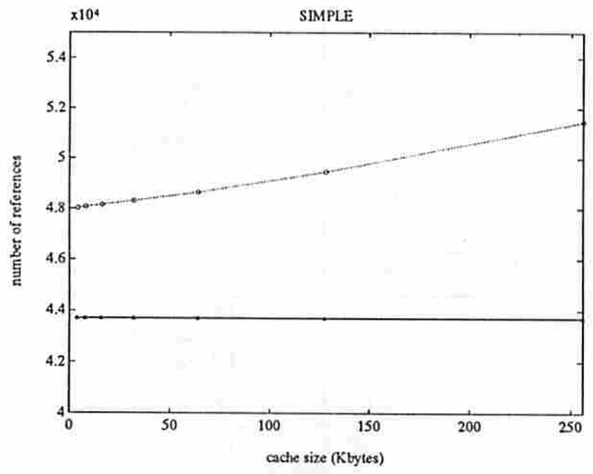
cache size \ miss rate	minimum	maximum	average	estimated average
4K, 256 sets, 1-way	23.19%	24.90%	23.95%	23.93%
8K, 256 sets, 2-way	8.09%	14.31%	11.20%	11.31%
8K, 512 sets, 1-way	23.09%	24.43%	23.74%	23.69%
16K, 256 sets, 4-way	6.50%	12.57%	11.02%	10.28%
16K, 512 sets, 2-way	7.67%	14.12%	11.58%	10.94%
16K, 1024 sets, 1-way	23.01%	24.32%	23.56%	23.53%
32K, 512 sets, 4-way	6.49%	12.55%	11.00%	10.25%
32K, 1024 sets, 2-way	7.27%	14.08%	11.34%	10.67%
64K, 1024 sets, 4-way	6.49%	12.49%	10.95%	10.22%

Figure 6. Number of references in trace T_2 vs. size of C_N

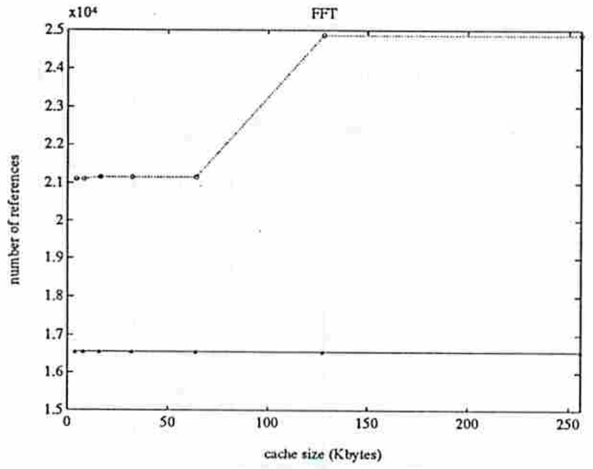
—— refs. by processor sampled .... total refs. in trace T_2



(a) WEATHER



(b) SIMPLE



(c) FFT

individual processor can be done as efficiently as in uniprocessors. Similarly, entire subsystems, such as clusters of processors in hierarchical architectures, can be efficiently designed by extracting their trace.

Besides their application to trace extraction and processor sampling, the conditions for cache inclusion could also be useful in the context of one-pass simulations of multiple cache architectures. The problem here is to find an efficient way to maintain the global stack in the presence of invalidations.

8. REFERENCES

[1] A. Agarwal, "Analysis of Cache Performance for Operating Systems and Multiprogramming". Kluwer Academic Publishers, 1989.

[2] J. Archibald and J. L. Baer, "Cache-Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model". *ACM Transactions on Computer Systems*, 4(4):373-298, November 1986.

[3] P. Bitar, "A Critique of Trace-Driven Simulation for Shared-Memory Multiprocessors". M. Dubois and S. S. Thakkar, editors, *Cache and Interconnect Architectures in Multiprocessors*, pp. 37-52, Kluwer Academic Publishers, 1990.

[4] Chaiken, D. *et al*, "Directory-Based Cache Coherence in Large Scale Multiprocessors". *IEEE Computer*, Vol. 23, No. 6, pp. 49-59, June 1990.

[5] Covington, R.G., *et al*., "The Rice Parallel Processing Testbed". *Proceedings of the 1988 ACM Sigmetrics Conference*, May 1988, pp. 4-11.

[6] M. Dubois, L. Barroso, Y. Chen and K. Oner, "Scalability Problems in Multiprocessors with Private Caches". *Proceedings of PARLE'92*, June 1992.

[7] M. Dubois and J-C Wang,"Shared Data Contention in a Cache Coherence Protocol". *IEEE Transactions on Computers,* Vol. 40, No. 5, pp. 640-645, May 1991.

[8] A. Gupta and W.-D. Weber, "Analysis of Cache Invalidation Patterns in Shared-Memory Multiprocessors". *Proceedings of the 3rd Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 243-256, April 1989.

[9] E. Hagersten, S. Haridi, and D. Warren, "The Cache-Coherence Protocol of the Data Diffusion Machine". M. Dubois and S. S. Thakkar, editors, *Cache and Interconnect Architectures in Multiprocessors*, pp. 165-188, Kluwer Academic Publishers, 1990.

[10] M. D. Hill and A. J. Smith, "Evaluating Associativity in CPU Caches". *IEEE Transactions on Computers*, C-38(12):1612-1630, December 1989.

[11] K. Gharachorloo, A. Gupta and J. Hennessy, "Hiding Memory Latency using Dynamic Scheduling in Shared-Memory Multiprocessors". *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pp. 22-33, May 1992.

[12] E.J. Koldinger, S.J. Eggers, and H.M. Levy, "On the Validity of Trace-driven Simulations for Multiprocessors". *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pp. 244-253, May 1991.

[13] Lee, R., Yew, P-C. and Lawrie, D., "Multiprocessor Cache Design Considerations". *Pro-*

ceedings of the 14th International Symposium on Computer Architecture, June 198.

[14] R. L. Mattson, J. Gecsei, D. R. Slutz and I. L. Traiger, "Evaluation Techniques for Storage Hierarchies". *IBM System Journal*, 9(2):78-117, 1970.

[15] Puzack, T. R., "Cache Memory Design". Ph.D. thesis, ECE Department, University of Massachusetts, 1985.

[16] H. S. Stone, "High Performance Computer Architecture". Addison Wesley, Second Edition, 1990.

[17] J. G. Thompson and A.J. Smith, "Efficient (Stack) Algorithms for analysis of write-back and sector memories". *ACM Transactions on Computer Systems*, pp. 78-116, Feb. 1989.

[18] W. H. Wang, "Multilevel Cache Hierarchies". Ph.D. thesis, University of Washington, Seattle, September 1989.

[19] W. H. Wang and J. L. Baer, "Efficient Trace-Driven Simulation Methods for Cache Performance Analysis". *ACM Transactions on Computer Systems*, 9 (3), pp. 222-241, Aug. 1991,

[20] J. C. Wang, "Analytical Modeling for Shared Block Contention in Cache Coherence Protocols". Ph.D. thesis, University of Southern California, December 1990.

APPENDIX.

Relation between the cache coherence events of two multiprocessors under cache inclusion

In this appendix the cache coherence events that affect caches in two systems are compared. A few assumptions are made before presenting this comparison. It is assumed that the cache coherence protocol, the multiprocessor trace and the replacement algorithm are the same for the two systems. All caches in a given system are identical, and the cache block size is the same for both systems. The systems are named S_1 and S_2. Each processor j in system S_i has a private cache, and cache j in system S_i is denoted by C_{ij}, $i = 1, 2,\ \ j = 1, 2, \ldots$. It is also assumed that cache C_{1j} is included in cache C_{2j}, for all j. We further assume a write-invalidate cache coherence protocol[5], where a cache block can be in one of the following states: *invalid* (block does not contain valid data); *shared-unmodified* (some other caches may have a unmodified copy of the block, and data in block is consistent with main memory); *exclusive-modified* (no other cache has this block, and data in block is not consistent with main memory). The protocol action for each possible event are described as follows:

1. *read hit*. No action by the protocol.

2. *read miss*. If another cache has a modified copy of the block, the block is written back to the main memory, the remote cache sets the state of the block to shared-modified and the main memory supplies the block to the requesting cache. The block is loaded as shared-unmodified.

3. *write hit*. If the block is already exclusive-modified, the write can proceed locally without any

5. The propositions proved in this appendix are also true for other write-invalidate protocols, as the Write-Once, Synapse, Illinois and Berkeley protocols. The cache block states have different names for different protocols, but the coherence events are equivalent to those presented here.

action by the protocol. If the block is shared-modified, an invalidation signal is sent, and all other caches with a copy of the block set their states to invalid. The writing cache sets the block state to exclusive-modified.

4. *write miss*. Like in a read miss, if another cache has an exclusive-modified copy, the block is written back to main memory, and the main memory supplies the block to the requesting cache. All remote caches with a copy of the block set their states to invalid. The block is loaded as exclusive-modified.

In a write-invalidate coherence protocol as the above described, the coherence events that affect the state of a cached block are invalidations due to writes in remote caches, and change of block state, from exclusive-modified to shared-modified, due to read misses in remote caches.

The following statements are true under the above conditions, and will be used in the proofs of relations between cache coherence events of the two systems:

Fact 1. *If a block is present as shared-unmodified in more than one cache in S_1, it is also present as shared-unmodified in the same caches in S_2.*

Fact 2. *If a block is present as exclusive-modified in C_{1j}, it is also present as exclusive-modified in C_{2j}. However, a block may be present as exclusive-modified in C_{2j} and as shared-unmodified in C_{1j}[6]. In the later case, there can be only one shared-unmodified copy in S_1, that is, the one in C_{1j}.*

Proposition 1. *Every invalidation to a cache in S_1 corresponds to an invalidation to the same cache in S_2.*

Proof: It follows from inclusion that if a block present in C_{1j} is invalidated, the block is also present in C_{2j}, and therefore an invalidation must also occur in C_{2j}. However, a block may be present in C_{2j} at the time of the invalidation, but it may have been replaced from C_{1j}, and therefore there may be more invalidations to C_{2j} than to C_{1j}.

Proposition 2. *Every change of state of a block, from exclusive-modified to shared-unmodified, due to coherence, in a cache in S_1 corresponds to a change of state, from exclusive-modified to shared-unmodified in the same cache in S_2.*

Proof: Assume that C_{1j} has a modified copy of a block, when processor i needs to read it. Due to inclusion, the block is also present and modified in C_{2j}, and consequently a change of state from modified to shared-modified must also occur in C_{2j}.

From the two propositions above, it can be concluded that the cache coherence events that affect a cache in system S_2 are a superset of the cache coherence events that affect the same cache in system S_1, or, in other words, the coherence events that affect a given cache are a superset of the events that affect caches included by it.

6. A block may had been in both C_{1j} and C_{2j} as exclusive-modified, and have been replaced in C_{1j} and brought back to C_{1j} as shared-unmodified.