A Novel Lightweight Directory Architecture for Scalable Shared-Memory Multiprocessors

Alberto Ros, Manuel E. Acacio and José M. García

Departamento de Ingeniería y Tecnología de Computadores
Universidad de Murcia. 30071 Murcia (Spain)
{a.ros,meacacio,jmgarcia}@ditec.um.es

Abstract. There are two important hurdles that restrict the scalability of directory-based shared-memory multiprocessors: the directory memory overhead and the long L2 miss latencies due to the indirection introduced by the accesses to directory information, usually stored in main memory. This work presents a lightweight directory architecture aimed at facing these two important problems. Our proposal takes advantage of the temporal locality exhibited by the accesses to the directory information and on-chip integration to design a directory protocol with the best characteristics of snoopy protocols. The lightweight directory architecture removes the directory structure from main memory and it stores directory information in the L2 cache avoiding in most cases the access to main memory. The proposed architecture is evaluated based on extensive execution-driven simulations of a 32-node cc-NUMA multiprocessor. Results demonstrate that the lightweight directory architecture achieves better performance than a non-scalable full-map directory, with a very significant reduction on directory memory overhead.

1 Introduction

Particular implementations of cache coherence protocols are quite different depending on the total number of processors of a shared-memory multiprocessor. In systems with few processors, an interconnection network with a completely ordered message delivery (such as a bus) can be used. Cache coherence in these cases is ensured by making all processors snoop the bus to obtain information regarding the blocks that are being accessed (read or written) by the other processors. This implementation of the coherence protocol is known as snooping-based protocol whereas the term Symmetric Multiprocessors (SMP) is frequently used to designate the architecture of the multiprocessor [1].

On the other hand, systems with greater number of processors are organized around a scalable point-to-point interconnection network; besides, main memory in these machines is physically distributed to ensure that memory bandwidth also scales with the number of processors. Now a directory-based cache coherence protocol is used to ensure coherence [1]. Each node of the machine (which includes the processor and a fraction of the total main memory) has a directory structure which stores coherence information for the memory blocks that are

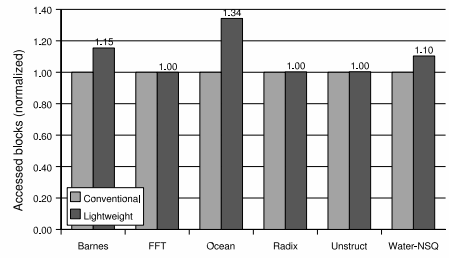

Fig. 1. Worst-case overhead introduced by the lightweight directory architecture.

allocated on it (the *home node*). In this way, L2 cache misses are sent to the corresponding home node, which acts as an ordering point and performs the actions needed to ensure coherence. Unfortunately, the accesses to the directory cause long L2 miss latencies since this structure is usually stored in main memory [2]. Additionally, the amount of extra memory required for storing directory information (directory memory overhead) could become prohibitive for a large-scale configuration of the multiprocessor if care is not taken [3]. In general, these multiprocessors have been called cc-NUMA (cache-coherent Non-Uniform Memory Access) and the best known example is the SGI Origin 2000/3000 [4].

In this paper, we propose the lightweight directory architecture, a novel architecture that takes advantage of on-chip integration to design a large scale cc-NUMA architecture with the best characteristics of SMP multiprocessors. Unlike conventional directories, which associate directory entries to memory blocks, our proposal moves directory information to the cache level where the coherence of the memory block is managed (the L2 cache in our particular case). In this way, directory information is removed from main memory. Our proposal is motivated by the observation that only a small fraction of the memory blocks are stored in the L2 caches at a particular time (temporal locality), and that in most cases, when a request for a memory block from a remote node arrives at the corresponding home node either the home node has recently accessed the block and it resides in the L2 cache, or the home node will request the block in a near future.

As in a conventional directory protocol, L2 cache misses are sent to the corresponding home node which is in charge of satisfying the miss (for example, by providing the memory block in case of a load miss). On the first reference to a memory block, however, the home node books an entry in the local L2 cache which is used to store directory information for the block and occasionally the own block. Subsequent L2 cache misses to the same block will find directory information and in some cases data in the L2 cache of the home node.

However, storing directory information in the L2 cache for each block requested by any remote node could result in a significant increase in the number of blocks being stored in the L2 cache of the corresponding home directory, and consequently, in its total number of replacements. Fortunately, the observation that motivates our proposal points out that it is not the case. We performed a preliminary study about the extra number of memory blocks in the worst case

that would be stored in the L2 cache when lightweight directories are used. This study has been carried out running several applications on top of the RSIM simulator assuming infinite caches. Figure 1 shows that in the worst case the increase in the number of blocks that are brought to the L2 cache does not exceed 34% and that in general good results could be expected.

Our proposal, therefore, brings two important benefits. First of all, since the total number of memory blocks is much larger than the total number of L2 cache entries, directory memory overhead is drastically reduced by a ratio of 1024 (or more) compared to conventional directory architectures. Second, since directory entries are stored in the L2 cache of the home node, and both the L2 cache and the directory controller are integrated into the processor chip (which is common in recent processors [5] [6]), the time needed to access the directory is significantly reduced, which translates into important reductions in the latency of L2 cache misses, and therefore, improvements of up to 26% in total execution time are obtained. Moreover, we develop a coherence protocol suited to the particularities of the new directory architecture.

The rest of the paper is organized as follows. In section 2 we present a review of the related work. In sections 3 and 4 we describe the lightweight directory architecture and the coherence protocol required by it, respectively. Section 5 introduces the methodology employed in the evaluation. In section 6 we show some performance results for our proposal. And finally, section 7 concludes the paper and points out some future work.

2 Related Work

In SMP multiprocessors a shared bus is typically employed to interconnect all the processors. In this way, every processor snoops all requests to memory in the order in which they appear on the bus. Unfortunately, the bus becomes a bottleneck when the number of processors increases. Martin *et al.* proposes timestamp snooping to avoid this bottleneck [7]. Timestamp snooping allows that a snoopy protocol is implemented on top of a scalable point-to-point interconnect network by using timestamp and reordering requests at the interconnect end points.

Bandwidth Adaptive Snooping Hybrid (BASH) [8] is a hybrid coherence protocol that dynamically decides whether to act like snooping protocols (broadcast) or directory protocols (unicast) depending on the available bandwidth.

Token coherence protocols [9] avoid both the need of a totally ordered network and the indirection caused by the directory by using N tokens per memory block. In this way, a node can read a block if it has at least one token and can update the block if it has all the tokens of that block.

On the other hand, cc-NUMA multiprocessors use a scalable point-to-point interconnection network and need a directory structure to guarantee ordered memory accesses. However, directory implies memory overhead and long L2 miss latencies. Directory caches can be used to reduce the latency of L2 misses by obtaining directory information from a much faster structure than main memory

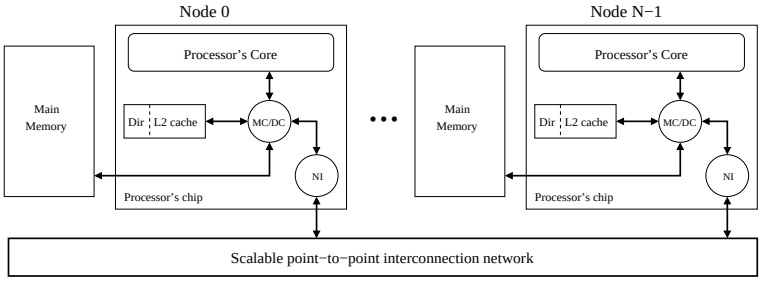

Fig. 2. The lightweight directory architecture.

[2]. Finally, several techniques have been proposed to reduce directory memory overhead. Usually, they are based on compressed sharing codes, such as *coarse vector* [10], which is currently employed in the SGI Origin 2000/3000 multiprocessor, *gray-tristate* [11] or *binary tree with subtrees* [3].

3 The Lightweight Directory Architecture

The lightweight directory architecture proposed in this paper removes directory information from main memory and stores it in the L2 caches to reduce its access time. Of course, this reduction would not be so effective if the directory controller were outside the processor chip. Fortunately, current integration scale allows the inclusion of some key components of the system such as the memory controller, the coherence hardware and the network interface and router inside the processor chip (see Compaq Alpha 21364 EV7 [5] or AMD Hammer [6]). Hence, we assume in this work that the directory controller and the L2 cache with directory information are on-chip.

Figure 2 shows the proposed architecture for a N-node multiprocessor. The nodes are connected using a scalable point-to-point interconnection network through the network interface (NI). The memory and directory controller (MC/DC) handles all inter-node memory references going into or out of the node. In this way, the L2 cache misses are sent to the memory controller of the corresponding home node, which looks for the block's directory information stored in the L2 cache tags structure speculatively in parallel with the access to the L2's structure where data is stored[1]. If the block is found in the L2 cache, directory information is obtained without going to main memory. On the other hand, if the block is not present at cache, the block is not cached by any node and it is necessary to accede to main memory.

Each cache block contains four main fields aside from the data of the block: the *tag* itself, used to identify the block, the *cache state*, the *directory state*, and the *sharing code*. The latter two fields are added by the lightweight directory structure proposed in this paper. The cache state field can take one of the four values (2 bits) used by the MESI protocol. Nevertheless, the invalid state has

[1] In this paper, we assume that the L2 cache is split into tags and data structures, as is commonly found in current designs.

two meanings: one of them is the same that in MESI protocol, and the other one means that this block has a valid directory information, and it takes place when there is some presence bit in the directory information. The directory state field can take two values (one bit):

- S (Shared): The memory block is shared in several caches, each one of them with a up-to-date copy. When needed, the L2 cache of the home node will provide the block to the requesters, since this cache has always a valid copy even if it has not used it.
- O (Owned): The memory block is in just one cache and could have been modified. The single valid copy of the block is held in the L2 cache of the home node, when its cache state is modified or exclusive, or alternatively, in one of the L2 cache of the remote nodes. In the latter case, the cache state for the memory block in the L2 cache of the home is invalid, and the identity of the owner is stored in the sharing code field.

Note that an additional directory state is implicit. The *U* state (Uncached) takes place when the memory block is not held by any cache and its only copy resides in main memory. This is the case of those memory blocks that have not been accessed by any node yet, or those that were evicted in all the caches.

The sharing code field keeps the identity of the L2 caches that hold a copy of the corresponding block. Although our lightweight directory architecture is compatible with any of the sharing codes proposed in the literature, for simplicity we have used the full-map sharing code in this paper. The election for this paper of the full-map sharing code instead of a compressed sharing code is to concentrate on the impact that lightweight directories have on performance, removing any interference caused by unnecessary coherence messages.

4 Coherence Protocol for Lightweight Directory

The proposed architecture requires a cache coherence protocol similar to `MESI` [1] with some minor modifications. These modifications are performed to ensure that for all memory blocks held in one or more L2 caches, directory information is present in the L2 cache of the home node. Moreover, when a memory block is evicted from the home cache, all the copies of this block must be previously invalidated. Next, we detail the modifications that are required.

We use the term local misses to refer to the L2 cache misses that take place in the home node. On the other hand, remote misses imply that the home node is not where the miss occurs. For local misses, the directory controller obtains directory information stored in cache tags, and then, the miss proceeds as usual.

On the other hand, remote misses are sent to the home node, where the directory controller checks the tags part of the L2 cache. If directory information is not in the home cache, the memory block is not cached by any node (the implicit uncached state mentioned above). Hence, the memory controller brings the block from main memory and stores an entry for it at the home cache in invalid state, just to hold directory information. Moreover, the directory state

		Directory States		
		Uncached	**Shared**	**Owned**
Conventional	**Dir. Inf.**	Memory	Memory	Memory
	Data	Memory	Memory	Owner Cache
Lightweight	**Dir. Inf.**	-	Home Cache	Home Cache
	Data	Memory	Home Cache	Owner Cache

Table 1. Where directory information and data are found when a L2 miss takes place in both conventional and lightweight directory protocols.

is set to owned because only one node will hold the copy of the block. Finally, the home node sends the block to the requester. If the directory information is in the home cache is not necessary to access to main memory. Moreover, if the directory state is shared, the home node has a valid copy and it can provide the block immediately if the request is a read one.

When a particular block in shared state is evicted from the L2 cache of its home node, the rest of the copies must first be invalidated to maintain coherence. In this way, the directory controller sends multiple invalidation requests to the sharers. Finally, the replacement proceeds once the home node has all the confirmations of the invalidations. If the evicted block has its directory state as owned, and the home node is not the owner, another node has the only valid copy of the block. Then, the directory controller requests the block to the owner. When the home node receives the block, it updates main memory.

The rest of cases are handled as in a conventional directory coherence protocol. Table 1 summarizes the advantages of our proposal. The lightweight directory avoids going to main memory when the directory state is shared, since the home node provides the block. Moreover, directory accesses in cache-to-cache transfers (owned state) are faster than in conventional architectures since the corresponding directory entry is stored in the L2 cache of the home node. Finally, we do not need directory information for uncached blocks, reducing the amount of extra memory that is required.

5 Simulation Environment

We have used a modified version of RSIM [12], a detailed execution-driven simulator, that our group has ported to the x86 architecture [13]. We have simulated a cc-NUMA system with 32 uniprocessor nodes that implements the lightweight directory protocol. Table 2 shows the parameters used to evaluate the lightweight directory architecture. We model the contention on tags and data cache accesses for the remote requests. In this way, those remote requests that try to access the tags at the same time that another request (local or remote) is in progress, will be delayed. Simulations have been performed using an optimized version of the sequential consistency model with speculative load execution following the guidelines given by Hill [14].

The benchmarks used in our simulations cover a variety of computation and communication patterns. `Barnes` (8192 bodies, 4 time steps), `FFT` (256K complex

32-Node System - Lightweight Directory Protocol			
ILP Processor Parameters		**Memory Parameters**	
Max. fetch/retire rate	4	Memory access time	80 cycles
Instruction window	128	Memory interleaving	4-way
Branch predictor	2 bit agree, 2048 count	**Internal Bus Parameters**	
Cache Parameters		Bus width	8 bytes
Cache block size	64 bytes	Bus cycles	1 cycle
Split L1 I & D caches	16 KB, direct mapped	**Network Parameters**	
	2 hit cycles	Topology	2-dimensional mesh
Unified L2 cache	64 KB, 4-way	Flit size	8 bytes
	15 hit cycles (6 + 9)	Non-data message size	2 flits
Directory Parameters		Router speed	250 MHz
Directory controller cycle	1 cycle (on-chip)	Router's internal bus width	64 bits
Directory access time	6 cycles (L2 cache tag)	Arbitration delay	4 router cycles
Message creation time	4 cycles first, 2 next	Channel bandwidth	2 GB/s

Table 2. Base system parameters.

doubles), `Ocean` (258x258 ocean), `Radix` (1M keys, 1024 radix), and `Water-NSQ` (512 molecules, 4 time steps) are from the SPLASH-2 benchmark suite [15] and `Unstructured` (Mesh.2K, 5 time steps) [16] is a computational fluid dynamics application. All experimental results reported in this work correspond to the parallel phase of these benchmarks. Input sizes have been also chosen commensurate to the total number of processors that are used, and cache sizes have been chosen so that the working set of the applications is greater than their capacity.

6 Simulation Results and Analysis

In this section, we evaluate the performance of lightweight directories in terms of total execution time as well as we analyze the effect that they have on the L2 caches, particularly whether the total number of replacements is increased or instead kept unchanged. We compare our proposal with a conventional directory based cache coherent multiprocessor, similar to the SGI Origin 2000/3000 [4], that uses full-map as the sharing code. Moreover, it is important to know the performance that can offer our proposal in ideal conditions. We called ideal case when the blocks used by a node are not affected by the allocation of remote blocks. That it is, we suppose an infinite cache size for those blocks allocated in the home node due to a request of a remote node, and a normal cache size for its local blocks [2].

Figure 3 shows the execution time for a conventional directory architecture and the ideal and realistic implementation of the lightweight directory architecture. As it can be observed, improvements in performance with the ideal implementation range from 6% to 29%. On the other hand, with the realistic 64KB L2 caches for all the blocks, reductions in terms of execution time are obtained for all the benchmarks except for `Radix`. Particularly, `Ocean` and `Barnes` obtain the most important reductions (20% and 19% respectively) whereas for the other benchmarks the reduction ranges from 4% to 11%. Only for `Radix` application execution time is increased and a degradation of 14% is observed.

Table 3 helps to understand the differences between the ideal and the realistic implementation. Overall, our proposal do not affect cache misses, since due to

[2] The size of the L2 cache for this paper is $64KB$

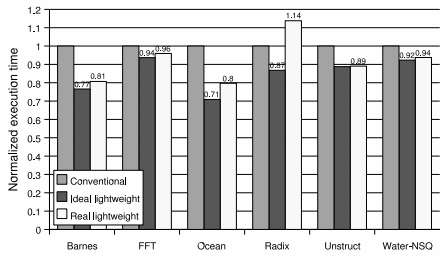

Fig. 3. Normalized execution time for conventional, ideal lightweight and real lightweight architectures.

the temporal locality exhibited by the references to memory it does not cause a significant increment in the L2 cache replacements.

`Unstructured` is the nearest benchmark to the ideal case. This is because it solves almost all the cache misses without accessing to main memory. `FFT` has a small number of replacements in conventional case, so the ideal case only has an improvement of 6% respect to conventional case. Moreover, `FFT` solves half of the misses in home cache and, therefore, the real case is 2% worse that the ideal one. `Barnes` solves almost all the L2 misses in home cache and also obtains a good performance very near to the ideal case. On the other hand, the performance of `Ocean` is a 7% worse than the ideal case because most misses are solved in main memory (75%). In addition, `Ocean` maintains constant the L2 miss rate and, therefore, the number of replacements, which is translated in a considerable improvement in performance (20%) respect to the conventional case. `Water-NSQ` cannot get a very high performance improvement because this benchmark spends just a short time to solve cache misses. Finally, the benchmark `Radix` increases its L2 miss rate and most the misses must often go to main memory to obtain the directory information. This causes that the real case obtains much worse performance than the ideal case. Moreover, the accesses for the directory information to main memory in lightweight directory architecture are greater than in the conventional architecture. Hence, the performance is a 14% worse than conventional case. In the section 1, we demonstrate that the total number of memory blocks that are brought to the L2 cache for a lightweight directory architecture is the same as for the conventional one (figure 1). The

	L2 Miss rate				**L2 Replacements**		
	Conv.	**Lightweight**			**Conv.**	**Lightweight**	**Ratio**
Benchmark	Total	Total	Cache	Memory	Repl / Node	Repl / Node	%
Barnes	0.16	0.17	0.15	0.02	26579	29471	1.11
FFT	0.04	0.04	0.02	0.02	6956	7514	1.08
Ocean	0.16	0.16	0.04	0.12	115957	116911	1.01
Radix	0.11	0.13	0.01	0.12	38575	56190	1.46
Unstruct	0.38	0.39	0.38	0.01	42116	43851	1.04
Water-NSQ	0.20	0.20	0.08	0.12	13348	13509	1.01

Table 3. L2 cache miss rate and replacement for conventional and lightweight architecture. In a lightweight architecture miss rate is separated into misses that found directory information in home cache or in home main memory.

only difference is in the order in which the blocks are allocated in L2 caches, so this case can be improved using other cache allocation policies.

Regarding the directory memory overhead, our proposal improves the scalability of the directory size by reducing the number of directory entries in a R ratio, where R is defined as the quotient between the main memory size and L2 cache size. According to current multiprocessors such as SGI Origin 2000/3000 [4] and AlphaServer GS320 [17], R can take a typical value of 1024, hence the memory reduction is very considerable.

7 Conclusions and Future Work

In this paper we have introduced the lightweight directory architecture, a scalable directory protocol that tries to achieve the best characteristics both of the snooping and of the directory-based protocols. Our proposal is based on current technology improvements to put the directory controller and directory information inside the processor chip. In this way, we remove the directory structure from main memory and we associate directory information to the L2 cache. Then, cache misses are satisfied by home node cache without accessing main memory, ever when some node has a valid copy of the block.

We have described the resulting architecture and a coherence protocol suited to the particularities of the architecture. In order to demonstrate the benefits derived from our proposal in terms of execution time, we have run several scientific parallel applications on top of a RSIM version that implements the lightweight directory protocol. The lightweight directory architecture presented in this paper obtains improvements of up to 20% in execution time compared to conventional architectures. Moreover, directory memory overhead is reduced by a R ratio respect to conventional directory architectures, where R is computed as the quotient between main memory size and L2 cache size. This means that our proposal drastically reduces the directory memory overhead, and in most cases improves performance.

As part of our future work, we plan to design a cache allocation algorithm, which only stores in cache some remote blocks for reducing the memory overhead caused by these blocks. Another area of interest is to study the impact of a victim cache for blocks whose replacements cause coherence actions. These blocks are those that maintain directory information in the home node cache and they have a copy in some remote node. In this way, it would not be necessary to performance coherence actions. Finally, in order to reduce even more directory memory overhead, we would like to evaluate the effect of limited pointers or compressed sharing codes.

Acknowledgments

This work has been supported by the Spanish Ministry of Ciencia y Tecnología and the European Union (Feder Funds) under grant TIC2003-08154-C06-03.

References

1. Culler, D., Singh, J., Gupta, A.: "Parallel Computer Architecture: A Hardware/Software Approach". Morgan Kaufmann Publishers, Inc. (1999)
2. Acacio, M., González, J., García, J., Duato, J.: "An Architecture for High-Performance Scalable Shared-Memory Multiprocessors Exploiting On-chip Integration". IEEE Transactions on Parallel and Distributed Systems **15** (2004) 755–768
3. Acacio, M., González, J., García, J., Duato, J.: "A Two-Level Directory Architecture for Highly Scalable cc-NUMA Multiprocessors". IEEE Transactions on Parallel and Distributed Systems **16** (2005) 67–79
4. Laudon, J., Lenosky, D.: "The SGI Origin: A cc-NUMA Highly Scalable Server". Proc. of the 24th Int'l Symposium on Computer Architecture (ISCA'97) (1997) 241–251
5. Gwennap, L.: "Alpha 21364 to Ease Memory Bottleneck". Microprocessor Report **12** (1998) 12–15
6. Ahmed, A., Conway, P., Hughes, B., Weber, F.: "AMD Opteron^{TM} Shared Memory MP Systems". Proc. 14th HotChips Symposium (2002)
7. Martin, M., Sorin, D., Ailamaki, A., Alameldeen, A., Dickson, R., Mauer, C., Moore, K., Plakal, M., Hill, M., Wood, D.: "Timestamp Snooping: An Approach for Extending SMPS". Proc. of the 9th Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX) (2000) 25–36
8. Martin, M., Sorin, D., Hill, M., Wood, D.: "Bandwidth Adaptive Snooping". Proc. of the 8th Int'l Symposium on High Performance Computer Architecture (HPCA-8) (2002) 251–262
9. Martin, M., Hill, M., Wood, D.: "Token Coherence: Decoupling Performance and Correctness". Proc. of the 30th Int'l Symposium on Computer Architecture (ISCA'03) (2003) 182–193
10. Gupta, A., Weber, W., Mowry, T.: "Reducing Memory Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes". Proc. Int'l Conference on Parallel Processing (ICPP'90) (1990) 312–321
11. Mukherjee, S., Hill, M.: "An Evaluation of Directory Protocols for Medium-Scale Shared-Memory Multiprocessors". Proc. of the 8th Int'l Conference on Supercomputing (ICS'94) (1994) 64–74
12. Hughes, C., Pai, V., Ranganathan, P., Adve, S.: "RSIM: Simulating Shared-Memory Multiprocessors with ILP Processors". IEEE Computer **35** (2002)
13. Fernández, R., García, J.: "RSIMx86: A Cost Effective Performance Simulator". Proc. of the High Performance Computing & Simulation (HPC&S) Conference (2005)
14. Hill, M.: "Multiprocessors Should Support Simple Memory-Consistency Models". IEEE Computer **31** (1998) 28–34
15. Woo, S., Ohara, M., Torrie, E., Singh, J., Gupta, A.: "The SPLASH-2 Programs: Characterization and Methodological Considerations". Proc. of the 22nd Int'l Symposium on Computer Architecture (ISCA'95) (1995) 24–36
16. Mukherjee, S., Sharma, S., Hill, M., Larus, J., Rogers, A., Saltz, J.: "Efficient Support for Irregular Applications on Distributed-Memory Machines". Proc. of the 5th Int'l Symposium on Principles & Practice of Parallel Programming (PPOPP'95) (1995) 68–79
17. Gharachorloo, K., Sharma, M., Steely, S., Doren, S.V.: "Architecture and Design of AlphaServer GS320". Proc. of the 9th Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX) (2000) 13–24