Available online at www.sciencedirect.com

SCIENCE DIRECT®

Journal of Systems Architecture 51 (2005) 451–469

JOURNAL OF SYSTEMS ARCHITECTURE

www.elsevier.com/locate/sysarc

Exploring the performance of split data cache schemes on superscalar processors and symmetric multiprocessors ☆

J. Sahuquillo [a,*], S. Petit [a], A. Pont [a], V. Milutinović [b]

[a] *Department of Computer Systems, Polytechnic University of Valencia, Cno de Vera s/n, 46022 Valencia, Spain*

[b] *Department of Computer Engineering, School of Electrical Engineering, University of Belgrade, PO Box 35-54, 11120 Belgrade, Serbia, Yugoslavia*

Received 25 August 2003; received in revised form 26 May 2004; accepted 1 December 2004

Available online 25 February 2005

Abstract

Current technology continues providing smaller and faster transistors, so processor architects can offer more complex and functional ILP processors, because manufacturers can fit more transistors on the same chip area. As a consequence, the fraction of chip area reachable in a single clock cycle is dropping, and at the same time the number of transistors on the chip is increasing. However, problems related with power consumption and heat dissipation are worrying. This scenario is forcing processor designers to look for new processor organizations that can provide the same or more performance but using smaller sizes. This fact especially affects the on-chip cache memory design; therefore, studies proposing new smaller cache organizations while maintaining, or even increasing, the hit ratio are welcome. In this sense, the cache schemes that propose a better exploitation of data locality (bypassing schemes, prefetching techniques, victim caches, etc.) are a good example.

This paper presents a data cache scheme called filter cache that splits the first level data cache into two independent organizations, and its performance is compared with two other proposals appearing in the open literature, as well as larger classical caches. To check the performance two different scenarios are considered: a superscalar processor and a symmetric multiprocessor.

The obtained results show that (i) in the superscalar processor the split data caches perform similarly or better than larger conventional caches, (ii) some splitting schemes work well in multiprocessors while others work less well because of data localities, (iii) the reuse information that some split schemes incorporate for managing is also useful for designing new competitive protocols to boost performance in multiprocessors, (iv) the filter data cache achieves the best performance in both scenarios.

© 2005 Elsevier B.V. All rights reserved.

☆ This work has been partially supported by the Generalitat Valenciana under Grant GV04B/487.

* Corresponding author. Tel.: +34 96 387 75 79; fax: +34 96 387 70 07.

E-mail address: jsahuqui@disca.upv.es (J. Sahuquillo).

1383-7621/$ - see front matter © 2005 Elsevier B.V. All rights reserved.

doi:10.1016/j.sysarc.2004.12.002

Keywords: ILP; Superscalar processors; Split data cache; Multiprocessors; Competitive coherence protocols; Performance evaluation

1. Introduction

Continuous advances in transistor technology bring two main consequences: (i) faster transistor switching time, and (ii) larger scale of integration. Both consequences have an important impact on architecture design. With respect to the first one, technology projections [1] predict that feature sizes will drop so reducing the fraction of the total chip area reachable in a single cycle. This fact will dramatically impact on architectural component latencies measured in processor cycles. For instance, a 4 KB cache will require three cycles with a 10 GHz clock [2] so technology constrains the cache geometry (mainly cache size and number of ways) in current and future microprocessors. In this sense, the Pentium 4 microprocessor [4] requires two processor clocks to access to its L1 data cache, whose 8 KB storage size is half the size of its predecessor Pentium III [3]. The second consequence permits current microprocessors to include two or more cache levels on-chip (e.g., the Itanium 2 [5] implements three levels) as the common adopted solution to reduce the latencies of the memory hierarchy. Despite these facts, the first level cache has changed little over the past two or three decades. This paper focuses on this level because it has a major impact on processor performance, and consequently, it has been the focus of many research efforts.

An important piece of research on cache performance has focused on splitting the first level data cache in two or more independent organizations in order to improve performance [7,8,10–12]. By using several organizations at the first level, data can be classified according to the behavior shown, and a given organization can be devoted to storing a particular data group. By proceeding in this way, each cache (organization and management) can be tuned to the data characteristics it stores, in order to maximize performance. For instance, if one cache is targeted to store data blocks exhibiting spatial locality then such a cache could have a large line size. In addition, these schemes can work faster because the first-level cache storage is usually split in smaller organizations.

The idea of using split data caches has been widely explored in the literature [14]. Because caches base their efficiency on the exploitation of reference locality (temporal and spatial) that data exhibit, a representative subset of proposals split the cache, and consequently classify data to be stored, according to this criterion. Unfortunately, despite the fact that this criterion has had a good acceptance among the research community, empirical results show that less than half of the data brought into the cache gets used before eviction therefore showing no locality [13]. Some of the proposals to split the first-level cache in the literature introduced extra hardware to move data blocks from one cache to the other, but no additional information about the block behavior is saved. For instance, the victim cache [15] exploits temporal behavior but no data is labeled as temporal. Other proposals classify and label data blocks at run time according to the observed dynamism in the data access patterns using additional hardware to save block behavior related information. These schemes make latter use of this information, called then reuse information (discussed in Section 2). This paper focuses on those schemes managing reuse information to improve cache performance. In [17] we introduced the filter data cache scheme and explore its performance in a simple single-issue microprocessor. Performance was compared against two other schemes that split the cache according to the criterion of the data locality (the NTS [7] and the SSNS [8]). Since advances in technology permit much more complex cache design than in previous cache generations, in [30] we analyzed the impact on performance of split data caches on a state-of-the-art superscalar processor, where modern cache features were also taken into account. For instance, non-blocking caches enable multiple memory references to be outstanding, and multiple ports enable several memory references to be satisfied in the same processor clock.

Nowadays, the use of symmetric multiprocessors begins to be familiar in certain scenarios (scientific labs, web servers, database servers, etc.) and it is quite easy to find computers with 2, 4, or 8 processors. Therefore, it would be worth that a new cache organization should also be suitable for this kind of system.

In this paper we extend the work in [30] in several ways. We investigate the use of split data caches in multiprocessor systems and analyze their impact on performance. Due to the key characteristic of multiprocessors is the mechanism for coherence maintenance (that directly impacts on system performance), we also discuss how the reuse information can help boost the performance of coherence protocols. Finally, we use a larger set of benchmarks (i.e., SPEC2000). Results show that: (i) in general, split data cache schemes achieve better performance in both superscalar processors and multiprocessors than conventional cache organizations by using less die area, and (ii) among the analyzed schemes the filter data cache we propose achieves better performance in both kinds of systems.

The rest of this paper is organized as follows. Section 2 discusses the related work. Section 3 describes the proposed scheme. Section 4 analyzes the performance results in superscalar processors. Section 5 analyzes the impact on performance of these proposals when used in multiprocessor systems, as well as discusses how to design competitive coherence protocols. Finally, conclusions are summarized in Section 6.

2. Related work

One of the earlier attempts to split the cache, extensively referenced in the literature, has been the victim cache proposed by Jouppi in [15]. This proposal includes a large direct-mapped cache joined to a small fully associative cache. The goal is for the smaller cache (buffer) to retain the most recent conflict lines that do not fit in the largest or main L1 cache (from now on, the largest cache will be referred to as the main cache). The goal behind this working mode is to exploit the temporal locality that data exhibits. Another proposal, implemented in a commercial processor, is the Assist Cache of Chan et al. [16], where the idea is to have a large space for a large amount of referenced data and another smaller cache to help the large one, by reducing conflict misses.

To avoid introducing pollution into the cache, some schemes such as the CNA proposed by Tyson et al. [6], chose bypassing those cache lines that are infrequently referenced. Other schemes, like the Memory Address Table (MAT) [9], proposed by Johnson and Whu, propose storing these lines in a small bypass buffer.

The Allocation by Conflict (ABC) scheme proposed by Tam [29] is a similar solution to the victim cache but it tries to take replacement decisions based on the behavior of the conflict block allocated in the main subcache (under the term subcache we refer to a specific organization located in the first level). For this purpose, it gathers information about the block behavior (just one bit) to be used while the block is in cache (current information), but in this scheme this information is reset when the block is evicted, so there is no information about the block behavior when it is fetched again.

An interesting idea exploited among many proposals appearing in the open literature has been the use of reuse information to improve performance. For this purpose, the current information attached to a block must be saved when the block is evicted from the first-level cache. This information can be stored in a decoupled hardware structure or attached to the corresponding L2 cache line (see Fig. 1). When a block is evicted from the L1 data cache, a later reference to such a block will result in a miss and the previously stored information, i.e., reuse information, is used for subsequent decisions about where (i.e., in which subcache) a data block must be placed.

Among the schemes exploiting reuse information appearing in the literature, the criterion of data locality has been widely exploited. Spatial locality assumes that when a memory location is referenced, there is a high probability that a neighboring location will be also accessed. Temporal locality assumes that if a memory location is referenced, there is a high probability that the same location will be accessed again. Two earlier

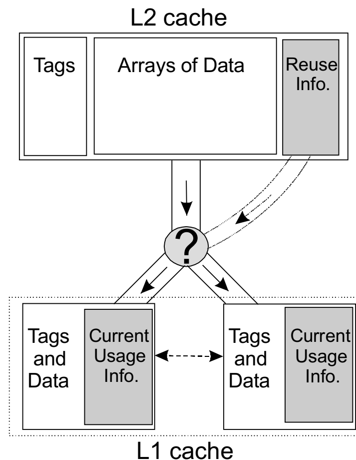

Fig. 1. Block diagram of the placement related decision process in a split data cache.

attempts for handling spatial and temporal localities in separate caches were the dual data cache (DDC) model introduced in [10] by Gonzalez et al., and the split temporal spatial model (STS) proposed by Milutinovic et al. in [11].

However, both localities can appear together or not appear at all. This fact is considered in other schemes, such as the non-temporal streaming cache (NTS) proposal of Rivers and Davidson in [7] where the data cache is split by giving priority to the temporal locality. In contrast, a scheme giving priority to the spatial locality called the split spatial non-spatial cache (SS/NS) is proposed by Prvulovic et al. in [8]. Another latter option is to consider three different caches, such as the locality sensitive module (LSM) scheme of Sanchez and Gonzalez presented in [12].

As mentioned above, in order to check the performance of our proposal we selected two schemes managing reuse information and that split the cache according to the criterion of data locality, the NTS and the SS/NS. Despite the fact that these schemes use different criteria to divide data in the first level of memory hierarchy they are quite similar in hardware and organization to our proposal (see Section 4.4), which make them suitable for comparison purposes. In order to better understand the performance comparison study below we detail both schemes.

In the NTS scheme, data is dynamically tagged as temporal or non-temporal. The model includes a large temporal cache placed in parallel with a small non-temporal cache. Each block in the temporal cache has a reference-bit array attached, in addition to a non-temporal (NT) bit. When a block is placed in the temporal cache, each bit in the reference bit array is reset; and the NT bit is set. Then, if a hit occurs in this cache, the bit associated with the accessed word is set. If that bit was already set (i.e., that the word had already been accessed), the NT bit is reset to indicate that the block is showing temporal behavior. When a block is removed from the first level cache, its NT bit is flushed to the second level. Then, if the block is referenced again, this bit is checked to decide where (i.e., in which subcache) the block must be placed.

The SS/NS scheme makes a division between spatial and non-spatial data blocks, giving priority to those blocks exhibiting spatial locality. The model introduces a large spatial cache in parallel with a non-spatial cache that is four times smaller. The spatial cache exploits both types of spatial locality (only spatial, or both spatial and temporal). The block size in the non-spatial cache is just one word; thus, only temporal locality can be exploited in this cache. In the spatial cache, the block size is larger than four words. The spatial cache uses a prefetch mechanism to assist this type of locality. A hardware mechanism is introduced to recompose blocks in the non-spatial cache and moves them (by a unidirectional data path) to the spatial cache. This proposal uses a reference bit array similar to that discussed for the NTS scheme. Blocks are tagged as spatial if more than two bits are set; otherwise, they are tagged as non-spatial.

3. Proposed caching technique: the filter data cache

In the strictest worst case, one can affirm that if one data item is twice referenced while the block is in cache, and no other data item in the same block is referenced, then the block exhibits temporal locality. Analogously, in the strictest worst case spatial locality is exhibited if only two words are

referenced just once. Logically, these two extreme cases introduce pollution in the cache. The ideal situation would be that a data block exhibits both data localities. In such a case, in the strictest worst case the block must be referenced at least three times (two for temporal, and once more for spatial). To identify and cache these blocks we talk about the most referenced ones, labeled as "High Frequency" in Fig. 2. Notice that these blocks could be easily detected by using a hardware counter attached to each block in cache. Although, a high counter value could mean that the block exhibits only one locality (e.g. temporal), there is a high probability that the block exhibits both localities.

The filter cache (FC) model we propose presents a very small "filter" cache working in parallel with a larger classic "main cache". The idea behind this scheme is to place in a small cache the most heavily referenced blocks. The scheme tries to identify these blocks and to cache them together. To this end, each cache block incorporates an associated counter.

When the execution of a memory reference instruction hits any subcache, the correspondent block counter is increased. If the access results in a miss in both subcaches, the counter of the referenced block (ctrmissed) is compared with the counter of the conflict block in the filter cache (ctrfilter), in order to decide in which subcache the missed block must be placed. If ctrmissed is greater or equal to the ctrfilter, then the missed block is placed in the filter cache; that is because the model assumes that the missed block is more likely to be referenced again than the conflict block in the filter cache. Otherwise, the missed block is placed in the main cache.

When a block is placed in the filter cache it is because it was highly referenced; therefore, it seems worthwhile keeping it longer in the first level when it is replaced from the filter subcache. For this reason, blocks move to the main cache using a unidirectional datapath instead of leaving the first level. The counter value (four bits) is the only information that is saved to be reused later when blocks are evicted from the first level cache. Fig. 3 shows a block diagram of the scheme.

Data blocks can dynamically change their localities along the program execution. The block counter values must be decremented from time to time, to avoid blocks showing good locality during just an interval of its execution remain in the filter cache. For instance, if a block shows good locality during a certain period of time it will saturate its counter. If that counter is not decremented, the block will always be placed in the filter cache, even if it shows poor or no locality during the remainder of the execution.

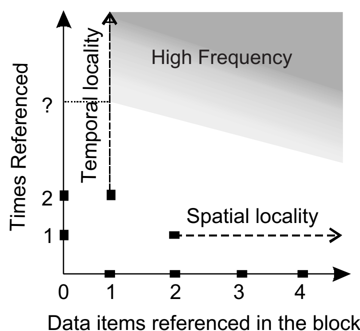

Fig. 2. Temporal locality, spatial locality, and high frequency.

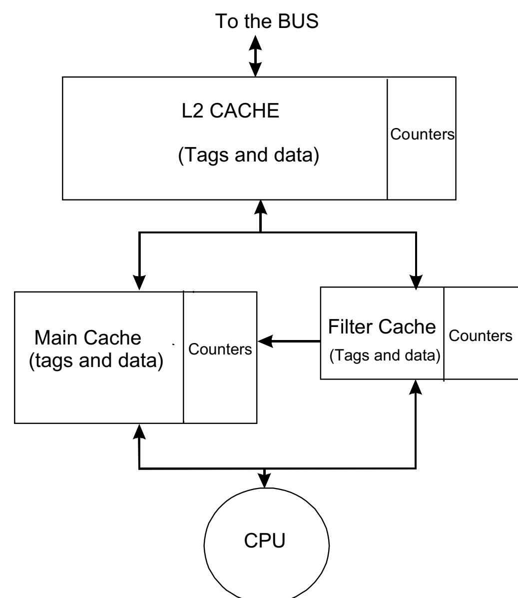

Fig. 3. The filter data cache scheme.

4. The use of split data caches in superscalar processors

As the detailed schemes are a particular kind of cache memory organization, they could be implemented in all systems incorporating caches, which are practically all the systems manufactured nowadays, such as superscalar processors, multi-threaded processors, SMP systems, etc.

Data localities vary among the kind of workload executed in the different systems, e.g., the localities presented by parallel workloads running in SMP systems are not the same as those presented by sequential workloads running in a superscalar processor.

This feature means that the effectiveness of data cache schemes will not remain homogeneous among the different systems but depend both on the system architecture and the data localities of the running workload. In this section we check the performance of the selected data cache schemes on a superscalar state-of-the art processor.

4.1. Experimental framework and workload

To run our experiments we use both the Simplescalar [20] and the mlcache [19] simulators. The Simplescalar tool set provides an execution-driven simulator that models an out-of-order superscalar processor. Its source code simulates a classical six stage pipeline processor: fetch, decode, issue, execute, writeback and commit. This code can be modified to check the behavior of new microarchitectural proposals. The simulator includes specific modules to deal with the memory subsystem (cache.c and cache.h) which are linked to other modules to run the selected benchmarks. The mlcache cache simulator focuses on cache organizations so that cache schemes can be implemented avoiding other micro architecture details. Once cache proposals are implemented, this simulator can be integrated with the out-of-order module provided by the Simplescalar by simply replacing the cache modules. Proceeding in this way, we check the performance of different cache proposals on superscalar processors.

Experiments have been carried out by using the SPEC95 and SPEC2000 benchmark suites. Table 1 shows the selected benchmarks for this study, the number of instructions issued, and the number of loads and stores committed, as well as their percentage over the total instructions.

4.2. Superscalar processor architecture model

Simulation parameters must be chosen according to the technology under study. To select the most important characteristics of the modeled processor, we have considered the features of some current processors [21,23,3]. As we are interested in learning how these schemes impact on a hypothetical future microprocessor, the selected parameters have been inflated. Table 2 shows the characteristics of the modeled processor.

Table 1
Workload characteristics

Suite	Benchmark	Input data	# Instructions	# Loads and stores (%)
SPEC95	099.go	9 9	1,500,000,001	18.19
	126.gcc	training	252,987,341	37.06
	129.compress	training	35,683,369	37.43
	130.li	training	183,708,317	42.43
	132.ijpeg	training	24,506,886	29.19
	134.perl	jumble	1,500,000,001	44.44
	134.perl	primes	10,510,668	44.73
	134.perl	scrabbl	40,482,647	45.71
SPEC 2000	164.gzip	combined	1,500,000,000	34.60
	176.gcc	cp-decl	1,500,000,000	41.05
	197.parser	train.in	1,500,000,001	35.74
	255.vortex	lendian.raw	1,500,000,000	54.17

Table 2
Parameters of the processor and memory hierarchy model

Machine unit	Component	Parameter value
Processor	Functional units	6 ALUs, 4 FPUs
	Issue width	8 instructions per cycle
	Issue mechanism	Out-of-order
	Instructions latencies	Same as those of MIPS R10000
	Register update unit size	256
	Branch prediction scheme	Perfect
Cache hierarchy	L1 main cache	8 KB, direct mapped, write-back, write-allocate, non-blocking, 4 ports, 32-byte line, 1 cycle hit
	L1 assistant	1 KB, fully associate, write-back, write-allocate, non-blocking, 4 ports, 32-byte line,1 cycle hit
	L2 cache	Infinite cache, 22 cycles
	TRI	128 entries

Due to the small working set of the chosen benchmarks, the selected cache sizes do not match current sizes. However, they are adequate for comparison purposes [18,12].

4.3. Simulation results

In this section we compare the performance of the three schemes that split the first level cache. In addition, we include the results of two conventional caches; the first one is as large as the main subcache (8 KB), and the second one is twice as large (16 KB) as the main subcache.

To evaluate the effectiveness of the small assist cache in the three schemes we measure the hit ratio in these small caches. Table 3 shows the results. The NTS and SSNS schemes show a hit ratio of about 20% while the filter scheme doubles this ratio (about 45%).

Accesses to a non-blocking cache in ILP processors may result in one of the following: a hit, a miss, or a delayed hit. A delayed hit is "virtual hit" if a returning miss is on its way back to the cache. That means that the hit ratio in non-blocking caches is not as decisive as it is in blocking caches, where its value indicates which scheme performs better. Thus, it is necessary to use other complementary performance indexes. In the results shown in Table 3 the virtual hits have been omitted.

Table 4 shows the hit ratio in L1 in the split schemes and two conventional organizations when using 8 and 16 KB caches. The NTS and filter schemes with only 9 KB data storage capacity in L1 achieves hit ratios higher than a conventional scheme with 16 KB of cache (almost double size). The SS/NS is the splitting scheme that works worst, but it performs better than the conventional organization.

Table 3
Hit ratio obtained using 1 KB cache organized according to the specifications of the NTS, the SSNS and the filter cache schemes

Suite	Benchmark	NTS	SSNS	Filter
SPEC95	099.go	0.47	49.97	50.50
	126.gcc	18.99	14.18	45.00
	129.compress	23.25	19.58	62.07
	130.li	11.16	17.00	33.46
	132.ijpeg	16.50	17.65	64.78
	134.perl jumble	15.56	13.64	30.84
	134.perl primes	29.22	15.97	53.06
	134.perl scrabbl	18.95	17.17	32.48
SPEC 2000	164.gzip	23.46	30.09	50.24
	176.gcc	19.26	14.32	37.65
	197.parser	19.15	18.85	36.92
	255.vortex	28.64	27.20	48.16
Average		18.72	21.30	45.43

Table 5 shows the IPC of all modeled schemes. On the average, the filter and NTS schemes achieve the best IPC, even better than the 16 KB conventional cache.

Fig. 4 shows the speedup obtained by the three splitting schemes and the conventional 16 KB cache with respect to a 8 KB conventional cache.

Table 4
L1 hit ratio for the three split schemes (8 KB + 1 KB) and two 8 KB and 16 KB conventional organizations

Suite	Benchmark	8 KB	NTS	SSNS	Filter	16 KB
SPEC95	099.go	93.355	93.49	93.73	93.73	93.51
	126.gcc	79.868	87.41	84.69	88.80	87.06
	129.compress	73.605	75.97	75.97	76.72	75.90
	130.li	83.992	86.88	86.72	87.49	87.88
	132.ijpeg	87.315	89.19	88.89	91.86	91.79
	134.perl jumble	88.875	92.57	92.36	94.37	91.81
	134.perl primes	85.524	96.78	94.04	97.53	90.52
	134.perl scrabbl	84.310	90.90	89.96	91.61	91.32
SPEC 2000	164.gzip	75.064	76.27	76.42	76.79	76.71
	176.gcc	69.057	74.63	74.24	75.53	74.75
	197.parser	82.070	87.183	86.61	87.89	86.16
	255.vortex	75.128	89.08	88.05	89.35	88.38
Average		81.51	86.70	85.97	87.64	86.32

Table 5
IPC achieved in the three split schemes in comparison with two traditional caches of 8 KB and 16 KB each

Suite	Benchmark	8 KB	NTS	SSNS	Filter	16 KB
SPEC95	099.go	3.31	3.33	3.32	3.32	3.31
	126.gcc	3.85	4.02	4.00	4.04	4.12
	129.compress	3.94	4.07	4.06	4.11	4.14
	130.li	4.17	4.24	4.24	4.26	4.31
	132.ijpeg	5.02	5.08	5.07	5.11	5.12
	134.perl jumble	4.22	4.44	4.45	4.63	4.41
	134.perl primes	4.23	4.85	4.59	5.00	4.61
	134.perl scrabbl	3.68	4.18	4.07	4.22	4.17
SPEC 2000	164.gzip	3.18	3.22	3.23	3.23	3.25
	176.gcc	3.17	3.29	3.29	3.31	3.37
	197.parser	2.86	3.02	3.01	3.03	3.06
	255.vortex	3.61	4.20	4.16	4.15	4.21
Average		3.77	4.00	3.96	4.03	4.01

Notice that all the splitting schemes perform closer and sometimes better than a conventional cache with almost double capacity. This happens because movements of blocks prolong the stance of the block in the first level cache where accesses are quicker. On the other hand, some slow speedups, like those achieved when using the benchmark go, cannot be stored in only 9 KB because of their working set.

From the total simulation time (measured in processor cycles) used to calculate the speedups, we estimated the theoretical capacity of a conventional cache that would obtain the same performance as the splitting cache schemes. To do this, we assumed a linear relation between performance and capacity. In other words, using the 8 and 16 KB caches, we have estimated where the performance of the splitting data cache schemes would be on the line. Table 6 shows the results. The filter data cache achieves the best performance, obtaining with just 9 KB similar performance to a hypothetical 18 KB data cache.

4.4. Hardware cost

In this section we estimate the extra hardware cost needed to implement the split schemes with respect to a conventional cache. We assume

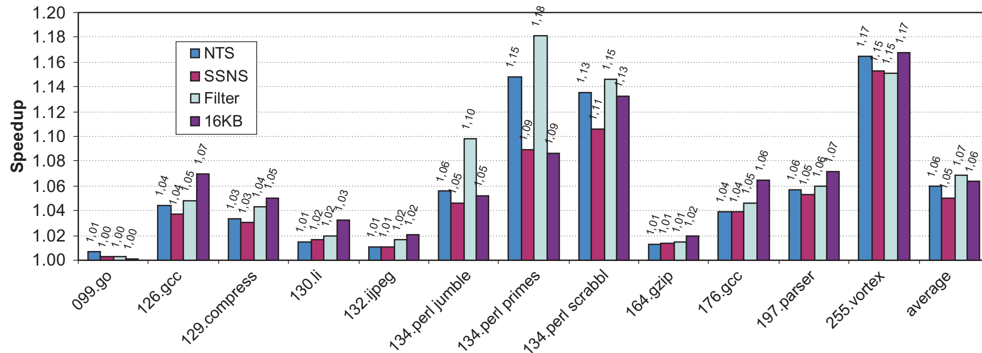

Fig. 4. Speedups achieved by the three split data cache schemes with respect to the conventional 16 KB cache.

Table 6
Theoretically equivalent conventional cache capacities

Suite	Benchmark	NTS	SSNS	Filter
SPEC95	099.go	49	27	27
	126.gcc	13	12	14
	129.compress	13	13	15
	130.li	12	12	13
	132.ijpeg	12	12	15
	134.perl jumble	17	15	22
	134.perl primes	21	16	23
	134.perl scrabbl	16	15	17
SPEC 2000	164.gzip	24	27	34
	176.gcc	13	14	14
	197.parser	13	13	14
	255.vortex	14	14	15
Average		17	15	18

Each value means the theoretical capacity of the conventional cache needed to achieve the same performance as the corresponding split data cache scheme.

a simplified model ignoring the hardwired control costs so that our results only concentrate on data storage, reuse information, and tags.

Assuming a 32 bits address bus, a 16 KB conventional direct mapped conventional cache that uses a 32 bytes block size (eight words), the cost incurred by tags are 18 bits per block and that supposes a total of 1.125 KB. These values are shown in Table 7.

For the split schemes, we consider a 8 KB direct mapped main cache, therefore we need 19 bits to tag each of the 256 blocks and that supposes a total tag cost of 608 bytes. The hardware costs of the splitting schemes differ slightly. This is the reason why we only show one of them. For illustrative purposes we select the cost of the filter cache. To estimate the counter cost we assume a 4-bit counter size attached to each cache block, and that supposes a total of 128 bytes. For the 1 KB fully associative filter cache we need 27 bits for a tagging its 32 blocks, that means a total of 108 bytes plus 16 bytes used to implement the associated counters. In both memories the cost incurred by the hardwired control and replacement algorithms have been omitted. Table 8 shows these values.

Table 7
Data and tag hardware cost for a direct mapped conventional 16 KB cache

Data cost (bytes)	Tags (bits)	Total tags (bytes)	Total cost (bytes)
16 K	18	1.125 K	17.125 K

Therefore, the splitting scheme, i.e. the filter in this case, requires much less cost, by about 57%, than the conventional larger cache (which practically doubles its data storage capacity) while achieving similar performance. Of course, in a more precise study it would be necessary to consider the cost incurred by the control logic, connections, data paths, etc.

5. The use of split data caches in multiprocessor systems

Advances in technology have meant that high performance microprocessors can be easily used

Table 8
Data, tags, and counter hardware cost for the filter scheme

Cache	Data cost (KB)	Tag (bits)	Total tags (bytes)	Counters cost (bytes)	Total approx. cost (KB)
Filter	1	27	108	16	1.12
Main	8	19	608	128	8.72
				Total	9.84

in symmetric multiprocessors (SMPs). Examples can be found in commercial products like the AMD Athlon and Intel Xeon for multiprocessor servers, which include two and four processors, respectively. This trend continues with the AMD Opteron processor for multiprocessor servers, which incorporate up to eight processors connected through an irregular network topology [22].

As a consequence of this current trend, SMP systems represent a large segment of commercial products; therefore, it would be worthwhile checking the impact on performance of the split data cache schemes in SMPs. In these systems, performance can change because of given conditions such as coherence problems and false sharing. In this section we explore the impact on performance of the filter and NTS schemes with respect to conventional caches.

In addition to the filter cache, we selected just the NTS—because as shown in Section 4.3 the NTS scheme performs better than the SSNS.

Fig. 5 shows a block diagram for a symmetric multiprocessor using a split data cache in the first level. A common second level cache connected to the memory bus takes care of coherence maintenance.

5.1. Experimental framework and workload

Performance results have been obtained using the LIMES [24] execution-driven simulator using five SPLASH-2 [25] benchmarks (FFT, Radix, FMM, LU and Barnes). The problem size of the selected benchmarks always exceeds 60 million memory operations.

We compile the running benchmarks with a modified version of GCC v2.6.3, applying the O2

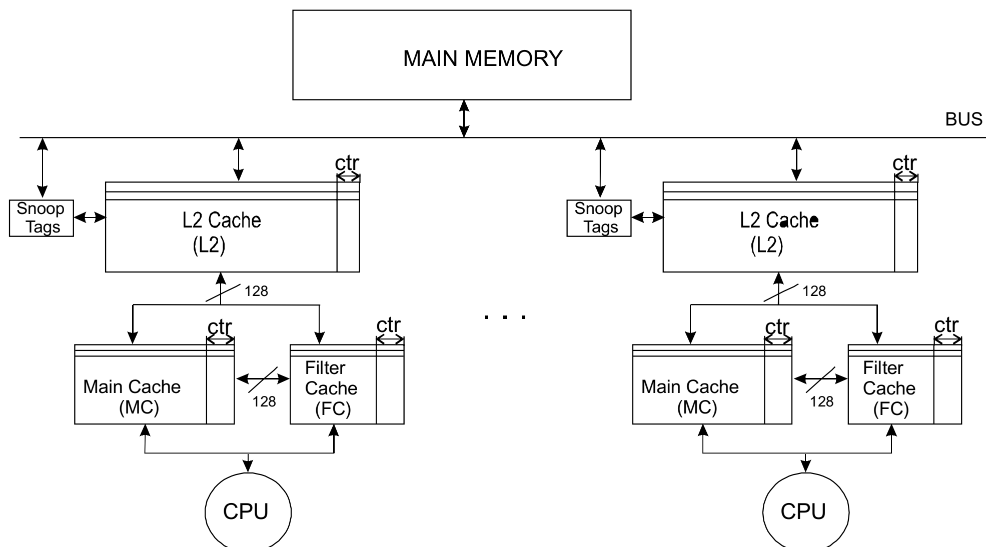

Fig. 5. Multiprocessor system block diagram with a split data cache in L1.

optimization flag. LIMES modifies the compiler in order to instrument memory accesses by adding augmentation code that calls the memory simulator after each memory reference. The synchronization operations can also be trapped by redefining the ANL macros (used by the SPLASH applications) to memory simulator calls.

5.2. SMP system model

All the cache memories in the hierarchy are two-way set associative with 32 bytes (eight words) block size. The first level includes a 16 KB main cache and an assistant or filter cache of only 2 KB, which means 18 KB memory data storage.

In this study we consider two conventional caches in L1 of 16 KB and 32 KB data storage, which are used to estimate the theoretically equivalent caches as in Section 4.3. Therefore, we present also results for two conventional schemes using caches in L1 of 16 and 32 KB, respectively. When a miss occurs in the first level cache the cache controller applies the non-write allocate policy.

We assume a 256 KB L2 cache connected to L1 through a 256 bits bandwidth bus. A hit in any L1 cache takes 1 cycle and 10 cycles in L2. Data transferences between L1 take 4 cycles for the first word if it comes from L2 and 16 cycles if it comes from main memory, the remaining words can be transferred in 1 cycle each one. For coherence purposes, the system implements the Berkeley write-invalidate protocol [26].

5.3. Simulation results

To compare the performance obtained by the different schemes, the performance index used is the Tour of a Line. The tour of a line or block is defined in [18] as the interval of time from when the line or block is placed in the first-level cache until it is evicted from that level. The tour length is measured as the mean number of accesses that result in hits per tour. This parameter and the number of tours are used to evaluate the effectiveness of data cache management. A good cache management will offer a low number of tours and a high tour length.

Table 9 shows the number of tours in a system with 1, 2, and 4 processors when using a conventional cache of 16 KB. Notice that the number of tours is quite similar independently of the number of processors in the system.

Table 10 shows the percentage of reduction in tours offered by the filter and NTS schemes and the conventional 32 KB cache, with respect to the 16 KB conventional cache. A negative value means that the scheme increases the number of tours with respect to the baseline scheme.

Results show that both schemes splitting the cache present a significant reduction in the number of tours. The 18 KB storage of the filter cache offers significant improvements in three of the six benchmarks used (Barnes, FMM and LU). In

Table 9
Number of tours when using a conventional cache memory of 16 KB

Benchmark	1 Processor	2 Processors	4 Processors
Barnes	2,015,920	2,011,519	2,156,089
FMM	887,092	876,865	887,142
LU	2,353,847	2,360,376	2,345,583
FFT	1,433,600	1,434,839	1,431,357
RADIX	1,028,906	1,036,306	1,043,864

Table 10
Percentage of reduction in tours offered by the splitting schemes and the 32 KB cache with respect to the 16 KB conventional cache

Benchmark	FILTER			NTS			32 KB		
	1 p.	2 p.	4 p.	1 p.	2 p.	4 p.	1 p.	2 p.	4 p.
Barnes	31.93	31.87	30.43	23.01	22.98	27.68	59.46	54.68	61.85
FMM	20.46	19.89	19.50	10.17	−7.65	−11.33	35.28	36.27	36.49
LU	7.74	11.62	19.12	3.48	3.68	4.19	3.75	3.85	4.27
FFT	2.13	2.20	2.00	2.34	2.52	2.47	12.27	12.24	11.56
RADIX	0.22	0.28	0.18	0.20	0.19	0.17	0.26	0.27	0.27
Average	15.27	13.60	14.27	19.29	6.23	6.00	18.53	17.91	19.07

the FMM and LU benchmarks, the reduction that the filter achieves doubles that offered by the NTS scheme. Furthermore, in the case of LU, the results surpass those presented by a conventional scheme with almost double capacity.

Results show that the NTS scheme is very sensitive to the number of processors in the system. This is because this scheme is quite sensitive to data locality, which tends to vary notably with the number of processors in the system. The results offered by the filter cache are more homogeneous because it manages the information according with the frequency of data and this criterion is more uniform than locality and less dependent on the number of processors.

Table 11 presents the estimation of the theoretical memory capacity that would offer the same performance (tour reduction) as the split schemes. The filter cache (with only 18 KB capacity) achieves, on average, a tour reduction similar to that obtained by 25 KB, 33 KB, and 36 KB conventional caches when working with 1, 2, and 4 processor, respectively.

Table 11
Theoretically equivalent capacities for the split data cache schemes

Benchmark	FILTER			NTS		
	1 p.	2 p.	4 p.	1 p.	2 p.	4 p.
Barnes	25	25	24	23	23	23
FMM	25	25	25	24	13	11
LU	49	64	88	35	31	32
FFT	19	19	19	16	19	19
RADIX	30	33	27	16	27	26
Average	25	33	36	20	22	21

Fig. 6 shows the tour length measured in number of load instructions handled while the block is in cache. Results are presented for 1, 2, and 4 processors. In FFT and Radix, the performance of both the splitting schemes and the larger 32 KB cache are on par. In Barnes, FMM, and LU, the filter scheme achieves better performance than the NTS, and even in LU surpasses the performance achieved by the larger 32 KB conventional cache. Notice that in FMM, the NTS scheme decreases its performance in comparison with the 16 KB organization as the number of processors increases, showing once more that this scheme is less adequate for SMP systems, because of its high sensitivity to data localities.

5.4. Designing competitive coherence protocols

The experiments discussed in the previous section show that tour behaviour (length and amount) depend on the number of processors. In the write-invalidate protocols (like that used in the previous section) when a given processor writes in a block, the protocol invalidates all the copies of the same block loaded in the caches of the remaining processors, independently of the use of such processors. In contrast, the write-update protocols update the data in the remaining caches containing the block—even though such a block was not to be referenced again. Write-update protocols usually

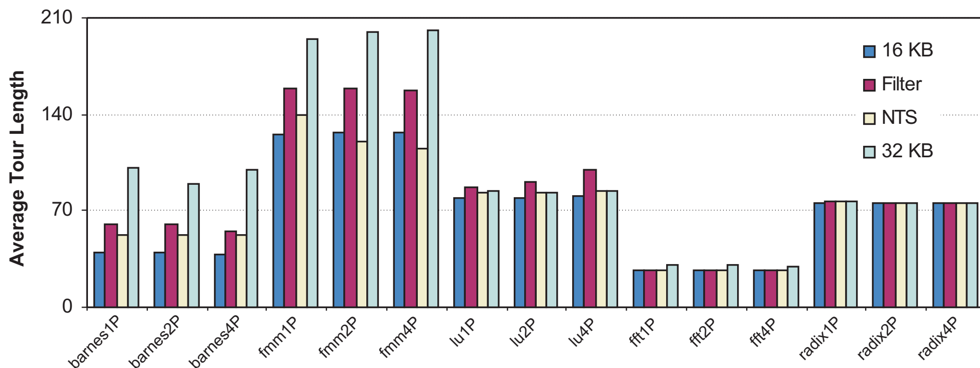

Fig. 6. Average tour length measured in load instructions handled while the block is on tour.

increase the hit ratio but at the expense of increasing the bus traffic. Therefore, the relative performance of write-invalidate versus write-update protocols strongly depends on the sharing patterns exhibited by the workload. In [27], Eggers and Katz compare the performance of write-invalidate and write-update policies. They conclude that neither protocol is better than the other. A third kind of protocols are the competitive or adaptive, which invalidate or update data according to the workload behaviour. For instance, it is possible to use a selective write invalidate policy when there is a high number of write operations on a given block, moving to a write update policy for those blocks showing fewer write operations.

As mentioned earlier, all the split data cache schemes managing reuse information can store such information in the L2 caches. Therefore, when designing coherence protocols for SMPs, the system can make use of this information to boost the protocol performance. For illustrative purposes in this section we explore the potential of this idea using the filter scheme and we present two new competitive protocols, called Competitive_1 and Competitive_2. Both protocols slightly differ with respect to the baseline write-invalidate Berkeley protocol, although the idea can be easily exported to many other protocols. Details of the block transition diagram between the different states in the Berkeley protocol can be found in [26].

5.4.1. Competitive_1 protocol

In the filter scheme, the reuse information maintained is just a counter attached to each block. A high value of the counter means that such block was highly referenced (as many times as its counter value) by the corresponding processor, while a zero value means that the block was referenced only once by the processor.

When a snoopy cache controller invalidates a block the processor may be heavily referencing that block (e.g., the block has a high counter value); therefore, the invalidation of the block is not the most appropriate action. It is very useful to know if a block is heavily referenced before invalidating it, and this can be discovered by testing the corresponding counter. But, a crucial aspect is to decide when is the counter value high enough to update the block; and when it is enough low to invalidate it.

A previous test study was made in order to check the appropriate counter size and the value that marks the threshold. When the cache controller detects that a load instruction is issued in a remote processor, the block counter values are checked. This experiment was performed for a system with 32 processors, and 512 KB L2 caches. Table 12 classifies the invalidated blocks according to their counter value. Results show that a high number of blocks were invalidated with a counter value equal to zero—from 48.9% in the Barnes application to 95.86 in the LU kernel. This last value indicates the poor data locality exhibited in the LU kernel. As zero is the minimum counter value, the protocol assumes that these blocks are unlikely to be referenced again; thus, in an initial run of a competitive protocol, we decided to invalidate them.

An infinite counter was assumed to carry out the experiment. Looking at the last column in Table 12, it is possible to observe that close to 21% of the blocks in Barnes and 14% of the blocks in FMM are invalidated with counter values greater than 31. This shows the high data locality of these kernels when their blocks are invalidated.

Table 12
Number and percentage of invalidated blocks classified according to different intervals of their counter value

Bench	*ctr* = 0		0 < *ctr* <= 7		7 < *ctr* <= 15		15 < *ctr* <= 31		*ctr* > 31	
	# inv	% inv	# inv	% inv	# inv	% inv	# inv	% inv	# inv	% inv
Barnes	93,791	48.90	24,884	12.97	17,194	9.01	16,899	8.81	39,334	20.515
FMM	119,209	62.98	23,033	12.17	11,681	6.53	10,661	5.63	25,724	13.599
LU	65,217	95.86	329	0.48	1334	1.98	1241	1.82	3	0.00
FFT	84,987	54.71	2306	1.48	32,133	20.15	33,623	21.65	793	0.51

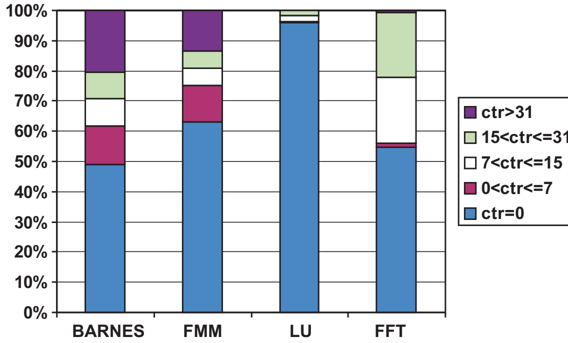

Fig. 7. Cumulative invalidated lines classified by counter values.

Thus, when designing competitive protocols, these blocks will not be invalidated in order to benefit from their excellent data localities. In general, a five bit counter size would capture almost the 80% of the invalidate requests in each application used.

Fig. 7 plots these percentages cumulatively to make visual analysis easier.

Using the local counter one can see which blocks are the least referenced. Of course, the invalidated blocks with a count equal to zero have poorer locality that those invalidated with a higher value. On the other hand, the number of invalidated blocks with a count equal to zero always exceeds 48% of the total invalidated blocks. This is a considerable percentage, and must be taken into account when a protocol is designed. In accordance with this observation, the Competitive_1 protocol invalidates a block when its counter value is zero, otherwise, it updates the word.

The base idea of this protocol can be easily generalized by replacing the zero with a constant value, and this leads towards a more conservative protocol. Logically, the higher the constant is, the fewer the number of updates the algorithm will perform; thus, it will work using an invalidation policy.

5.4.2. Competitive_2 protocol

This protocol proposes that the requesting cache also sends on the bus the counter value ($Counter_{local}$) of the block. Of course, if the operation is a local miss then this value will be zero. The snoopy cache controllers of the remote L2 caches will compare the counter value of the block ($Counter_{remote}$), meaning the candidate block to be invalidated, with the counter value of the local cache (read from the bus). The update condition is shown by the equation: $Counter_{remote} >= Factor \times Counter_{local}$ where *Factor* is a number for weighting the value of $Counter_{local}$. This value has been set to one in this study.

In this manner, a relative comparison is made. A higher counter in a local block generally indicates that the probability of being referenced again in this cache is higher than in the requesting cache.

Table 13 shows the total amount and percentage of invalidated blocks, according to the results of a counter comparison ($Counter_{local}$ versus the $Counter_{remote}$). The results show that the percentage of invalidated blocks when both values are equal to zero is a little higher than 34%—which supposes about a third of the total cases. The LU kernel shows the other side; because the percentage exceeds the 85%. This is because the blocks have poor locality and there is no benefit if an update policy is taken. Statistically, the greatest advantage from updating a block is obtained when the $Counter_{remote}$ value is greater or equal to the $Counter_{local}$ value; but not zero. FMM and Barnes kernels exhibit the highest values—up to

Table 13
Number of invalidate blocks and their percentage classified according their counter values

Bench	$ctr_l = ctr_r = 0$		$ctr_r = ctr_l > 0$		$ctr_r > ctr_l$		$ctr_r < ctr_l$	
	# inv	% inv	# inv	% inv	# inv	% inv	# inv	% inv
Barnes	67,943	34.89	5987	3.07	53,413	27.43	67,369	34.60
FMM	90,002	48.29	6982	3.75	36522	19.60	52,870	28.37
LU	57,330	85.82	646	0.97	861	1.29	7962	11.92
FFT	51,591	42.28	474	0.39	19,709	16.15	50244	41.18

23%. These two applications benefit most from this protocol.

It can be observed that by making slight modifications to the competitive update condition, a large variety of competitive update policies can be obtained. By choosing the condition, the protocol can work like an invalidation policy; or like to an update policy. For instance, if a factor of 3 is used, the resultant protocol will work in a similar way to a write-invalidate protocol—as discussed above. On the other hand, by setting *Factor* to zero, the protocol works like an update protocol.

Details of the state transition diagram for both competitive protocols can be found in [28].

5.4.3. Experimental results

To check the performance of both protocols there must be both enough traffic on the bus and enough reuse information on the second level caches. Therefore, we selected two small first-level caches (1 KB direct-mapped filter cache and 4 KB two-way set-associative main cache) and two four-way set-associative L2 cache sizes (256 and 512 KB). Cache block sizes are 16 bytes wide and each has a 4-bit counter attached.

The effectiveness of the proposed protocols in comparison with the write-invalidate and write-update protocols is evaluated. Bus utilization and speedup have been chosen as performance indexes to test the system behavior. Bus utilization is often used to carry out performance comparison studies in shared memory systems, because performance drops when the bus reaches a high utilization. In this paper, results for 16 and 32 processors are presented, while keeping the problem size constant.

First, the number of tours of blocks in the L2 caches has been measured in order to test the influence of the protocols updating the words. Fig. 8 shows the results. As expected, the update protocol generates fewer tours while the write-invalidate policy presents more. The Competitive protocols fall in between. Competitive_1 works more similarly to the write invalidate protocol and Competitive_2 more similarly to the update protocol. In FFT kernel, any protocol presents advantages over the others. This figure does not plot the effectiveness of the tours, only the total number.

Fig. 9 plots the bus utilization for the selected benchmarks. In Barnes and FMM, the update protocol reduces the amount of time the bus is used in the four studied cases, (16 and 32 processors with 256 and 512 KB cache sizes). In these cases, the write-invalidate protocol shows the worst performance; and the proposed schemes fall in between. In the LU kernel, the update protocol drops in performance compared to the write-invalidate protocol; while the proposed schemes have a quite good performance. This significant drop in the LU protocol performance is due to the large amount of "zero counter value cases". In other

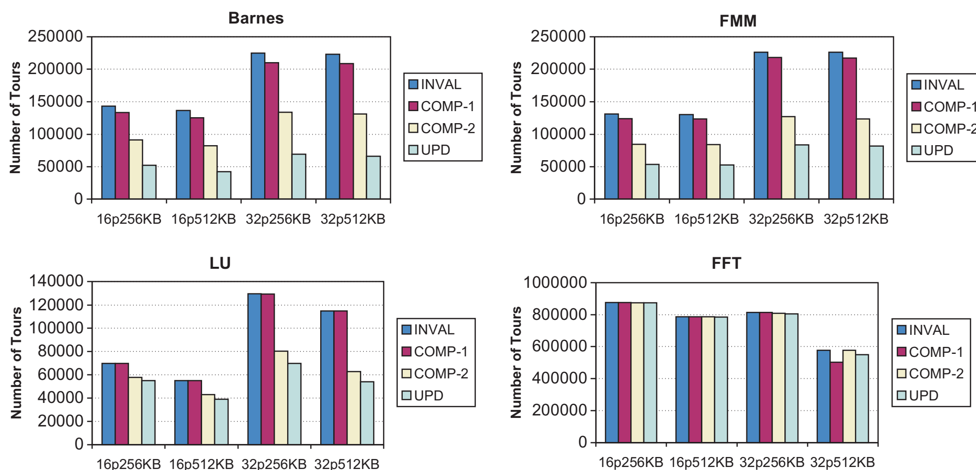

Fig. 8. Total tours of blocks in the L2 cache.

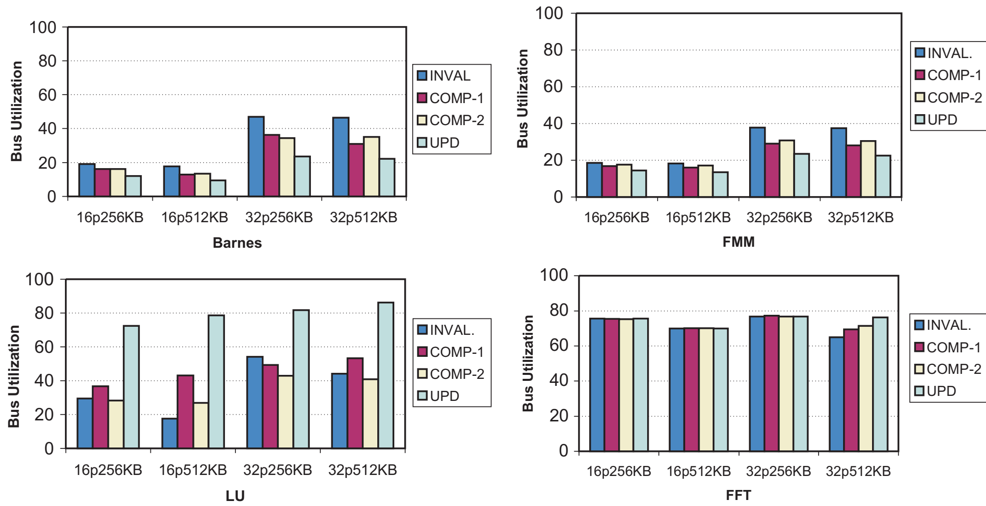

Fig. 9. Bus utilization in the four evaluated protocols.

words, there are many blocks that are updated and have a low probability of being referenced again.

Also notice that the Competitive_2 performs better than the write-invalidate in three of the four cases. Finally, in the FFT kernel, all the protocols obtain a similar performance, except in the last case (32p512 KB), where the write-invalidate performs slightly better.

Fig. 10 shows the relative execution times with respect to the write-invalidate protocol as unit time. Notice that Competitive2 protocol only exceeds the value of 1 three times; once in FFT and twice in LU kernel. On the other hand, one can see that Competitive_1 works closer to the write-update protocol than the Competitive_2.

In general, the proposed competitive protocols perform better, or similarly, than those using an invalidate policy; and their performance never drops dramatically—unlike the write-update protocol.

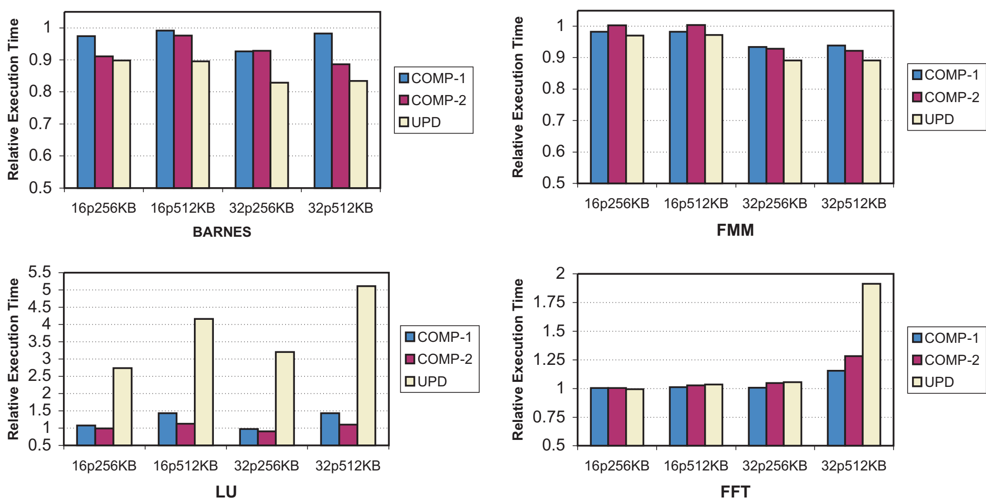

Fig. 10. Relative execution times, taking the execution time employed by the write-invalidate as unit.

6. Conclusions

In this paper we have discussed the essence of the split data cache schemes, we have analyzed what each scheme pursues and the different steps of the design process. The paper has exhaustively evaluated the performance results of some split data cache schemes by considering different scenarios: superscalar processors using the SPEC95 and SPEC2000 benchmarks, and bus-based multiprocessors using the SPLASH2 benchmark suite. To run simulations on multiprocessors we considered the nodes (processor plus caches) connected to the system bus by means of a classical write-invalidate protocol.

Experimental results have shown that during the different scenarios, in general, the split data cache schemes perform better than conventional caches with almost double the data storage capacity. Moreover, these schemes use much less die area and this can be used to improve the performance of other processor features issues.

Among the split data cache schemes, results show that the filter data cache—using the frequency of use as the criterion to select the data—achieves better performance than other schemes that split the cache according to the criterion of data localities. These differences grow when working in SMPs, because localities tend to vary with the number of processors so that schemes splitting the cache according to the criterion of data localities become inadequate.

Finally, we explore how the reuse information that some split schemes include to manage the information according to the past pattern block behaviour can help the design of coherence protocols. Results show that schemes handling reuse information are especially useful to help the design of new coherence protocols in order to boost the performance and increase the scalability of the system.

References

[1] V. Agarwal, M.S. Hrishekesh, S.W. Keckler, D. Burguer, Clock Rate versus IPC: the end of the road for conventional microarchitectures, in: Proceedings of the 27th International Symposium on Computer Architecture, June 2000, pp. 248–259.

[2] V. Agarwal, S.W. Keckler, D. Burguer, The effect of technology scaling on microarchitectural structures, Technical Report TR2000-02, Department of Computer Sciences, The University of Texas at Austin, April 2000.

[3] J. Keshava, V. Pentkovski, Pentium III Processor Implementation Tradeoffs, Intel Technology Journal Q 2 (1999).

[4] G. Hinton, D. Sager, M. Upton, D. Upton, D. Boggs, D. Carmean, A. Kyker, P. Rousell, The michroarchitecture of the Pentium 4 processor, Intel Technology Journal Q 1 (2001).

[5] C. McNairy, D. Soltis, Itanium 2 processor microarchitecture, IEEE Micro 23 (2) (2003) 44–55.

[6] G. Tyson, M. Farrens, J. Matthews, A.R. Pleszkun, A modified approach to data cache management, in: Proceedings of Micro-28, December 1995, pp. 93–103.

[7] A. Rivers, E.S. Davidson, Reducing conflicts in direct-mapped caches with a temporality-based design, in: Proceedings of the 1996 ICPP, August 1996, pp. 151–160.

[8] M. Prvulovic, D. Marinov, Z. Dimitrijevic, V. Milutinovic, The split spatial/non-spatial cache: a performance and complexity analysis, IEEE TCCA Newsletter (July) (1999) 8–17.

[9] T. Johnson, W.W. Whu, Run-time adaptative cache hierarchy management via reference analysis, in: Proceedings of the ISCA-24, June 1997, pp. 315–326.

[10] A. Gonzalez, C. Aliaga, M. Valero, A data cache with multiple caching strategies tuned to different types of locality, in: Proceedings of the ACM International Conference on Supercomputing, Barcelona, Spain 1995, pp. 338–347.

[11] V. Milutinovic, B. Markovic, M. Tomasevic, M. Tremblay, The split temporal/spatial cache: initial performance analysis, in: Proceedings of the SCIzzL-5, Santa Clara, CA, USA, March 1996, pp. 63–69.

[12] J. Sanchez, A. Gonzalez, A locality sensitive multi-module cache with explicit management, in: Proceedings of the ACM International Conference on Supercomputing, Rhodes, Greece, June 1999.

[13] S. Kumar, C. Wilkerson, Exploiting spatial locality in data caches using spatial footprints, in: Proceedings of the 25th ISCA, June 1998.

[14] J. Sahuquillo, A. Pont, Splitting the data cache: a survey, IEEE Concurrency (September) (2000).

[15] N. Jouppi, Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers, in: Proceedings of the ISCA-17, June 1990, pp. 364–373.

[16] K.K. Chan, C.C. Hay, J.R. Keller, G.P. Kurpanek, F.X. Schumacher, J. Zheng, Design of the HP PA 7200 CPU, Hewlett-Packard Journal (February) (1996) 1–12.

[17] J. Sahuquillo, A. Pont, V. Milutinovic, The filter data cache: a comparison study with splitting L1 data cache schemes sensitive to data localities, in: Proceedings of the 3rd International Symposium on High Performance

Computing (ISHPC2K), Tokyo, Japan, October 2000, pp. 319–327.
[18] E.S. Tam, J.A. Rivers, V. Srinivasan, G.S. Tyson, E.S. Davidson, Active management of data caches by exploiting reuse information, IEEE Transactions on Computers 48 (11) (1999) 1244–1259.
[19] E.S. Tam, J.A. Rivers, G.S. Tyson, E.S. Davidson, mlcache: A flexible multilateral cache simulator, in: Proceedings of MASCOTS'98, pp. 19–26, 1998.
[20] D.C. Burger, T.M. Austin, The SimplesSalar Tool Set, Version 2.0, Computer Architecture News 25 (3) (1997) 13–25.
[21] http://www.sun.com/processors/UltraSPARC-II/.
[22] C.N. Keltcher, K.J. McGrath, A. Ahmed, P. Conway, The AMD opteron processor for multiprocessor servers, IEEE Micro 23 (2) (2003) 66–76.
[23] R.E. Kessler, The Alpha 21264 microprocessor, IEEE Micro 19 (2) (1999) 24–26.
[24] D. Magdic, LIMES: a multiprocessor simulation environment, IEEE TCCA Newsletter (March) (1997).
[25] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, A. Gupta, The SPLASH-2 programs: characterization and methodological considerations, in: Proceedings of the 22nd ISCA, pp. 24–36, June 1995.
[26] R.H. Katz, S.J. Eggers, D.A. Wood, C.L. Perkings, R.J. Sheldon, Implementing a cache consistency protocol, in: Proceedings of the ISCA-12, 1985, pp. 276–283.
[27] S.J. Eggers, R.H. Katz, A characterization of sharing in parallel programs and its application to coherency protocol evaluation, in: Proceedings of the ISCA-15, Honolulu, May 1988.
[28] J. Sahuquillo, A. Pont, Designing competitive coherence protocols taking advantage of reuse information, in: Proceedings of the 26th IEEE Euromicro Conference, September 2000.
[29] E.S. Tam, Improving cache performance via active management, Ph.D. Dissertation, University of Michigan, June 1999.
[30] J. Sahuquillo, S. Petit, S. Petit, A. Pont, V. Milutinovic, Performance study of the filter data cache on a superscalar processor architecture, IEEE Computer Society Technical Committee on Computer Architecture Newsletter (TCCA) News (January) (2001).

Julio Sahuquillo received his BS, MS, and Ph.D. degrees in Computer Engineering from the Polytechnic University of Valencia (UPV), in Valencia, Spain. Since 2002 he is an associate professor at the Computer Engineering Department at the Polytechnic University of Valencia. His research topics have included clustered microarchitectures, multiprocessor systems, cache architecture design, distributed shared memory, multithreading microarchitectures, and power dissipation. Recently, a part of his research has also concentrated on the web performance field.

Salvador Petit received his Ph.D. degree in Computer Engineering from the Polytechnic University of Valencia, Spain. Currently he is an assistant professor at the Computer Engineering Department at the Polytechnic University of Valencia. His main research topics are distributed memory systems and cache architecture design. Recently, he performed a research stay as invited scholar in the Electrical and Computer Engineering department in Northeastern University, Boston, USA.

Professor Ana Pont-Sanjuán received her MS degree in Computer Science in 1987 and a PhD in Computer Engineering in 1995, both from Polytechnic University of Valencia. She joints the Computer Engineering Department in the UPV in 1987 where currently she is full professor of Computer Architecture. Since 1998 until 2004 she was the head of the Computer Science High School in the UPV. Her research interest include multiprocessor architecture, memory hierarchy design and performance evaluation, web and internet architecture, proxy caching techniques, CDN's, communication networks. She also has participated in a high number of research projects financed by Spanish Government and Local Valencian Government. Since January 2005 she is the Chairperson of the IFIP TC6 Working Group: Communication Systems for Developing Countries.

Professor Veljko Milutinović received his Ph.D. in Computer Engineering in 1982 from the University of Belgrade, Serbia. From 1982 till 1990 he was on the faculty of Purdue University. He was coarchitect of the World's first 200MHz RISC microprocessor, for DARPA. Member of the advisory board and active consultant in a number of high-tech companies (TechnologyConnect, BioPop, IBM, AT&T, NCR, RCA, Honeywell, Fairchild, etc.). Since 1990 he is a full professor in the University of Belgrade. In 90s he was responsible for the silicon-compiler based design of a clone of Intel i860 (the first 64-bit microprocessor). He has coauthored about 50 IEEE journal papers, plus in about 50 papers in other journals or book chapters. He has presented over 300 invited

talks at all major universities in the World, in both computer engineering and business administration. He also has served as guest editor for a number of special issues in various IEEE journals: Proceedings of the IEEE, IEEE Transactions on Computers, IEEE Concurrency, IEEE Computer, etc. More recently he is active in infrastructure for e-business on the Internet. Professor Milutinovic is a Fellow of the IEEE since 2003.