Performance Assessment of Contents Management in Multilevel On-Chip Caches *

Pablo Ibáñez and Víctor Viñals
Univ. of Zaragoza
Dpto. de Informática e Ingeniería de Sistemas
C/ María de Luna 3, Zaragoza 50015. Spain
{imarin, victor}@posta.unizar.es

Abstract

This paper deals with two level on-chip cache memories. We show the impact of three different relationships between the contents of these levels on the system performance. In addition to the classical Inclusion contents management, we propose two alternatives, namely Exclusion and Demand, developing for them the necessary coherence support and quantifying their relative performance in a design space (sizes, latencies, ...) in agreement with the constraints imposed by integration. Two performance metrics are considered: the second-level cache miss ratio and the system CPI. The experiments have been carried out running a set of integer and floating point SPEC'92 benchmarks. We conclude showing the superiority of our improved version of Exclusion throughout all the sizing and workload spectrum studied.

1. Introduction

Main memory and processor cycle times keep on diverging increasingly, mostly due to the growing integration scale and to the use of new organization techniques such as superpipelining. Experts foresee a further increase in this speed gap and the use of multiple levels of on-chip cache [8]. Current integration technology makes it possible to devote large chip areas to cache memory, allowing then a two-level organization: the first level is split into data and instructions and works at processor rate; the second level is a slower unified one. This organization has the following advantages versus the classical single-level split cache:

i) Given a fixed total size, it usually has a better overall hit ratio, due to dynamic space reallocation between data and instructions in the second level [14].

ii) Each level can be devoted to different goals, and be specifically designed to attain high performance for each target goal. For instance, the size and associativity of the first level can be limited, resulting in shorter access times and allowing to access it using virtual addresses. This contributes to decrease CPU cycle time and/or memory access instructions latency. On the other hand, in order to reach high hit ratios, the second level may have higher associativity, and even make use of prefetch hardware.

iii) The second level can watch the system bus or reply to the coherence commands sent from an external level-three cache. This may allow a further reduction in the complexity and cycle time of the first level. In addition, and depending on the implementation, a snooping level-one cache could have a higher latency in responding coherence commands due to its high utilization by the processor.

In this paper we shall assume that the integration scale and/or the cycle time suggest the use of two levels of on-chip cache [11]. It will also be assumed a RISC processor able to work stand-alone or in a shared memory multiprocessor system. A recently marketed DEC chip, the Alpha 21164, has these characteristics, with a $96KB$ second-level and $8KB + 8KB$ first-level caches [5]. Due to the area constraints imposed by the joint integration of both levels, their contents relation can have a considerable influence on the system performance. Our goal is to determine that influence in a wide and representative enough design space.

*This work was supported by the CONAI (Diputación General de Aragón) under grant PIT 09/93

So far, three relations between the contents of two consecutive cache memory levels have been defined: 1) Level i (L_i) is a superset of level i-1 (L_{i-1}). 2) The contents of L_i and L_{i-1} are disjoint most of the times. and 3) The only criterion that determines the contents of L_i is the sequence of the L_{i-1} demands; therefore, a block that is heavily reused by L_{i-1} can be excluded from L_i.

The two former relations have been called Multilevel Inclusion Property (MLI) and Two-Level Exclusive Caching (TLEC), respectively. The later relation has no established name; we will call it Demand hereafter. The MLI Property has been suggested by Baer and Wang to ease the coherence maintenance in uni and multiprocessors in an efficient way [2, 3]. Their performance evaluation was made using stochastic or analytic models, assuming that Inclusion management introduces a penalty on the global miss ratio. Two-level Exclusive caching has been suggested by Jouppi and Wilton to achieve a better use of the on-chip area [11]. This work makes a miss ratio based comparison with Demand management, but does not take into account different write costs between levels or coherence support cost. A similar concept, *Exclusion* management, has been independently proposed but not evaluated in [20]. Demand management has been used in some quantitative studies with the goal of optimizing the parameters of two-level cache memory systems [15, 21, 13, 4, 6, 16]. All of them consider an external second level, therefore with temporal or sizing parameters that are far from our scope. On the other hand, coherence maintenance cost and support were no studied.

The goal of this study is to compare Inclusion, Demand and Exclusion management in an general purpose —symbolic and numerical processing— uniprocessor environment. System must maintain coherence with a third-level external cache in order to ensure DMA correctness and to allow building a multiprocessor. For that purpose we will extend Demand and Exclusion definition with the mechanisms needed to maintain coherence. We will define a design space (sizes, latencies, ...) in agreement with the constraints imposed by integration, simulating these three strategies in it.

The following section introduces the Reference Module selected for the simulation, recalling Inclusion management and giving special attention to our Demand and Inclusion basic definitions. Section 3 presents the coherence problem and the added support to Demand and Exclusion for solving it as efficiently as in Inclusion. The workload is presented and the results on miss ratios are analysed in section 4. In section 5 we present the detailed timing model for the cycle-by-cycle simulator, analysing the CPI results and pointing out their differences to those of the miss ratio analysis. Finally, our conclusions are summarized in section 6.

2. On-chip Inclusion, Exclusion and Demand management basics

In this section we present the basic protocols for managing the contents of on-chip caches by Inclusion, Exclusion or Demand. To compare the different managements we take a Reference Module into which the three strategies can be embedded (Figure 1). Because components and interconnects are essentially the same for all strategies, the hardware costs and speeds remain unchanged, thus allowing the isolation of the contents management on performance. Throughout the work we assume a copy-back policy and a single block size inside the chip.

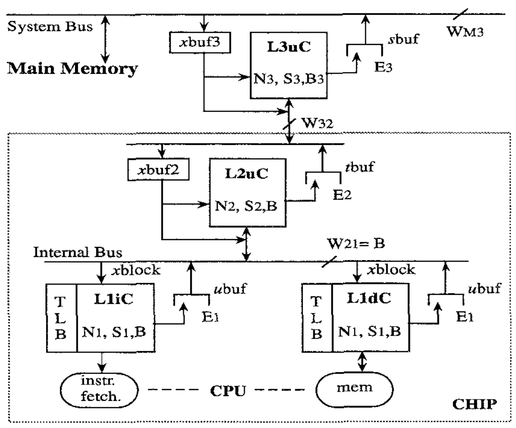

Figure 1. Reference Module used for comparing alternative Level-2 contents management.

The first level is made up of equally sized on-chip instruction and data caches: L1iC and L1dC, of N_1 sets, S_1 blocks per set (associativity) and B bytes per block. There is also an on-chip second level unified cache: L2uC, with sizing parameters N_2, S_2 and B. A third level unified cache, L3uC, includes all the blocks present on chip.

Buffers are provided to speed-up interlevel transfers: *xbuf2* and *xbuf3* assemble incoming words to levels 2 and 3, allowing to write blocks at once; *ubuf*, *tbuf* and *sbuf* entries hold what is needed to interact with the upper level. The first letter of each buffer indicates the type of block it contains: x blocks keep the word that is accessed by the processor; u, t and s blocks are the replaced from levels 1, 2 and 3 respectively. The *ubuf*, *tbuf* and *sbuf* sizes are E_1, E_2 and E_3 entries respectively.

Static priorities arbitrate simultaneous access to shared buses. Note that the transfer width between L2 and L1, W_{21}, is always set up to the chip block size, making use of having both caches always on-chip (as an example of a large internal bus consider the Power PC 604, which has a bus size equal to 128 bits [17]).

L1 caches are virtually addressed but they keep physical tags. This offers all the advantages of a physical addressing without the cost of translation in the path between L1 and the processor, but *puts an upper limit to L1 cache sizes* equal to page-size times associativity [21]. L2 and L3 are physically addressed and keep physical tags. On the other hand, we will assume a one-to-one mapping between virtual and physical addresses because our trace system provides just the virtual ones.

2.1. Inclusion management

As mentioned in section 1, the Inclusion management of L2 is the *space inclusion* suggested by Baer and Wang for copy-back caches [2]. L2 must always hold a —not necessarily updated— superset of L1. The protocol to achieve it is as follows: 1) If a miss occurs both in L1 and L2, the missed block x is copied into both cache levels. 2) If L1 misses but L2 hits, the block is copied from L2 to L1. 3) The replacement of an L1dC dirty block requires a copy-back operation on L2 that always hits. And 4) Any replacement policy can be used for L1, but only the blocks not present in L1 can be replaced from L2. This last rule requires a *present-in-L1* bit to be added to every L2 block; this bit is set when delivering the block to the lower level, and reset when replaced by that level. Therefore, L1 has to report on dirty and clean block replacements to L2.

To fulfill space inclusion L1 and L2 must observe the following sizing constraint:

$S_2 \geq 2 * S_1 * max(1, N_1/N_2)$. If there are write buffers between L1 and L2, Inclusion must be also observed on the blocks they keep, what forces to add their sizes to S_2. Under these circunstances minimum L2 associativity must be 4, (e.g. $E_1 = 1$, $S_1 = 1$ and $N_1 = N_2$).

2.2. Exclusion management

Exclusion keeps L1 and L2 contents always disjoint, so that the useful information equals the sum of their sizes. To achieve this, a miss in L1 that hits in L2 is solved by *swapping* blocks between these two levels: the missed block x is moved from L2 into L1, while the replaced block u is moved (being it either clean or dirty) to the position where x was in L2. Its *dirty* bit must be sent with block u, so that dirty blocks can be copied back to L3 when replaced from L2. A miss in both levels loads the missed block x directly into L1, displacing the block u from L1 to L2 (instruction or data, dirty or not). Therefore, the L2 contents are made up from L1 u-victim blocks. The L2 incoming block can cause a second replacement from L2. Replacement algorithms for L1 and L2 have no constraints.

According to Jouppi and Wilton (TLEC), strict exclusion is only achieved if $N_1 \geq N_2$, otherwise the sets where x and u are mapped onto L2 may be different, thus disabling swapping. In that case they explicitly assume that a block can be present in both levels at the same time. In our Exclusion protocol, an L1 miss followed by an L2 hit always forces the invalidation of the block in L2 thus keeping strict exclusion under any configuration. As we will see later on, the invalidation of the read block allows simpler coherence support and a reduction of the time to receive the u-block.

Moreover, a better hit ratio can be achieved since the hole created when invalidating can prevent another useful block from being replaced. As an example let us assume $N_1 = N_2$, $S_1 = 1$ and $S_2 = 2$. Figure 2 shows the blocks present in a given set for all caches. If no invalidation is used, a miss for block x in L1iC that hits in L2uC can leave the cache contents as shown in fig. 2a. A later access to data block c causes block a to be replaced from L1dC, which in turn can cause the replacement of block b, as shown in fig. 2b. Thus, the effective capacity of the set for the two levels is reduced to 3 blocks: x, a and c. However, with strict Exclusion, block a is placed into the hole left by x in L2uC, resulting in an effective capacity of 4 blocks.

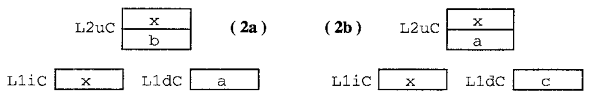

Figure 2. Non-strict exclusion. Blocks x, a, b and c are mapped onto the same L2 set.

2.3. Demand management

Our Demand contents management is based upon one of the organizations proposed by Baer and Wang for uniprocessors: *Write-back and no MLI* [2]. The protocol suggested is identical to the Inclusion one (section 2.1) except for the last point: 4) Upon Demand, every L2 block, either belonging to L1 or not, is subject to be replaced. Therefore neither warning from L1 to L2 is needed if a clean block is replaced from L1, nor *presence* bits are required in L2.

This causes another change in point 3; when a dirty block is replaced from L1, a copy-back is made into the upper level, the same as in Inclusion. However, the write can now miss in L2, requiring an additional replacement in L2. In this case we propose to send the block directly to L3 instead of writing it in L2; we call this L2 write policy *block write-around* (such expanding to block level the *word write-around* with copy-back policy described in [10] and previously used in [13] in level 1 caches).

By studying Demand we consider the simplest way to manage L2 contents. It is an interesting compromise, since a priori it can perform better or worse than Inclusion or Exclusion: it can achieve smaller miss ratios than Inclusion and less interlevel traffic than Exclusion.

3. Coherence support

We assume that the System Bus has a snoopy protocol [19] that ensures I/O DMA correctness with no O.S. intervention. Also, if the Reference Module is used to build a shared memory multiprocessor, this protocol, 1) guarantees correct and efficient handling of shared variables and, 2) allows process migration without O.S. started flushes [18]. In this context, the Inclusion management of all the three levels ($L1 \subset L2 \subset L3$) guarantees correctness and is efficient: coherence commands spread into the bus are caught and resolved in the furthest possible level from the processor. Therefore, word broadcasts (updating protocol) or block invalidations (invalidating protocol) do not disturb the closest levels to processor that have replaced the item. Besides, if the protocol requires block transfers among processors, Inclusion allows delivery with the least possible latency from the closest to System Bus level.

In this section we show how to adapt L2 Exclusion and Demand management to achieve the same efficient behaviour in broadcasts or block transfers. For the upper level (L3) the only possible policy is some kind of inclusion of all chip contents (L1 and L2). In our case (*Space MLI*), a *present-on-chip* bit is added to every L3uC block; it is managed as stated in section 2.1. Therefore, L3 must receive a *replacement warning* each time a block is replaced from the chip.

3.1. On-chip Exclusion and coherence

Due to the strict exclusion imposed by our protocol, an L2 block replacement is equivalent to a replacement from the chip. Therefore, L3 must be only warned about each L2 replacement. On the other hand, a coherence command sent from L3 can be either a hit or a miss in L2. If it hits, the command must be captured not disturbing L1. On the contrary, if it misses the command must be sent to L1 where it will certainly hit.

If exclusion were not strict, as proposed by Jouppi and Wilton in their protocol for $N_1 < N_2$, the previous scheme would be wrong; it would be necessary to add a copy of L1's directory to L2uC in order to correctly send (from chip) replacement warnings and to achieve the as-soon-as-possible command capture goal.

3.2. On-chip Demand and coherence

Using Demand and assuming a block write-around write policy, a block is replaced from the chip and consequently L3 must be warned in the following cases:

i) A block that only exists in L1 is replaced. But if L1 ignores the contents of L2 it cannot decide whether L3 must be warned or not.

ii) A block that only exists in L2 is replaced. But if L2 ignores the contents of L1 it cannot decide whether L3 must be warned or not.

On the other hand, if L2 ignores the contents of L1 it cannot properly capture the commands of L3.

The most immediate solution is to add to L2 a replicated directory of the L1 caches. A much simpler solution consists of adding a *present-in-L1* bit to every L2 block. Since there are blocks that are present in L1 but not in L2 this is a partial information, but it suffices:

i) A block replacement from L1 can hit or miss in L2. If it hits the presence bit is modified and if the block was dirty it is brought into L2. If it misses a warning is sent to L3 (if the block was dirty the warning is piggybacked on the block).

ii) When a block is replaced from L2, L3 is warned only if the *present-in-L1* bit is not set. The replaced block may be dirty in L2 and present in L1; in this case a new command is required to send to L3 a block with the warning that it is still present in the chip. If that copy of the block in L1 is not modified again, there will be a replacement warning only when it is replaced. On the contrary, if it has been written, the previous copy on L3 was useless.

iii) A coherence command sent from L3 to L2 can be either a hit or a miss. If it is a hit, it will be forwarded to L1 only if the presence bit is set. If it is a miss, the command must be forwarded to L1, where it will hit.

3.3. Block transfer service between caches

If a given protocol encourages cache-to-cache transfers, a mechanism to minimize latency transfer can be added. This can be achieved by including a *modified* bit in each L3uC block. This bit indicates whether its content is updated (bit not set) or the updated copy is in the chip (bit set). This bit is set when a *first write* command is received from the chip, and is reset when a dirty block is received. Obviously L1 does not need such a bit, since the *dirty* bit means the same[1].

L2 contents management determines the details on the existence or handling of the *dirty* and *modified* bits:

- In Inclusion and Demand, the same as in L3uC, a *modified* bit that is set and reset according to the previous rules must be added to L2uC.
- In Exclusion only a single copy of each block exists, and therefore a *modified* bit is not required. It is enough with the *dirty* bit management shown in section 2.2.

4. Experimental results about L2 miss ratio

The workload we have used throughout the simulations consists of a subset of C and Fortran programs belonging to the SPEC'92 suite, compiled for SPARC V8 under SunOS 4.1 and linked with the *static* option to avoid writes in the instructions space. Selection has been made using the data published in [7] by searching for some of the most cache-pressuring programs. Trace generation is performed using Shadow [9], a tool that allows step by step execution of the user code of a process. For each program a maximum of 200 million instructions is executed; it is enough to completely fill the largest of the second level caches simulated. Programs *cc*1 and *compress* have been executed until completion. The experiments carried out do not show the effects of either multiprogramming or system code. Table 1. summarizes some characteristic of the used workload. (i) means integer intensive workload; (f) means floating-point intensive workload.

Program	# Instr.	# Reads	# Writes	# Refs.
compress (i)	90.559	13.533	6.763	110.855
espresso (i)	200.000	40.853	4.721	245.574
xlisp (i)	200.000	40.806	17.835	258.641
cc1 (i)	149.043	28.261	10.663	187.967
doduc (f)	200.000	58.770	16.240	275.010
fpppp (f)	200.000	106.603	22.305	328.908
spice (f)	200.000	42.369	24.523	266.892
swm (f)	200.000	54.261	21.222	275.483

Table 1. Workload used for computing miss ratio and CPI. Numbers in thousands.

All simulations keep constant: 1) A single on-chip block size ($B = B_1 = B_2$), 2) direct mapping in L1 and, 3) write policy (copy-back, fetch-on-miss). L2uC and L3uC use Random replacement policy; at the end of this section we will show some LRU replacement results supporting this choice.

In this section we compare the L2 global miss ratio (*gm2*) for the three strategies. *gm2* is computed by dividing the number of misses in L2uC by the total number of instruction and data references[2]. All measures are obtained in a cold-start fashion, that is, not excluding from the count the initial cache load transient.

L1 cache size varies from $2KB + 2KB$ to $16KB + 16KB$, with a $16B$ block size. For each L1 size, five L2 *four-way* set-associative caches are simulated by varying the number of sets from $N_1/2$ to $8N_1$ ($S_2 = 4$ is the minimum to fulfill Inclusion with buffering). Therefore, L2uC size varies from $4KB$ to $512KB$. Only four sizes are simulated for L2 Inclusion, since the configuration $L2size = L1size$ [3] violates the size constraints needed for Inclusion (see section 2.1).

Figure 3 shows separately for integer and floating-point the average *gm2* for each contents management. This average assumes every program equally weighted, no matter its actual number of executed instructions. Each plot consists of four data groups, corresponding to the L1 four possible sizes ($2KB + 2KB - 16KB + 16KB$). In each group N_2 ranges from $N_1/2$ to $8N_1$ ($L2size$ varies from $L1size$ to $16L1size$). In

[1] *Modified* is a status which is different from *dirty* for L2 and L3. A block can be dirty and modified (the updated copy is present in one of the lower levels) or dirty and not modified (the block is incoherent with respect to upper levels but coherent with respect to the lower ones)

[2] In order to obtain the number of misses for Exclusion and Demand with LRU replacement, extensions of the algorithm proposed by Mattson et al. [12] have been used. They make it possible to simulate caches with several L2 sizes and associativities for a given L1 size in a single pass throughout the trace. Inclusion needs a different simulation for each sizing because the mechanism used by L2 to keep the blocks that are present in L1 breaks the stack behaviour of this replacement algorithm

[3] $L2size = B * N_2 * S_2$; $L1size = 2 * (B * N_1 * S_1)$

general we see that Exclusion achieves substantially smaller miss ratios than Demand and Inclusion for nearly every size and program. Demand generally achieves better results than Inclusion but with much less differences. It can be seen the large locality offered by the SPEC programs compared to nowadays cache sizes; this agrees with Gee, Hill and Smith in [7]. For instance, a first level with $L1size = 8KB+8KB$ serves 93.2% of the floating-point accesses and 97.2% of the integer ones.

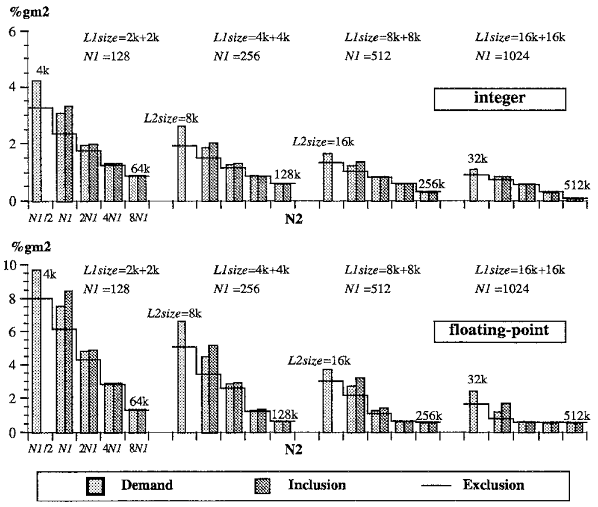

Figure 3. L2 global miss ratios for all three contents management policies expressed as a percentage ($\%gm2$)

The closer L1 and L2 sizes, the bigger the differences, since the *effective* on-chip cache size is relatively more dependent on the contents management policy. For instance, if we compare Exclusion and Inclusion for the integer workload with $L1size = 4KB + 4KB$ and $L2size = 16KB$, $gm2$(Inclusion) is 36% greater than $gm2$(Exclusion) (effective on-chip size is $16KB$ for Inclusion and $24KB$ for Exclusion). Whereas with $L2size = 128KB$, $gm2$(Inclusion) is only 2% greater than $gm2$(Exclusion) (the effective sizes are now $128KB$ and $136KB$). In the same way, $gm2$(Demand) is 25.9% greater than $gm2$(Exclusion) for $L2size = 16KB$ and 2.46% for $L2size = 128KB$. Floating-point code exhibits similar behaviour for the two considered L2 sizes: 50.9%-1.82% for Inclusion and 30.6%-1.33% for Demand.

Additional simulations have been carried out keeping L1 and L2 sizes unchanged but varying 1) block sizes $B_1 = B_2$ from $16B$ to $32B$, 2) L2uC replacement policy from Random to LRU and 3) L2 associativity S_2 from 2 to 32, keeping $N_1 = N_2$. Tables 2 and 3 show a subset of the miss ratio results for this broad design space. The subset has been selected to highlight the configurations where Exclusion performs best. For every L1size the tables show (from left to right) Exclusion miss ratio(Ex), the difference between Demand and Exclusion miss ratios as a percentage of the later($\Delta(Dm)$), and the same figure between Inclusion and Exclusion($\Delta(In)$). The row (*base*) are the baseline results showed in figure 3, rows ($B = 32$), (*LRU*) and ($S_2 = 8$) are the results for the additional simulations presented in this paragraph.

Variants ($B = 32$) and ($S_2 = 8$) achieve smaller miss ratios than the base simulation for nearly all programs (except for *compress*, where an increase of the block size leads to higher miss ratios). On the other hand, the performance comparison trend among contents managements remains as previously described. Regarding the replacement algorithm, LRU is slightly better for integer programs. However, Random is better for floating-point with sometimes quite remarkable differences. Comparisons between Exclusion, Inclusion and Demand remain the same when the replacement algorithm is changed; there is just one substantial difference: while LRU Exclusion miss ratio keeps always strictly smaller than LRU Inclusion and Demand, this is not always true for Random.

To sum up, Exclusion achieves smaller miss ratios due to a better use of space over the tested design space. The less the chip area globally devoted to cache and the closer L1 and L2 sizes, the more the difference; in other words, the size constraints imposed by the integration of L1 and L2 favour Exclusion. Demand behaves between Exclusion and Inclusion closer to the later.

5. Experimental results about CPI

In this section we compare the performance of Inclusion, Exclusion and Demand measured as cycles per instruction (CPI). It is not possible to analytically compute CPI from miss ratios, since the timing of an event depends on the previous state of whole the hierarchy (items inside buffers, concurrent activity of each level, etc.) Therefore, we have built a discrete event time-driven simulator which, coupled to Shadow, obtains the CPI for our workload when executing in the Reference Module.

The simulator includes a model of a SPARC V7 processor which issues one instruction per cycle, has a 5 stages integer pipeline and two floating-point units: a fully-pipelined adder/multiplier (4 stages) and a pipelined divider/square-root unit (4 stages, with a sec-

	L1size=2KB+2KB			L1size=4KB+4KB			L1size=8KB+8KB			L1size=16KB+16KB		
int($base$)	2,35	31,9%	42,97%	1,52	25,9%	35,99%	1,02	23,76%	33,62%	0,73	17,78%	19,44%
int($B = 32$)	1,96	30,13%	36,18%	1,28	25,26%	32,67%	0,90	20,52%	27,03%	0,65	17,16%	19,77%
int(LRU)	2,27	32,33%	49,15%	1,45	23,96%	41,84%	0,97	23,6%	37,67%	0,7	13,99%	20,87%
fp($base$)	6,13	23,5%	39,25%	3,45	30,65%	50,94%	2,14	25,72%	51,88%	0,80	50,51%	114,66%
fp($B = 32$)	4,15	31,16%	48,61%	2,14	40,47%	62,76%	1,34	30,56%	45,80%	0,57	44,12%	93,68%
fp(LRU)	6,81	18,44%	28,74%	3,68	25,59%	43,02%	2,74	23,6%	26,26%	0,89	63,52%	121,85%
	Ex	$\Delta(Dm)$	$\Delta(In)$	Ex	$\Delta(Dm)$	$\Delta(In)$	Ex	$\Delta(Dm)$	$\Delta(In)$	Ex	$\Delta(Dm)$	$\Delta(In)$

Table 2. $gm2$ **additional simulations with** $L2size = 2L1size$ **(**$8KB - 64KB$**).**

	L1size=2KB+2KB			L1size=4KB+4KB			L1size=8KB+8KB			L1size=16KB+16KB		
int($base$)	1,77	12,61%	12,78%	1,18	10,88%	11,83%	0,81	8,78%	8,34%	0,55	10,21%	9,87%
int($B = 32$)	1,64	16,77%	14,98%	1,11	15,09%	14,8%	0,78	10,9%	9,76%	0,53	13,81%	13,27%
int(LRU)	1,64	12,38%	18,13%	1,12	9,47%	14,46%	0,76	6,17%	9,8%	0,53	7,43%	10,97%
int($S_2 = 8$)	1,49	12,11%	11,2%	1,02	9,95%	10,13%	0,71	8,26%	8,49%	0,48	9,68%	10,43%
fp($base$)	4,29	12,71%	14,5%	2,61	9,26%	12,89%	1,06	21,11%	35,92%	0,61	6,79%	11,88%
fp($B = 32$)	3,93	21,4%	22,73%	2,38	16,13%	20,17%	0,9	29,14%	37,72%	0,57	4,56%	6,67%
fp(LRU)	4,39	14,73%	18,47%	3,26	6,43%	6,58%	1,13	37,48%	50,04%	0,63	13,09%	18,23%
fp($S_2 = 8$)	2,83	14,27%	17,25%	1,65	10,84%	11,11%	0,74	18,23%	29,57%	0,37	9,39%	20,04%
	Ex	$\Delta(Dm)$	$\Delta(In)$	Ex	$\Delta(Dm)$	$\Delta(In)$	Ex	$\Delta(Dm)$	$\Delta(In)$	Ex	$\Delta(Dm)$	$\Delta(In)$

Table 3. $gm2$ **additional simulations with** $L2size = 4L1size$ **(**$16KB - 128KB$**).**

ond stage latency of 4/7 cycles for divisions and 6/10 for square roots, simple/double precision respectively)

5.1. Components service times

For each cache level i , we define a base or characteristic time, T_{Li} processor cycles. It corresponds to the parallel access of the S_i tag and data arrays to deliver the requested item. Thus, T_{Li} is the Read-hit time for a level i cache. We estimate the Write-hit time to be $2T_{Li}$: the first T_{Li} to find the desired item and the second T_{Li} to write into the selected data array. L3uC and Main Memory cannot read or write a block in a single access, therefore their characteristic times must be multiplied by the required number of sub-operations. Besides, accesses to these levels will be delayed the penalty time of the corresponding bus.

On a miss in level i , T_{Li} is also the time spent on writing the block on level i once served by level $i+1$, in parallel to its transfer to $i-1$. Finally, we define T_{MLi} as the time to modify any of the state bits attached to each level i block (in general, $T_{MLi} \leq T_{Li}$).

Appendix A offers a more detailed insight of the times of each of the components of the model.

5.2. Baseline simulation model

To limit simulation time the baseline model sets all the parameters of the Reference Module, except for those belonging to L2uC, to values which are a good compromise among 1) nowadays machines sizes and timings, 2) limited size and on-chip integration, and 3) the high locality of the traces we have observed in the previous section, which imposes Mc sizes so that its variation has some influence on our system performance.

Baseline parameters are the following: $L1size$ has been set to $2KB+2KB$, direct mapping and $B = 16B$. $L3size$ is $512KB$ with $S_3 = 16$ and $B3 = 32B$. The bus between L1 and L2 is equal to the block size ($W_{21} = 16B$). Between L2 and L3, and between L3 and Main Memory, busses are $8B$ wide ($W_{M3} = W_{32} = 8B$) with a penalty of one and four cycles respectively ($Tbus_{32} = 1$, $Tbus_{M3} = 4$). Basic times for each level (T_{Li} and T_{lat}) are 1, 2, 3 and 20 for levels 1, 2, 3 and main memory[4], while state modification times have been set to 1 for all levels ($T_{ML2} = T_{ML3} = 1$). The sizes of the output Buffers are 1, 2 and 1 respectively for levels 1, 2 and 3. According with the section 4 results, Random replacement policy has been chosen for all contents management.

$L2size$ is varied from $8KB$ to $64KB$, keeping $S_2 = 4$ y $B_2 = 16$. Each plot presents both the CPI for each simulated configuration and two lines representing 1) the ideal CPI (assuming a memory system which always responds in a single cycle) and 2) the CPI without L2uC (assuming that each L1 miss is served directly from the external cache). As in previous section, measurements for each workload are combined in a non-weighted up way.

The first obvious conclusions about the baseline model are: 1) System performance is limited by the

[4]Times T_{lat}, T_{L3}, $Tbus_{32}$ and $Tbus_{M3}$ agree with those of the server DEC 7000/10000 [1]

memory hierarchy ($CPIideal < 1.25$), and 2) adding up L2 does not degrade this performance[5].

As shown in figure 4, Exclusion achieves better results than Inclusion and Demand for those L2 sizes closest to L1 ($8KB$ and $16KB$); individually, maximum difference is achieved 1) for integers in *cc1* with $L2size = 8KB$, where Inclusion needs 0.25 CPI and Demand 0.19 CPI more than Exclusion (10.5% and 7.9% worse, respectively) and 2) for floating-point in *doduc* with $L2size = 8KB$ where Inclusion needs 0.41 CPI and Demand 0.35 CPI more than Exclusion (14.2% and 12%). Unlike to what the miss ratio analysis showed and in contrast to that exposed in [11], the CPI achieved by Exclusion is greater than in Inclusion and Demand for relatively large L2uC sizes. This is because, in spite of achieving smaller miss ratios, L2uC Exclusion wastes more time by receiving all the blocks replaced from L1; however, Inclusion and Demand only need to bring into L2 the blocks which are dirty, while the rest of them needs just a replacement warning. In section 5.3.1 we evaluate an improvement in the L2 Exclusion block reception mechanism, which effectively decreases the swapping overhead.

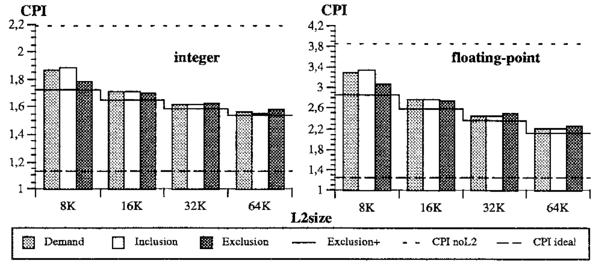

Figure 4. Baseline model CPI. $S_1 = 1$, $S_2 = 4$, $B = 16B$, $L1size = 2KB + 2KB$

We recall that the miss ratio analysis indicated always a little advantage of Demand over Inclusion. However, in section 2.3. we saw that Demand can cause some traffic increase between L2 and L3 when compared to Inclusion. Simulation results prove this compromise. Only when $L2size$ is similar to $L1size$, Demand achieves a smaller CPI than Inclusion due to its smaller miss ratio. For large $L2size$ —with comparable miss ratios in Inclusion and Demand— Inclusion performs sligthly better than Demand for most of the programs; in any case, differences are quite small.

[5]Except for *compress*, where some configurations show a worse CPI than that without second level. *Compress* has a high instruction locality which is absorbed by L1, but in practice no data locality. Local miss ratios for L2uC Exclusion are 78%, 72%, 65% and 54% for $8KB$, $16KB$, $32KB$ and $64KB$ respectively, hence many more of the L2uC accesses are delayed rather than helped

5.3. Improvements in L2uC Exclusion

Next, we introduce two improvements to increase the Exclusion performance at a very low cost that, if combined, become a solid alternative to on-chip contents management.

5.3.1 Block reception mechanism

To maintain strict exclusion, our protocol invalidates each block that is read and brought into L1. Therefore, invalid blocks may appear as a result of the exchanges between L1 and L2 that do not map into the same L2 set. Then a block replacement from L1 does not require a new replacement from L2 to L3 if there are such invalid blocks. It suffices to implement the replacement algorithm so it considers the valid bits in addition to its basic replacement policy. Thus, the whole operation can be performed in just T_{L2} cycles, the writing time, instead of the full $T_{L2}+T_{L2}$ cycles needed to search and write a set without invalid blocks. A similar mechanism in Inclusion or Demand would cause wrong behaviours since in these policies a replaced block from L1 can hit on L2, so the search can not be avoided.

5.3.2 Non-blocking scheme

In Exclusion, an L1 miss followed by an L2 miss results in a block read from L3 that is directly loaded into L1. So, L2uC remains free when the miss is detected and the request is sent to the upper level. We propose to modify the L2uC control to accept other L1 requests since then. A similar mechanism in Inclusion or Demand requires a more complex control and possibly the ability to read and write in parallel the data arrays, since the service to L1 can overlap the pending block reception.

In figure 4 the CPI computed for this improved version named Exclusion+ is shown. The decrease achieved by Exclusion+ in comparison to the basic model is clear for all the points of the simulation, with an average improvement of 6.2% in floating point and 2.9% in integers. The CPI for Inclusion and Demand is greater than that of Exclusion+ for all the simulations and all the programs. The maximum difference is achieved 1) for integers in *cc1* with $L2size = 8KB$, where Inclusion needs 0.41 CPI and Demand 0.35 CPI more than Exclusion+ (18.5% and 15.6% worse respectively) and 2) for floating-point in *fpp* with $L2size = 8KB$ where Inclusion needs 0.76 CPI and Demand 0.9 CPI more than Exclusion+ (19.1% and 21.9%).

5.4. Baseline variations

Some parameters of the baseline model have been individually changed: We have doubled 1) L3uC service time, 2) L2uC associativity, 3) block and bus sizes and 4) all cache sizes by increasing their number of sets. Table 4 shows some figures for these simulations. As in the baseline model, $L2size$ is set to 2, 4, 8 and 16 times $L1size$. For every selected point the tables show (from left to right) Exclusion+ CPI (Ex+), the difference between Demand and Exclusion+ CPI as a percentage of the later($\Delta(Dm)$), and the same figure between Inclusion and Exclusion+($\Delta(In)$). The row ($base$) means the same as in figure 4; rows ($T_{L3} = 6$), ($S_2 = 8$), ($B = 32$) and ($Sizes * 2$) are the results of the baseline variations presented in this paragraph. We can add the following conclusions to those of the previous subsections:

i) When increasing the L3uC service time, the differences between Exclusion and the rest of managements are also increased, because those which miss more often suffer from a greater penalty. So, due to the current trend of the increasing divergence between on-chip and off-chip times, Exclusion will become even more useful in the future.

ii) The benefits obtained using Exclusion+ are greater when moving towards smaller capacities. Presumably, a workload with less locality (increasing problem and code size) would have the same effect. Of course, if the performance of the system is far above the requirements of the workload, the contents management would not be an issue.

6. Conclusions

In this work we compare alternatives to two level on-chip cache contents management. Three alternative schemes namely Inclusion, Exclusion and Demand have been specified in the context of a Reference Module to compare their miss ratios and temporal performance. All previous quantitative studies about multilevel-cache memory assume an external second level, and/or ignore coherence support and/or do not compare different contents managements. Up to this work, the only coherent multilevel on-chip solution has been Inclusion.

Keeping Inclusion in an external third level, new protocols to manage lower levels in a coherent way have been developed. The protocol for Exclusion and its hardware requirements have turned up to be even simpler than those for Inclusion. We have carried out a quantitative analysis to each alternative in a uniprocessor environment. From the simulations it is clear that, in spite of achieving always a lower miss ratio, Exclusion can lead to worse CPI results than Inclusion and Demand due to the high cost of the L1 replacements. We propose a solution that decreases this overhead by improving the block reception mechanism and adopting a non-blocking scheme for L2 Exclusion. Apart from that, in the light of the obtained data, Demand seems to be discarded by its similar to Inclusion behaviour and its greater implementation difficulty when coherence support must be added. Our improved Exclusion scheme achieves 7.9% better CPI than Inclusion for floating-point and 4.12% for integers in the baseline simulation.

The foreseeable increase of the timing differences between on-chip and off-chip caches makes Exclusion even more attractive to manage on-chip contents. A criticism about our work is the high locality of the traces we have used, which has forced us to experiment with small first level caches; although we believe that our conclusions can be extended to larger sizes (lower localities), we are starting up experimentation using SPEC'95 to verify this presumption.

References

[1] B. R. Allison and C. van Ingen. Technical description of the DEC 7000 and DEC 10000 AXP family. *Digital Technical Journal*, 4(4):1–11, 1992.

[2] J. L. Baer and W. H. Wang. Architectural choices for multilevel cache hierarchies. In *Proc. Int. Conf. on Parallel Processing*, page 258, Aug. 1987.

[3] J.-L. Baer and W.-H. Wang. On the inclusion properties for multi-level cache hierarchies. In *Proc. 15th Ann. Int. Symp. on Computer Architecture*, pages 73–80, May 30–June 2, 1988.

[4] H. O. Bugge, E. H. Kristiansen, and B. O. Bakka. Trace-driven simulations for a two-level cache design in open bus systems. In *Proc. 17th Ann. Int. Symp. on Computer Architecture*, pages 250–259, May 28–31, 1990.

[5] J. Edmondson, P. Rubinfeld, and R. Preston. Superscalar instruction execution in the 21164 Alpha microprocessor. *IEEE Micro*, pages 33-43, April 1995.

[6] B. Ewy and J. Evans. Secondary cache performance in risc architectures. *Computer Architecture News*, 21(3):33–39, June 1993.

[7] J. D. Gee, M. D. Hill, D. N. Pnevmatikatos, and A. J. Smith. Cache performance of the SPEC92 benchmark suite. *IEEE Micro*, 13(4):17–27, Aug. 1993.

[8] P. Gelsinger, P. Gargini, G. Parker, and A. Yu. Microprocessors Circa 2000. *IEEE Spectrum*, 26:43–47, October 1989.

[9] P. Y.-T. Hsu. Introduction to SHADOW. Technical report, Sun Microsystems Inc, July 1989. Revision A.

	L2size=2L1size			L2size=4L1size			L2size=8L1size			L2size=16L1size		
int($base$)	1,73	8,40%	9,74%	1,65	3,51%	3,36%	1,59	2,09%	1,89%	1,54	1,61%	1,50%
int($T_{L3}=6$)	1,93	11,86%	13,06%	1,81	4,54%	4,11%	1,70	2,45%	2,08%	1,61	1,75%	1,56%
int($S_2=8$)	1,72	7,57%	9,50%	1,63	4,08%	3,68%	1,58	2,08%	1,85%	1,54	1,60%	1,49%
int($B=32$)	1,62	6,63%	6,98%	1,56	2,58%	2,33%	1,51	1,33%	1,14%	1,47	0,92%	0,85%
int($Sizes*2$)	1,55	4,81%	5,79%	1,50	2,16%	2,16%	1,46	1,29%	1,16%	1,43	1,04%	1,00%
fp($base$)	2,85	16,28%	17,49%	2,60	6,96%	6,93%	2,37	3,86%	3,62%	2,13	3,76%	3,73%
fp($T_{L3}=6$)	3,50	25,03%	21,48%	3,06	8,17%	6,98%	2,67	3,37%	2,70%	2,24	3,54%	3,47%
fp($S_2=8$)	2,81	15,08%	16,91%	2,50	12,26%	10,36%	2,35	3,83%	3,43%	2,10	3,85%	3,75%
fp($B=32$)	2,34	11,70%	15,74%	2,15	5,15%	5,20%	1,99	2,71%	2,52%	1,85	2,44%	2,37%
fp($Sizes*2$)	2,30	12,65%	14,92%	2,20	3,65%	3,97%	1,97	3,24%	3,37%	1,89	2,33%	2,35%
	$Ex+$	$\Delta(Dm)$	$\Delta(In)$	$Ex+$	$\Delta(Dm)$	$\Delta(In)$	$Ex+$	$\Delta(Dm)$	$\Delta(In)$	$Ex+$	$\Delta(Dm)$	$\Delta(In)$

Table 4. CPI baseline variations.

[10] N. P. Jouppi. Cache write policies and performance. In *Proc. 20th Ann. Int. Symp. on Computer Architecture*, pages 191–201, May 17–19, 1993.

[11] N. P. Jouppi and S. J. E. Wilton. Tradeoffs in two-level on-chip caching. In *Proc. 21st Ann. Int. Symp. on Computer Architecture*, pages 34–45, Apr. 18–21, 1994.

[12] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[13] S. Przybylski. *Cache and Memory Hierarchy Design: A Performance-Directed Approach.* Morgan Kaufman, San Mateo, California, 1990.

[14] C. Rodriguez, V. Viñals, J. Labarta, and R. Beivide. Cache memory reasignation models and their impact on multiprocessor performance. *Int. Journal of Mini and Microcomputers*, 11(1):9–12, 1989.

[15] R. T. Short and H. M. Levy. A simulation study of two-level caches. In *Proc. 15th Ann. Int. Symp. on Computer Architecture*, pages 81–88, May 30–June 2, 1988.

[16] R. Smith, J. Archibald, and B. Nelson. Evaluating performance of prefetching second level caches. *ACM Performance Evaluation Review*, 20(4):31–42, May 1993.

[17] S. Song, M. Denman, and J. Chang. The PowerPC 604 RISC microprocessor. *IEEE Micro*, pages 8–17, October 1994.

[18] P. Stenstrom. A survey of cache coherence protocols for multiprocessors. *IEEE Computer*, 23(6):12, June 1990.

[19] P. Sweazey and A. J. Smith. A class of compatible cache consistency protocols and their support by the IEEE Futurebus. In *Proc. 13th Ann. Int. Symp. on Computer Architecture*, pages 414–423, June 2–5, 1986.

[20] V. Viñals. Diseño de memoria cache multinivel con restriccion de capacidad. Proyecto DGA, convocatoria 1993 de proyectos de investigacion, 1993.

[21] W.-H. Wang, J.-L. Baer, and H. M. Levy. Organization and performance of a two-level virtual-real cache hierarchy. In *Proc. 16th Ann. Int. Symp. on Computer Architecture*, pages 140–148, May 28–June 1, 1989.

A. Component timing

Next, to help reproduce the simulation results, we give a detailed description of the service times of all the system components.

L1iC, L1dC Read-hit time = T_{L1}. Write-hit time = $2T_{L1}$. Block-loading time = T_{L1}, is the time needed for loading a block into L1 after delivered by the upper level.

L2uC Inclusion The same as L1dC using T_{L2} instead of T_{L1}. **L2uC Demand:** the same as Inclusion except for the possibility of a copy-back miss from L1dC; in that case the block is copied into *tbuf* at a cost T_{L2}. **L2uC Exclusion:** Read-hit time = T_{L2}, Block-loading from L1 time = $2T_{L2}$, the first T_{L2} to read and replace, if necessary, a block t and the other T_{L2} to load block u. State-change time (when a replacement warning is received): in Inclusion, $T_{L2}+T_{ML2}$, T_{L2} cycles to find the block and T_{ML2} to modify state. $T_{L2} \geq T_{ML2}$; in Demand, the same if it hits but if it misses, the warning is forwarded to *tbuf* at a cost T_{L2}.

L3uC Read-hit time = $\beta_{32}(Tbus_{32}+T_{L3}+Tbus_{32})$, the number of cycles since an address is sent until *xbuf2* has been loaded. An external L3 cache which can offer W_{32} bytes per access is assumed, being $\beta_{32}=B_2/W_{32}$. That is, a bus cycle ($Tbus_{32}$) to send the address, a cache access time (T_{L3}) and another bus cycle to send back the data are spent for each piece of a block being read. Write-hit time = $Tbus_{32}+T_{L3}+\beta_{32}(Tbus_{32}+T_{L3})$, where the first access ($Tbus_{32}+T_{L3}$) is needed to check tags, and the following β_{32} accesses are for writing. Both timings follow the interface model between an Alpha 21064 and its external cache in the system DEC 7000/10000 [1]. State-modification time (replacement warning) = $Tbus_{32}+T_{L3}+T_{ML3}(T_{L3} \geq T_{ML3})$. Block-loading time = T_{L3}.

Main memory Read time = $Tbus_{M3}+T_{lat}+\beta_{M3}*Tbus_{M3}$: number of cycles since the address is sent until *xbuf3* is loaded, where we assume a bus cycle to send the address, followed by a memory access time (T_{lat}) and β_{M3} subsequent transfers at a bus rate. Writing time = $\beta_{M3}*Tbus_{M3}+T_{lat}$. It corresponds to an interleaved Main Memory or a Dynamic RAM operating in some optimized mode.

Output buffers (*tbuf*, *ubuf* and *sbuf*) Writing time = 0 cycles if the buffer is not full, because writing into the buffer overlaps the miss resolution. If the buffer is full, Writing time = 1 since the moment when an entry is freed. A hit on an output buffer stalls a level until the entry which was a hit is downloaded into the upper level. The block will be recovered later on from the upper level in a conventional way.