Unbalanced Cache Systems

David L. Rhodes*
US Army CECOM
Research, Development and Engineering Center
Fort Monmouth, NJ 07703
d.l.rhodes@ieee.org

Wayne Wolf
Department of Electrical Engineering
Princeton University
Princeton, NJ 08544
wolf@ee.princeton.edu

Abstract

*The new concept of an unbalanced, hierarchically-divided cache memory system is introduced and analyzed. This approach generalizes existing cache structures by allowing different memory references (*e.g. *as possibly unevenly divided within an address-space) to be subject to various levels of caching as well as varied amounts of cache at each level. Under the assumption that the total cache size at a particular level is fixed, it is easily shown that at least one divided cache structure exists for which the miss-rate is the same as a single unified cache. By using alternate implementations, however, the method may provide a significant decrease in miss-rates as is shown via simulations. Specifically, SPEC95 benchmarks are used to demonstrate that the technique is effective for general usage but it may be even more useful for embedded systems where memory access patterns can be more fully controlled (*i.e. *via the compiler). In addition to improved miss-rates, another advantage is that the hit-time for multiple smaller caches may be smaller than for a single larger cache. Disadvantageous, but readily surmountable, electrical aspects are also discussed.*

1. Introduction

As implemented with VLSI technology as it has existed over the last few decades, the gap between processor speed and main memory latency has been ever widening [4]. One of the primary techniques used to address memory's inability to respond in a timely manner to ever speedier processors has been the addition of one or more levels of **cache** (the distance of the cache from the processor is called its **level**). A cache is an intermediate memory with interfaces to each of lower level caches or the processor on one side and to the next higher level cache or main memory on the other. The term processor here represents any type of computational element—for example a microprocessor, digital signal processor (DSP), a very-long instruction word (VLIW) special-purpose processor—which reads or writes to some form of memory.

Most processors are synchronous designs which perform (or execute) particular 'sub-components' of each instruction on different (CPU) cycles. Furthermore, many processors are pipelined or super-scalar, which means that sub-components of multiple instructions are executing simultaneously. Such designs place even further demands on the memory system, usually requiring multiple data read/writes and instruction fetches within the same cycle. With respect to a particular processor, the latency (delay) of a memory operation is effectively measured in terms of cycles, rather than time. So the effectiveness of a particular memory system may be evaluated or estimated in terms of latency measured in cycles of reads/writes/fetches for a particular execution **trace** (a sequence of references) or for some expected average reference mix.

As will be elaborated, the proposed hierarchically-divided cache structure generalizes existing cache structures by allowing uneven memory space and cache size **division** at each cache level. With respect to a particular memory reference, two forms of imbalance arise: in the amount of cache experienced at each cache level; and in the number of cache levels (before reaching main memory). Although some background is presented, in general it is assumed that the reader is well aquainted with memory system design for example as contained in [4].

2. New Structure

From the perspective of the processor, the basic memory operations are the reading/writing of data from/to a memory reference. A slightly finer distinction of memory operation types is usually used, including: **read** implying a data read; **write** implying a data write; instruction **fetch** implying the read of a instruction (self-modifying code, which would re-

*Mr. Rhodes is also with the Department of Electrical Engineering, Princeton University.

quire the ability to write instructions, is not common and requires special handling in many current processors). There is sometimes a further distinction regarding whether or not memory operations occur on behalf of the operating system or for a particular application, but this level of detail will not be necessary here.

For completeness of discussion, the operation of a typical two-level cache design as shown in Figure 1 is briefly described. A memory reference (read/write/fetch) is issued from the processor (P) to the first level cache (L1). If the reference is present in the L1 cache, then the response (read/write) takes place after the L1 **hit-time**. Otherwise, this reference causes an L1 **miss** and the reference goes to the L2 cache. The same action occurs for the L2 level; misses at the L2 level are finally fulfilled by the main memory (M). Note that references only travel towards the memory, while data travels in both directions depending if the request is a read/fetch or a write. As is depicted here, often two or more cache levels are employed to reduce the average processor-experienced hit-time as much as possible while maintaining reasonable costs and power consumptions (in current VLSI technologies, faster memories are usually more expensive and greater consumers of power).

For *address-based* (versus content-addressable) memories [4], the address is broken into three fields; the `<block>` which addresses the particular byte in the **cache line** (for byte addressable machines); the `<index>` which addresses 1 or more (for set-associative caches) cache lines; and the `<tag>` which is the remainder of the address. The `<tag>`, along with a few status bits, is contained in the cache along with the data for each cache line. The **associativity** of the cache is the number of distinct places a particular cache line can occupy, a cache with an associativity of 1 is normally called direct-mapped. A cache miss occurs if a reference's `<tag>` field does not match that of the `<tag>`(s) in the cache for the same index bits (within the associativity of the cache).

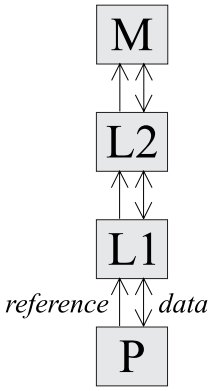

Figure 1. Typical 2-level cache design

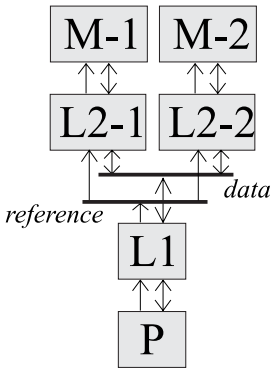

Figure 2. One possibility of the new cache/memory structure

The new, proposed cache memory structure fully generalizes typical multiple level cache memory structures and, conversely, existing (uniprocessor) cache structures are a subset of the new structure. First, at any one or all cache levels, the cache and subsequent memory space (either address or content-based) may be **divided**. As an initial example, one possibility for the new structure is shown in Figure 2. In this case, the L2 cache is divided (into parts called L2-1 and L2-2). In order to logically divide references, either the L2 caches must filter incoming references or a partial *decode* or similar operation must occur at the division point (not shown). It is important to recognize that there are *two* types of divisions occurring, the division in memory space, and the division in size of the caches (if any) allocated to each part of the division. Dividing a cache based on addresses is a well-known technique, as a recent example the PowerPC 604e processor uses odd/even cache line splits (a 1 bit decode) at the L1 level [10]. But in this part as well as all existing address-split caches (to our knowledge), both the cache size and memory space divisions are split evenly. In this new structure, *uneven* splits, in either or both of memory space and cache size, are possible.

Secondly, in all existing (uniprocessor) split-cache structures the memory space is effectively *re-merged* at the next level. While the unbalanced structures proposed allow for such re-merging, they more importantly can be used to keep the memory space (logically and physically) separated which then allows for an unbalanced *number of cache levels* in each 'caching path' (from processor to memory). The benefits derived from such imbalance(s) will be demonstrated in ensuing examples.

So in its general form, the hierarchically divided cache/memory scheme allows memory space division at any point, including 0 or more caches before or after the division. Memory space division at any point may or may not

be equal but is *orthogonal* meaning that every pairwise intersection of each divided memory space is null and that the union of the divided spaces is the entire memory space (at the division point). The orthogonal memory space division at each level implies that each memory reference has a single caching path between the processor and the main memory. Non-orthogonal cache/memory space, as used for various multiprocessor memory systems such as cache-coherent, non-uniform memory access (CC-NUMA) [13], cache-only memory architecture (COMA) [12], and reactive NUMA [3], do not readily appear useful for uniprocessor systems.

Note that the proposed unbalanced, hierarchically divided cache system is essentially structural from a physical viewpoint and transparent from a logical addressing perspective. This makes them *compatible with all existing cache protocols*, including various write policies (*e.g.* write-through, write-back), associativities, sub-block policies, both content and address-based referencing, associative and non-associative schemes, inclusive/non-inclusive caches, paged or virtual memory translations, various prefetching schemes, etc.

Several types of cache modifications, especially at the L1 level, have been proposed; most of these are aimed at improvement of *conflict* cache misses (a conflict miss is one which is caused due to collisions, rather than compulsory or capacity types, see [4]). For example, in recognition of the different characteristics of instruction fetching and data reads/writes, the L1 cache is sometimes split based upon memory access *type* (the instruction, I, and data, D, caches) and this results in the so-called 'Harvard architecture' (strictly speaking a Harvard architecture is one where instructions and data memory spaces are completely separated, but the term is also used in the case where just the L1 cache is split). Extending this further, the Non-Temporal Streaming (NTS) cache makes uses of a parallel structure for the data cache, with references separated based on a dynamic assessment of their (temporal) 'access patterns' [11].

The memory technology called Enhanced DRAM (EDRAM) [6] integrates static RAM (SRAM) and dynamic RAM (DRAM) on a single part where, in one implementation, the SRAM acts as a cache. Although this technology aids in the flexibility of placing cache in the memory hierarchy, it lacks the features of presented here, namely arbitrary structure and imbalance. Several other related modifications have been studied as well, but the structure proposed here is different in that it separates caches strictly by reference. Furthermore, this is generalized to an hierarchical, possibly unbalanced, form at all cache levels.

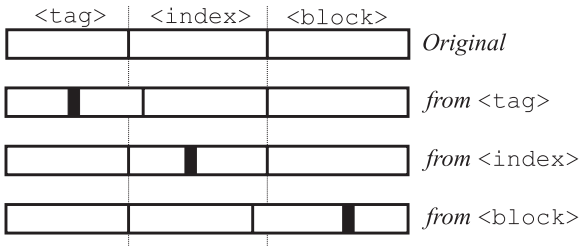

Figure 3. Comparison of tag, index and block fields at a 1-bit divided memory point. Dashed lines show alignment to original and differences from this are 1-bit.

2.1. Considering Divisions from 1-Bit Decoding

Given this generalized structure, consider a more specific situation. Suppose that at a 2-way cache division point that the total amount of cache at the division remains constant and that an addressed-based memory space division is done using a single 1-bit decode. For simplicity at this point, consider a division assuming that the block size is kept constant, that a single address bit is decoded, and that the cache size is halved for each division. We then have the following possibilities:

Case–I. Bit is in original `<tag>` field. The miss-rate may be higher or lower. Increased collisions in the smaller caches may or may not be mitigated from the memory space division that occurred in the `<tag>` field.

Case–II. Bit is in original `<index>` field. Here the miss-rate is *exactly* the same as it was in the original case. Addresses which would 'collide' in the smaller caches are separated into the two divided caches.

Case–III. Bit is in original `<block>` field. The miss-rate may be higher or lower just as in Case I.

These situations may be visualized with the aid of Figure 3. The dark box represents the bit selected for the memory space division. For the cases here, it was assumed that the block size is constant and that the cache for each division is equal, therefore the `<index>` field is 1-bit shorter and the `<tag>` field length is the same in all cases. Unbalanced divided structures are more general than the cases suggested above, but these assumptions serve as a good starting point for investigation.

3. Results

Trace-based simulations are used to demonstrate the advantages of imbalanced cache structures. All of the simulations use Sparc-processor traces of SPEC95 programs as instrumented with the QPT2 tool [8, 9]. In cases where the SPEC benchmark was composed of several program runs with different input files, an arbitrary one was chosen to generate the traces for that benchmark (*e.g.* `expr.i` in the case of `126.gcc`). Benchmarks were compiled using `egcs v1.03` [2] in accordance with the mandated SPEC compiler and application options—the one exception was for the `130.li` (LISP interpreter) example where optimization was turned off. For each of six integer and six floating-point benchmarks, 100,000,000 (100M) traces were generated for a total of 1.2G traces total. Each of these twelve benchmarks were individually simulated using a combination of DineroIV [5] and PDATS-based [7] cache simulators. Since the miss/hit rates presented include all 100M traces, cold-start/compulsory miss effects are included. Although all data presented is for the Sparc architecture, several experiments using MIPS-3000 traces revealed similar advantages—implying in an ad hoc way that unbalanced cache structures may be applicable to many architectures.

In the ensuing sub-sections, two main results are presented; the first highlights the potential advantage of memory space imbalance at a single cache level, while the second shows cache size imbalance advantages. The first comparison concerns the L1 level only where total amount of cache is constant for several designs—while the second subsection considers the case where cache size imbalance is used, in this case at the L2 level.

3.1. Imbalanced memory space example

Using the SPEC95 benchmarks mentioned above, Table 1 compares a Harvard architecture, an uniform and several divided L1 cache structures, where a total of 32 KB of L1 cache is present in all cases. Specifically, the comparison considers:

- a typical 32KB unified cache (labeled 'Base Case').
- a Harvard architecture with 16KB for each of the instruction and data caches.
- 2-way and 4-way divided caches where both the memory space and cache sizes are divided equally (the 2-way case is the same as an address-split cache, albeit the bit used for the division has been optimally selected). The 2-way case uses bit `<1>` while the 4-way case uses bits `<0,1>` for memory space division (note that bit `<0>` is the least significant bit). To maintain a constant L1 total size, each of the 2-way L1 caches are 16KB while each are 8KB for the 4-way division.
- a divided cache where address space is unequally divided into two equally sized caches. Bits `<1,15>` are used for the (uneven) address split; one-quarter of the address space, bits `<1 = 0,15 = 0>`, is directed to one side while the remaining three-quarters of the space, `<1 = 0,15 = 1>` and `<1 = 1,15 = 0>` and `<1 = 1,15 = 1>`, is directed to the other side. Note here that the memory space is unevenly divided but that the cache sizes are evenly divided.

In these cases, all caches have a block size of 32 bytes and are direct mapped (associativity = 1). The table appears in order from worst to best, with respect to both read and total miss-rates. The Harvard architecture is an improvement over the unified ('Base Case') cache; the 2-way equally split and 4-way equally split cases follow. However, as is readily apparent the '2-way unequal' case is shown to be far superior to all—it approximately *halves* the number of both total and read misses over the typical uniform cache.

3.2. Imbalanced cache size example

Results exploiting unequal cache sizes are are done at the L2 level to highlight some additional advantages of imbalanced structure. As a baseline comparison, a typical 2-level, write-back, inclusive cache system (as is shown by Figure 1) with the following characteristics is used: L1 is unified of size 32KB, has an associativity=1 and block size=32 and has an 256KB L2 cache. This case is denoted as 'Base Case' here—the L1 cache is indeed the same as that of 'Base Case' in Table 1. The first divided cache structure under consideration, shown in Figure 4, divides the L1 cache using bit `<1>` and has 16KB on either side of the division (the graphical image is simplified from the previous figures which showed detailed reference and data connections). Both the memory space and cache size division here are equal—note that the L1 level is the same as in the case labeled '2-way equal' of Table 1. Finally, Figure 5 shows an unbalanced (with respect to cache size) divided cache. Here the entire 1MB of L2 cache is devoted to the '-1' side. For now, each of the three structures have a total of 256KB L2 and all L2 caches have associativity of 2 and a block size of 32 bytes.

The upper part of Table 2 compares these structures with respect to the read miss-rates and memory traffic. At the L2 level, the only advantage that Case A has over the Base Case is merely that of *maintaining* the separation started at the L1 level. That is, if the memory space were re-merged

Table 1. Comparison of L1 cache types with 32KB total memory

	Base Case	Harvard	2-way equal	4-way equal	2-way unequal
Total Misses	19,998,682	19,760,884	17,365,880	16,674,166	10,807,592
Read Misses	16,008,762	15,812,916	14,097,994	13,630,374	7,982,621

Table 2. Some L2 Comparisons

COMPARISON WHEN TOTAL L2 SIZE IS 256KB

		Base Case	Case A	Case B
L2	Read Misses	4,053,266	1,428,252	764,090
Mem	Total Refs	6,355,516	4,606,729	3,039,955

COMPARISON WHEN TOTAL L2 SIZE IS 1MB

L2	Read Misses	2,407,168	458,863	286,880
Mem	Total Refs	3,922,016	2,519,561	2,016,077

Note: L1 performance for Case A and Case B is that listed as '2-way equal' in Table 1

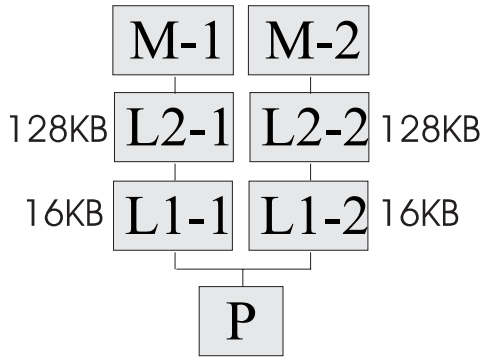

Figure 4. Case A

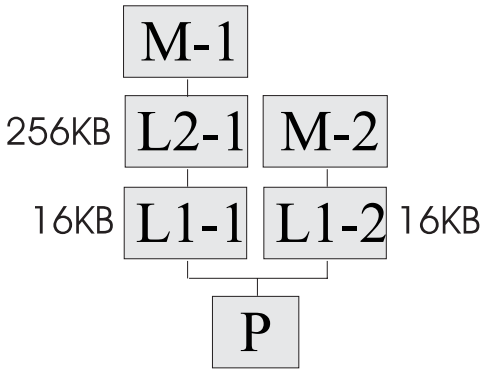

Figure 5. Case B

after L1, as would be the case in say the address split cache of the PowerPC 604e, then the (inferior) Base Case results would apply. Returning to the focus on unbalanced systems, Case B is an example of unbalanced cache size structure (in this case to the extreme of having no cache on one side) with an even memory space division. The lowered miss-rates and lowered main memory references—summed from from both the L2-1 and L1-2 of course—afforded by this case clearly demonstrates the benefit of unbalanced cache size structures.

Table 2 shows only read/fetch misses since these are more critical to performance; the unbalanced systems also improve other performance metrics, *e.g.* write miss-rates, castouts (dirty line write-backs), etc. but these are not shown here. For an L2 cache with total size of 256KB, the unbalanced structure reduces the read miss-rate from 4,053,266 to 764,090 an improvement of over a factor of 5 and reduces memory traffic by a factor of 2. The lower portion of Table 2 shows the same information, but now assuming the total amount of L2 cache is 1MB rather than 256KB. A factor of 2 times improvement in memory traffic remains (approximately) and a factor of over 8 times improvement is observed in read miss-rate.

4. Electrical Considerations

In addition to reduced miss-rates, the proposed method may offer lower cycle-times (average hit-time); although there are competing engineering trade-offs to be considered in trying to reach this objective. Increasing the cardinality of a division at any memory point is hampered by

Table 3. Net Cycle Time Comparisons, Total L2 is 256KB

		Base Case	Case B
L1	Net Hit Rate	98.343%	98.571%
	Hit Time	0	0
	Miss Penalty	0.0	0.0
L2	Net Hit Rate	1.319%	1.279%
	Hit Time	5	5
	Miss Penalty	0.06595	0.06395
Main	Net Hit Rate	0.337%	0.150%
	Hit Time	24	24
	Miss Penalty	0.0811	0.0360
Avg. Penalty (cycles)		0.14705	0.09995

two factors. The first is the latency introduced by decoding or gating memory space splits (*i.e.* for typical address-based memory this involves a Boolean-valued functions of address bits while for content-addressable memory it may be Boolean-valued hash functions of keys). Much or all of this latency may be *hidden* in that for many designs the result of the decision need not be valid until the *end* of the access—all branches can begin access at the beginning of the access leaving the final resolution toward the end of the cycle. Keep in mind that these bits must be decoded eventually anyway. Furthermore, when divided the smaller caches may have faster hit-times. Speedier access from multiple, parallel memory as well as the possibility of *overlapping* sequential memory access requests will also aid in latency improvements in divided caches (much the same as in interleaved memories). So the net result of this adverse 'decoding' factor may well be nil. However, the second adverse factor is the increased capacitance in the reference and data busses, which results in additional delay. Additional bus driver circuits may be able to overcome some of this up to sensible limits. Regardless, both of these factors place effective limits on the amount of division possible and each is ultimately very much dependent on the particular implementation technology.

Consider some typical memory characteristics as implemented using current technologies: the latency of DRAM main memory may be in the 10–100 cycle range and it may be sized in the 4MB to 1GB (or more) range. The hit-time of the L1 cache, perhaps sized in the 4KB to 128KB range, is often times 0 (cycles). The second level cache, with a size 32KB to 1MB, may have a hit-time in the 2–20 cycle range. Using representative numbers from these ranges, cycle time comparisons between the Base Case and Case B, with miss-rates from Table 2, is summarized in Table 3. The average memory access time penalty is reduced from about 0.147 to about 0.1, approximately a **32% reduction**. Note that this reduction is strictly due to miss-rate improvements; further improvements would apply if cache size splitting resulted in caches with lower hit-times.

5. Unbalanced Cache System Design

A good question is: *How can unbalanced, hierarchically divided caches be designed?* The use of bit `<1>` for Case A and Case B discussed previously was chosen using the aid of a program devised to assist in designing divided memory structures. This design aid determines the effective miss rates for possible single bit divisions under the assumption that the subsequent amount of cache will be divided equally. This tool, coupled with a tool to extract cache misses and castouts, was used in a hierarchical manner to design hierarchically divided cache structures by first deciding level 1, and recursively using it on subsequent levels. If at any point 'reasonable' improvements are not made by using a divided scheme, then this point is left undivided.

However, this locally greedy design approach, iteratively applied at each cache level, is far from optimal (since this design problem does not exhibit optimal sub-structure, a greedy approach in general will not be optimal [1]). As such, a more comprehensive tool will ultimately be needed which includes the following features:

- Looks at *n*-way divisions rather than only 2-way.
- Simultaneously and completely considers various unequal memory space and cache size divisions.
- Considers situations other than when total cache memory at each level is fixed. That is, design tradeoffs might be made *between* the amounts of L1 and L2 cache.
- Considers the total cache/memory system rather than working from the processor outwards.

Such features, as well as some others, would enable a more comprehensive design for unbalanced divided cache/memory structures.

6. Concluding Remarks

The new idea of using cache imbalance in a hierarchical manner has been introduced. It was shown that such structures can outperform Harvard (in the sense of I/D split L1 caches) and traditional linear cache memory systems, even

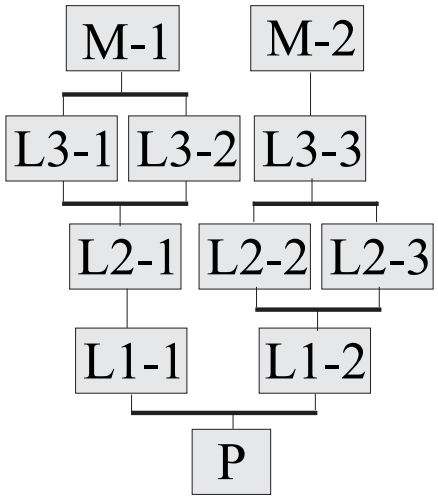

Figure 6. More Generalized Possibility

within the assumption that the total amount of cache memory at each level is constant. Such structures, however, offer the ability to more flexibly position memory and cache, perhaps with varying response times, in a very generalized hierarchy. They also offer the ability to possiblity partition the logical address space in an uneven manner. They are applicable to both address-based and content-based systems, and are compatible with any memory protocols. Although shown beneficial for general-purpose computing via use of SPEC95 traces, this approach may be especially effective in embedded systems where memory access patterns are more predictable. Conversely, in an unbalanced cache system, compilers could take advantage of the imbalance by placing often-accessed parts of the datum in the 'most cached' memory space(s).

Two forms of imbalance were independently shown beneficial, memory space imbalance with evenly sized cache division and cache size imbalance with even memory space division. Of course, these forms of imbalance may be combined (even at a single caching level) for potential further gains. The assistance of more advanced design tools, as discussed in the previous section, would be critical to such an endeavor. Figure 6 shows a more generalized possibility. Here the various caches may be sized differently and note that the possibility of re-merging either at the memory level (L3-1 and L3-2 into M-1) or at a cache level (L2-2 and L2-3 into L3-3) is shown. Creating such a general structure including cache sizing for some optimal condition (*i.e.* cost, power, speed) is a challenging problem as has been mentioned.

It is interesting to think about why unbalancing, in either of memory space or cache size (or both), works. Caching itself works only because real programs exhibit the principle of "locality of reference" (over temporal frames) [4]. While balanced memory-space/cache-size division can reduce conflict misses (as is witnessed in Table 1), there is no guarantee of this (see Section 2). But unbalancing can improve this even further—as demonstrated with the SPEC95 traces. This must mean that different parts of the address space have different caching requirements. In some sense this should be expected, it merely means that the degree of locality of reference exhibited is a function of the area of memory space.

All results here assumed that the cache structure and division logic were constant throughout *all* examples (SPEC benchmarks). However, a dynamic (or configurable computing) version would allow the cache system to be configured on say a per application basis or possibly even at a finer granularity. Such a dynamic approach would, of course, further aid the technique in reducing hit-time making it even more attractive.

In summary, the prime advantages of hierarchically divided cache/memory structures are:

- They are rather simple to implement.
- A configuration may be found in which the miss-rate improves.
- Each of the smaller caches in the division may be electrically faster.
- Different parts of the memory space can be cached differently including both different amounts of caches as well as with varying number of levels.

While the only disadvantages seem to be those related to electrical considerations:

- Additional decoding/gating is needed at memory space division points. This does not seem too significant in that (for address-based caches/memory) these bit(s) would eventually have to be decoded anyway.
- Additional capacitance at division points—may have negligible effect depending on drivers/technology.

A US Patent has been filed based on this work.

References

[1] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

[2] Cygnus Solutions, Inc. `//egcs.cygnus.com/`.

[3] B. Falsafi and D. A. Wood. Reactive NUMA: A design for unifying S-COMA and CC-NUMA. In *24th ACM Int. Symp. on Computer Architecture*, pages 229–40, 2–4 June 1997. Denver, CO.

[4] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Francisco, CA, 2^{nd} edition, 1996.

[5] M. Hill and J. Elder. DineroIV trace-driven uniprocessor cache simulator. See: `//www.cs.wisc.edu/~markhill/DineroIV`, 1998.

[6] S. S. Iyer and H. L. Kalter. Embedded DRAM technology: opportunities and challenges. *IEEE Spectrum*, 36(4):56–64, Apr. 1999.

[7] E. E. Johnson and J. Ha. PDATS: Lossless address trace compression for reducing file size and access time. 13^{th} *IEEE Int. Phoenix Conf. on Computers and Communications*, pages 213–9, 12–15 April 1994.

[8] J. R. Larus. Efficient program tracing. *IEEE Computer Magazine*, 26(5):52–61, May 1993.

[9] J. R. Larus and E. Schnarr. EEL: Machine-independent executable editing. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI'95)*, pages 291–300, La Jolla, CA, 18–21 June 1995.

[10] Motorola Inc. *PowerPC 604e: RISC Microprocessor User's Manual*, Mar. 1998.

[11] J. A. Rivers, E. S. Tam, and E. S. Davidson. On effective data supply for multi-issue processors. In *IEEE International Conference on Computer Design (ICCD'97)*, pages 519–28, 12–15 October 1997.

[12] A. Saulisbury, T. Wilkinson, J. Carter, and A. Landin. An argument for Simple COMA. In *First IEEE Int. Symp. on High-Performance Computer Architecture*, pages 276–85, 22–25 January 1995. Raleigh, NC.

[13] Z. Zhang and J. Torrellas. Reducing remote conflict misses: NUMA with remote cache versus COMA. In *Third IEEE Int. Symp. on High-Performance Computer Architecture*, pages 272–81, 1–5 February 1997. San Antonio, TX.