# Caches Versus Object Allocation

Jochen Liedtke
IBM T. J. Watson Research Center *
GMD — German National Research Center for Information Technology †
jochen@watson.ibm.com

## Abstract

*Dynamic object allocation usually stresses the randomness of data memory usage; the variables of a dynamic cache working set are to some degree distributed stochastically in the virtual or physical address space. This interferes with cache architectures, since, currently, most of them are highly sensitive to access patterns. In the above mentioned stochastically distributed case, the true capacity is far below the cache size and largely differs from processor to processor. As a consequence, object allocation schemes may substantially influence cache/TLB hit rates and thus overall program performance.*

*After presenting basic cache architectures in short, we sketch an analytical model for evaluating their true capacities. Some industrial processors are evaluated way and potential implications for memory management techniques are discussed.*

## 1. Rationale

This paper deals with the *secondary* costs of memory allocation. Does a program the objects which of have been dynamically allocated and perhaps reallocated by garbage collectors behave and perform like other programs? And can we reduce negative effects by modification of memory management algorithms and/or by hardware?

Suppose that you use a simple block structured programming language which does not support pointers, allocates variables solely on the stack and passes parameters and results always by value. When running a program written in such a language, select by random a sequence of a few thousand instructions and mark all data (variables) accessed in the sequence. There is a good chance that this (fine-grained) working set has a highly systematic structure: all addresses fit into a relatively small interval which is the stack's hot part, and there are only few unused holes in it. If the size of the interval is less than or equal to the data cache size, you can expect a hit rate of nearly 100%.

(Un)fortuantely, programming languages are not as restricted as assumed above. They have reference parameters, pointers, heaps and sometimes use rather sophisticated memory management mechanisms including garbage collection. An extreme example might be a concurrent logic programming language, where all variables are written at most once and data structures are implemented as pointer arrays. As a consequence, a data working set is usually spread over a fairly large interval and has a stochastically influenced structure. Dynamic memory management usually leads to more or less randomly allocated variables.

Since caches and translation lookaside buffers (TLBs) are in most cases not fully-associative, the effect of stochastically structured working sets is not obvious. As will be shown later, random influences lead to an increase of cache conflicts and thus to reduced hit rates.

Cache and TLB performance is crucial for today's systems and will become even more crucial for tomorrow's processors. For illustration: on a fast 3-issue processor, a primary cache miss (and secondary cache hit) may lead to a 20 cycle delay corresponding to a delay of 50 to 60 instructions, even if a few subsequent instructions may be executed during miss handling. TLB misses induce similar costs. In this situation, reducing both hit

*30 Saw Mill River Road, Hawthorne, NY 10532, USA
†GMD SET–RS, 53754 Sankt Augustin, Germany

rates by only 1%, from 99% to 98%, can make the processor run 1.3 times slower.

## 2. Cache associativity

A general introduction into caches can be found in [14]. Special architectures are described in [2, 18, 16, 1, 3, 10]. This section deals only with aspects related to associativity, since they determine how a cache reacts to usage patterns. Cache addressing and tagging (virtual or physical), line size, replacement algorithms, write strategies and coherence protocols are not discussed here.

In the case of a *direct-mapped* cache, some bits of the physical or virtual address $a$ are used to form a cache index. This index selects a single cache entry which then is checked against the address $a$. Direct-mapped caches are simple, fast and cheap. For a given die size, the direct-mapped architecture permits the fastest [17] and the largest [13] cache (the cache with the most entries). On the other hand, they tend to *cache conflicts* or clashes, i.e. cache misses caused by two or more addresses which are mapped to the same index and thus cannot be held in the cache simultaneously.

An *$n$-way set-associative* cache can contain up to $n$ memory entities with map-equivalent addresses per set, because the index is used to select a set of $n$ entries instead of a single one. This $n$-fold associativity reduces the conflict probability and accordingly improves the hit rate. On the other hand, an $n$-way cache needs more die size than a direct-mapped one and is not quite as fast.

In practice, direct-mapped (Mips R4000), 2-way (Pentium), 4-way (486, PowerPC 604) and 8-way caches (PowerPC 601) are used.

## 3. Probabilistic capacity

Numerous studies use the cache hit rate (the ratio of accesses which hit in the cache to accesses in total) as a measure of the cache's quality. Unfortunately, the hit rate not only depends on the cache architecture but also heavily on the dynamic program or system behaviour. We cannot predict hit rates; we can only measure them for a given program or a given set of programs and given data sets. Large amounts of heuristic work has been invested to find benchmarks which are in some respect "representative". Some people hope that the results of such benchmarks are valid also for "similar" programs and that many practically relevant programs are "similar". This approach has at least two weak points:

- It cannot be predicted how upcoming new applications, new programming styles, even new programming languages or code generators will effect the hit rate.
- It cannot be predicted how the combination of two or more applications will effect the hit rate.

| Processor | Size | Ways | Page Size | Use |
|---|---|---|---|---|
| 486 | 8 K | 4 | 4 K | I+D |
| Pentium | 8 K | 2 | 4 K | I |
| | 8 K | 2 | 4 K | D |
| PowerPC 601 | 32 K | 8 | 4 K | I+D |
| PowerPC 604 | 16 K | 4 | 4 K | I |
| | 16 K | 4 | 4 K | D |
| Alpha 21064 | 8 K | 1 | 8 K | I |
| | 8 K | 1 | 8 K | D |
| Mips R4000 | 8-32 K | 1 | 4 K | I |
| | 8-32 K | 1 | 4 K | D |

**Table 1.** *First Level Processor Caches.*

Measuring hit rates is not sufficient to understand caches. We need a measure which gives us more insight into the cache properties and is not as program dependent as the simple hit rate. There are strong similarities between caches and paging. The most important idea to understand paging was introducing *working sets* [4]. This abstraction turned out to be both sufficiently independent of concrete program behaviour and sufficiently expressive for performance evaluations.

Accordingly, we define the cache working set of a sequence of $n$ memory accesses to be the set of $\omega$ (different!) memory entities accessed by this sequence. (A memory entity is the memory unit which can be held in one cache entry.) If the complete cache working set fits completely into the cache, we can be sure that the instruction sequence can be executed fast. Besides potentially loading

the cache working set, no cache miss at all will occur: the worst case miss rate is $\omega/n$. Otherwise, if the complete cache working set does not fit simultaneously into the cache, we cannot make relevant statements about cache hits and misses: the worst case miss rate is $n/n$.

Now, we argue the other way around. Assume an infinite sequence of accesses. $N_\omega$ denotes the maximum prefix length of this sequence so that its cache working set contains $\omega$ entities. The corresponding cache working set is denoted by $W_\omega$. The cache capacity related to the given sequence of accesses is the value $C$ such that the first $C$ members of the working set fit simultaneously into the cache whereas the first $C+1$ do not, i.e., $W_C$ fits into the cache and $W_{C+1}$ does not.

This *capacity* seems to be an essential cache property. Unfortunately, for most cache architectures, it heavily depends on the working set structure: the capacity of a direct-mapped cache can range from 1 (all accesses mapped to the same cache entry) up to $n$ (all mapped to different entries). To get rid of this dependency, we use *expected capacity* and *probabilistic capacity*. These terms are defined more precisely in [11]; here we describe them informally:

**Expected Capacity** is the "average" capacity $\widetilde{C}$ over the working sets of all possible access sequences. If you select such a working set at random, $\widetilde{C}$ is the expected value of the corresponding capacity.

**Probabilistic Capacity** is the maximal capacity $C_p$ such that with probability $p$, a randomly selected working set relates to a capacity of at least $C_p$. Usually, $p$ is chosen very close to 1.

Accordingly, we use the term *systematic capacity* as a synonym for the cache size.

An important parameter for determining the expected and the probabilistic capacity is the method for selecting the working set. Generally, any method can be specified by an according probability distribution.

We concentrate on equally distributed working sets, more precisely, we assume that any working set with a predefined size has the same probability of occuring. In the case of stochastically selected working sets, the probabilistic capacity is also called *stochastic capacity* and the expected capacity is called *expected stochastic capacity*.

Why did we choose the stochastic model? There are presumably no hard mathematical arguments for this choice but some serious intuitive and pragmatic arguments:

- In practice, stochastic selection is presumably the worst case. A systematic selection is either better than a stochastic one or can be randomized, e.g. by a hash function or even by simply xoring the address by a bit mask. Therefore a cache architecture well-suited for stochastic selection should perform well in most cases.
- In practice, stochastic influences become more and more important. Among other reasons, increasing cache and TLB size, increasing concurrency and object-oriented programming techniques are responsible for this effect. Therefore, a stochastically bad-performing cache architecture will presumably not be very efficient in practice.

In [11], we show how expected and probabilistic capacity of various cache types can be calculated analytically.

Assume that for 8K cache, we find an expected stochastic capacity $\widetilde{C} = 50\%$ and a stochastic capacity $C_{99\%} = 25\%$. What does this mean? The naive interpretation of the expected capacity is: we expect that programs with cache working sets up to 4K perform fast. This interpretation is wrong!

We can be relatively sure (precisely 99%-sure) that programs with cache working sets up to 2K, the stochastic capacity, perform fast. For a stochastic capacity $C_p$, we can expect a worst case miss rate of

$$p\,\frac{C_p}{N_{C_p}} \;+\; (1-p)\,\frac{N_{C_p}}{N_{C_p}} \quad\approx\quad \frac{C_p}{N_{C_p}} + (1-p) \quad .$$

We do not have a similar approximation based on the expected capacity. In our example, a cache working set of 3.5K may lead to a horrible miss rate, although it is not larger than the expected capacity.

**Pragmatic conclusion:** Cum grano salis, we can use probabilistic and expected capacity as probable lower and upper bounds for efficiently performing cache working sets. As long as the working sets do not exceed the probabilistic capacity (with $p \approx 1$), we can be relatively sure that the program performs fast. On the other hand, we should be surprised, if a program heavily using cache working sets beyond the expected capacity performs well.

## 4. Capacity analysis

For comparing some cache architectures, in this section always 32-byte cache lines are assumed. Figure 1 shows the stochastic capacities with $p = 99\%$ for conventional direct-mapped, 2-way, 4-way and 8-way associative caches from 4K up to 32K size.

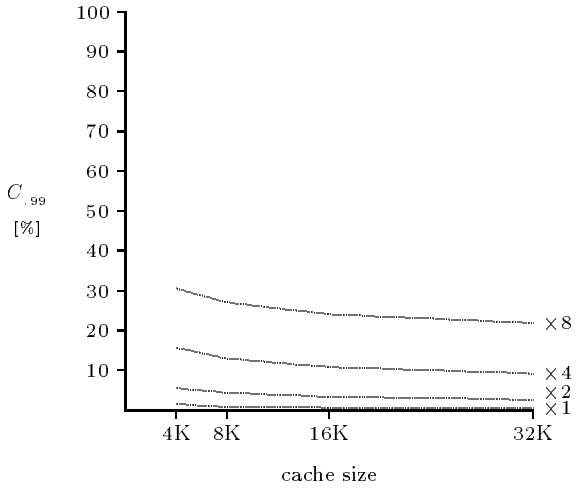


**Figure 1.** *Stochastic Capacity, n-way Caches*

The capacity is given as relative capacity, where 100% denotes the complete cache, i.e. the size given by the x-axis. Direct-mapped caches (×1) have an extremely low stochastic capacity, mostly below 1%; 2-way caches are slightly better with 4%. Although 4-way and 8-way caches have a 15 respectively 40 times higher stochastic capacity than direct-mapped caches, their absolute values, 11% and 25% respectively, are still not very high.

Figure 2 shows the effects of complementing a 4-way cache by various overflow caches. (An overflow cache is similar to a victim cache [8].) Only 4 overflow entries increase the stochastic capacity by roughly 20%, i.e. more than doubles it. 50% can be reached by 16 overflow entries.

All the capacity evaluations discussed until now assume purely stochastic cache or TLB working sets. In practice, stochastic (e.g. on the heap) and systematic (e.g. on the stack) influences coexist. Does this substantially increase the capacities? We examine working sets built by two simultaneously active mechanisms: the first is a pure stochastic selection, the second a pure systematic selection which chooses subsequent adjacent entries, i.e. a

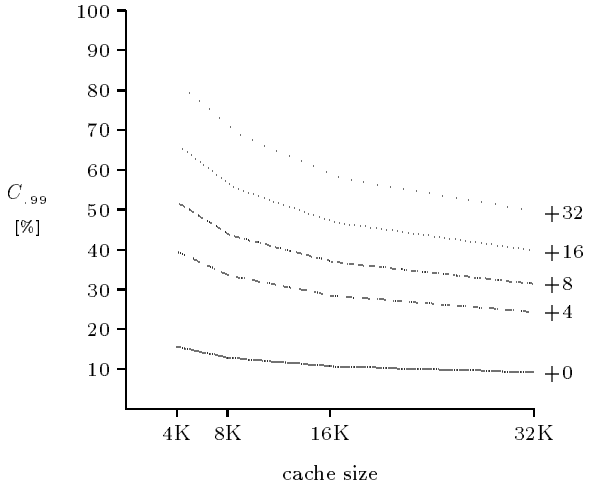


**Figure 2.** *Stochastic Capacity, 4-way Cache*

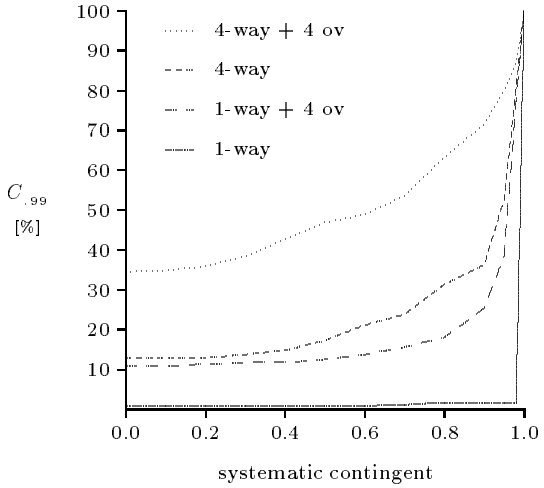


**Figure 3.** *Probabilistic Capacity of 8K-Caches for Mixed Stochastical and Systematic Selections*

compact part of memory. Now we start with a pure stochastically determined situation and then increase the systematic contingent. A systematic contingent of 0.6 means that 60% of any cache working set is chosen systematically and the remaining part is chosen stochastically. Figure 3 shows probabilistic capacities ($p = 99\%$) for direct-mapped and 4-way caches without and with a 4-overflow cache. $C_p$ is measured for systematic contingents from 0.0 (purely stochastic) up to 1.0 (purely systematic). In the latter case, cache capacity is of course always 100%; but even limited stochastic influences, like systematic contingents of 0.7 or 0.8, reduce the capacity nearly to the purely stochastic case. From this, we conclude that stochastic capacity is an acceptable measure for programs which are influenced by dynamic memory management and garbage collection.

**Counterarguments**

Many existing processors have direct-mapped or 2-way caches. Do these caches really perform as bad as the above cacpacity analysis suggests? There are two obvious counterarguments:

1. The mentioned processor vendors made benchmark-based hit-rate measurements for various cache architectures. Obviously, some of them concluded that improved associativity does not pay in relation to the improved hardware costs.

2. Measurements and simulations, especially Hill and Smith [5], state that improving associativity beyond 2 ways has only very limited effects.

Indeed, Hill and Smith show that the influence of associativity is limited *in scenarios where capacity misses (which here should better be called size misses) dominate conflict misses*. They explicitly say that "trace samples that exhibit unstable behaviour (e.g., a particular doubling of cache size or associativity alters the miss ratio observed by many factors of two) have been excluded from both groups [of trace samples]" [5] (p. 1615). Not surprisingly, under this premise size misses dominate and enlarging size or increasing associativity has only smoothing effects; otherwise increasing size or associativity would produce "unsteady" effects.

Due to instruction prefetching, speculative execution and non blocking caches, the delay effects of instruction cache size misses may substantially decrease. Larger register sets, new compiling techniques and perhaps data prefetching may also decrease data cache size misses. Conflict misses remain.

Furthermore, it should be mentioned that these cache miss rate measurements always show rates averaged over a variety of programs. They do not predict the behaviour of a single program. From a software architect's point of view, a 50% performance difference in programs of his favoured type is important, even if the hardware architect realizes only a 2% effect in his (conservative) overall benchmark suite.

A further remark: a stochastic cache working set is chosen out of an infinitely large address interval. In practice, twice the cache size is already "infinite". On the other hand, if you select variables within a memory interval smaller than or equal the total cache size, the capacity is always 100%. This means that a 32K cache works perfectly as long as the hot data variables lie within one 32K interval, no matter what architecture the cache has.

## 5. Conclusions

Table 2 and figure 4 show systematic, expected and stochastic data cache capacity of various available processors. For the processors using a unified instruction and data cache (486 and PowerPC 601), it is assumed that half of the cache is used for instructions.

| Processor | Cache Workingset |
|---|---|
| 486 | 55–139 × 16 B = 0.88–2.22 K |
| Pentium | 11–45 × 32 B = 0.35–1.44 K |
| PowerPC 601 | 61–112 × 64 B = 3.90–7.17 K |
| PowerPC 604 | 55–139 × 32 B = 1.76–4.44 K |
| Alpha 21064 | 2–20 × 32 B = 0.06–0.64 K |
| Mips R4000 | 5–40 × 32 B = 0.16–1.28 K |

**Table 2.** *Concrete Cache Capacities.*

1. Analytically or heuristically derived values of the cache working sets of concrete programs may help the user to select the most appropriate hardware.

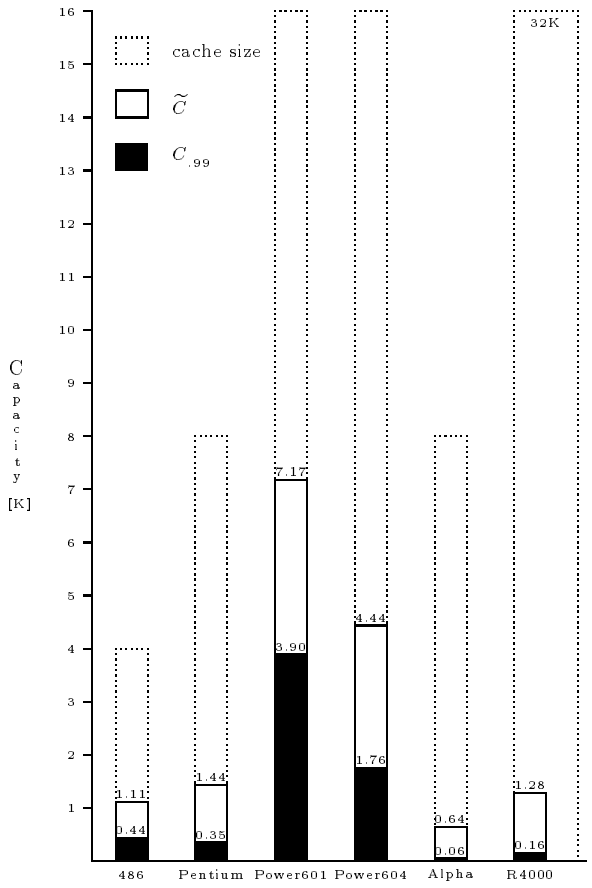


**Figure 4.** *Concrete Cache Capacities*

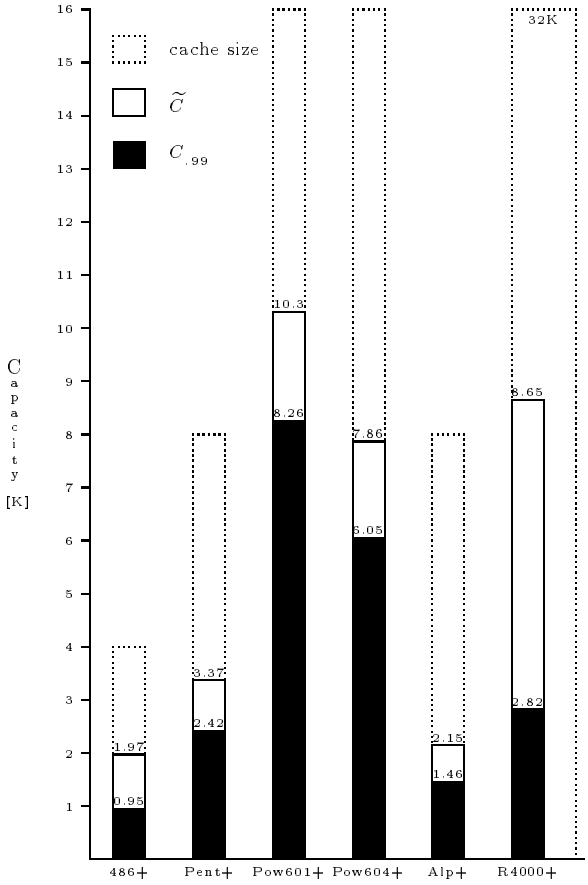


**Figure 5.** *Hypothetical Cache Capacities When Adding an 8-overflow Cache*

2. The cache capacity characteristics of the processors differ largely. We should not expect to find processor independent optimization strategies for memory management algorithms.

3. For programs with a relatively small data set (and processors with a fairly large cache), it might be a good strategy to concentrate the complete data set into a virtual memory region smaller than the cache size.

4. This strategy will presumably not work for larger data sets, especially in the case of object-oriented systems and databases or in the case of single address space operating systems. All these applications will profit from higher associativity and thus higher stochastic capacity like on the PowerPC.

5. A strategy for PowerPCs: try to cluster related objects in one page. As long as the data working set consists of only up to 4 pages, you have 100% capacity, i.e. 16K (provided that not more than 4 instruction pages are required on the 601).

6. Intuitively, we doubt that effects comparable to increased associativity can be obtained by software, mainly due to the costs of dynamic detection of working sets and the required rearrangement.

7. Increasing the stochastic capacity of caches seems to be the most promising way. Figure 5 shows the hypothetical effect of adding only an 8-entry overflow cache to the primary cache of the processors mentioned above.

## References


[1] J. L. Baer and W. H. Wang. On the inclusion properties for multi level cache hierarchies. In *15th Annual International Symposium on Computer Architecture (ISCA)*, pages 73–80, Honolulu, HA, June 1988.

[2] S. Bederman. Cache management system using virtual and real tags. *IBM Technical Disclosure Bulletin*, 21(11):4541, Apr. 1979.

[3] T. Chiueh and R. H. Katz. Eliminating the address translation bottleneck for physical address cache. In *5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 137–148, Boston, MA, Oct. 1992.

[4] P. J. Denning. The working set model for program behaviour. *Commun. ACM*, 11(5):323–333, May 1968.

[5] M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. *IEEE Trans. Comput.*, 38(12):1612–1630, Dec. 1989.

[6] Intel Corp. *i486 Microprocessor Programmer's Reference Manual*, 1990.

[7] Intel Corp. *Pentium Processor User's Manual, Volume 3: Architecture and Programming Manual*, 1993.

[8] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *17th Annual International Symposium on Computer Architecture (ISCA)*, pages 364–373, Seattle, WA, May 1990.

[9] G. Kane and J. Heinrich. *MIPS Risc Architecture*. Prentice Hall, 1992.

[10] R. E. Kessler, R. Joos, A. Lebeck, and M. D. Hill. Inexpensive implementations of set-associativity. In *16th Annual International Symposium on Computer Architecture (ISCA)*, pages 131–139, Jerusalem, May 1989.

[11] J. Liedtke. Potential interdependencies between caches, tlbs and memory management schemes. Arbeitspapiere der GMD No. 962, GMD — German National Research Center for Information Technology, Sankt Augustin, Dec. 1995.

[12] Motorola Inc. *PowerPC 601 RISC Microprocessor User's Manual*, 1993.

[13] J. Mulder. An area model for on-chip memories and its applications. *IEEE Journal of Solid States Circuits*, 26(2):98–106, Feb. 1991.

[14] A. J. Smith. Cache memories. *ACM Comput. Surv.*, 14(3):473–530, Sept. 1982.

[15] S. P. Song, M. Denman, and J. Chang. The PowerPC 604 risc microprocessor. *IEEE Micro*, 14(5):8–17, Oct. 1994.

[16] W. H. Wang, J. L. Baer, and H. Levy. Organization and performance of a two-level virtual-real cache hierarchy. In *16th Annual International Symposium on Computer Architecture (ISCA)*, pages 140–148, Jerusalem, May 1989.

[17] S. J. E. Wilton and N. P. Jouppi. An enhanced access and cycle time model for on-chip caches. Technical Report 93/5, Digital Western Research Laboratory, Palo Alto, CA, July 1994.

[18] D. A. Wood, S. J. Eggers, G. Gibson, M. D. Hill, J. M. Pendleton, S. A. Ritchie, G. S. Taylor, R. Katz, and D. A. Patterson. An in-cache address translation mechanism. In *13th Annual International Symposium on Computer Architecture (ISCA)*, pages 358–365, Tokyo, June 1986.