Intended for further publication.
Do not copy! Do not circulate!

Potential Interdependencies Between Caches, TLBs and Memory Management Schemes

Jochen Liedtke
GMD — German National Research Center for Information Technology *
jochen.liedtke@gmd.de

Abstract

Dynamic memory management 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 and TLB architectures, since, currently, most of them are highly sensitive to access patterns. In the above mentioned stochastically distributed case, the true capacity is (a) far below the cache or TLB size and (b) largely differs from processor to processor. As a consequence, dynamic memory management schemes may substantially influence cache/TLB hit rates and thus overall program performance.

After presenting basic cache and TLB architectures in short, an analytical model for evaluating their true capacities is developed and applied to various architectures. Some industrial processors are evaluated in the same way and potential implications for memory management techniques are discussed. Furthermore, a new architecture for caches and TLBs is presented which improves their true capacity and reduces their dependence on usage patterns.

1 Rationale

This paper does not deal with the primary costs of garbage collection or other dynamic memory management mechanisms but with their secondary costs. Does a program the variables which of have been dynamically allocated and subject to garbage collection 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. (Note that even a perfectly compactifying garbage collector does not lead to compact dynamic working sets.)

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

*GMD I5.RS, 53754 Sankt Augustin, Germany

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

Hence, it might be relevant to examine (a) to what extent current caches and TLBs support or impede MM, (b) whether there exist MM guidelines that lead to better performing programs and (c) how caches and TLBs can be made more efficient when dynamic memory management is used. This paper examines these issues. Section 2 gives a short introduction to basic cache and TLB architectures and describes a new overflow cache mechanism. Section 3 informally introduces various *capacity* cache measures; more precise definitions and an analytical model of them are given in appendix A. The model was originally developed to examine the ability of caches and TLBs to handle fine-grained virtual memory [Liedtke 1994]. Although it thus is stronger in giving upper complexity bounds than precise costs which can be expected, it shows limitations of different processors and gives some insight into the problem's structure. In the remaining sections, some cache architectures and concrete processors are evaluated according to the model, and some conclusions with respect to memory management schemes are drawn.

2 Basic Cache Architectures

A general introduction into caches can be found in [Smith 1982]. Special architectures are described in [Bederman 1979; Wood et al. 1986; Wang et al. 1989; Baer and Wang 1988; Chiueh and Katz 1992; Kessler et al. 1989]. 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.

2.1 Cache Associativity

This section can be shortened, if cache architecture is widely known.

A *fully-associative* cache of n entries is a content addressable memory. Each entry consists out of a *tag* field and a data field. The tag field contains the (main memory) address of the data held in a given cache entry; the data fields contain the corresponding data. Upon a cache access, the address a is compared with all n tags in parallel. If one tag matches, a hit is detected and the corresponding data field is used. Otherwise the access leads to a cache miss which must be handled by the cache replacement mechanism.

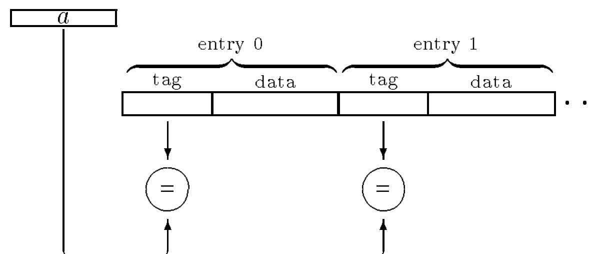

Figure 1: *Fully Associative Cache.*

From a software engineer's point of view, a fully-associative cache is the most convenient architecture, since there are no restrictions on the set of addresses which can be cached simultaneously. Any n addresses can be held; clashes never occur.

Unfortunately, fully-associative caches are very expensive *and become too slow when they are designed sufficiently large.* In practice, this type is not used for data or instruction caches, but sometimes for TLBs.

In the case of a *direct-mapped* cache (see figure 2), a map function forms a cache index from the physical or virtual address a. The index selects a cache line. Each (valid) cache line contains both data and the address of the associated main memory as a tag. a is compared with the tag stored in the selected cache entry. If it matches, we have a hit and the cache data field is used instead of the memory. Otherwise, we have a cache miss, and some cache replacement mechanism will solve the problem in such a way that a restarted access a results in a hit.

In most cases, $(a \bmod \mathit{cachesize})/\mathit{linesize}$ is used as map function. Then it is sufficient to store only $a/\mathit{cachesize}$ instead of the complete address in the tag field.

Direct-mapped caches are simple, fast and cheap. For a given die size, the direct-mapped architecture permits the fastest [Wilton and Jouppi 1994] and

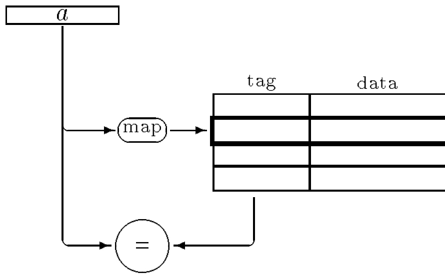

Figure 2: *Direct-mapped Cache.*

the largest [Mulder 1991] cache (the cache with the most entries). On the other hand, it tends 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.

Direct-mapped Caches are simpler but lead to higher miss rates than *n-way set-associative caches.* In principle, n-way caches consist of n accordingly smaller direct-mapped cache blocks. Figure 3 shows a 2-way set-associative cache. The replacement mechanism of the cache ensures that any main memory entity resides at most once in the cache. The map function always adresses a complete row, i.e. n cache entries simultaneously. All n tags are read and compared with the address a in parallel. The row consisting of n tag and data fields is called a *set.* Thus an n-way cache with s sets (or rows) contains sn entries. If all comparisons fail, a cache miss is signaled. If one tag matches, a cache hit is detected and the cache entry of this *way* is used. In the figure, a way is represented by its tag and data column.

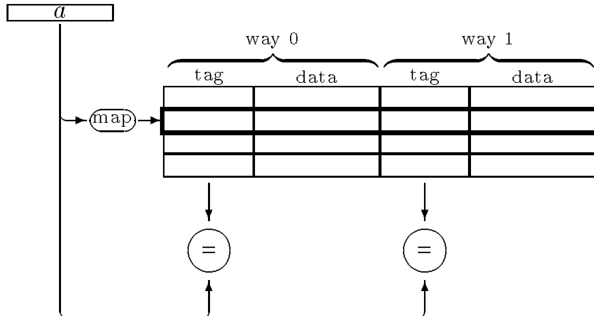

Figure 3: *2-way Set-associative Cache.*

The n-way set-associative cache can contain up to n memory entities with map-equivalent addresses per set. 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.

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.*

2.2 TLBs

A *Translation Lookaside Buffer* (TLB) is a special cache which is used for translating virtual to physical addresses. Some processors use fully-associative TLBs, e.g. Mips R4000 , but in most cases, the TLBs are n-way set-associative caches.

Although in practice, TLBs are at least 2-way set-associative, we use a direct-mapped one (figure 4) for explaining the principle functions. The diagram is simpler and can easily be extended to the n-way case.

The uppermost bits v' of the virtual address v are used to select an entry of the TLB (in the n-way case, to select a row of n entries simultaneously). The entry i holds the virtual page number v_i, its associated physical page number r_i and further status bits (e.g. valid, writable, user/kernel) which are omitted in the figure. If the entry is valid

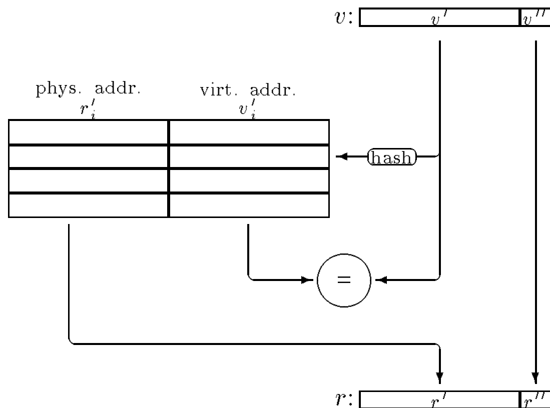

Figure 4: *Direct-mapped TLB.*

and $v' = v_i$, we have a TLB hit, and the physical address is composed of the lowermost bits v'' of the virtual address v (also called page offset) and the physical page number r_i'.

Processor	Entries	Ways	Page Size	Use
486	32	4	4 K	I+D
Pentium	32 64	4 4	4 K 4 K	I D
PowerPC 601	256	2	4 K	I+D
PowerPC 604	128 128	2 2	4 K 4 K	I D
Alpha 21064	32	32	8 K	I+D
Mips R4000	48	48	4 K	I+D

Table 2: *TLBs.*

2.3 Overflow Caches and TLBs

Jouppi [1990] placed a small, fully-associative *victim cache* between ordinary direct-mapped primary and secondary caches for tolerating a few primary cache conflicts. As a generalization and extension of this idea, we propose to extend a conventional i-way set-associative cache or TLB by a *parallel*, small fully-associative *overflow cache* or TLB.

The mechanisms described in the following sections can be applied to both caches and TLBs. Since usually, stochastic influences are stronger for TLBs than for data or instruction caches, we first describe the TLB case.

A direct-mapped or n-way set-associative TLB is complemented by a small fully-associative *overflow TLB* which is used to hold the "set overflows". Assume that a new (v, r) pair should be entererd into the TLB and that the map function maps v to a set which is already full, i.e. consists already of n valid entries. Instead of flushing one entry of the set, the new (v, r) pair can also be entered into the overflow TLB. Thus, some sets may overrun without inducing clashes.

The overflow TLB operates in parallel with the conventional TLB. Figure 5 shows a system composed of an overflow TLB with j entries and a conventional n-way set-associative TLB. When trans-

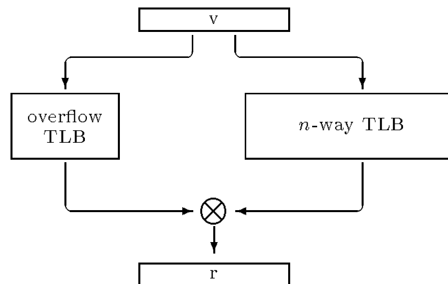

Figure 5: *n-way j-overflow TLB.*

lating a virtual address v, both the n-way TLB and the overflow TLB work in parallel. If one of the two units signals a hit, the corresponding physical address is the resulting physical address r and the whole TLB signals a hit. If no unit signals a hit, we have a TLB miss. Figure 6 shows a 2-way set-associative, 3-overflow TLB. Note that small content addressable memories can be very fast so that an 4-way 4-overflow cache should be simpler and faster than an 8-way cache.

The mechansims described in section 2.3 can also be used to improve the associativity of an instruction or data cache. An access with address a (virtual or physical) will be handled in parallel by a conventional n-way or direct-mapped cache and the new overflow cache. If one of them signals a hit, the corresponding data field will be used. Otherwise, if neither of the two parts signals a hit, the composed cache signals a miss.

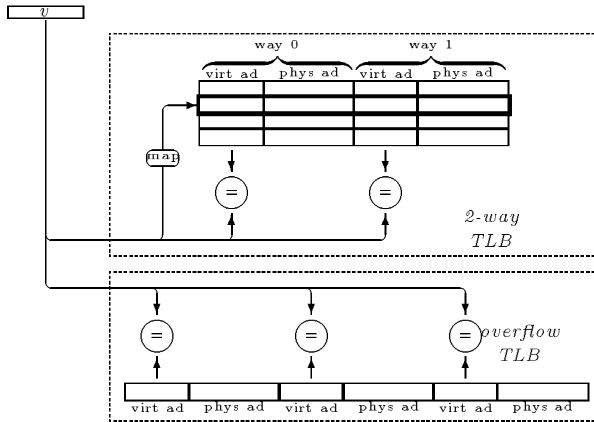

Figure 6: *2-way 3-overflow TLB.*

3 Probabilistic Capacity

3.1 Evaluations

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.

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* [Denning 1968]. 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 ω (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 ω/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_ω denotes the maximum prefix length of this sequence so that its cache working set contains ω entities. The corresponding cache working set is denoted by W_ω. 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 appendix A; 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 appendix A 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

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

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 7 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.

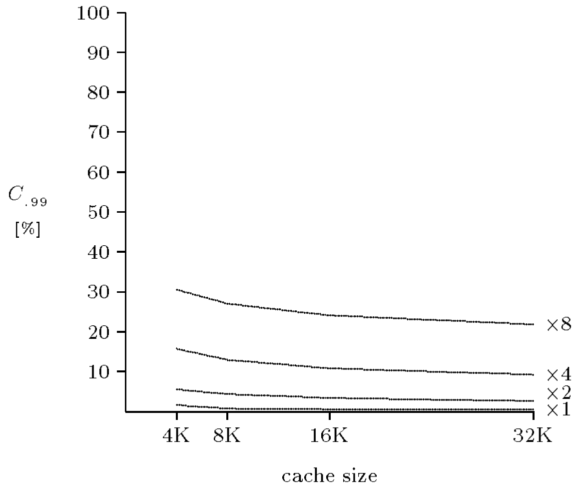

Figure 7: *Stochastic Capacity, n-way Caches*

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 8 shows the effect of adding overflow caches with 4, 8, 16 and 32 entries to a direct-mapped cache. The effect is surprisingly strong: even 4 entries

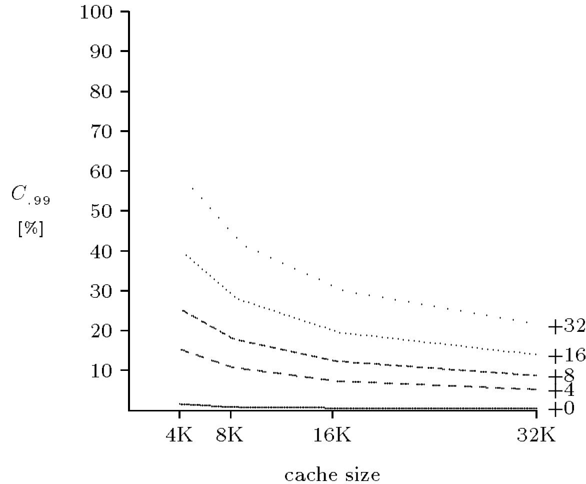

Figure 8: *Stochastic Capacity, Direct-Mapped Cache*

are enough to compete with a 4-way set associative cache, 32 overflow entries outperform the 8-way cache.

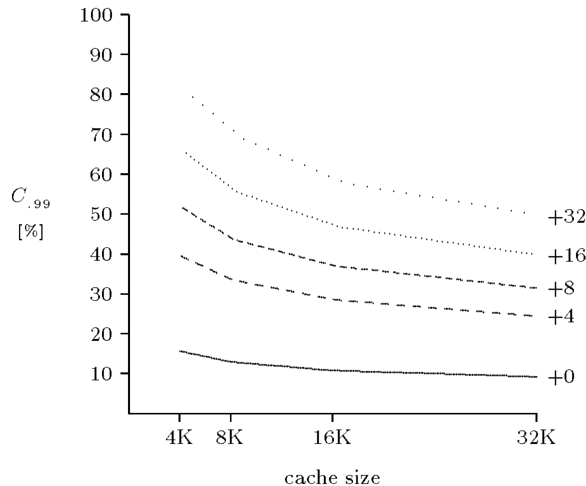

Figure 9: *Stochastic Capacity, 4-way Cache*

When a 4-way cache is complemented by an overflow cache (see figure 9), the first 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 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 10 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.

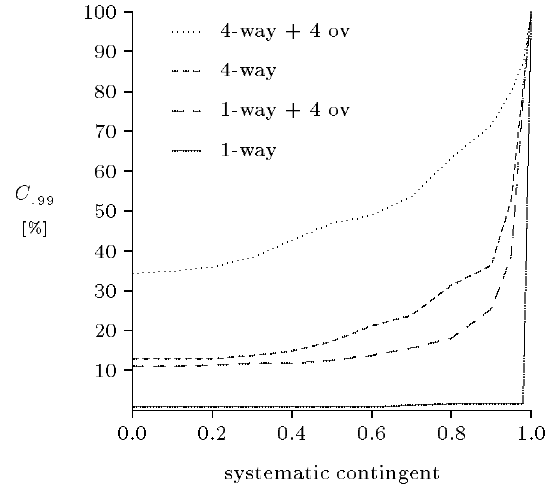

Figure 10: *Probabilistic Capacity of 8K-Caches for Mixed Stochastical and Systematic Selections*

For a more comprehensive comparison, figure 11 shows stochastic and *expected* capacity of 1-, 2-, 4- and 8-way associative caches, each for in isolation and also when combined with 4-, 8-, 16- and 32-overflow caches. The general result is that to achieve higher capacities, multiple ways and an overflow cache are required. For capacities beyond 50%, the potential combinations are 2-way+32, 4-way+16 or 8-way+8. Note that the gap between expected and stochastic capacity becomes smaller when using overflow caches. In the case of a 16-entry overflow cache e.g., the difference is only 10% or even less.

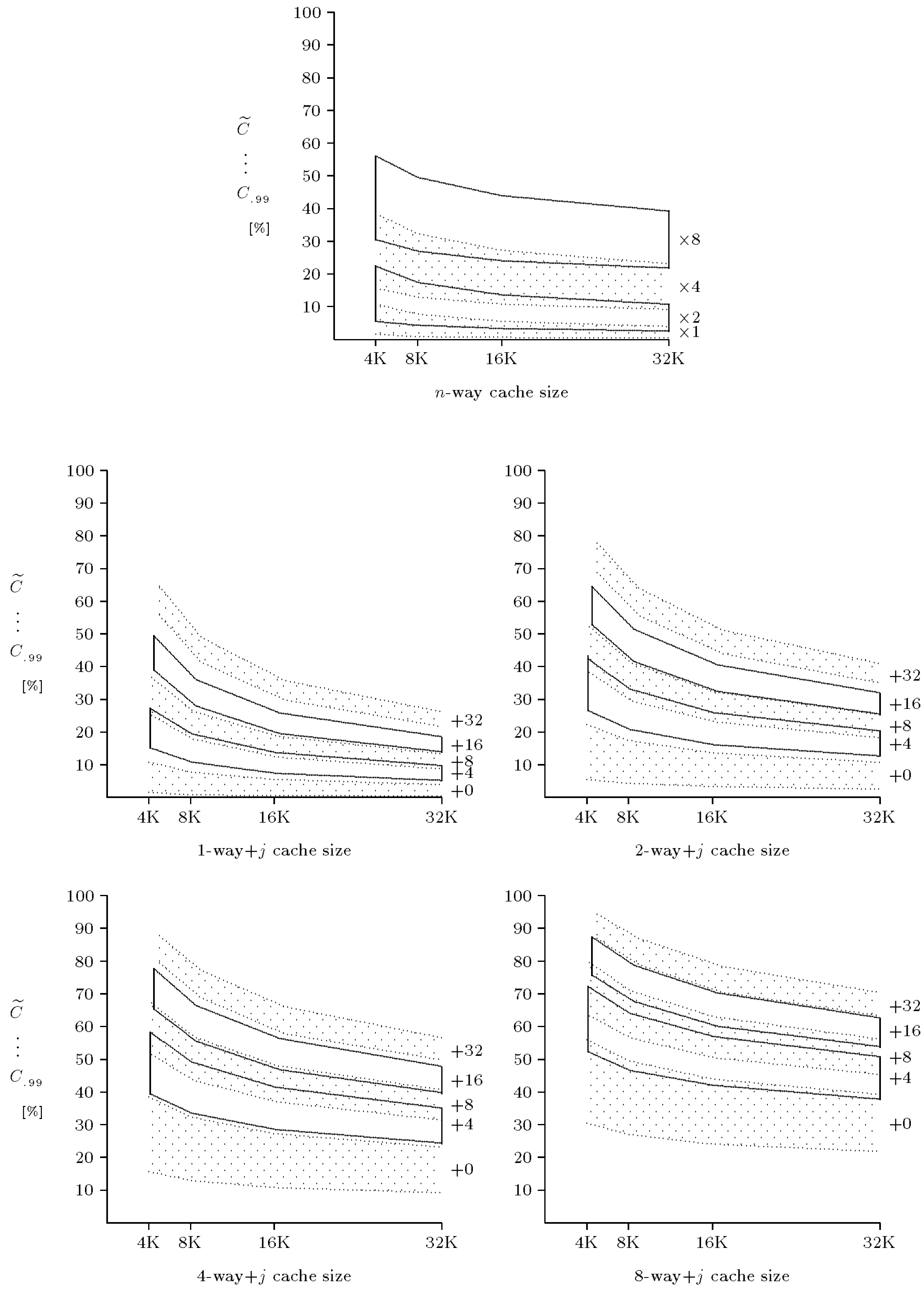

Figure 11: *Cache Capacities.*

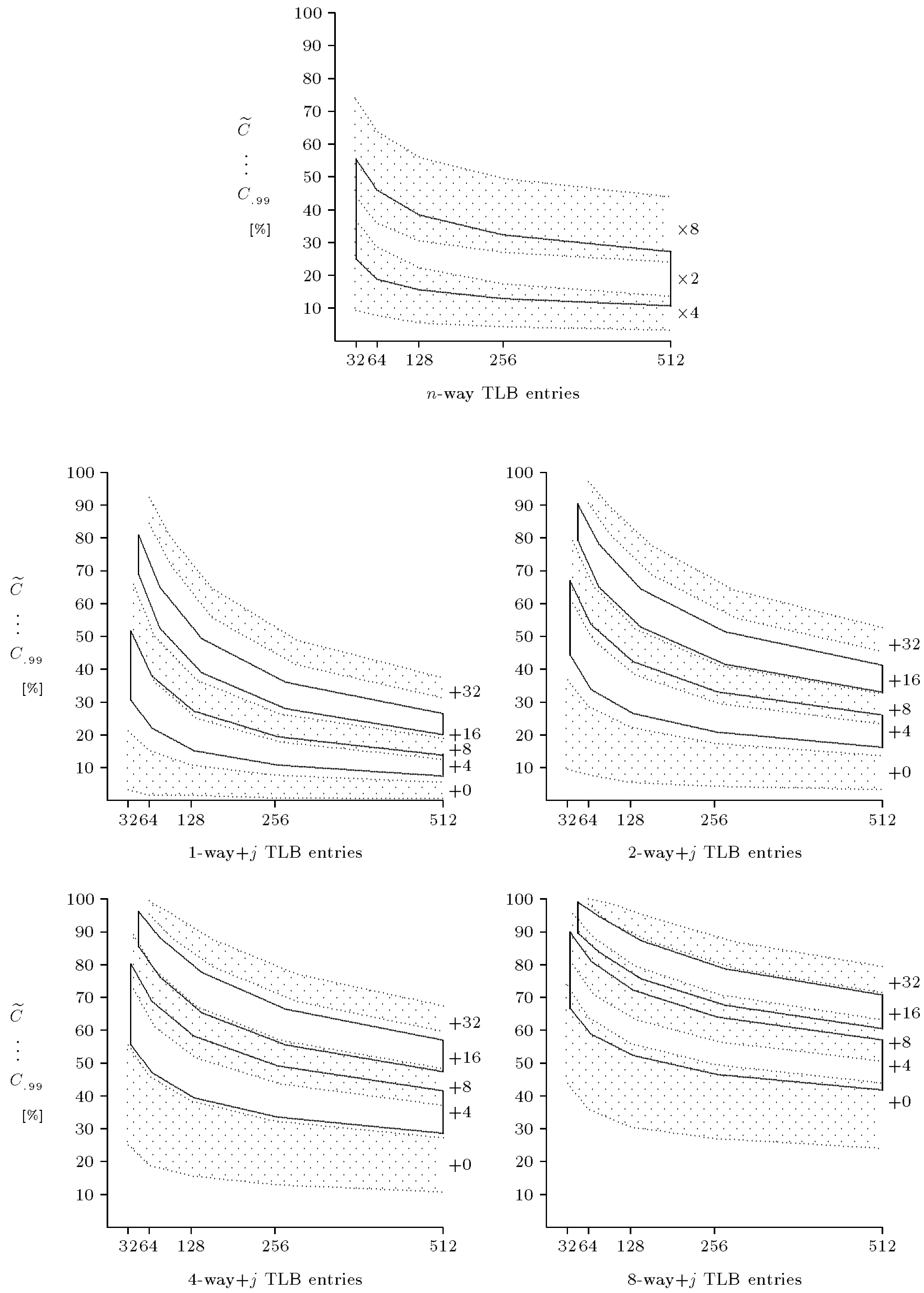

Figure 12: *TLB Capacities.*

TLBs have a behaviour similar to caches. Figure 12 shows expected capacity and stochastic capacity ($p = 99\%$) for TLBs from 32 up to 512 entries. To ensure capacities beyond 50%, TLBs also require overflow TLBs.

4.1 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 [1989], 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]" [Hill and Smith 1989, 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

5.1 Interdependencies with Caches

Table 3 and figure 13 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 $\times$ 16 B = 0.88–2.22 K
Pentium	11–45 $\times$ 32 B = 0.35–1.44 K
PowerPC 601	61–112 $\times$ 64 B = 3.90–7.17 K
PowerPC 604	55–139 $\times$ 32 B = 1.76–4.44 K
Alpha 21064	2–20 $\times$ 32 B = 0.06–0.64 K
Mips R4000	5–40 $\times$ 32 B = 0.16–1.28 K

Table 3: *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.
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.

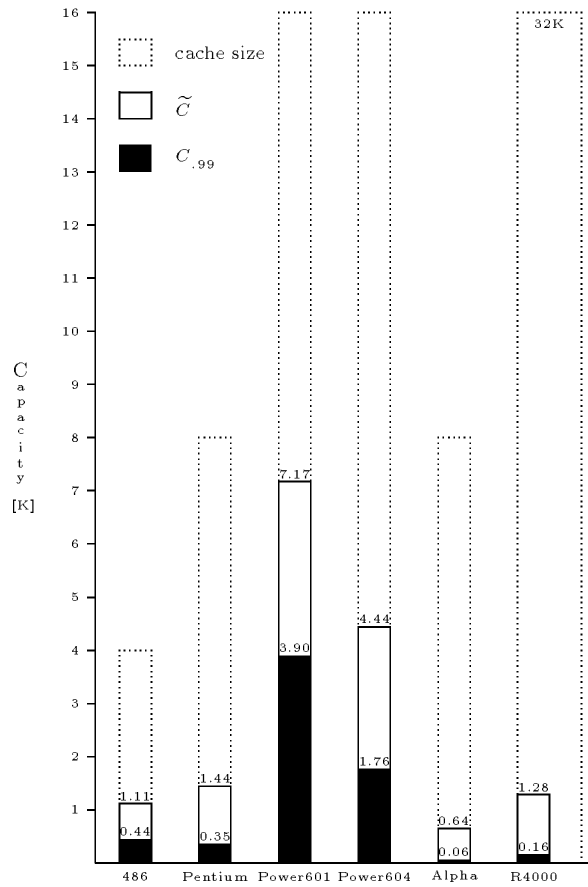

Figure 13: *Concrete Cache Capacities*

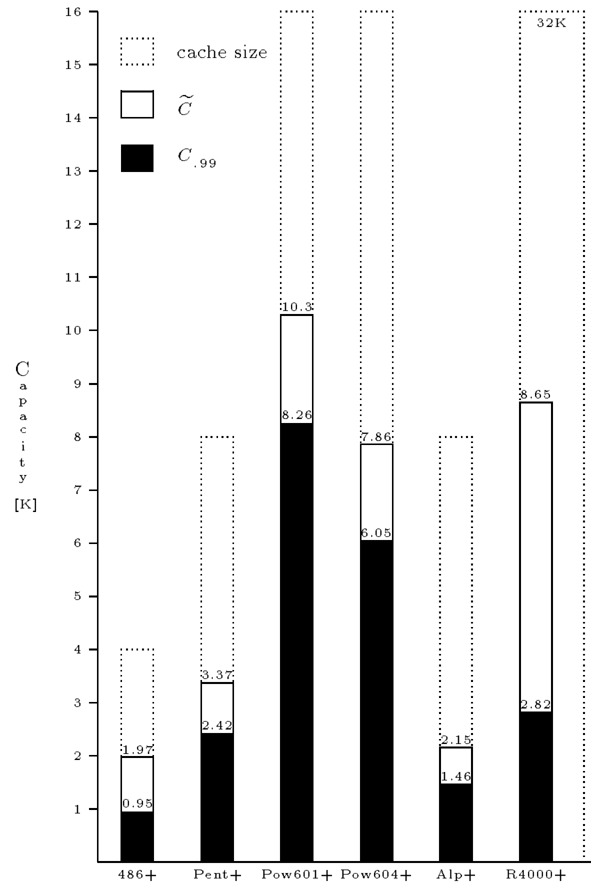

Figure 14: *Hypothetical Cache Capacities When Adding an 8-overflow Cache*

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 14 shows the hypothetical effect of adding only an 8-entry overflow cache to the primary cache of the processors mentioned above.

5.2 Interdependencies with TLBs

Table 4 and figure 15 show systematic, expected and stochastic data TLB capacity of the same processors. The capacity is given in Kbyte which can be mapped by the TLB. The Alpha and R4000[1] TLBs are fully-associative and therefore have high stochastic capacity. Note that these are both processors with software controlled TLBs, where TLB misses are extremely expensive. Again, in the case of unified instruction and data TLBs (all but Pentium and PowerPC 604), it is assumed that half of the TLB is used for instruction pages.

1. Similar to the cache case, TLB working set data of his program may help the user.

2. The capacity characteristics of the processors vary less than in the cache case. The chance is higher to find optimization strategies which can be applied to multiple processor types.

[1] The R4000 TLB has 48 entries, each of which can hold a pair of two adjacent pages. Therefore the total TLB capacity is at least $48 \times 4K$ and at most $48 \times 8K$.

Processor	TLB Workingset
486	4–9 × 4 K = 16–36 K
Pentium	12–29 × 4 K = 48–116 K
PowerPC 601	6–23 × 4 K = 24–88 K
PowerPC 604	7–29 × 4 K = 28–116 K
Alpha 21064	16–16 × 8 K = 128–128 K
Mips R4000	24–24 × 4 K = 96–96 K

Table 4: *Concrete TLB Capacities.*

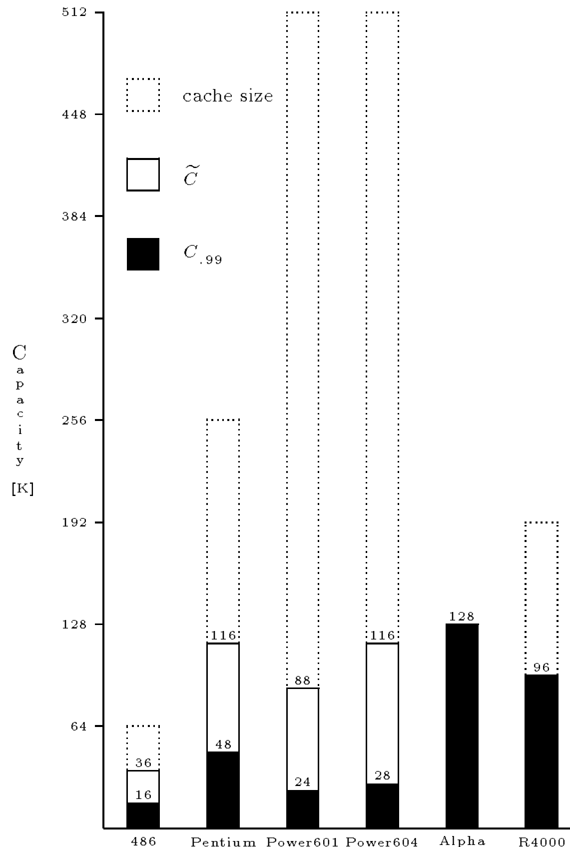

Figure 15: *Concrete TLB Capacities*

3. There are processors (Alpha and R4000), where the TLBs permit arbitrary stochastic distributions, provided that not too many pages are used. Unfortunately, exactly these processors are characterized by low cache capacity.

4. On the PowerPC and the Pentium, restricting the data memory used to a region of 256K (Pentium) or 512K (PowerPC) will eliminate data TLB conflicts. Such an attempt to improve locality is e.g. described in [Lam et al. 1992]. The stochastic capacity permits also a certain unlimited random use.

6 Open Questions

This is a workshop paper and does not describe completed research. It leaves a many questions open:

1. How stochastic are data working sets of programs running under various memory management mechanisms? Are they similar to Lisp and Smalltalk programs, to OO databases and single address space operating systems?

2. Performance research has concentrated on the efficiency of the garbage collection process itself. Are the existing algorithms capable of usage optimization strategies without loosing their performance?

3. Is the situation different in uniprocesors, multiprocessors and distributed systems?

4. What are the real costs (in terms of performance degradation) of capacity overflows, i.e. when the working set exceeds stochastic or expected capacity?

5. Will the importance of conflict misses really increase in future processors?

6. Can overflow caches/TLBs be added to normal caches/TLBs without slowing them down?

Last, but not least, it should be mentioned that the discussed problems may be as important for large second and third level caches, since the costs per cache miss then increase dramatically, size misses occur more seldom and randomess grows, because the caches are shared by multiple address spaces. Since these caches are indexed by physical addresses, virtual memory mapping adds additional stochastic effects.

Acknowledgement

Many thanks to Richard Uhlig for proofreading and critical comments.

A Theorems

For sake of simplicity, we assume that the smallest addressable unit is one cache line. Let us regard a sequence $\dot{a}_i$ of addresses by which the cache is accessed. We eliminate all multiple occurrences of addresses from $\dot{a}_i$ so that in the resulting sequence a_i, all elements are pairwise different. A *working set* of cache entries can be characterized by a set of pairwise different addresses a_i with the property that *the cache is able to hold all a_i simultaneously.*

From the sequence a_i of addresses, the cache's map function derives a sequence x_i of cache indices by $x_i = \text{map}\,(a_i)$. Note that the elements of this sequence are *not* necessarily pairwise different: $x_i = x_j$ models two used cache entries addressed by the same index. For capacity modeling, we can describe a cache working set as well by the sequence $\bar{x} = (x_1, x_2 \ldots x_n)$.

For a complete cache description, we additionally need a *fit function* which specifies whether any given $\bar{x}$ simultaneously fits into the cache or not.

Definition 1 (Abstract Cache) *An* abstract cache *is a pair (X, f) consisting of an* index set X *and a* fit function $f : X^* \to \{0,1\}$. *For all sequences $\bar{x} \in X^*$ which fit completely into the cache, $f(\bar{x}) = 1$ holds; otherwise $f(\bar{x}) = 0$.*

Usually, a cache's index set consists of the indices $0, 1, \ldots |X| - 1$. Its fit function is determined by the cache architecture. Fit functions for some architectures are given below.

The length n of a sequence $\bar{x} = (x_1, x_2 \ldots x_n) \in X^n$ is denoted by $|\bar{x}|$. For $\bar{x} \in X^*$ and $y \in X$, the number of x_i with $x_i = y$ is denoted by $\bar{x}||_y$.

Then the fit functions for a fully-associative cache of m entries ($f_{1\times m}$), an m-way set-associative cache with s sets and sm entries ($f_{s\times m}$) and an m-way j-overflow cache with s sets and $sm + j$ entries ($f_{s\times m+j}$) can be written as

$$f_{1\times m}(\bar{x}) = \begin{cases} [] \ |\bar{x}| \le m & \to 1 \\ [] \ |\bar{x}| > m & \to 0 \end{cases}$$

$$f_{s\times m}(\bar{x}) = \begin{cases} [] \ \forall i : \ \bar{x}||_{x_i} \le m & \to 1 \\ [] \ \exists i : \ \bar{x}||_{x_i} > m & \to 0 \end{cases}$$

$$f_{s\times m+j}(\bar{x}) = \begin{cases} [] \ \sum_i \max(\bar{x}||_{x_i} - m\,,\, 0) \le j & \to 1 \\ [] \ \sum_i \max(\bar{x}||_{x_i} - m\,,\, 0) > j & \to 0 \end{cases}$$

Definition 2 (Probabilistic and Stochastic Capacity) *Assume an abstract cache with index set X and fit function f. Let $D = (d_1, d_2 \ldots)$ be a family of probability distributions $d_n : X^n \to [0,1]$ with $1 = \sum_{\bar{x} \in X^n} d_n(\bar{x})$ for all d_n. Then*

$$\varphi_n = \sum_{\bar{x} \in X^n} d_n(\bar{x}) f(\bar{x})$$

is called the corresponding fit probability. *For $p>0$, the corresponding* probabilistic capacity *with limit probability p is defined as*

$$C_p = \max \{ n \mid \varphi_n \geq p \} \quad .$$

For $p=0$, the probabilistic capacity is

$$C_0 = \max \{ n \mid \varphi_n > 0 \} \quad .$$

If D is the family of equal distributions $d_n(\bar{x}) = |X|^{-n}$, C_p is also called the stochastic capacity *of the cache.*

Remark: C_0 is the best case capacity of the cache, since it is the largest n such that there is *at least one* sequence of length n fitting into the cache. Usually, C_0 equals the number of cache entries.

C_1 is the worst case capacity of the cache, since it is the largest n such that *any* sequence of length n fits into the cache. Usually, C_1 equals the cache's maximum degree of associativity.

Definition 3 (Expected Capacity) *Assume an abstract cache with index set X, fit function f and a maximum capacity of n entries ($f(\bar{x}) = 0$ for all $\bar{x}$ with $|\bar{x}|>n$). The function $F : X^n \to N$ with*

$$F(x_1, \ldots x_n) = \max \{ i \leq n \mid f(x_1, \ldots x_i) = 1 \}$$

delivers the length of the longest prefix of a given $\bar{x}$ which fits into the cache. If the cache is not even able to hold only the first element ($f(x_1) = 0$), $F(x_1, \ldots) = 0$ is defined.

The expected capacity *of the given cache with respect to a probability distribution $d_n : X^n \to [0,1]$ is defined to be*

$$\widetilde{C} = \sum_{\bar{x} \in X^n} d_n(\bar{x}) F(\bar{x}) \quad .$$

If d_n is the equal distribution $d_n(\bar{x}) = |X|^{-n}$, $\widetilde{C}$ is also called the expected stochastic capacity *of the cache.*

Theorem 1 (Expected Stochastic Capacity) *Assume an abstract cache with index set X, fit function f and maximum capacity n. Let φ_k be the corresponding fit probability. Then the expected stochastic capacity can be calculated by*

$$\widetilde{C} = \sum_{k=1}^{n} \varphi_k \quad .$$

Proof: By definition, the expected stochastic capacity is

$$\widetilde{C} = \sum_{\bar{x} \in X^n} d_n F(\bar{x}) \quad ,$$

where $d_n \ |X|^{-n}$. Now, we construct the pairwise disjoint sets

$$Y_k = \{ \bar{x} \in X^n \mid F(\bar{x}) = k \}$$

which form a decomposition of X^n ($\bigcup Y_k = X^n$ and $Y_i \cap Y_j = \phi$ for $i \neq j$). Using this decomposition, we can rewrite the expected stochastic capacity as

$$\begin{aligned} \widetilde{C} &= \sum_{k=1}^{n} \sum_{\bar{x} \in Y_k} d_n F(\bar{x}) = \sum_{k=1}^{n} \sum_{\bar{x} \in Y_k} d_n k \\ &= \sum_{k=1}^{n} d_n |Y_k| k \end{aligned}$$

Note that $d_n Y_k$ is the probability that a randomly selected $\bar{x}$ is member of Y_k. By presupposition, we know the probability φ_k that $\bar{x}$ is member of $Y_k \cup Y_{k+1} \cup \cdots Y_n$. Since the Y_i are pairwise disjoint, the probability that $\bar{x}$ is member of Y_k is

$$d_n Y_k = \varphi_k - \varphi_{k+1} \quad .$$

Note that $\varphi_{n+1} = 0$, i.e. $d_n Y_n = \varphi_n$. Therefore

$$\widetilde{C} = \sum_{k=1}^{n} (\varphi_k - \varphi_{k+1}) \, k = \sum_{k=1}^{n} \varphi_k \quad .$$

q.e.d

A.1 m-way Set-associative Caches

An m-way set-associative Cache with s sets has sm entries in total. We model this cache as follows:

There is an infinite set of balls, each one marked by an index. There are s different indices which are equally distributed. A ball is drawn by randomly selecting one and removing it from the infinite set of balls. The probability to draw a ball with a specific index is $\frac{1}{s}$ and remains stable over subsequent drawings. Furthermore, there are s bins, one per index value. Each bin is able to hold up to m balls, i.e. altogether there is room for sm balls. Now, n balls are drawn and each drawn ball is put into the appropriate bin.

Select one of the s indices and then draw n balls. How large is the probability $\Pi_{n,s}^{(k)}$ that exactly k of the drawn balls have the preselected index? Since the probability to get a preselected index in one drawing is $\frac{1}{s}$ and there are $\binom{n}{k}$ possibilities to select k elements out of n, the answer is

$$\Pi_{n,s}^{(k)} = \binom{n}{k} \left(\tfrac{1}{s}\right)^k \left(1 - \tfrac{1}{s}\right)^{n-k}$$

We define $0^0 = 1$ so that $\Pi_{n,1}^{(k)}$ correctly evaluates to 1 if $k=n$ and 0 otherwise.

How large is the probability $P_c^{(s)}$ that at most m balls of each index are drawn in n drawings, i.e. that no bin runs over? Assume that $0 \leq k \leq m$ balls of the first index are drawn. Then $n-k$ balls are left for the remaining $s-1$ indices. For $c>0$, $P_s^{(n)}$ can be calculated recursively by summing over all legal values of k:

$$P_s^{(n)} = \sum_{k=0}^{m} \Pi_{n,s}^{(k)} \ P_{s-1}^{(n-k)}$$

For proper end of recursion, we define $P_0^{(n)} = 1$. Note that $P_1^{(n)}$ is 1 in case of $n \leq m$ and 0 otherwise.

Theorem 2 (Stochastic Capacity of m-way Cache) *Let (X, f) be an abstract cache with $|X| = s$ and fit function $f = f_{s\times m}$ as defined in (1). The stochastic capacity of this cache is determined by $\varphi_n = P_s^{(n)}$.*

Proof: See construction above.

$P_s^{(n)}$ is hard to calculate for large n and s. In the case of $P_s^{(n)} \approx 1$, it can also be approximated much simpler:

Theorem 3 (Approximation $P_s^{(n)}$)

$$P_s^{(n)} \geq 1 - sY \quad \textit{where} \quad Y = \sum_{k=m+1}^{n} \Pi_{n,s}^{(k)} \quad .$$

Proof: Here Y is the probability that a given bin runs over. The probability that at least one of s bins runs over is certainly less or equal than sY.[2]

The simple to calculate lower bound $1 - sY$ is obviously only good for $cY \ll 1$. Fortunately, we are interested exactly in these cases.

A.2 m-way j-overflow Caches

Now we extend the model to describe also m-way j-overflow caches. We introduce an additional overflow bin which is able to hold up to j balls of arbitrary index.

At first, we generalize $\Pi_{n,s}^{(k)}$: instead of 1, select i of the s indices and then draw n balls. How large is the probability $\Pi_{n,s,i}^{(k)}$ that exactly k of the drawn balls are marked with one of the preselected indices? Since the probability to get a ball with one of the preselected indices in one drawing is $\frac{i}{s}$ and there are $\binom{n}{k}$ possibilities to select k elements out of n,

$$\Pi_{n,s,i}^{(k)} = \binom{n}{k} (\tfrac{i}{s})^k \, (1 - \tfrac{i}{s})^{n-k} \quad .$$

Similar to $P_s^{(n)}$ in (A.1), we calculate the probability $Q_s^{(n)}$ that more than m balls of *each* index are drawn in n drawings, i.e. that all bins run over. Here the lower bound of k is $m + 1$. The upper bound is $n - (s - 1)(m + 1)$, since otherwise at least one of the remaining $s - 1$ bins cannot run over. Therefore,

$$Q_s^{(n)} = \sum_{k=m+1}^{n-(s-1)(m+1)} \Pi_{n,s}^{(k)} \; Q_{s-1}^{(n-k)}$$

for $s > 0$ and $Q_0^{(n)} = 1$ holds.

At last, we want to know the probability that no more than j overruns occur when n balls are drawn. To express this more precisely, for every experiment of drawing n balls, we denote by N_i the number of actually drawn balls of index i. If $N_i \leq m$, bin i did not run over; otherwise it produced $N_i - m$ overruns. We are interested in the probability $R_s^{(n)}(j)$ that $\sum_{i=1}^{s} \max(N_i - m\,,\, 0) \leq j$.

For the calculation, we assume that exactly i bins run over and the remaining $s - i$ do not. Furthermore, we assume that exactly k balls are drawn which belong into the overrunning i bins. Due to Bayes' formula, the probability for this special situation is the product

$$\Pi_{n,s,i}^{(k)} \; Q_i^{(k)} \; P_{s-i}^{(n-k)} \quad .$$

The lower bound for k is $i(m+1)$, since i bins must run over; its upper bound is $im + j$, since exactly i bins run over and at most j overruns are permitted. The number i of overrunning bins may vary from 0 to j. Therefore, the final result is

$$R_s^{(n)}(j) = \sum_{i=0}^{j} \binom{s}{i} \left(\sum_{k=i(m+1)}^{im+j} \Pi_{n,s,i}^{(k)} \; Q_i^{(k)} \; P_{s-i}^{(n-k)} \right) \quad .$$

Theorem 4 (Stoch. Cap. of m-way j-overflow Cache) *Let (X, f) be an abstract cache with $|X| = s$ and fit function $f = f_{s\times m+j}$ as defined in (1). The stochastic capacity of this cache is determined by the fit probability $\varphi_n = R_s^{(n)}(j)$.*

Proof: See construction above.

[2] The different possible bin overflow events during *one* drawing experiment are not independent. Therefore, we cannot simply calculate $P_s^{(n)}$ by $(1 - Y)^s$. This expression gives the probability that in s subsequent drawing experiments, bin i does not overflow *during the ith drawing experiment*, while other bins may overrun in the experiment.

References

BAER, J. L. AND WANG, W. H. 1988. On the inclusion properties for multi level cache hierarchies. In *15th Annual International Symposium on Computer Architecture (ISCA)*, Honolulu, HA, pp. 73–80.

BEDERMAN, S. 1979. Cache management system using virtual and real tags. *IBM Technical Disclosure Bulletin 21,* 11 (April), 4541.

CHIUEH, T. AND KATZ, R. H. 1992. Eliminating the address translation bottleneck for physical address cache. In *5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Boston, MA, pp. 137–148.

DENNING, P. J. 1968. The working set model for program behaviour. *Commun. ACM 11,* 5 (May), 323–333.

HILL, M. D. AND SMITH, A. J. 1989. Evaluating associativity in CPU caches. *IEEE Transactions on Computers 38,* 12 (Dec.), 1612–1630.

Intel Corp. 1990. *i486 Microprocessor Programmer's Reference Manual.* Intel Corp.

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

JOUPPI, N. P. 1990. 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)*, Seattle, WA, pp. 364–373.

KANE, G. AND HEINRICH, J. 1992. *MIPS Risc Architecture.* Prentice Hall.

KESSLER, R. E., JOOS, R., LEBECK, A., AND HILL, M. D. 1989. Inexpensive implementations of set-associativity. In *16th Annual International Symposium on Computer Architecture (ISCA)*, Jerusalem, pp. 131–139.

LAM, M. S., WILSON, P. R., AND MOHER, T. G. 1992. Object type directed garbage collection to improve locality. In *Memory Management (IWMM 92)*, St. Malo, pp. 404–425. Springer.

LIEDTKE, J. 1994. Address space sparsity and fine granularity. In *6th SIGOPS European Workshop*, Schloß Dagstuhl, Germany, pp. 78–81. also in *Operating Systems Review 29*, 1 (Jan. 1995), 87–90.

Motorola Inc. 1993. *PowerPC 601 RISC Microprocessor User's Manual*. Motorola Inc.

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

SMITH, A. J. 1982. Cache memories. *ACM Computing Surveys 14,* 3 (Sept.), 473–530.

SONG, S. P., DENMAN, M., AND CHANG, J. 1994. The PowerPC 604 risc microprocessor. *IEEE Micro 14,* 5 (Oct.), 8–17.

WANG, W. H., BAER, J. L., AND LEVY, H. 1989. Organization and performance of a two-level virtual-real cache hierarchy. In *16th Annual International Symposium on Computer Architecture (ISCA)*, Jerusalem, pp. 140–148.

WILTON, S. J. E. AND JOUPPI, N. P. 1994. An enhanced access and cycle time model for on-chip caches. Tech. Rep. 93/5 (July), Digital Western Research Laboratory, Palo Alto, CA.

WOOD, D. A., EGGERS, S. J., GIBSON, G., HILL, M. D., PENDLETON, J. M., RITCHIE, S. A., TAYLOR, G. S., KATZ, R., AND PATTERSON, D. A. 1986. An in-cache address translation mechanism. In *13th Annual International Symposium on Computer Architecture (ISCA)*, Tokyo, pp. 358–365.