Reorganisation in the skewed-associative TLB

Thorild Selén

Reorganisation in the skewed-associative TLB

Thorild Selén

Master's Thesis

Supervisors: Prof. Erik Hagersten, Dr. Andreas Ermedahl

September 29, 2004

Abstract

One essential component and a common bottleneck in current virtual memory systems is the *translation lookaside buffer* (TLB), a small, specialised cache that speeds up memory accesses by storing recently used address translations. A TLB can be viewed as a hash table that only has the capacity for holding a subset of the actively used address translations.

The traditional way to increase the performance of a TLB (other than making it larger) is to increase associativity, typically performing multiple comparisons in parallel to avoid slowing down lookups; however, this is expensive in terms of chip area and energy consumption. Skewed associativity, i.e. using several different hash functions for parallel lookups, has been demonstrated to yield good results with less parallelism and therefore at a lower cost.

In skewed-associative models, the sets of possible placements for two entries may only partially overlap. Thus, the current placement of entries will limit future replacement possibilities. This is an inherent inflexibility in traditional skewed-associative models, since we cannot predict which placements will enable the most desirable future replacement choices.

This thesis demonstrates how the performance of skewed-associative TLB models can be enhanced further by *reorganisation* — moving old entries around to allow for more efficient replacements. This gives even more efficient usage of TLB locations, increasing performance without further complicating lookups. The thesis introduces and demonstrates a *collision tendency* metric that enables simple comparison of the conflict miss vulnerability for a multitude of associativity models and degrees of associativity over a large range of sizes.

Simulations demonstrate that using skewed-associative techniques and reorganisation, efficient TLBs can be implemented with far less parallelism in hardware, allowing for more compact and much less energy-consuming designs without sacrificing performance.

Additionally, this thesis discusses adapting the skewed-associative TLB with reorganisation to handle real-time requirements, notably in applications where tasks with different real-time needs are run concurrently.

1

Contents

1 Introduction

Today, virtually every non-embedded computer, ranging from personal computers and low-end workstations to supercomputers, utilises a virtual memory system. This concept is essentially based on separating the manner in which memory is addressed by a process from the physical addresses used by memory hardware. This introduces a security and abstraction layer between these separate address spaces, in which a memory access by a process includes an address translation, hidden from the process. Trivially implemented, this address translation is very costly; however, it can be sped up greatly by dividing the address space into pages, indexed by page tables, and caching recently used page table entries in a translation lookaside buffer (TLB).

A slow address translation will in turn cause a slow memory fetch and stall the processor. Since address translations are very common, the TLB may become a significant performance bottleneck of a computer system, and the time required for handling TLB misses may add up to a sizeable portion of the processor cycles consumed by a set of processes. This makes TLB design and construction an interesting area for system performance improvement.

When comparing and evaluating TLB designs, one commonly looks at such properties as size (how many entries can it hold at one time?), reach (how much memory can the entries refer to simultaneously?), speed (how costly is a lookup?) and associativity (how many alternative locations are available for storing an entry?) and replacement algorithm (what old entries do we replace when the TLB is full?). However, making a good choice for a system involves certain tradeoffs; an increase in size implies a larger circuit, which leads to longer signal paths through the TLB; also, an increase in associativity calls for a greater number of simultaneous comparisons, which infers similar costs. It is clear that there is no way to completely avoid these tradeoffs; rather, the choice of a good TLB design is a compromise by nature – how do we make a TLB that for the types of applications we design the system for, within certain lookup time and chip area limits, will perform as good as possible with a maximal probability, and perform badly with a minimal probability? An often recited motto in computer architecture is "make the common case fast"; conversely, we also want to make the pessimal case rare.

Adding reorganisation to the skewed-associative TLB makes the fast case — a TLB hit — more frequent without slowing it down, at the cost of adding some complexity to handling the rare case — a miss.

1.1 Contributions

This thesis:

- describes models and discusses issues of comparing and evaluating TLB models, with a focus on avoidance of conflict misses

- introduces *collision tendency*, a method for evaluating and scoring different associativity models within a range of sizes

- presents and evaluates the *reorganising skewed-associative TLB*, comparing it to other well-known associativity models, and discusses tradeoffs for implementation of the model

- discusses and evaluates real-time enhancements to the *reorganising skewed-associative TLB*, proposing it as a cheap and energy-efficient design for real-time and embedded applications.

2   Related work

TLBs have not been studied as extensively as memory caches, although the concept is about as old as virtual memory using paging [TK86]. A detailed study of TLB behaviour and simulation in a modern virtual memory system was written by Clark and Emer in 1985 [CE85]; although rather specific to the contemporary implementations of the VAX architecture with its peculiarities, many of the observations still hold. A coeval study by Alexander, Keshlear and Briggs [AKB85] has a somewhat more general approach, describing the traditional associativity models and several issues of TLB design and measurement.

Attempts have been made to avoid using a TLB altogether and move the address translation into the cache. One such approach was described by Wood et al [WEG+86]; their method is probably not well suited to the heavily pipelined architectures of today, and furthermore requires a single, global virtual address space (which is very uncommon in modern virtual memory systems).

Several studies and articles of importance concerning associativity in caches, most of which apply to TLBs to a large extent, were written in the late 1980s: Hill studied the advantages and problems with direct-mapped caches [Hil88] and later, together with Kessler, Joos and Lebeck, devised methods to decrease the cost of set-associative models [KJLH89]; then, together with Smith, compared direct-mapped and set-associative models and presented methods for simulation and evaluation of these models [HS89].

An early proposal of a cache design using multiple simple hash functions for placement, named *hash-rehash*, was described by Agarwal, Hennessy and Horowitz [AHH88], inspired by a TLB design proposed by Thakkar and Knowles [TK86]. The idea was later improved upon by Agarwal and Pudar [AP93].

Skewed associativity was introduced for caches by Seznec in 1993 [Sez93]. Seznec, together with Bodin and others, examined the concept in further detail in several papers [SB93] [BS94] [DGJS93] [BS95], and Michaud [Mic01] described it from a theoretical statistical point of view. Seznec also elaborated on the choice of replacement strategies for skewed-associative caches and branch target buffers (BTBs) [Sez97].

The matter of choosing appropriate hash functions for skewing was more closely studied by Topham, González and González [TGG97], and by Topham and González [TG99]. Reorganisation in skewed-associative caches was described and evaluated by Spjuth [Spj02] and studied from a power consumption perspective by Spjuth, Karlsson and Hagersten [SKH03].

3   Background

3.1   Virtual memory systems

A virtual memory system handles two kinds of addresses; *physical* (used by the memory circuits and describing where an item is physically located in memory) and *virtual* (also called *logical*), used by software. This has several purposes; it allows the operating system to place and rearrange data used by processes in memory as it sees fit, even between primary and secondary storage, without the process needing to know about it (paging and swapping); it also provides a straightforward way to bar a process from accessing memory belonging to another process or even the operating system itself.

The mappings between virtual and physical addresses are stored in *page tables*. The set of page tables can be very large, and may therefore in their turn need to be split up into many smaller sets indexed by higher-level page tables [NUS+93] [Tal95] [WZ97].

As memory is accessed, the virtual addresses used by software must be converted into physical addresses in order to locate the data items. Such an address conversion may involve one or several accesses to page tables; the time that this takes may have a considerable impact on program execution speed in a computer system of today, and may well render the system unacceptably slow [WZ97] [SDS00]. To overcome this problem, a virtual memory system usually contains a small and fast cache for page table lookups, called a *Translation Lookaside Buffer (TLB)*[1]. When a memory page has been accessed recently, its mapping is likely to be in the TLB, speeding up accesses to that page in the near future. A processor may employ several TLBs; there is often one TLB for instructions and one for data.

3.2   The Translation Lookaside Buffer (TLB)

A Translation Lookaside Buffer (TLB) (fig. 1) stores copies of *page table entries (PTEs)* which are determined likely to be used soon. It is generally not feasible to know exactly which pages will be accessed in the near future, so this must be approximated in some way. Since a page that has just been accessed usually is accessed again soon, it normally makes sense to update the TLB on every TLB miss (i.e. an access to a virtual address that wasn't in a page already stored in the TLB), and then keep it there until it seems advantageous to overwrite the entry occupied by the address with something else (or until the mapping represented by the entry is no longer valid). When a new entry is about to be stored in a TLB which doesn't have any suitable free space, a decision must be made which old entry to remove in order to provide a location where the new entry can be stored. This is done by a *replacement algorithm*. Replacement can be done independently of what is already contained in these entries, such as by *random replacement*, or using some heuristic to guess which of these entries that are the least likely ones to be used again soon, or otherwise determined to be less desirable to keep.

---

[1] *Translation Lookaside Buffer (TLB)* is the common term in current literature, although other names have also been used for the same concept: *Translation Buffer (TB), Directory Look-Aside Table (DLAT), Translation Cache (TC)* [AKB85] [CE85]

TLB hit: address translation through the TLB

page tables

virt page addr   TLB

TLB lookup

(hit)
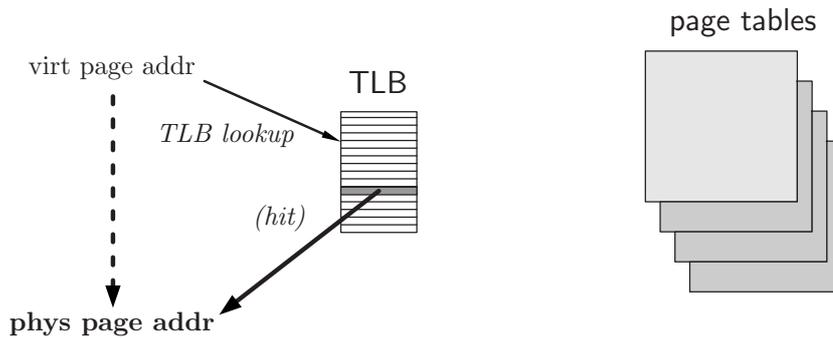
phys page addr

TLB miss: address translation through page tables, updating the TLB

page tables

virt page addr   TLB   *(miss)*   *lookup*

TLB lookup

fill

phys page addr

Figure 1: Address translation using a TLB: A virtual page address is first looked up in the TLB. On a hit, the corresponding physical page address can be returned immediately, avoiding the much slower path through the page tables. On a miss, the page tables are accessed; the TLB is then updated with the result to avoid further misses on future accesses to the same memory page. This is called a *TLB fill*.

The set of pages actively used by an execution thread (or sometimes a set of threads) is called its *working set*, and the amount of memory that a TLB may index concurrently is called its *reach*. Preferably, a TLB in a processor should be able to contain all of the working sets of the currently executing threads; this implies that the 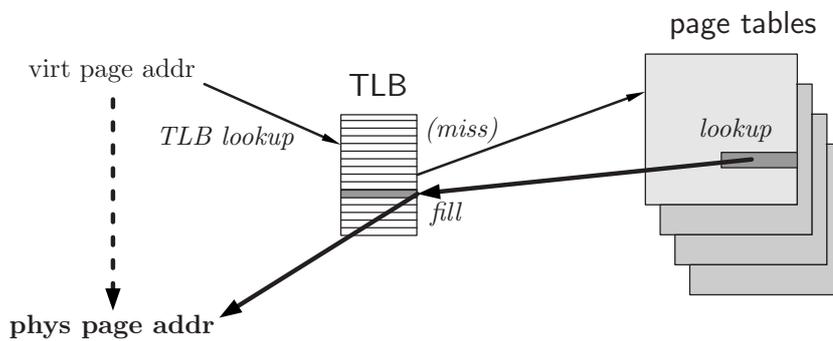*TLB reach* should preferably be at least as large as the working sets of the executing threads in addition to the working set of the operating system kernel. When the union of these working sets becomes much larger than the TLB reach, the TLB may exhibit *thrashing*, i.e. cause the CPU to spend more time on handling TLB misses than actually executing code.

The opposite of a miss is a *hit*, i.e. a case where an item is found in the cache/TLB as a lookup. The fraction of the accesses are that cause misses and hits are called *miss rate* and *hit rate*, respectively.

3.3   TLB entries

Some TLB entries map virtual page addresses[2] to physical page addresses; these entries are called *valid*. An entry may also be *invalid*, meaning that it does not contain any mapping, or that any mapping that it might contain is not to be used. Further, TLB entries may contain some additional data such as information on how recently pages that the entries correspond to have been accessed.

On a TLB miss, the page mapping is loaded from page tables. This is called a *TLB fill*[3]. Both hardware and software implementations of this procedure are common [NUS+93] [JM98].

To efficiently keep track of several different threads of execution with different memory mappings, TLB entries may furthermore contain context information. For associativity and replacement purposes, this can be seen as an extension of the address space[4].

When page mappings become invalid, the corresponding entries in the TLB must be invalidated. An implementation may support invalidation of individual entries, or invalidation operations that invalidate all or a subset of of the entries in the TLB, for example based on context information for TLBs that support multiple contexts.

To achieve better throughput, TLBs may be multiported, i.e. able to do several simultaneous lookups. An even more common way to avoid letting the TLB become a bottleneck is to in fact have two separate TLBs, one for instructions and one for data [JLN97]. This works well in practice, as data and instruction fetches are largely independent, and data and instruction working sets rarely overlap.

---

[2]By *page address* we mean an address that, within its address space, uniquely identifies one page. The page address is typically identical to the lowest address within the page.

[3]Some literature uses the term *bump* to denote a fill that replaces a valid entry, and reserves the term *fill* for those that replace an invalid entry [CE85]. This distinction is rarely made, and will not be used here.

[4]This context information is usually in the form of an address space identifier (ASID), a number identifying the virtual address mappings belonging to a certain process (or kernel) context. From a TLB point of view, the ASID can essentially be seen as part of each virtual address. [HH93]

3.4 Impact of TLB performance on system performance

TLB miss rates are typically under one percent [MKGS97]. TLB misses can still cause a sizable performance loss, because of the costs of handling a miss. For some applications, the percentage of CPU time spent on handling TLB misses can easily reach 10%–20% of the total CPU time [SDS00] [KS02], and numbers beyond 40% are not unheard of [HH93]. Some processors may continue executing instructions independent of the one causing a TLB miss during miss handling, and even instructions belonging to other threads of execution, but this can only partly compensate for the miss costs; the instruction stream is ultimately delayed in any case.

The speed differences in memory hierarchies have been increasing for quite some time now [WM95]. This is to some extent compensated for by larger caches and more cache levels, but we can nevertheless expect that that the relative costs for handling TLB misses will increase as well.

3.5 Associativity

As has already been mentioned, translation time is a very important property of a TLB. When a lookup is performed, any matching valid entry should be found as soon as possible; any unnecessary delay is likely to delay the entire CPU. For every valid entry in the TLB that could contain a matching page address, it has to be determined whether it contains the sought after data or not; at least if the entries are tested in parallel, which is normally the only reasonable choice for reasons of speed.

Finding a matching valid entry naturally become more costly (in time or hardware) the more entries that have to be tested for a match. We define *degree of associativity* (or just *associativity*) as the number of possible placements for each logical page address; less formally, we will also use the word *associativity* to designate the property of allowing many possibilities for where to place such an address. One extreme in this property is the *direct-mapped TLB* [AKB85], which uses a hash function to decide a single possible placement for each address (fig. 2). The hash function used is typically a very simple one, such as masking away all but the very last n bits of the page part of the address, for a TLB size of 2^n entries. The other extreme is represented by the *fully-associative TLB* [AKB85] [MKGS97], where addresses can be placed totally freely (fig. 3). We will refer to these limitations of where an entry may be stored as *placement restrictions*.

3.6 Placement and replacement of TLB entries

Abstractly, the contents of a TLB T of size s can be seen as an array of entries, $T_0 \ldots T_{s-1}$. For some associativity models, it is more appealing to view the TLB contents as a multi-dimensional array, i.e. an array indexed by pairs of integers. Let T_{loc} be the set of locations of each TLB T, and Π_V and Π_P be the sets of virtual pages and physical pages, respectively. We can then describe these placement restrictions for T as a *placement function*

$$p_T \quad : \quad \Pi_V \quad \longrightarrow \quad \wp(T_{loc})$$

that maps each virtual page to a *placement set*, a subset of the set of locations (i.e. to an element in $\wp(T_{loc})$, the power set of T_{loc}). For theoretical descriptions
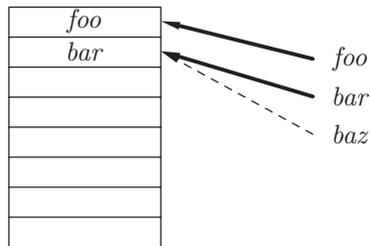
Figure 2: A direct-mapped TLB: Each virtual page address has one possible placement. If two page addresses map onto the same location, it is impossible to keep them in both in the TLB simultaneously. The thick arrows show how some of these entries can be stored the TLB; obviously, *bar* and *baz* can't be there at the same time, as *baz* can only be stored by overwriting *bar*.
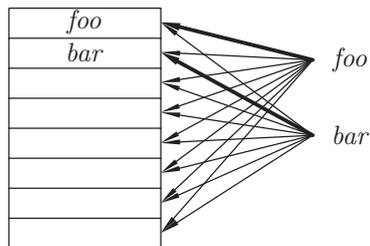
Figure 3: A fully-associative TLB: Each virtual page address maps onto every location in the TLB, so the a TLB of size n may contain any n page addresses. Thus, the TLB can contain any set of addresses that isn't larger than the size of the TLB. The thicker arrows show the placements for *foo* and *bar* in this example; however, any two different locations could have been chosen.

of associativity and replacement we can largely disregard what information an entry actually contains and how it is represented, but we will assume that there is a set of possible entries E and *virtual address selection* and *physical address selection* functions.

$$\pi_V \quad : \quad E \quad \longrightarrow \quad \Pi_V$$

$$\pi_P \quad : \quad E \quad \longrightarrow \quad \Pi_P$$

that, given a valid TLB entry, tell which virtual page it maps to which physical page. We must also be able to decide whether an entry is *valid* or *invalid*:

$$valid \quad : \quad E \quad \longrightarrow \quad \{\text{true}, \text{false}\}$$

The contents of an entry e will then, if valid, map the virtual page $\pi_V(e)$ to the physical page $\pi_P(e)$, and its possible placements[5] in a TLB T are given by $p_T(\pi_V(e))$; if e is invalid, its possible placements are given by T_{loc}. A mapping for a virtual page a is thus in T if there is a $p \in p_T(a)$ such that $valid(T_p)$ and $\pi_V(T_p) = a$. Updating of the TLB should always preserve *consistency*, i.e. make sure that there are not multiple concurrent mappings associated with a virtual page that do not agree with each other. (Several mappings to the same physical page is however not an error; it is often useful to permit such mappings.)

A TLB lookup looks approximately as follows (in pseudo-code; note that this is done in hardware and that the comparisons are almost always done in parallel):

```
TLB_lookup(T, x):                        # x is a virtual address
  let a = virtual_page_of(x)
  let o = page_offset_of(x)              # offset within page
  for any placement p ∈ p_T(a)
      such that valid(T_p) and π_V(T_p) = a :
    update meta-data for T_p
    return π_P(T_p) + o                  # Hit!
call TLB_miss_handler(a)                  # Otherwise: miss
```

Finding out which virtual page an address belongs to gets a bit more complicated if multiple page sizes are allowed (see 4.4.1, pg. 21); for simplicity, we assume one single page size in this description. The meta-data update on a hit includes bookkeeping for replacement purposes, such as marking the entry as recently used. On a miss, a *miss handler* is called, which performs page table lookup and fills the TLB with an entry for the virtual page where the miss occurred; if there is no valid entry for the page in the page tables, then the operating system is notified. (The miss handler can be implemented partly or entirely in software; how a miss handler operates depends much on page table layout and is outside the scope of this thesis.)

A lookup as above usually also involves fetching some meta-data from the entry on a hit. Such meta-data may for example be used to determine whether a process is allowed to read from or write to the page in question. A page table entry often also contains a "dirty bit" that is set whenever something is written

---

[5] "Possible placements" are here defined as associated with a virtual page. We will sometimes talk about possible placements for an entry; by that we mean the possible placements for the virtual page that is mapped by the entry, or, if the entry is invalid, any location in the TLB (since an invalid entry may be placed anywhere).

to the page. When an entry in the TLB is overwritten, some dynamic meta-data such as this bit may have to be written back to the page tables. For simplicity, we ignore these issues, which are irrelevant to our comparisons of associativity models.

Since a TLB contains an entry (valid or invalid) in each location, and any entry may need to be invalidated, we can assume that placement functions always allow an invalid entry to be placed anywhere. The contents of invalid entries are not used and thus do not matter, except that we must be able to keep track of their status as invalid. As invalid entries are not reused but only kept or entirely overwritten, their representation does not matter much; for the purposes of describing or simulating associativity models we can therefore allow arbitrarily placements of invalid entries even at locations where their (bogus) contents would otherwise not be permitted.

As for replacement, an associativity model also decides which replacements are possible. Abstractly, a replacement consists of overwriting a TLB entry — the *victim* — with new contents, and possibly other actions specific to the associativity model and the replacement algorithm. (In a concrete implementation, there is a writeback step before the victim is overwritten; if the cached victim PTE is valid and has been modified, the page tables need to be updated so that these modifications are not lost.)

3.7   Replacement policies

An important property of a TLB or memory cache is its *replacement policy* (or *replacement algorithm*), i.e. how the entry to replace is chosen when a new entry is to be stored. *Direct-mapped* models have only one possible placement for each entry; thus, there will only be one possible choice for replacement. Many models, however, allow for more replacement choices. This issue is largely orthogonal to the associativity model used, even though some combinations are simpler to implement than others.

A common replacement policy is *LRU*, where the *least recently used* entry is overwritten at replacement. (An invalid entry can be viewed as less recently used than any valid entry, and will therefore always be preferred for replacement.) With a greater degree of associativity it becomes increasingly more expensive to implement genuine LRU, so it is common to choose some approximation to this heuristic. It should be noted that while LRU is often seen as a virtually optimal replacement strategy, it is not always optimal. For example, imagine that a program repeatedly loops through an array of sequential pages accessing each page once, and the array is somewhat too large for all its pages to fit in the TLB. Then the least recently accessed parts of the array will be accessed again very soon, contrary to the LRU principle. In such a case, a direct-mapped TLB may perform remarkably better than a fully-associative one; pathological examples like this are however not very common.

This phenomenon does make it a valid observation that a lower degree of associativity leads to less TLB misses in some rare cases — this should not be interpreted as an advantage of a design with a lower degree of associativity, but rather as a failure of LRU as a replacement heuristic; by providing a very limited choice of entries to replace (only one in the case of a direct-mapped TLB), the design will only rarely allow the least recently used entry of the TLB to be replaced, and this might actually be advantageous in some cases. This failure of

the TLB to follow the LRU heuristic, combined with this rare situation where the LRU heuristic gives very bad replacement suggestions, is an unusual case where two wrongs actually make a right.

Although enhancements and alternatives have been proposed, the LRU heuristic is commonly considered very good, and it is regarded as a very natural way of trying to predict future accesses. For some simple associativity models, such as 2-way set-associative models, it is very cheap to implement; however, as associativity increases, it quickly gets more complicated to keep track of the order in which entries have been recently accessed. Therefore, an approximation is often chosen; a heuristic of this type is called "pseudo-LRU".

Another common replacement strategy is *random replacement*, where an entry to replace is picked at random (in practice generally pseudo-randomly). This is simple to implement as it does not require any additional per-entry state. The unpredictability of the strategy means that it rarely performs pessimally, but it is generally far from optimal since entries recently used are just as likely to be replaced as entries that have not been used recently. Another simple strategy requiring very little state is *FIFO (first-in-first-out)*, where the oldest (least recently stored) entry is replaced; there are multiple variations on this theme, sometimes including some minimal information about the most recently accessed entries in order to avoid replacing those. This modifies the strategy to a crude approximation of LRU.

Abstractly, we may view the replacement strategy as an algorithm cycling through and comparing or otherwise selecting one replacement from the replacements permitted by an associativity model. In practice, these tests are usually parallelised to some extent.

3.8   Miss categories and associativity

TLB and cache misses are typically categorised as follows [HP96] [Jou98]:

- Capacity misses: *Capacity misses* are misses caused by a working set being too large to represent within the available locations. These are simply the misses that a conflict-free (i.e. fully-associative) model of the same size wouldn't be able to avoid regardless of replacement policy.

- Coherence misses: *Coherence misses* occur in multi-processor systems where entries may need to be invalidated to preserve cache consistency between processors. This is more of a problem for caches than for TLBs, since cache contents change more often than TLB contents do (except for internal bookkeeping data for replacement purposes and the like, that is often merely used for optimisation and not critical to consistency). This class of misses is unrelated to size, associativity and replacement, and will not be further elaborated on in this thesis.

- Compulsory misses: Sometimes a data item isn't cached simply because it has not been used before; this is called a *compulsory miss*. Compulsory misses cannot be avoided by increasing size or associativity. Therefore, they tend to constitute a relatively larger part of the miss total as both these parameters are increased, bringing down the overall miss rate.

- Conflict misses: *Conflict misses* are misses caused by suboptimal placement of entries, typically due to placement restrictions; these are the

13

misses that a fully-associative model, i.e. a model with no placement restrictions, could have avoided. Hence, a fully-associative model is considered not to have any conflict misses.

An important class of misses is misses due to suboptimal replacement choices. These can be seen as "avoidable" conflict misses, and are usually considered as such; it is a bit unfortunate that these classifications do not specifically distinguish conflict misses caused by the replacement policy from conflict misses due to the associativity model used. This thesis focuses on exploiting associativity to decrease conflict misses, so replacement policy issues will only be discussed briefly. (Agarwal, Hennessy and Horowitz [AHH88] use another classification, calling misses caused by replacements *interference misses*. By this definition, both conflict and capacity misses are interference misses.)

To decrease the TLB miss rate we must reduce one or several of these types of misses, possibly at the cost of increasing the rate of one or more of the others by a smaller amount. Constraints such as a limited chip area, a maximum acceptable lookup time and the acceptable energy consumption must be considered when doing this. Note that an increased energy consumption makes the chip hotter, increasing cooling requirements.

Capacity misses can trivially be reduced by increasing the size of the TLB. However, increasing the size will increase the cost in chip area by about the same factor, and will also lead to an increased energy consumption. However, the size of a TLB entry does not increase with a larger page size; thus capacity misses can also be decreased without increasing TLB area by using larger pages and similar techniques. This is not the main subject of this thesis, but it will be discussed briefly later (see 4.4, pg. 20).

Compulsory misses can in some sense be partially avoided by not completely invalidating the contents of the TLB at a task switch (when multitasking). This can be done rather efficiently by adding context tags to TLB entries and restrict lookups to specific contexts, or by other techniques; these methods typically do not depend on size or associativity, and are outside the scope of this thesis. (However, if the context is simply seen as an extension of the virtual address space, then the inability to mix contexts arbitrarily in the TLB constitutes a kind of placement restriction; so it would also make sense to classify these context switch misses as conflict misses. This thesis does not attempt to further analyse or classify these misses.)

Conflict misses can be reduced without changing anything except how data is organised, so they can often be significantly reduced at a comparatively low cost. This thesis will focus on how this can be done by skewing (using multiple hash functions in parallel) and extensions of this technique. When the TLB is just large enough for the working set to fit, much of the remaining misses tend to be conflict misses (this is sometimes called "the unit working-set problem" [Mic01]). Therefore, associativity is especially important for TLBs that are "just large enough".

A direct-mapped TLB D of size s will use one hash function f_D : to decide one single possible placement for each virtual page address:

$$f_D \quad : \quad \Pi_V \longrightarrow \big\{0 \ldots (s-1)\big\}$$
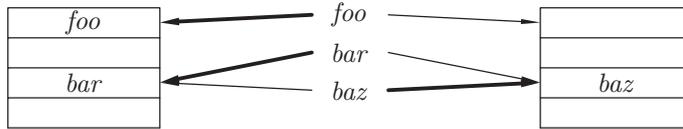
$$p_D(a) = \{D_{f_D(a)}\}$$

14

Figure 4: A 2-way set-associative TLB (the simplest non-trivial case of n-way set associativity). Each virtual page address maps onto n locations, i.e. two locations in this example. The thicker arrows indicate an attempt to insert these entries into the TLB; the TLB can only contain n virtual addresses that hash onto the same locations, so there are still cases where the TLB can't contain a set of entries which actually isn't larger than the size of the TLB. However, these cases are not as common as for the direct-mapped TLB.

A fully-associative TLB F does not need or use any hash function for placement, since any entry, valid or invalid, may be in any location:

$$p_F(a) = F_{loc}$$

In both cases, there are as usual as many possible replacements for a fill as there are possible placements for the new entry. Each of the possible placements for the new entry corresponds to one possible replacement, overwriting the entry at that placement.

3.9   The traditional compromise: n-way set associativity

For both TLBs and memory caches, full associativity has often been seen as too expensive, while direct-mapped models are much too plagued by conflict misses. (In a set of entries, the number of conflicts is often larger than one might think; Michaud [Mic01] notes that this is another incarnation of the famous "birthday paradox".) A simple and often used compromise is *n-way set associativity* (fig. 4), where the TLB (or cache) is divided into n equally large columns (or *ways*), searched in parallel using the same hash function [AKB85]. This is actually a generalisation of the two aforementioned associativity models, since a direct mapping is just the same as 1-way set associativity, and a full associativity is nothing but n-way set associativity where n equals the total number of entries.

For a n-way set-associative TLB T with $m{\times}n$ locations, using a hash function f_T for deciding one placement in each of the n columns, the possible placements for a virtual address a are:

$$f_T \quad : \quad \Pi_V \longrightarrow \big\{0\ldots(m-1)\big\}$$

$$p_T(a) = \bigcup_{i=0}^{n-1} \big\{T_{f_T(a),i}\big\}$$

For an n-way associative TLB to be considerably faster (on a hit) and simpler to build, n should be fairly small, and the hash function that decides where entries may be located must be computed very quickly. The common choice of hash function is one that just picks the least significant bits of the page part of the address [CE85]. Such a function is extremely fast and simple to compute,
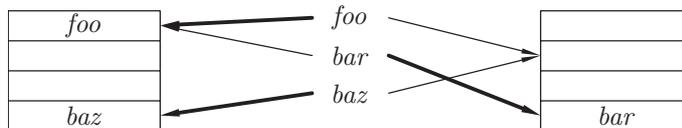
Figure 5: A 2-way skewed-associative TLB; different functions are used for deciding possible placements in the two columns. As entries that collide in one column (such as *foo* and *bar* above, or *foo* and *baz*) are unlikely to also collide in the other column, the risk for conflict misses is significantly lower than that of a 2-way set-associative TLB of similar size.

and also distributes page addresses very well so that pages close to each other in the virtual address space will not map onto the same set of locations. However, such simple hash functions cause unnecessary conflict misses; for example, when looping through large arrays, most notably in the kind of algorithms used in scientific computing, one often gets working sets containing several pages spread out with a fixed distance between them. If these distances have large common divisors with the size of the TLB, several virtual page addresses in the working set will often map onto one small set of TLB locations — if this set of locations is smaller than the set of page addresses that conflict for those locations, then entries for these locations will often need to be overwritten even though there may be less needed entries elsewhere in the TLB. Thus this leads to conflict misses, and is one of the main reasons why fully-associative TLBs are used so often. Still, fully-associative models are more complicated and often slower on a hit as compared to the alternatives, because of the many comparisons that have to be done when doing lookups.

Manne, Klauser, Grunwald and Somenzi [MKGS97] have described a *banked associative* TLB design, that uses a hash function to map each virtual page address to one of m small fully-associative banks of size n. Associativity-wise, this is equivalent to a n-way set-associative model.

Another "mixed breed", combining full associativity with direct-mapping, is used in the *victim cache* described by Jouppi [Jou98]. Although described as two parallel caches with different associativity models, one large direct-mapped and one fully-associative, it can just as well be viewed as a single entity where each entry has one possible placement in the large direct-mapped part, and in addition can be placed anywhere in the smaller fully-associative part. This is combined with a simple reorganising replacement technique that always stores a new entry in its only possible position in the direct-mapped part, letting the tossed out entry replace one entry in the fully-associative part. Thus, the fully-associative part will only be used to avoid collisions. This scheme avoids much of the collisions in a direct-mapped model of about the same size, assuming that the fully-associative part is large enough; it is mostly suitable for small cache/TLB sizes where the fully-associative part can be made relatively big (compared to the size of the entire cache/TLB) without becoming large in absolute numbers [AP93].

3.10   The skewed-associative TLB

A proposed solution to the problems with conflict misses in n-way set-associative TLBs is the *skewed-associative TLB* (fig. 5). The basic idea behind this associativity model is to try to avoid associating many virtual page addresses in a working set with one and the same small set of locations by using different hash functions $f_0, f_1, f_2 \ldots f_{n-1}$ for the different columns of the TLB. The n-way skewed-associative TLB is thus, just as in the normal n-way set-associative case, split into n columns of equal size (to be searched in parallel at lookup), each using one of these hash functions. A virtual address a may then be placed into one of the locations $(f_0(a), 0), (f_1(a), 1), (f_2(a), 2) \ldots (f_{n-1}(A), n-1)$. This means that when some addresses collide (i.e. are hashed to the same location) in one column of the TLB, they are likely to be hashed to different locations in another, making it much less probable that a working set won't fit into the TLB because of a subset that causes collisions in multiple columns.

For a n-way skewed-associative TLB T of $m \times n$ entries, the possible placements for a virtual address a are:

$$p_T(a) = \bigcup_{i=0}^{n-1} \{T_{f_i(a), i}\}$$

Lookup clearly becomes a bit more complicated for a skewed-associative TLB than for a set-associative TLB, but computation of the different hash functions can (and certainly should) be done in parallel, reducing the extra cost at lookup to the cost of actually computing these functions. André Seznec demonstrates (for skewed-associative caches) that skewing works quite well even when very simple and easy to compute hash functions are used [Sez93], and suggests an appropriate class of functions. More elaborate (and more expensive) hash functions for skewed-associative caches have been suggested and described by Topham and González and others [GVTP97] [TGG97] [TG99]. However, the strict timing constraints may make these more complicated functions less suitable for TLBs.

An associativity model closely related to skewed associativity is used in the *column-associative* cache design described by Agarwal and Pudar [AP93]. This design uses two or more hash functions for lookups within a single array, with a simple reorganisation technique to partly compensate for the fact that the design only does one comparison at a time; this means that a lookup often will take more than one cycle even on a hit, so column-associativity is probably less well suited for TLBs.

3.11   Hash functions

It is important that the hash functions used are fast and cheap enough to compute for a realistic TLB implementation; for models using multiple hash functions, these should also be different enough to minimise the risk that the same collisions occur in several columns for set of addresses. Ideally, the hash functions should be computed using only a few layers of gates. It has been demonstrated [GVTP97] [TGG97] that polynomial hash functions may achieve better results. However, they are rather expensive to compute (both in terms of time and chip area), and it is essential that computing the hash functions does not take more time than is absolutely necessary; slow to compute hash

functions will slow down *every* lookup and may easily lower the speed of the entire processor noticeably.

Seznec and others [DGJS93] [Sez93] [BS94] [BS95] have proposed constructing simple hash functions as a simple combination of exclusive-or (XOR) operations and bit permutations. These operations are trivially implemented in hardware. For the tests performed for this paper, it has been desirable to get a parameterised hash function that can be used as multiple similar (but different) hash functions. A function of this class, generating hash functions similar to those used by Seznec et al., is

$$\text{xor3}(n)(a) \quad = \quad xor3_{17}(n)(a_{16..0})$$

where n selects one of the hash functions generated, $a_{16..0}$ refers to the 17 least significant bits of the virtual address a, and $xor3_{17}(n)$ is defined by

$$\text{xor3}_{17}(n)(a) \quad = \quad a \oplus \overleftarrow{R}_{17}(\lfloor \frac{n}{2} + 1 \rfloor)(a) \oplus \overrightarrow{R}_{17}(n \oplus 11_b)(a) \quad ,$$

$\overleftarrow{R}_{17}(n)$ and $\overrightarrow{R}_{17}(n)$ being the n steps rotate left and rotate right operations, respectively, operating on 17-bit words[6]. $\oplus$ is bitwise exclusive or (XOR). As n is constant for each hash function generated by xor3, all the rotates are constant for each function; the hash functions generated simply construct each bit in the result by an exclusive or operation on three different bits of the input. Thus, each hash function can be implemented in hardware using simple bit permutations and one single layer of 3-input XOR gates, operating in parallel, computing one bit each of the result.

The hash functions $xor3(0)\ldots, xor3(n-1)$ are used as $f_0\ldots f_{n-1}$ for evaluating skewed-associative designs in this thesis.

It should be noted that the commonly used hash functions in n-way associative TLBs, bit selection functions, have the small advantage that some bits of the address stored in an entry trivially can be computed from the hash (which is given by the TLB location) and therefore do not need to be represented in the TLB. This property requires that the hash contains as much information as the bits that it depends on; an evenly distributing hash function with the range $\{0\ldots 2^{n-1}\}$ may then only depend on n bits of information. Hash functions depending on so few bits of input can be expected to yield significantly worse results, making them less appropriate for placement [Sez93] [GVTP97].

3.12 Skewed associativity and reorganisation

A new class of associativity models has been spawned by the observation that skewed associativity allows further conflict reduction by moving entries around to give more replacement choices and a higher apparent degree of associativity. The *elbow cache* described by Spjuth [Spj02] takes advantage of such reorganisation to allow for far lower miss rates than conventional n-way set-associative caches with the same number of columns; instead of overwriting an entry, that entry can be moved to another location not conflicting with the entry to be

---

[6]The number 17 was chosen rather arbitrarily; it was deliberately chosen to be large enough to hash based not only on the very lowermost bits of the page address, and also to give a reasonably large number of different hash functions — more than ten are more than enough for the tests done for this thesis.

inserted. This was first suggested as a way to increase performance at a low cost in terms of hardware complexity; this cache design was later demonstrated by Spjuth, Karlsson and Hagersten [SKH03] to be cost efficient also in a power consumption perspective.

Most work in this area has been done on caches; this thesis focuses on the application of skewed associativity with reorganisation in TLBs.

4 TLBs and other caches

Per definition, a TLB is a kind of cache, a very fast memory to speed up retrieval of recently seen data. Memory caches have been studied more than TLBs; while there is much to learn from that area, and many concepts and enhancements relating to caches apply just as well to TLBs, we should also be aware of some differences — not only in size and speed requirements, but also in locality properties.

Several performance-enhancing techniques for memory caches are also applicable to TLBs; there are also some possible enhancements that are specific to TLBs. The enhancements discussed in this section are largely orthogonal to the associativity model and the choice of replacement policy.

4.1 Size

While the size of a memory cache ranges from tens of kilobytes to several megabytes, the storage capacity of a TLB is more likely to be measured in kilobytes or merely hundreds of bytes. Reasons for this, and some consequences, are discussed below.

Primarily, however, the job of the TLB is to *map* memory contents, not to store or buffer them, and this is what sets TLBs apart from instruction and data caches. The amount of memory that can be concurrently mapped by a TLB is commonly referred to as *TLB reach*. For example, a 64-entry TLB mapping 8kB pages has a TLB reach of $64 \cdot 8$ kB, i.e. 512kB; if each entry uses eight bytes of storage, it will however only need to contain $64 \cdot 8$ bytes (512 bytes) of storage in total to map this amount of memory. (As we see, TLB reach depends to a large extent on other factors than TLB size.)

The size of a TLB is largely limited by its speed requirements. Larger TLBs occupy more chip area and have longer signal paths; this tends to make them slower.

4.2 Speed

For memory to be really fast you usually have to fit it on the same chip as the processor itself, which puts severe restrictions on size. Therefore, there are often several levels of cache in a computer system; one small and extremely fast cache (the L1, i.e. level 1) cache, and one or two much bigger but slower cache levels; in a multi-processor system some levels may be private to each processor and others shared between some or all of the processors. A read from cache may take from one or two cycles for a fast L1 cache to tens of cycles for slower levels in the hierarchy; just as cache contains copies of items recently used in memory, the L1 cache will typically be used for keeping copies of recently used items from

the L2 cache, and so forth. (Sometimes, some reorganisation technique is used to avoid this replication of cached data.)

In the same manner, it is possible to design a hierarchy of TLBs of different speeds and sizes. In a sense, memory caches become part of such a hierarchy, since page tables that need to be accessed when handling at a TLB miss may already reside in memory cache.

However, the memory cache will not do all the work of the TLB; much desired features like high associativity (with parallel comparisons) and such are in general expensive to implement for memory, and can often only be afforded for small caches such as TLBs. As a TLB lookup is needed for effectively every memory operation, it is critical that a TLB lookup doesn't take more time than absolutely necessary; therefore, the benefits of implementing a fast, multiply-associative TLB often outweighs the cost. Another consequence is that hash functions used for placement in TLBs may need to be simpler and faster to compute than hash functions used in other contexts, such as for slower levels of a memory cache hierarchy.

4.3   Locality

The concept of caching is based on two assumptions about the behaviour of the programs being executed; *time locality* and *space locality* (or *spatial locality*). *Time locality* means that a recently accessed item is likely to be accessed again soon.

Space locality is a related concept. It means that for recently accessed items, items at nearby addresses are likely to be accessed soon. When implementing memory-intensive applications, software developers will often make an effort to make sure that their routines exhibit good space locality, so that the caches can do their job better and thus make the application run faster.

In the absence of locality, the benefits of caching are very small; first, because there probably is no way to guess what will be accessed soon and therefore should be kept in the cache; second, because the speed of the slower levels in a memory hierarchy will limit the number of operations that can be performed within a certain amount time even if we have some means to foresee what data we want to fetch soon — if this doesn't become an issue, there is probably not much point in caching as the slower levels are fast enough for our needs. Luckily, most programs exhibit good or even very good locality properties; in practice, there will usually be some small set of data that is accessed very often.

4.4   Exploiting space locality — lines, blocks and pages

There is a simple method to exploit space locality; namely, to divide data into chunks of an appropriate size, such that when two items reside close to each other in memory, they are likely to end up in the same chunk. When data is mapped into an address space or moved in a memory hierarchy, this is done one chunk at a time. These chunks are usually called *cache lines* for memory caches, *pages* for random-access memory and *blocks* for disk devices, but the idea is basically the same. Typically, the chunk sizes are chosen so that the size of a chunk is a power of two; for a binary machine, this makes it trivial to decide which chunk an address belongs to.

Cache line sizes are usually measured in tens or at most hundreds of bytes; larger sizes tend to make the traffic between cache and memory excessive [CGS97]. TLB entries, on the other hand, do not store the contents of pages but merely references to them; to some extent, larger page sizes are more likely to instead decrease traffic and miss costs, as they allow for an increased TLB reach without increasing the size of the TLB [TH94]. However, larger pages give less fine-grained control over permissions, as access permission bits are usually specific to each page; larger pages also make loading and storing pages more expensive. Several schemes have been devised to increase TLB reach without sacrificing too much granularity; some of these are described briefly below.

The different spatial scales of cache lines and memory pages may give TLBs different space locality properties than caches, as strides that are small from a page (TLB) perspective may seem large from a memory cache point of view [AHH88].

4.4.1 Increasing TLB reach: superpages

A compromise between choosing a small page size (which gives better granularity and makes it less costly to load and store a page) and, on the other hand, choosing a large page size (which increases TLB reach and often exploits spatial locality better) is to allow for multiple page sizes [TKHP92]. These systems commonly have a base page size but also allow for larger pages, often referred to as *superpages*. These larger page sizes require some effort on part of the operating system to be of much use, and also add some complexity to miss handling; it is important that the advantage gained by having multiple page sizes isn't lost due to miss handling becoming more expensive [Tal95].

Multiple page sizes add some complexity to paging, and also to the construction of a TLB; given a virtual address, the TLB must be able quickly identify which virtual page the address belongs to, and this can no more be done by just masking off a fixed number of bits. Seznec [Sez03] has demonstrated how this can be done with skewed-associative TLBs with certain rather strict restrictions on page placement. Pages of certain sizes are restricted to certain columns of the TLB, depending on some rather high bits in the address (this spreads the load evenly over the columns, even when the pages in use are unevenly distributed over the available page sizes). Thus, an 8-way skewed-associative TLB can appear as four 2-way skewed-associative TLBs, each one for pages of a specific size. Associativity is sacrificed for page-size flexibility, trading conflict misses for capacity misses.

4.4.2 Increasing TLB reach: TLB sub-blocking

Another way to achieve the low granularity of small pages and still increase TLB reach is *TLB subblocking*, described by Talluri and Hill [TH94], where several pages adjacent in virtual memory space share one entry, with some separate meta-data for each. *Partial sub-blocking* (as opposed to *full sub-blocking*) requires the physical pages referenced by such an entry to reside at corresponding positions in a same-sized set of physical pages; thus memory is essentially divided into large pages, but each of these pages is divided into multiple "sub-pages", which share a TLB entry but with some separate meta-data for each. Talluri and Hill demonstrate that this can be done efficiently at a small cost in chip area,

especially with some clever memory management by the operating system. Talluri [Tal95] reports achieving better simulation results from subblocking designs than from multiple-page-size models that would occupy a similar chip area.

These techniques avoid certain problems with varying page sizes, such as the complexity of deciding which virtual page a virtual address belongs to. This suggests that it would be suitable enhancement to a skewed-associative TLB.

4.4.3 Lowering miss costs

A different approach to increasing VM performance is lowering the cost of a TLB miss. This can be done by using *page table pointer caches (PTPCs)* [WZ97]. A PTPC is a small table containing physical address pointers to recently accessed page tables. At a TLB miss, the PTPC is consulted. If the PTPC contains a pointer to the right page table, then the lookup can be done accessing only that physical page instead of going through a complete hierarchy of paged multi-level page tables. This decreases the costs of a large majority of TLB misses to one single access outside the TLB.

This method can take advantage of spatial locality at a higher level; one page table at the lowest level typically contains hundreds of page table entries for pages adjacent in the virtual memory address space. If a TLB miss causes an entry to be loaded from such a page, the PTPC will then contain a pointer to that page, decreasing the costs of subsequent accesses to page mappings contained in the same page table. Because so many virtual pages correspond to a single PTPC entry, a large PTPC hit rate can be achieved with a relatively small PTPC.

As TLB misses will still occur and be more costly than TLB hits, a PTPC will not entirely compensate for bad TLB performance. It can, however, serve as a complement. Since PTPCs are consulted more rarely than TLBs and a PTPC miss is followed by a generally more expensive page table reference in any case, PTPC lookups are not as time critical as TLB lookups, and should not have the same need for efficient associativity models.

An idea, similar to the PTPC concept, that applies to multi-level page table hierarchies, is to reserve a certain portion of the TLB to second-level page table entries [NUS⁺93]. Higher-level PTEs will not need to be accessed as long as the working set otherwise fits into the TLB, but keeping a few recently accessed higher-level PTEs in the TLB is likely to decrease most miss costs to just a single memory access.

Other ways to decrease miss costs include page table layouts that further exploit locality in order to lower the memory cache miss rate while servicing TLB misses [Tal95], and preloading techniques, where one tries to predict future TLB misses and handle them before they actually occur [SDS00].

4.5 Separation of instructions and data

Instruction and data working sets are usually independent of each other, with a few notable exceptions such as self-modifying code and just-in-time compiling environments (JIT). This means that separate caches can be used for instructions and data — these caches can work independently of each other, avoiding pipeline stalls when an instruction and a data word need to be fetched simultaneously [HP96].

For basically the same reasons, a CPU often has one TLB for instructions and one for data. This paper focuses on avoiding TLB conflict misses for data; instructions often have different time and space locality properties, so results based on data TLB behaviour might not necessarily apply to instruction TLBs. TLB performance for data is generally more interesting than for instructions, since high TLB miss rates for data tend to be much more common than high miss rates for instructions [KS02].

5   Reorganisation in the skewed-associative TLB

The skewed-associative TLB can be further improved by reorganisation, which generally increases the number of replacement choices for each TLB fill. This decreases the risk for conflict misses, giving a lower miss rate.

This section:

- elaborates on placement and replacement in the skewed-associative TLB

- describes single-step reorganisation, a simple way to increase the apparent associativity and thereby lower the miss rate of a skewed-associative TLB without increasing parallelism

- describes further miss rate reduction through multiple-step reorganisation

- discusses reorganisation decisions and replacement selection in the reorganising skewed-associative TLB, elaborating on some implementation issues.

5.1   Replacement choices

The advantages of the skewed-associative TLB come from the property that page addresses are hashed differently in each column of the TLB. This means that potential collisions, i.e. the hashing of several addresses to the same position, are likely to occur differently in the different columns. When two addresses A and B collide (i.e. are hashed to the same entry) in one column (which we choose to call column 0), they may not collide in another column (which we may call column 1). If A is stored in column 0 and we also want to store B into the TLB, we can't just write it into column 0 without overwriting the entry for A. We have to overwrite one of the entries indexed by $(f_0(B), 0), (f_1(B), 1), (f_2(B), 2) \ldots (f_{n-1}(B), n-1)$ (where $f_0(B) = f_0(A)$), one in each column of the TLB, meaning that we may overwrite any of n entries (for a n-way skewed-associative TLB). If any of these entries are invalid, i.e. do not contain currently valid mappings from a virtual page addresses to a physical page addresses, we can store B without having to overwrite a valid entry. This is good, because it won't make the next miss happen sooner (since all valid entries currently in the TLB will be preserved). However, it affects the possible choices at later occasions where it might be necessary to overwrite a valid entry.

Overwriting a valid entry can cause a miss to occur if the corresponding page is accessed again soon. Choosing a valid entry to overwrite will also affect future replacements, since one would later be likely to prefer another entry to overwrite than the one recently overwritten, which at that time contains information about a recently accessed page. In a normal n-way set-associative TLB, the

"is-possible-replacement-for"-relation is an equivalence relation, which can be broken down into equivalence sets of virtual addresses (the placement sets), each such set associated with one "row" of the n-way TLB of size $m \times n$, when thought of as a table of n columns and m rows. Thus, the TLB actually works as m parallel fully-associative "sub-TLBs", each one operating on a certain subset of the virtual address space, disjoint from the subsets operated on by the other sub-TLBs (as the banked TLB design proposed by Manne, Klauser, Grunwald and Somenzi [MKGS97]). A bad replacement decision in one sub-TLB can only affect the future hits and misses of that sub-TLB[7].

In the skewed-associative TLB, overwriting a valid entry is different. The entry to be overwritten could also be overwritten with entries with different sets of possible placements. This suggests that it could be desirable to be able to later revise and redo replacements. Since we cannot go back in time and change what has already happened, nor skip ahead in time to see what is going to happen (not at a reasonable cost, anyway), this would mean trying to change the state to one that would have been a result of past choices that, by reasonable heuristics, appear to be better than the ones actually made. This would mean reorganising the contents of the TLB to allow for better replacement choices.

When some entry is to be inserted, it may be the case that a reorganisation of the TLB makes it possible to find a more suitable replacement than would otherwise be found at that occasion. This could be done again when other similar situations later occur, which suggests a TLB where the contents are moved around dynamically to give better choices for what to overwrite. Being able to make better choices for overwriting would roughly be the same as having more to choose from when overwriting, i.e. a higher degree of associativity, so we should expect this n-way associative TLB to, in most cases, behave comparable to a k-way set-associative TLB, for some $k > n$ (not necessarily an integer). If this would be reasonably cheap and simple to implement in a real computer system, it could combine the advantages of a TLB of low associativity (speed and simplicity for lookups) with the advantages of a high-associativity TLB (low miss rate and a stable behaviour, i.e. a low risk for an occasional bad working set to cause lots of conflict misses).

5.2 Simple possibilities of reorganisation

Consider a 2-way TLB and the situation that the page address A can be placed at one of the locations α, β and that the page address B can be placed at at α or γ (fig. 6). It is possible to keep them both in the TLB, if (A, B) are placed in (α, γ), (β, α) or (β, γ), respectively. Assume that they are placed in, for example, (α, γ), and that the next TLB miss that occurs is for the page address C with α and γ as valid placements. If overwriting an old entry is all that can be done, then either A or B must be overwritten, which could have been avoided if a different placement had been chosen for (A, B). When A and B was written into the TLB, it was not known that C would be accessed soon. However, now that this is observed, there could still be a possibility to find a better placement for (A, B) and keep them both, instead overwriting some other (valid or invalid) entry. Moving A from α to β and then writing C into α would result in (A, B, C)

---

[7]As a simple example, a 8-way set-associative TLB with 16 entries, using the lowermost bit in the page part of each address for placement, is logically equivalent to two separate fully-associative TLBs of size 8, one for "even pages" and one for "odd pages".

Before reorganisation:
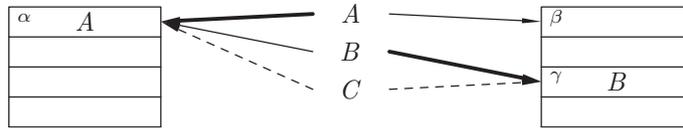
After reorganisation and insertion:
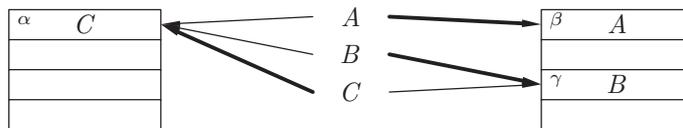
Figure 6: A simple example of collision avoidance by reorganisation in a 2-way skewed-associative TLB: The TLB contains the entries A and B, and the entry C is to be inserted. All these entries share a possible placement in the left column, here occupied by A. Normally, we would be forced to overwrite either A or B to insert C; by relocating A to its other possible placement, this can be avoided, as shown in the lower picture.

Before reorganisation:
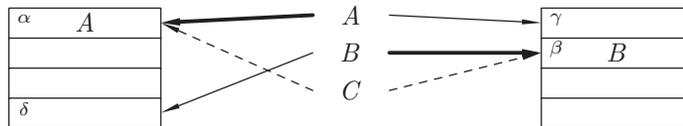
After reorganisation and insertion:
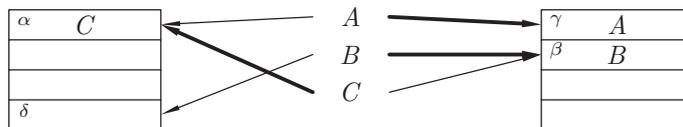
Figure 7: More possibilities for reorganisation in a 2-way skewed-associative TLB: The TLB already contains the entries A and B, and the entry C is to be inserted. This can be done by overwriting either A or B. However, there is another possible placement for A — if A is moved there, then C can be written into the previous location of A (as in the lower picture). Similarly, B could be moved to its other possible placement, yielding yet another replacement possibility.

being stored in (β, γ, α), respectively, respecting placement restrictions for these addresses. The cost for saving all three is overwriting the (probably valid) entry at β, but if that entry has been accessed less recently than A or B, this is clearly preferable by the LRU criterion. This very simple reorganisation gives an extra possibility for replacement, which is what one would have expected from a 3-way TLB of some sort. This is an example of how a dynamically reorganised n-way skewed TLB may exhibit greater apparent associativity than n.

Let us consider the case of choosing an entry to overwrite more closely: We may for example have an address A that can be placed in α in column 0 of the TLB or in γ in column 1, and an address B which can be placed in δ or β in column 0 and 1 respectively (fig. 7). Now assume that (A, B) are placed in (α, β), respectively, and that the next miss calls for the insertion of C into either α or β. Accepting that the data in all of $\alpha, \beta, \gamma, \delta$ can be overwritten, there are now actually four replacement choices, assuming that the four locations are distinct:

- Storing C in α, overwriting A

- Storing C in β, overwriting B

- Moving A to γ and then storing C in α, overwriting whatever was in γ

- Moving B to δ and then storing C in β, overwriting whatever was in δ

If α and δ are distinct, and if β and γ also are, then $\alpha, \beta, \gamma, \delta$ will be distinct (since α and δ are in column 0 of the TLB, whereas β and γ are in column 1), giving four possible choices for which entry to overwrite in order to make space for the new entry. In a reasonably large TLB, this situation is very probable, meaning that the TLB will likely resemble a 4-way TLB in behaviour.

5.3   Further reorganisation

The reasoning so far describes how a 2-way skewed-associative TLB quite often can be made to behave as a 4-way TLB at an insertion, by permitting relocation of the entries occupying the valid TLB locations for the new entry. Such a relocation causes another (often valid) entry to be overwritten. However, that entry will often in turn have another valid placement, and if this is different from the location where the new entry would be stored, this gives even more candidates for overwriting to choose among (fig. 8).

Consider a 2-way skewed-associative TLB where the entries in column 0 are labelled $\alpha, \beta, \gamma \ldots$, and the entries in column 1 are labelled by $\bar{\alpha}, \bar{\beta}, \bar{\gamma} \ldots$, and where the next page address to be stored can be placed at α or $\bar{\alpha}$. The addresses there (if any) can be moved to $\bar{\beta}$ and β, respectively, to avoid overwriting them when storing the new address into α or $\bar{\alpha}$. The addresses there will of course also have other possible placements γ and $\bar{\gamma}$, respectively, so if the address in $\bar{\beta}$ is moved to γ and the address in α to $\bar{\beta}$, then the new address can be stored in α, at the cost of removing the contents of γ from the TLB. Alternatively, the contents of β could be moved to $\bar{\gamma}$ to make place for moving the address in $\bar{\alpha}$ to β and store the new address in $\bar{\alpha}$, only sacrificing the contents of $\bar{\gamma}$. This could be carried yet one step further, by moving the address in $\bar{\gamma}$ to its other possible location before overwriting it, and so on. We can call these sequences of moves *push paths*, since addresses are pushed onto other locations one after another as

one would push around crates and boxes in a very full warehouse or cellar; to gain access to this box, that box has to move there, but in order to get it over there, this nearby box has to get out of the way, and so forth. In this example, all push paths are prefixes of these sequences:

$$\alpha \longrightarrow \bar{\beta} \longrightarrow \gamma \longrightarrow \bar{\delta} \longrightarrow \epsilon \longrightarrow \cdots$$

$$\bar{\alpha} \longrightarrow \beta \longrightarrow \bar{\gamma} \longrightarrow \delta \longrightarrow \bar{\epsilon} \longrightarrow \cdots$$

where α and $\bar{\alpha}$ are the possible placements for the new entry, and every successor in the sequences is another valid placement for the address located in its predecessor. These sequences are considered to end where an invalid entry appears or where they start to loop; for a TLB of finite size, this must eventually happen. (For multiple-step reorganising models, care should be taken to avoid paths that loop.) With a good hashing function for the skewed-associative TLB, the addresses should be effectively evenly distributed over the TLB, which makes it less probable that this will happen very early in the sequence. Two such sequences may join into one; it may, for example, be the case that $\epsilon = \beta$. In a case where this holds, and $\bar{\gamma} = \bar{\epsilon}$, the sequences would look like:

$$\alpha \longrightarrow \bar{\beta} \longrightarrow \gamma \longrightarrow \bar{\delta} \longrightarrow \beta \longrightarrow \bar{\gamma} \longrightarrow \delta$$

$$\bar{\alpha} \longrightarrow \beta \longrightarrow \bar{\gamma} \longrightarrow \delta$$

Either α or $\bar{\alpha}$ could be chosen for storing the new address. Also, a victim for overwriting has to be selected from a sequence containing that entry; if possible an invalid entry should be picked, since such a selection would not cause any valid entry to be overwritten, but such a choice is rarely available. For each location in the sequence, from the beginning of the sequence up to and including the predecessor to the location of the victim, its contents are moved to its successor in the sequence, so that the contents of the sequence are pushed one step down towards the victim. This will preserve the consistency of the TLB, since, by the definition of the sequence, each successor of a location is another possible placement for its contents. (This property probably doesn't hold anymore after the push (so push paths are generally not "reusable"), but consistency will be preserved since the push won't cause an entry to be written to a location that isn't a possible placement for it.) Finally, the new entry will be written to the location starting the sequence, which also is a possible placement for it by the definition. Thus, by this method:

- Any entry that is stored in a location appearing in such a sequence can be chosen for overwriting when a new entry is to be stored, so that the TLB is consistent and contains the new entry afterwards, and that no entry that was in the TLB before the operation will be lost, except for the one chosen for overwriting (the victim).

- If any location containing an invalid entry appears in such a sequence, the new address can be stored in the TLB while keeping the rest of the addresses in the TLB and preserving consistency.

In a 2-way TLB, there will be exactly two sequences as those described above, since a new entry can be placed at exactly two locations (the starting points of the sequences), and because each entry in the TLB has 2 possible placements;

Before reorganisation:
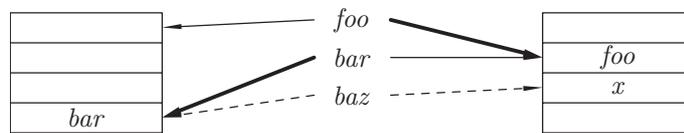
After one-step reorganisation and insertion (sacrificing foo*):*
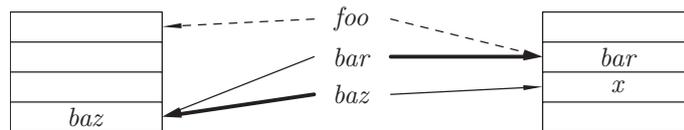
After two-step reorganisation:
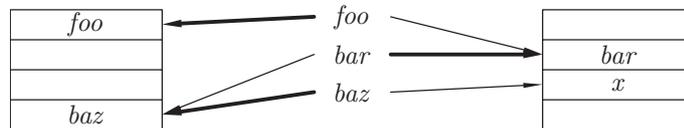
Figure 8: More replacement choices by multiple-step reorganisation: Directly storing *baz* in the TLB means overwriting either *bar* or *x*. One possibility to keep both *x* and *bar* is to move *bar* out of the way, overwriting *foo*. However, *bar* can be moved to the former location of *foo* without overwriting *foo* by first moving *foo*. In this manner, the number of replacement choices can be further increased by reorganisation in multiple steps.

the entry where it is stored and another entry that becomes the single successor to the location where it is stored (which can't be the same entry, since the two possible placements are in different columns of the TLB). So few sequences means that a relatively large search depth is needed to get a decent number of replacement choices, and if the paths happen to end quickly due to loops the number of choices will be low in any case. The gains of reorganisation can therefore be expected to be more limited for 2-way models than for 3-way models and up.

In an n-way skewed-associative TLB, $n > 2$, addresses have more possible placements than these two. This makes the sequences form a forest of n $(n-1)$-ary trees, and causes the number of possible push paths to grow exponentially, although limited by the size of the TLB. This provides a good picture of how complex it may be to find the optimal way to reorganise a n-way $(n > 2)$ skewed-associative TLB, and, on the other hand, how much greater the possibilities of doing a close-to-optimal replacement may be.

5.4   Reorganisation decisions

To store an entry at one of its possible placements, without having to remove another entry at one of those locations from the TLB, one those entries must be moved elsewhere. This means storing it at another of its possible placements, so something there has to be removed or overwritten, and so forth. However, each of these moves will consistute another step in one of the sequences just discussed. This informal reasoning suggests that any TLB reorganisation work that would be immediately rewarding follows a push path, i.e. a prefix of one of these sequences. Doing anything in addition to the reorganisation described by that push path could affect later reorganisations and can therefore not be excluded as completely useless; but the time for performing the reorganisation is limited in a real application, so any unnecessary or not immediately rewarding work should probably be avoided. It is also not obvious how to know in advance which reorganisations will be useful for storing address mappings that we do not yet even know that we will use soon.

This strongly suggests that the problem of performing efficient reorganisation is indeed the problem of quickly finding a good push path. The push path that yields the least recently used (LRU) victim, or the shortest of those push paths if there are more than one, can probably be considered the best choice or at least a very good choice, if one accepts that LRU is the generally most reasonable replacement strategy in a TLB. Deciding which push path to use could of course be done using another heuristic than LRU or pseudo-LRU, possibly also taking into account how the relocation of other addresses along the path could simplify further reorganisations. However, since the sets of possible placements for future insertions likely will appear to be random (which is the very point in using reasonably good hash functions), the possibilities for such an approach look less promising.

5.5   Selecting a replacement

Two properties are of obvious importance concerning how good a push path is; path length, which should be as short as possible, and how acceptable it is that the victim is replaced (which by the LRU criterion depends on how recently

the victim was accessed). The pseudo-random placement possibilities of the skewed-associative TLB makes it hard to direct the search towards finding a good victim candidate, so it seems reasonable to try short push paths in the first place. Then one of those is picked, for example according to the LRU criterion, possibly also taking path length into consideration.

Breadth-first search through the forest that is the union of all possible push paths would find shorter paths before longer ones. However, breadth-first search algorithms tend to require more state. A cheaper approach in terms of memory would be to perform a depth-first search down to some maximal depth, or some kind of compromise between these two methods. A suitable search depth could be decided based on the expected time it would take to do the needed page table lookups. The most simple solution would be to just pick an appropriate constant search depth. It should be feasible to test and compare a set of push paths that only differ in the last step in parallel, as an optimisation.

In the simple case of a 2-way skewed-associative TLB, the forest of possible push paths contains only two sequences (1-ary trees). A breadth-first search is far from costly in that special case, and might be regarded as the obvious choice.

5.6 Maximal and typical reorganisations

Some observations on the wins and costs of additional reorganisation steps can be done if the mean reorganisation depth is logged and compared to the maximal search depth. As it is possible that several reorganisation "push paths" lead to the same results, this adds another consideration for the replacement strategy; it might be appropriate to prefer the shortest path of several that yield the same victim. If this is done, the mean path length of the reorganisations performed (rounded up) will show how deep we typically need to search to find the optimal replacement at the set maximal search depth.

5.7 Non-blocking reorganisation

The reorganisation of a TLB, done by simply following a push path, will never move an address to a disallowed placement for that address, not even temporarily. This means that if a reorganisation of a skewed-associative TLB would be interrupted for a lookup, anything in the TLB at that moment could still be found.

If lookups can be done in the TLB while reorganising, then some reorganisation work could be done while code is being executed. It might only be feasible to move one single entry at a time, but the TLB can be designed so that all of the post-fill contents of the TLB may still be found at any time while reorganisation is performed, by doing a lookup in all columns of the TLB in parallel with a lookup in a buffer used for temporary storage during reorganisation. Since a miss normally is followed by a number of hits (often to the same page that caused the miss), there are several cycles to gain here by being able to continue executing code during the fill; by some prefetching mechanism, by dynamical reordering of instructions or by simultaneous handling of multiple threads of execution.

A TLB miss in a reorganising TLB of this kind leads to a TLB update operation consisting of two phases; deciding on a push path and then reorganising

according to that push path. A reasonable way of finding a possible push path is by starting at the possible placements of the virtual page address to insert and then search by following the sequences given by alternative placements, as has previously been argued[8]

Because of this, and because it isn't possible to know how good a push path is before knowing where it ends (which decides its victim) it would probably be very hard to perform these two phases in parallel. If the search depth is not limited, then the only certain limit to the search space is the size of the TLB, so the first phase could take almost arbitrarily long; at least this is true for n-way skewed-associative TLBs ($n > 2$), since there will very probably be many sequences from which to select push paths. At some point, the search phase might need to be aborted even though it is far from certain that the best possible push path has been found, so that the actual reorganisation can commence.

Preferrably, so much time would be allocated to searching for a replacement that reorganisation can be completed just before the next miss occurs, or possibly somewhat later if this would allow for a much better replacement. It will nearly always be virtually impossible to know the remaining number of cycles until the next miss, so the search phase could be aborted after a fixed number of cycles even if it is not known whether the best push path within the search space has been found. Alternatively, the maximal number of cycles for this could be adjusted dynamically depending on an observed typical number of cycles between misses, but since miss intervals vary much, this probably doesn't pay.

The search for push paths will likely be much more expensive than the actual reorganisation. While a replacement policy could be designed to prefer shorter paths, the gains of this approach seem rather small.

One argument for letting the processor do some reorganising during page table lookups is that this time is wasted anyway until the lookup is complete, so it could as well be used for something useful. It is becoming more common for processors to have some built-in support for managing several parallel instruction streams, so that the processor can just shift to another of these streams when one is stalled because of some memory operation — such as a page table lookup [EJK$^+$96]. This means that these cycles need not be wasted after all, so one could argue that this would reduce the gains of reorganising while performing lookups. However, the working set used by a processor executing multiple threads at once in this manner will likely be larger than if only one thread were executed, further stressing the need for an efficient TLB. One could also imagine a TLB reorganisation pseudo-thread executing virtually parallel to these threads, largely building on resources that are already present in such a processor.

---

[8]It might be possible to do some optimisations by deciding on suitable victims and searching backwards from those in advance, but it would probably be too expensive to do this on each TLB miss. It is not obvious how to efficiently keep that information up to date through reorganisations, which is required if it is to be kept and reused for future TLB fills.

6 A simple method for associativity comparisons

6.1 Parameters for TLB comparisons

When comparing the miss rate of two TLB:s for some sequence of virtual page accesses, there are several parameters to take into account. Associativity models may have different constraints that affect the range of available choices for TLB parameters.

6.1.1 Sizes

A larger TLB will generally behave better than a smaller one. This is not always the case; a TLB of a different size may need a somewhat different hash function, because the number of sets of TLB locations that page addresses map to may be different. For example, a direct-mapped TLB of size 48 and one of size 64 cannot use the same hash function, because the latter needs a hash function with a larger range (see 3.8, pg. 14). This means that the choice of size affects not only the number of capacity misses (misses because the TLB simply isn't large enough), but also the number of conflict misses (misses because too many page addresses map onto the same location or set of locations).

6.1.2 Hash functions

Parameters such as size and associativity model determine the range of the hash functions used, but not the hash functions themselves. The number of virtual pages potentially used in a system are magnitudes larger than the number of TLB locations, so there are obviously many different classes of hash functions that could be used.

   A simple way to create hash functions with different ranges and of about the same quality, is to start with a hash function with a large range and then map it down to the desired range by a applying the modulo operation. One obvious argument against this is that such a method would hardly be used in practice for most sizes, since computing modulo some number is generally too expensive. This might be seen as a moot point, since most sizes within the interesting range of sizes would hardly be used anyway, just because of the difficulty of designing evenly distributing hash functions with the appropriate ranges that are very simple and fast to compute.

   One important note here is that this method can be expected to favour hash functions with odd range cardinalities somewhat, especially with prime range cardinalities, since this may yield a smaller probability for accesses in common access patterns to map onto the same location, i.e. it could decrease the risk of conflict misses. With this in mind, we can use this method for easily testing cache/TLB behaviour at many different sizes. We can expect this unfairness to be less pronounced when comparing TLBs of the same sizes with each other, since the range cardinalities of their hash functions will tend to have common divisors.

6.1.3 Replacement policies

Comparing replacement policies is rather off-topic for this thesis. Nevertheless, we should keep in mind that the although a replacement policy can be

chosen largely independently from the associativity model used, some associativity models will gain more from a good replacement policy than others. In a fully-associative TLB, the replacement policy may replace any entry without affecting the others in any way, but in a direct-mapped TLB, there is always just one possible replacement choice — clearly, there is more room for a clever replacement policy to make wise (or unwise) replacement decisions in the former case.

LRU yields good results and is simple to use and understand, and is therefore believed to be a good replacement policy for comparing different associativity models and sizes of TLB:s. However, other policies that sometimes perform better than LRU have been suggested, most notably for disk and memory caches.

One can argue that it might be easier to implement genuine LRU or a good pseudo-LRU for a set-associative model than for a skewed-associative one — on the other hand, a set-associative model will often need to be designed with higher associativity than a skewed-associative model achieving the same performance, and this decreases this advantage.

6.1.4 Associativity models

A choice of associativity model (with parameters for the model, such as the choice of n for an n-way associative TLB), typically restricts the possible choices of size (for example: only multiples of n allowed) and hash functions (by deciding their required range together with the choice of size). This is, to some extent, an obstacle when comparing associativity models, as we want to test them under as fair conditions as possible. A few combinations are rather simple to test:

- A *direct-mapped* associativity model is simple to compare to any other model, as its size can be chosen freely. There is one possible placement for each page address, so replacement strategy is not an issue; however, the placement function, in this case a function hashing each page address to one single TLB location, will generally need to be more computationally complex for some sizes than for others to give an even distribution of page addresses — modulo will do, but is often too expensive to be a realistic implementation choice, except for sizes that are powers of two (where this function can be easily implemented by just picking the lowermost bits of the page address).

- A *fully-associative* model can very easily be compared to any other model. Its size can be chosen freely, and there is no hash function to adapt to the size; the possible placements are always the same for each page address.

- For each n: *n-way set-associative* TLBs, as well as *n-way skewed-associative* (with or without reorganisation) can easily be compared to each other, as they have the same size constraints. Given a replacement policy, all that has to be done is to choose appropriate hash functions.

6.2 Influence of TLB size on testing

Miss rates will generally depend a lot on TLB size (fig. 9). A fairly typical pattern known from memory caches is that the miss rate stays high for small sizes, then slopes down as the size is increased past a level where a reasonably
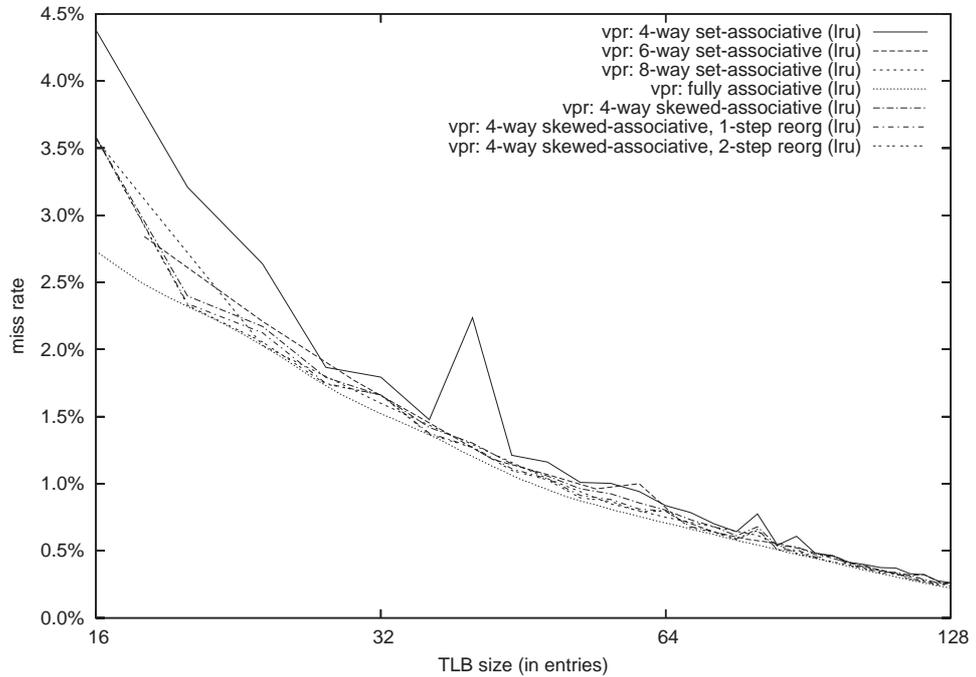
Figure 9: Comparing TLB models: Generally, a larger TLB will have a lower miss rate regardless of the associativity model; however, models rarely have entirely stable behaviour, so the decrease in miss rate is most often not monotonic. The differences between models of the same size are often most noticeable at the extremes — when the TLB is severely under-dimensioned, or when the TLB is so large that the capacity misses are few.

large part of the working set begins to fit in the cache, and then levels out where almost all of the working set fits in the cache; beyond this point, further size increases don't make as much difference. However, there is usually still some improvement as conflict misses also tend to get rarer with increased size [Hil88].

This pattern is not as often and as easily observed for TLB:s, likely because an entry represents an entire memory page, consisting of many cache lines. This tends to give rather different space locality patterns. Also, this larger granularity makes this curve (miss rate vs. total number of entries) very noisy, and often there is no such easily discernible slope at all. Still, we want to be able to compare the "general performance" of different associativity models without resorting to only comparing them for a few fixed sizes, an exercise that doesn't give any simple answer to the question of how good one model generally is as compared to another model. One might argue that this simplifies the issue too much, but one might always go back to comparing for specific sizes, to confirm the observations done with such a method.

6.2.1 Selecting reference models

To test how good an associativity model generally performs at a range of sizes, we will compare it to two reference models; a *direct-mapped* model and a *fully-*

34

associative model. The reasons for choosing these are several:

- The models are simple and well understood.

- They can be of any size, as opposed to n-way associative TLB:s that generally are required to have a size that is a multiple of n. Therefore, we can use them for comparisons at any TLB size.

- They can be seen as representing a "worst alternative" and a "best alternative", respectively. The *direct-mapped* model leaves nothing to the replacement strategy by only permitting one single replacement choice for each case, making it very sensitive to collision misses but thereby also detecting access patterns that are more likely to yield conflict misses in other models of about the same size. On the other hand, the *fully-associative* model gives the replacement strategy full freedom in choosing a replacement, thereby exploiting the TLB size as much as possible. Given a replacement strategy that is close to optimal, this should also give a result that is close to optimal for that TLB size.

One basic observation is that the expected patterns, such as 3-way associative TLBs performing worse than 4-way, do not always hold in all cases. If we regard the associativity models basically as limitations on how well we can follow a replacement strategy, then we can see this as a symptom of an imperfect replacement strategy. A perfect replacement strategy is in most cases unrealistic, as it would need to be able to see into the future, so we generally have to live with this.[9] However, if we view our measurements as testing how successful TLB model is at implementing a replacement strategy, then this makes perfect sense. If we assume that our replacement strategy is good, although not necessarily optimal, then these observations can be seen as reasonably good measurements of the overall goodness of the TLB model. The same methodology could of course be used with other replacement models as well.

6.2.2   A method for comparison: Collision tendency

In the following we present the method we have been using for scoring (comparing) TLB models:

- For each size within the range of sizes, we measure the miss rates of the *direct-mapped* and *fully-associative* reference models. We will view these as a "worst case" and a "best case" effort, respectively, for implementing the chosen replacement strategy at these sizes.

- For all possible TLB sizes within the range (or at least for a reasonably large and evenly distributed set of these sizes), miss rates for the other models are measured. (As mentioned above, this will usually not be possible for all sizes. In those cases, only possible sizes within the range are used.)

---

[9]It would actually be possible to simulate an optimal replacement policy, since having a trace makes it possible to "look into the future". While even more unrealistic for real implementations, this might be a better choice for comparing associativity models.

- The miss rate of the corresponding (same TLB size) sample for the *fully-associative* reference model is subtracted from each sample. This largely removes the influence of capacity misses from the measurement results. As we want to compare results for different models at fixed sizes, capacity misses are not interesting since they, by definition, are strictly size dependent.

- For each sample: If the fully-associative reference model performed equal or worse than the direct-mapped reference model (which very rarely happens), the result for this sample is considered undefined, and the sample is discarded[10]. Otherwise, the resulting value is scaled linearly by the difference between the two reference models, so that the fully-associative model would always yield a result of 0 and the direct-mapped always a result of 1.

- The scaled samples are clipped upwards at 1 and downwards at 0. Occasionally it happens that a model performs worse than the direct-mapped reference model. When this happens, it is a result of a not entirely optimal replacement strategy; we do not really want to punish a model for this. (For example, the direct-mapped reference model will now and then perform very well at some specific size for some specific trace.) Similarly, by clipping downwards at 0 we avoid rewarding (or punishing, if for example a quadratic mean is used) a model for occasionally performing better than the fully associative reference model — another artifact of a suboptimal replacement strategy.

- For each model, an average of the adjusted samples is computed, summarising the performance of the model in one single scalar value. Computing the harmonic mean is often not possible as it is undefined for values of zero, but arithmetic mean as well as quadratic can be used. (Arithmetic mean has been used in this thesis.) The resulting averages will be values in the range 0 to 1, somewhat crudely summarising the performance of the TLB model, within the chosen range of sizes, in one single value.

For each model, we get a value in the $0 \ldots 1$ range, summarising how sensitive the model is to conflict misses.

7 Simulation setup

7.1 Collecting data

Virtual address traces are all that is needed to simulate a single-page-size TLB in enough detail to determine typical miss rates. The corresponding physical addresses are of interest to a real TLB implementation, but since miss rates are all that we are interested in, there is no reason to collect physical addresses; all that matters is which virtual page addresses are accessed, not what they would map to in a real virtual memory system. Also, the presence of misses should

---

[10]Keeping those rare cases where the fully-associative reference model performs equal to the direct-mapped one would yield a division by zero when scaling. If it actually performs worse, then the scaling would give a model better results for worse performance, and the other way around, which doesn't seem right.

not be expected to affect execution significantly, apart from lookups that might be needed when finding page table entries. These rarely constitute a significant part of the memory accesses.

A traditional method to get data for TLB simulations is to modify hardware to intercept virtual page addresses [AKB85] [CE85]. Other methods that have been used are microcode modifications [AHH88] and not collecting traces at all but instead modifying a software TLB miss handler to simulate and collect statistics for a TLB model [NUS$^+$93] [Tal95]. In these experiments, traces have instead been collected using the tracing facilities of the Virtutech Simics system level simulator [MDG$^+$98], simulating a single-processor Sparc v9 system running Sun Solaris 7 (in 64-bit mode), with virtually no load except for the benchmarks run one by one. No context information have been collected, so the traces may contain fragments of exception handlers and other tasks; however, the traces ought to be sufficient for their purpose, namely, to represent realistic memory usage patterns for TLB testing and evaluation.

Within the simulator, run on a otherwise fairly lightly loaded Sun E450, the selected benchmarks have been run for about one minute (to skip initialisation of data structures and the like), and after that traces have been recorded. The first 200000 data accesses in each of the traces have been used to "warm up" the simulated TLB models, assuming that the TLB starts out empty and that no invalidations occur. (Again, not perfectly realistic in the strictest sense, but reasonably good enough to simulate TLB behaviour for normal access patterns.) After this warm-up, miss rates has been measured for another 1000000 data accesses.

Benchmarks have been chosen to test the models with varying types of access patterns; some of these benchmarks are very sensitive to associativity, while others are less so [KS02]. It has been noted that multiprogrammed workloads may be preferable for stressing TLBs [AHH88]; for simplicity, that has not been done here.

7.2 Simulation and evaluation of TLB models

Several methods for optimising parallel simulation of several caches have been devised, such as *forest simulation* and *stack simulation* [HS89]. These methods could just as well be applied to TLBs, but both place rather strict constraints on the properties of the caches or TLBs to be simulated, so they would only be usable for certain subsets of the simulations done for this thesis. Instead, the models have simply been simulated in separate processes without any shared data or state.

7.3 Hash functions

For the skewed-associative models, hash functions generated by xor3 (see 3.11, pg. 18) have been used, as these seem to be of realistic complexity. For the n-way set-associative models, the identity function has been used for "hashing", as this most closely emulates the scheme used in common implementations of such models.

The hash values have been adapted to the desired range using a modulo operation. While in some cases not entirely realistic for a TLB implementation, this allows us to test and compare the modules for a multitude of sizes.

7.4 Replacement policy

To compare the different associativity models, an LRU replacement policy has been used. The object of the comparisons is not to test specific TLB implementations, rather to specifically compare associativity models; thus LRU was chosen, being a simple and well-understood replacement policy. An alternative would have been to choose some common pseudo-LRU variant, or some enhanced variant; however, there are many such, and evaluating and comparing replacement policies is outside the scope of this thesis (see 6.1.3, pg. 33).

7.5 Benchmarking and comparison of TLB models

The different TLB models have been compared using the collision tendency metric described in 6.2.2 (p.35). Traces were collected from parts of the of the SPEC2000 [KS02] benchmark using the Simics system level simulator [MDG$^+$98] simulating a single-processor UltraSPARC (Sparc v9) system running Sun Solaris 7, and the simulations were performed on the virtual addresses of data accesses, assuming a single page size of 8k.

The reason for choosing this method for comparison, instead of directly comparing TLB models for certain sizes, is that such comparisons are generally not possible to do except for a few specific sizes. For example, if a 5-way associative model is to be compared to a 7-way associative model of the same size, only nonzero sizes that are both divisible by 5 and 7 are possible to compare; i.e. only sizes divisible by 35. If more models are tossed in for comparison, there may be even harder restrictions on for which sizes all of the models can be compared. The method intends to be a best effort to compare multiple associativity models without these strict size restrictions.

8 Simulation results

8.1 General observations

The goal of the simulations done for this paper is twofold; to show what can be achieved in theory by reorganisation of a skewed-associative TLB, and what can reasonably be achieved in practice. A practical implementation would most probably need to limit the search depth, and choose a reasonably simple hash function; the simulations confirm that good results can be achieved even using simple hash functions and shallow search depths.

The theoretical observation (see 5.3, pg. 27) that TLB reorganisation in an n-way skewed-associative TLB should pay off more for $n > 2$ than for $n = 2$ is confirmed by the simulations.

The reorganising TLB may in some cases perform approximately as well as a fully-associative TLB, and usually exhibits miss rates closer to those of a normal skewed-associative TLB of doubled or quadrupled associativity, compared to how it performs without any reorganisation. Another gain from reorganisation is stability in performance; the reorganising TLB will suffer less from unfortunate working sets that would normally cause conflict misses.

As the number of possible replacement choices increases with the reorganisation search depth, the miss rate also decreases due to better replacements. A search depth of one may make a huge improvement over the corresponding

reorgs → ↓ cols ↓	0	1	2	3	4	5
2	2…3	2…4	3…4	3…4	3…4	3…4
3	5…8	7…10	8…10	8…10	8…10	8…10
4	4…10	7…16	8…16	8…	8…	8…
5	7…16	8…	10…	10…	16…	10…

Figure 10: A skewed-associative model can not be determined to generally correspond to an n-way set-associative model for some specific value of n, since results vary considerably between different traces. Also, although performance for n-way set-associative models typically increases as n gets larger, this increase is very often nonmonotonic. Thus, the resulting measurements become rather noisy. One way to determine a "typical" associativity range that a model corresponds to is that, for each trace used, determine the largest a and the smallest b in the set of associativities used for comparison, such that such that no tested n-way set-associative model, $n \leq a$, performs better than the skewed-associative model, and that no tested n-way set-associative model, $n \geq b$, performs worse, according to the collision tendency metric. Discarding the lowest resulting value of a and the highest resulting value of b removes most of the noise, and the remaining lowest and highest values of a and b, respectively, give a rough picture of what range of values of n that the skewed-associative model can be said to typically correspond to. In this way, the results of this section can more or less be summarised in the table above.

non-reorganising model, and sometimes further searching will hardly lead to any improvement. In other cases, searching a few additional steps makes a noticeable difference, most notably for 3-way skewed-associative TLBs as they have fairly low associativity by themselves but still offer much more opportunity for reorganisation than 2-way skewed-associative TLBs.

A greater degree of associativity will increase the number of entries found at a given, reasonably low search depth. This decreases the gain of searching deeper, and also makes it more expensive. This could be taken into consideration when deciding on a certain fixed search depth, which has its advantages since it allows for a simple depth-first search algorithm, and also gives a predictable maximal search time. It must also be noted that the gain in searching several steps for reorganisation depends much on the memory usage patterns of running processes; different applications may vary a lot in the influence of associativity on TLB miss rate [JM98] [KS02].

8.2   Reorganisation yields fewer collisions

For this thesis, skewed-associative and set-associative models have been evaluated using the collision tendency method described in 6.2.2 (p.35). It is obvious that traces with different access patterns yield different results, and there appears to be no direct equivalence in performance between n-way set-associative and m-way skewed-associative models (with or without reorganisation) for certain values of n and m. However, some tendencies are clearly observable (see fig. 11, 12, 13, 14). A rough comparison is given by fig. 10.

Typical patterns in collision tendency are:

- Non-reorganising 2-way skewed-associative models are largely comparable to 3-way set-associative models. One or two steps of reorganisation makes a clearly visible difference, but generally does not correspond to an additional level of associativity.

- Non-reorganising 3-way skewed-associative models are comparable to 5-way to 8-way set-associative models. Adding one step of reorganisation gives an improvement corresponding to yet a few steps of associativity. A second step gives a clearly visible but not as large improvement.

- 4-way skewed-associative models resemble 4-way to 10-way set-associative models without reorganisation, one step of reorganisation giving performance roughly corresponding to 7-way to 16-way set-associative models. Additional reorganisation steps do not make much of a difference here.

- 5-way skewed-associative models without reorganisation correspond to 7-way set-associative models and above, typically closer to 16-way. One step of reorganisation gives results often surpassing 16-way set-associative models.

A general observation is that reorganisation can not always make up for low associativity. 2-way skewed-associative models are harder to improve by reorganisation because there is only one alternative placement for each entry; reorganisation still yields clear improvements, but 2-way models typically need several steps of reorganisation to get anywhere near 3-way skewed-associative models, if they ever do.

For 3-way models, one step of reorganisation gives a clear improvement; a second step usually also does, but not always. 3-way skewed-associative models with one step of reorganisation resemble 4-way non-reorganising models. For 4-way models and above, the gains of multiple-step reorganisation tend to be low; however, a single step of reorganisation still has a clearly noticeable effect.

For comparison, the miss rates for a TLB size of 120 entries are shown in fig. 15, 16, 17 and 18; the size 120 was chosen for this example as it permits many different degrees of associativity.

8.3 Associativity and reorganisation tradeoffs

One very appealing property of skewed-associative models with reorganisation is that the common case — a hit — is very fast; in addition to the lookup time of a n-way set-associative model, there is just the cost of computing the hash functions in parallel. To keep the common case fast and not have the rest of the processor wait for the TLB, very simple hash functions should be chosen.

The costs in terms of chip area and energy consumption are also lower than that of n-way set-associative models with similar miss rates (see 9.7, pg. 57). Reorganisation may make TLB fills more expensive; however, this additional cost would only occur at TLB fills, and TLB fills are much rarer than TLB lookups. A lower miss rate means that fewer misses occur and thus fewer fills need to be performed. Therefore, the total cost for TLB misses may well decrease even if fills become more expensive.

The costs of an implementation depend on several factors, such as:

- Size: Increased size means increased chip area.

- Associativity: The chip area needed for hashing and comparison logic can be expected to be largely proportional to the associativity of the model, as is the energy consumption of those parts. Also, a small improvement in associativity in skewed models generally corresponds to a much larger increase in a set-associative model with similar properties. Instead of increasing associativity in a set-associative model, the same low miss rates may be achieved by changing to a reorganising skewed-associative model, perhaps of an even lower associativity. The chip area saved could possibly be used to increase the size of the TLB instead, also reducing capacity misses.

Given the results of the tests described here, a suitable tradeoff between simplicity and complexity might be to go for a 3-way skewed-associative TLB with two steps of reorganisation, or a 4-way or 5-way skewed-associative TLB with one step of reorganisation.
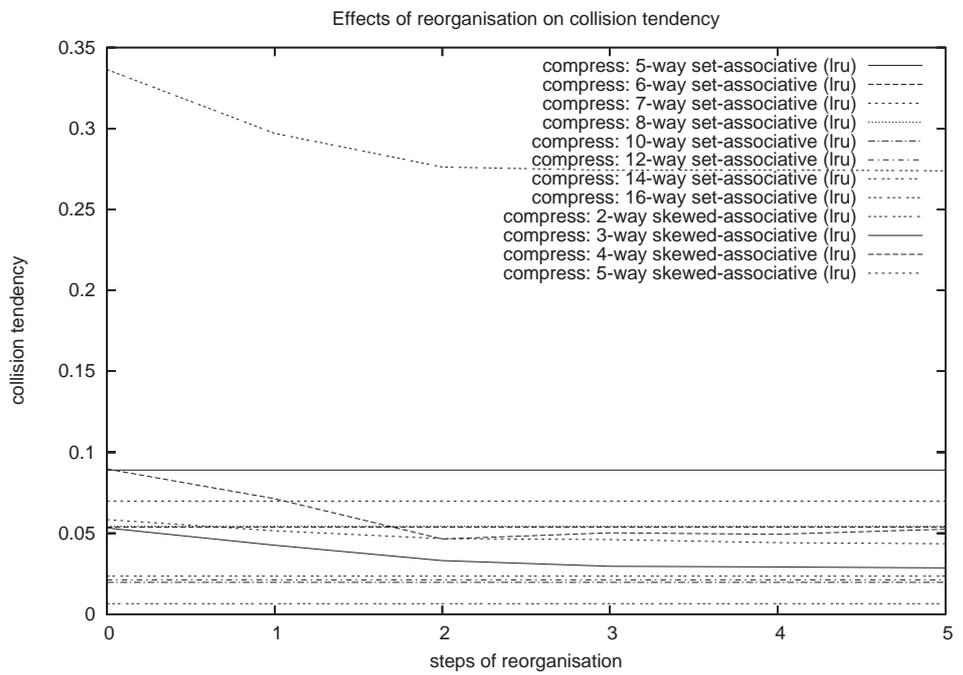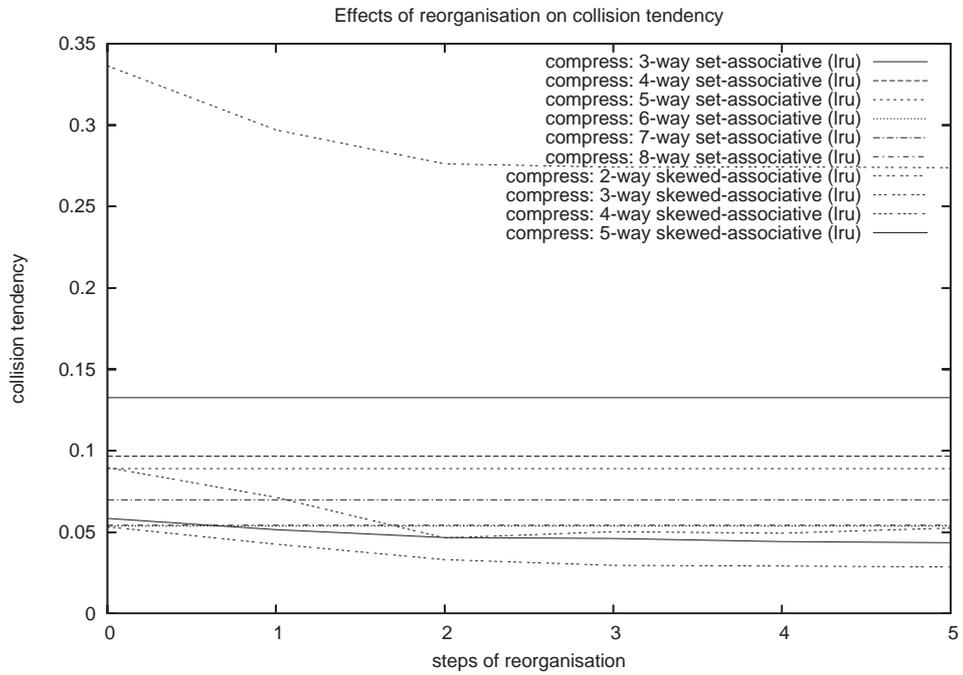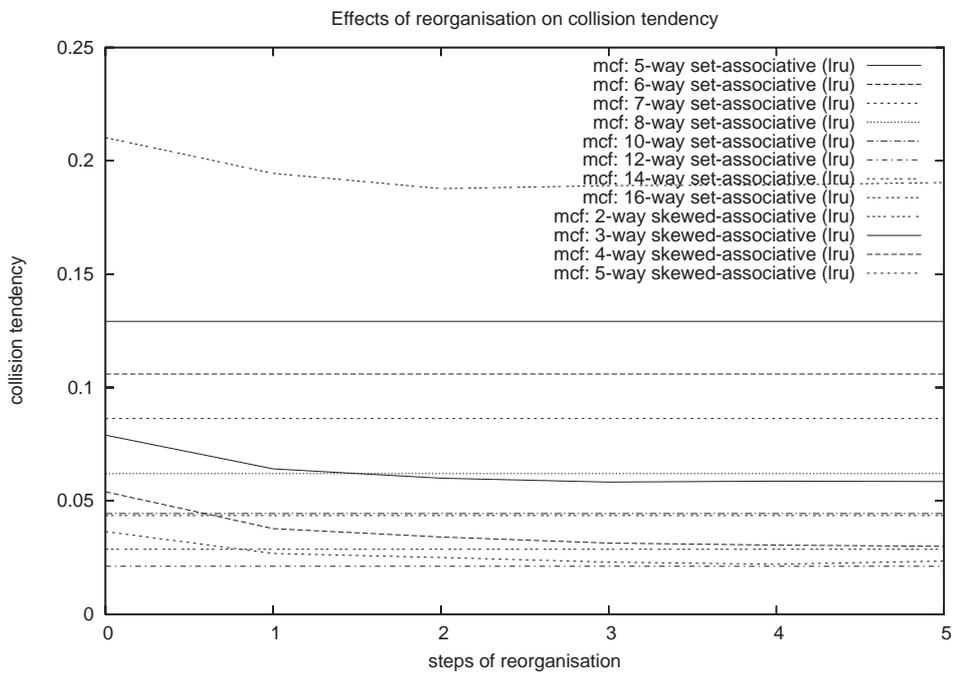
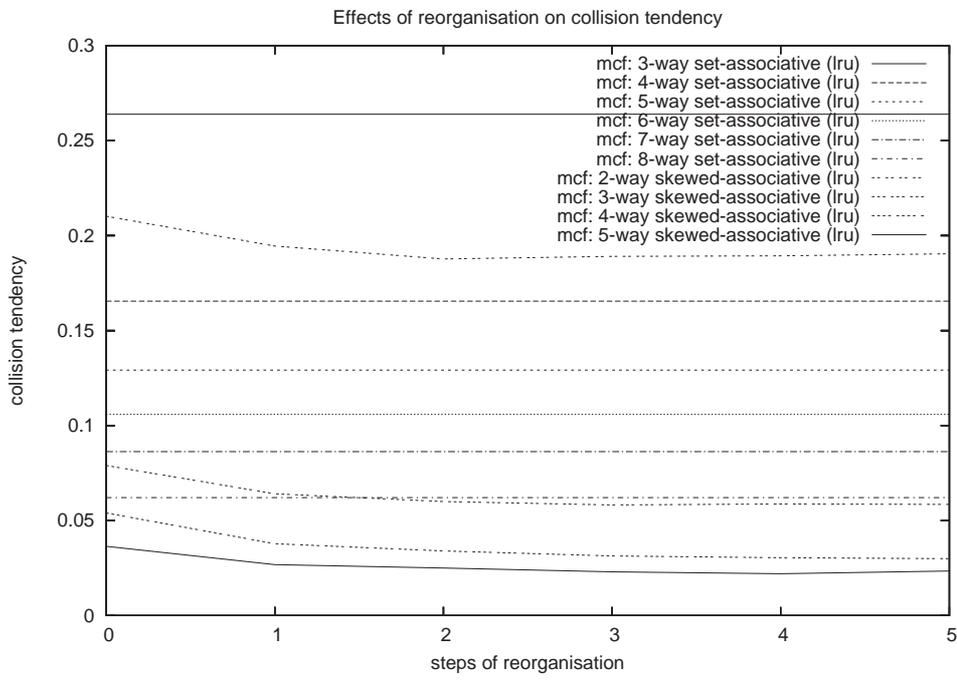Figure 11: Improvement by reorganisation for *compress*

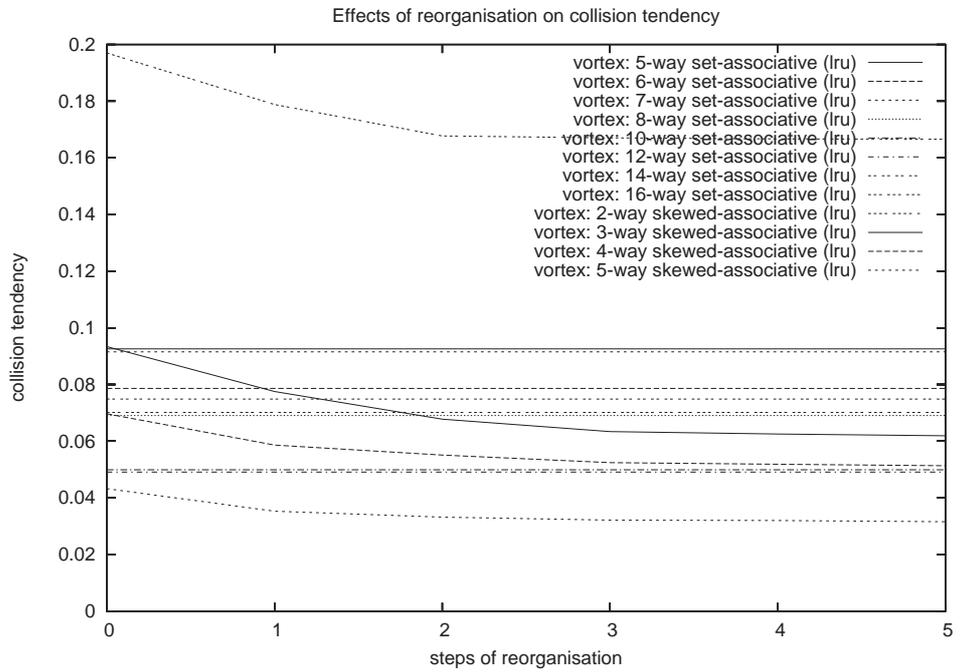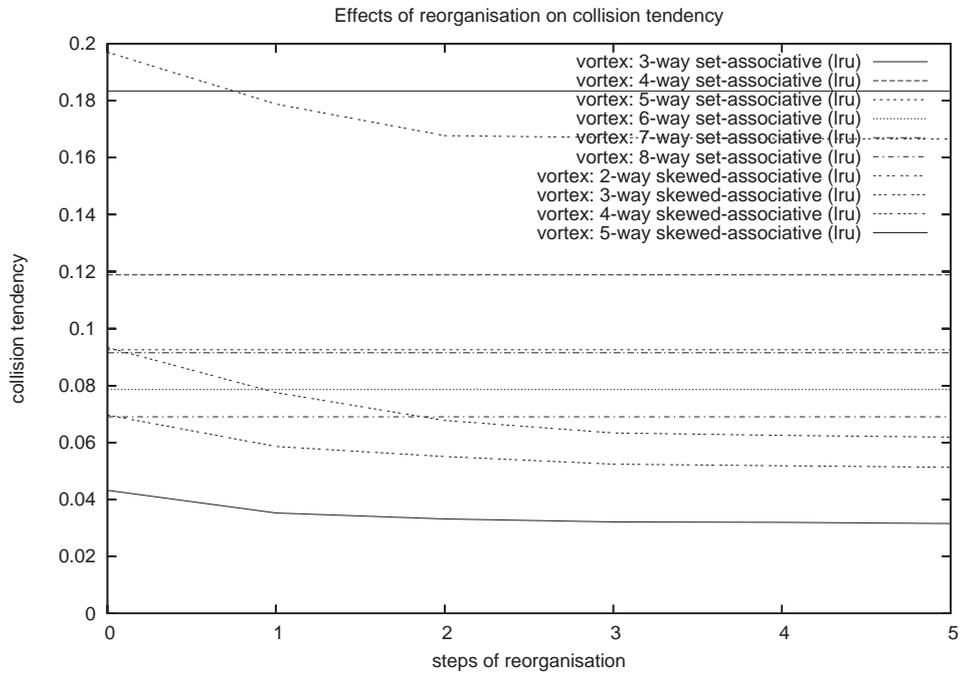Figure 12: Improvement by reorganisation for *mcf*

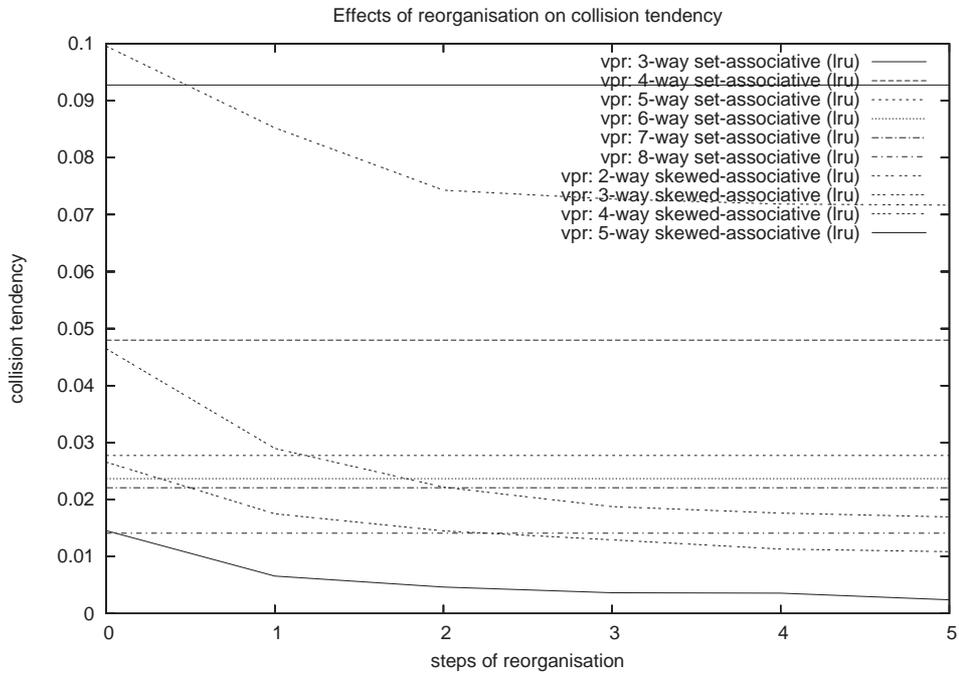Figure 13: Improvement by reorganisation for *vortex*
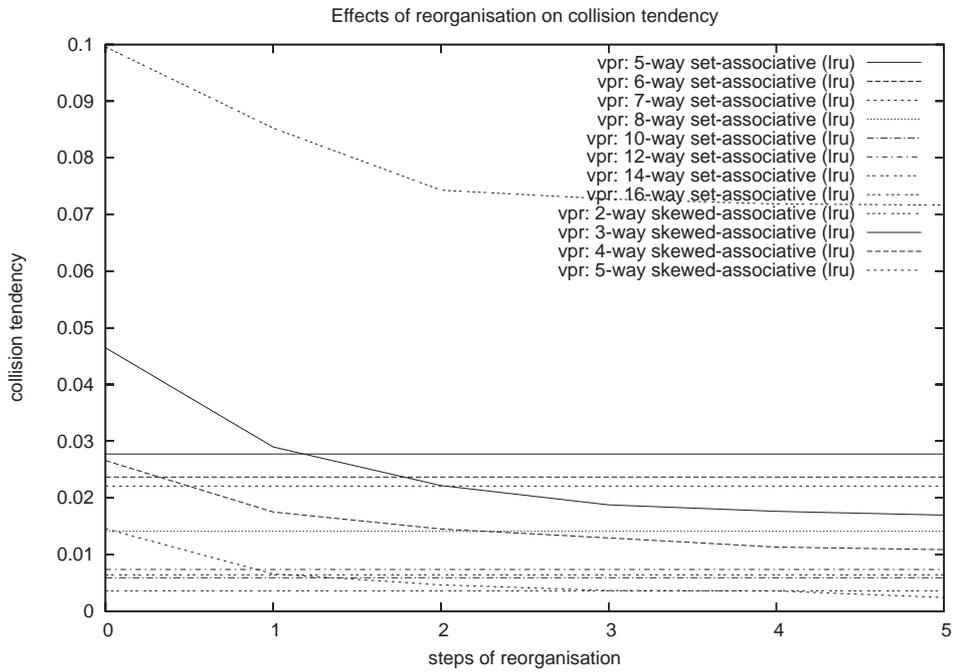
Figure 14: Improvement by reorganisation for *vpr*

compress: fully associative (lru)	0.022 %
compress: 5-way skewed-associative, 4-step reorg (lru)	0.026 %
compress: 5-way skewed-associative, 5-step reorg (lru)	0.026 %
compress: 5-way skewed-associative, 3-step reorg (lru)	0.027 %
compress: 10-way set-associative (lru)	0.027 %
compress: 5-way skewed-associative, 2-step reorg (lru)	0.027 %
compress: 5-way skewed-associative, 1-step reorg (lru)	0.028 %
compress: 12-way set-associative (lru)	0.028 %
compress: 5-way set-associative (lru)	0.029 %
compress: 5-way skewed-associative (lru)	0.029 %
compress: 4-way skewed-associative, 2-step reorg (lru)	0.030 %
compress: 4-way skewed-associative, 4-step reorg (lru)	0.030 %
compress: 4-way skewed-associative, 5-step reorg (lru)	0.031 %
compress: 4-way skewed-associative, 3-step reorg (lru)	0.031 %
compress: 6-way set-associative (lru)	0.032 %
compress: 4-way skewed-associative, 1-step reorg (lru)	0.032 %
compress: 8-way set-associative (lru)	0.032 %
compress: 3-way skewed-associative, 5-step reorg (lru)	0.032 %
compress: 3-way skewed-associative, 4-step reorg (lru)	0.032 %
compress: 3-way skewed-associative, 3-step reorg (lru)	0.033 %
compress: 3-way skewed-associative, 1-step reorg (lru)	0.033 %
compress: 3-way skewed-associative, 2-step reorg (lru)	0.033 %
compress: 4-way skewed-associative (lru)	0.033 %
compress: 3-way skewed-associative (lru)	0.036 %
compress: 3-way set-associative (lru)	0.037 %
compress: 2-way skewed-associative, 4-step reorg (lru)	0.037 %
compress: 2-way skewed-associative, 5-step reorg (lru)	0.038 %
compress: 2-way skewed-associative, 3-step reorg (lru)	0.038 %
compress: 2-way skewed-associative, 2-step reorg (lru)	0.038 %
compress: 2-way skewed-associative, 1-step reorg (lru)	0.039 %
compress: 2-way skewed-associative (lru)	0.041 %
compress: 4-way set-associative (lru)	0.044 %
compress: 2-way set-associative (lru)	0.048 %
compress: direct mapped (lru)	0.064 %

Figure 15: Miss rates for *compress* at 120 entries

mcf: fully associative (lru)	0.159 %
mcf: 4-way skewed-associative, 4-step reorg (lru)	0.162 %
mcf: 4-way skewed-associative, 5-step reorg (lru)	0.163 %
mcf: 4-way skewed-associative, 3-step reorg (lru)	0.164 %
mcf: 4-way skewed-associative, 2-step reorg (lru)	0.166 %
mcf: 12-way set-associative (lru)	0.168 %
mcf: 5-way skewed-associative, 5-step reorg (lru)	0.168 %
mcf: 5-way skewed-associative, 4-step reorg (lru)	0.169 %
mcf: 5-way skewed-associative, 3-step reorg (lru)	0.170 %
mcf: 4-way skewed-associative, 1-step reorg (lru)	0.171 %
mcf: 5-way skewed-associative, 2-step reorg (lru)	0.171 %
mcf: 10-way set-associative (lru)	0.174 %
mcf: 5-way skewed-associative, 1-step reorg (lru)	0.177 %
mcf: 5-way skewed-associative (lru)	0.180 %
mcf: 4-way skewed-associative (lru)	0.187 %
mcf: 3-way skewed-associative, 5-step reorg (lru)	0.209 %
mcf: 3-way skewed-associative, 4-step reorg (lru)	0.211 %
mcf: 3-way skewed-associative, 3-step reorg (lru)	0.212 %
mcf: 3-way skewed-associative, 2-step reorg (lru)	0.225 %
mcf: 3-way skewed-associative, 1-step reorg (lru)	0.231 %
mcf: 3-way skewed-associative (lru)	0.247 %
mcf: 8-way set-associative (lru)	0.262 %
mcf: 2-way skewed-associative, 5-step reorg (lru)	0.270 %
mcf: 2-way skewed-associative, 4-step reorg (lru)	0.270 %
mcf: 2-way skewed-associative, 3-step reorg (lru)	0.270 %
mcf: 2-way skewed-associative, 2-step reorg (lru)	0.272 %
mcf: 5-way set-associative (lru)	0.292 %
mcf: 4-way set-associative (lru)	0.305 %
mcf: 2-way skewed-associative, 1-step reorg (lru)	0.315 %
mcf: 6-way set-associative (lru)	0.341 %
mcf: 2-way skewed-associative (lru)	0.455 %
mcf: 3-way set-associative (lru)	0.654 %
mcf: 2-way set-associative (lru)	0.789 %
mcf: direct mapped (lru)	1.715 %

Figure 16: Miss rates for *mcf* at 120 entries

vortex: fully associative (lru)	0.326 %
vortex: 4-way skewed-associative, 5-step reorg (lru)	0.345 %
vortex: 4-way skewed-associative, 4-step reorg (lru)	0.345 %
vortex: 12-way set-associative (lru)	0.347 %
vortex: 4-way skewed-associative, 3-step reorg (lru)	0.348 %
vortex: 4-way skewed-associative, 2-step reorg (lru)	0.348 %
vortex: 4-way skewed-associative, 1-step reorg (lru)	0.353 %
vortex: 8-way set-associative (lru)	0.354 %
vortex: 4-way skewed-associative (lru)	0.367 %
vortex: 10-way set-associative (lru)	0.380 %
vortex: 4-way set-associative (lru)	0.397 %
vortex: 6-way set-associative (lru)	0.413 %
vortex: 2-way skewed-associative, 4-step reorg (lru)	0.427 %
vortex: 2-way skewed-associative, 5-step reorg (lru)	0.428 %
vortex: 2-way skewed-associative, 3-step reorg (lru)	0.428 %
vortex: 2-way skewed-associative, 2-step reorg (lru)	0.428 %
vortex: 2-way skewed-associative, 1-step reorg (lru)	0.436 %
vortex: 2-way skewed-associative (lru)	0.442 %
vortex: 3-way skewed-associative, 4-step reorg (lru)	0.467 %
vortex: 3-way skewed-associative, 5-step reorg (lru)	0.467 %
vortex: 3-way skewed-associative, 3-step reorg (lru)	0.471 %
vortex: 3-way skewed-associative, 2-step reorg (lru)	0.475 %
vortex: 5-way skewed-associative (lru)	0.479 %
vortex: 5-way skewed-associative, 5-step reorg (lru)	0.482 %
vortex: 3-way skewed-associative, 1-step reorg (lru)	0.482 %
vortex: 5-way skewed-associative, 4-step reorg (lru)	0.484 %
vortex: 5-way skewed-associative, 2-step reorg (lru)	0.486 %
vortex: 5-way skewed-associative, 3-step reorg (lru)	0.487 %
vortex: 3-way skewed-associative (lru)	0.488 %
vortex: 5-way skewed-associative, 1-step reorg (lru)	0.489 %
vortex: 2-way set-associative (lru)	0.516 %
vortex: 3-way set-associative (lru)	0.535 %
vortex: 5-way set-associative (lru)	0.552 %
vortex: direct mapped (lru)	1.775 %

Figure 17: Miss rates for *vortex* at 120 entries

vpr: fully associative (lru)	0.260 %
vpr: 5-way skewed-associative, 5-step reorg (lru)	0.265 %
vpr: 4-way skewed-associative, 5-step reorg (lru)	0.266 %
vpr: 10-way set-associative (lru)	0.267 %
vpr: 5-way skewed-associative, 3-step reorg (lru)	0.267 %
vpr: 4-way skewed-associative, 4-step reorg (lru)	0.267 %
vpr: 5-way skewed-associative, 4-step reorg (lru)	0.269 %
vpr: 4-way skewed-associative, 3-step reorg (lru)	0.269 %
vpr: 4-way skewed-associative, 2-step reorg (lru)	0.270 %
vpr: 4-way skewed-associative, 1-step reorg (lru)	0.271 %
vpr: 5-way skewed-associative, 1-step reorg (lru)	0.271 %
vpr: 5-way skewed-associative, 2-step reorg (lru)	0.272 %
vpr: 12-way set-associative (lru)	0.273 %
vpr: 5-way skewed-associative (lru)	0.277 %
vpr: 8-way set-associative (lru)	0.280 %
vpr: 4-way skewed-associative (lru)	0.285 %
vpr: 5-way set-associative (lru)	0.288 %
vpr: 4-way set-associative (lru)	0.323 %
vpr: 6-way set-associative (lru)	0.323 %
vpr: 2-way skewed-associative, 4-step reorg (lru)	0.336 %
vpr: 2-way skewed-associative, 5-step reorg (lru)	0.337 %
vpr: 2-way skewed-associative, 3-step reorg (lru)	0.338 %
vpr: 2-way skewed-associative, 2-step reorg (lru)	0.338 %
vpr: 3-way skewed-associative, 4-step reorg (lru)	0.358 %
vpr: 3-way skewed-associative, 5-step reorg (lru)	0.358 %
vpr: 3-way skewed-associative, 3-step reorg (lru)	0.359 %
vpr: 3-way skewed-associative, 2-step reorg (lru)	0.359 %
vpr: 3-way skewed-associative, 1-step reorg (lru)	0.361 %
vpr: 3-way skewed-associative (lru)	0.368 %
vpr: 3-way set-associative (lru)	0.388 %
vpr: 2-way skewed-associative (lru)	0.395 %
vpr: 2-way skewed-associative, 1-step reorg (lru)	0.427 %
vpr: 2-way set-associative (lru)	0.498 %
vpr: direct mapped (lru)	1.492 %

Figure 18: Miss rates for *vpr* at 120 entries

9 Adapting to real time applications

9.1 Real-time TLB requirements

Some systems, as well as applications and tasks in a computer system, have certain well-defined time constraints for performing their operations. These are called *real-time (RT) requirements*, and thus we talk about *real-time systems*, *real-time applications*, and *real-time tasks*. Real-time tasks and requirements may be *hard*, meaning that the time constraints are critical for correct operation, or *soft*, in which case it is considered acceptable to occasionally fail in satisfying the real-time constraints of a task. Typical examples of soft real-time applications are interactive games and multimedia players; among hard real-time applications we find vehicle control and security systems and industrial control systems, where failure to meet a deadline can incur huge material costs and sometimes even be deadly.

A typical workstation or server system often runs some real-time tasks (usually of the soft variety) together with some tasks that do not have similarly strict timing requirements. If the time allowed for performing a real-time task is not too short, it may be enough if the operating system gives the task a higher priority than less time-critical simultaneously active tasks. In other cases, it might be important to avoid cache and TLB misses by making sure that the cache always contains the memory that the real-time task needs to access, and that the TLB constantly maps the pages used by the real-time tasks[11]. Trivially, this requires that all the pages used by the real-time task (we'll simply call these *real-time pages*) can fit within the TLB[12]. Furthermore any single additional page that will be accessed by other tasks running on the system, including the operating system itself, must also fit in the TLB together with the real-time pages; otherwise we would not be able to run these other tasks on the system at all while keeping the real-time pages in the TLB. To get decent performance even for non-real-time tasks, the set of TLB locations not occupied by real-time page entries must be adequate for the remaining tasks in the system.

There are similar issues with memory caches; these have been subject to more research, since memory caches are far more common than TLBs in real-time systems. There are techniques to keep down the number of entries that need to be locked to satisfy real-time requirements [VLX03]; these are probably to a large extent applicable to TLBs as well.

9.2 Reserving resources for real-time tasks

A simple way to ensure that real-time tasks get the TLB resources they need would be to use an entirely separate address space with its own TLB for real-time tasks only; another way would be to reserve a number of entries for real-time

---

[11]Sometimes certain entries used by an operating system are locked in the TLB [SDS00], because they are expected to be referenced very often or because they are absolutely necessary to have there; a typical example is a software TLB miss handler, which must have its pages mapped in the TLB to prevent an infinite chain of TLB misses when the miss handler is called. (Another way to solve this case is to let the miss handler reside in unmapped address space, where memory can be accessed without TLB lookups [JM98].)

[12]If the real-time constraints are less strict, this requirement may possibly be relaxed, although this complicates proving that the real-time constraints will be satisfied. Note also that some pages used by the operating system itself may need to be classified as real-time pages to make sure that the operating system provides timely service to real-time tasks.

pages and make sure that other page entries are never placed there. However, methods like these reserve a certain amount of TLB capacity whether needed or not, and if that capacity is not needed by any real-time tasks, it is not used for other tasks; it is just lost. Therefore, it is not only simpler but probably also more efficient to keep real-time page entries in the same TLB as other entries, and use a replacement policy that avoids overwriting real-time entries.

9.3   Real-time-conscious placement and replacement

We can modify any replacement strategy to prefer replacing non-real-time pages. This can be done by doing the following when selecting a replacement:

- If there are any possible replacements that cause an invalid entry to be overwritten, pick one of those replacements arbitrarily.

- Otherwise, if there are any possible replacements that overwrite non-real-time page entries, pick one of those according to some replacement strategy, such as LRU.

- Otherwise, pick a replacement arbitrarily. This will cause a real-time page entry to be overwritten, but there is no other choice.

As seen from this, we might be forced to overwrite real-time page entries. This easily becomes unavoidable for the direct-mapped case; since there is always just one replacement to pick, we must pick that replacement regardless of what entry it may cause to be overwritten. Even in higher-associativity models, collisions involving real-time page entries may severely restrict our freedom to select an acceptable victim, possibly to the extent that a real-time page entry will sometimes have to be sacrificed. This effect might be alleviated by distributing the real-time pages so that collisions involving those are minimised. An operating system with real-time functionality could have some knowledge about the TLB parameters of the system, and allow applications and system tasks to explicitly ask for pages and sets of pages chosen so that the risks for collisions between real-time pages, as well as between real-time pages and other pages (see fig. 19), are kept to a minimum. For example, in a n-way set-associative TLB of size $m \times n$, real-time pages may be picked from the m different placement sets in a round-robin fashion.

This policy, in itself, makes no difference between hard and soft real-time tasks. An operating system may make this distinction itself and prioritise keeping pages used by hard tasks whenever forced to overwrite a real-time page entry. One should in any case make sure to limit the set of hard real-time pages so that it becomes possible to keep this set in the TLB under all circumstances.

As associativity increases, the risk for collisions decreases. This also means that real-time pages might no longer need to be chosen as carefully. For the skewed-associative case, it is much less apparent (and probably much harder) to minimise the collision risks, but this is likely compensated by collisions being so much rarer — hopefully, the distribution of real-time pages in virtual address space does not affect the conflict miss rate noticeably. One advantage of this is that if a certain amount of real-time pages can be given real-time status arbitrarily, then there is no need to know in advance when a page is to be allocated whether it is to be used for real-time purposes or not; this also simplifies efficiently handling pages with a temporary or intermittent real-time status.
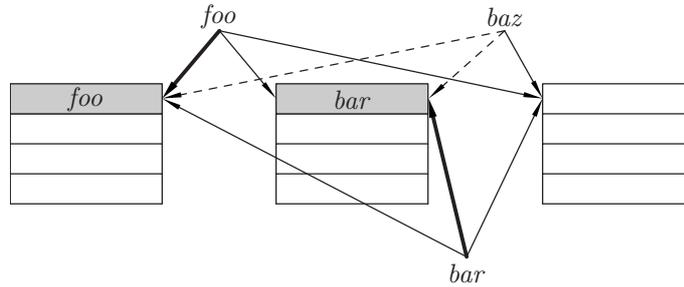
Figure 19: Collisions involving real-time entries: Consider a 3-way set-associative TLB where two real-time pages *foo* and *bar* (their locations shown greyed) share one placement set. For other page addresses that happen to map to the same placement set, there is only one single possible placement that does not overwrite a real-time page entry. As far as other page addresses that map to the same placement set (such as *baz* above) are concerned, the TLB will locally degenerate towards a direct-mapped behaviour. This effect can be decreased by clever allocation of real-time pages so that they rarely collide. If skewed associativity with reorganisation is used, then the risk of collision with multiple real-time page entries is significantly decreased. For soft-real-time applications, it may then be acceptable to allocate real-time pages arbitrarily.

9.4 Reliable handling of real-time page entries

For hard real-time applications, it will often be necessary to be able to determine in advance that a set of pages can be mapped by the TLB concurrently, together with any other page, so that we know that we can always avoid overwriting a real-time page entry at a TLB fill. Reorganisation complicates this somewhat.

Assuming an n-way reorganising skewed-associative TLB of $m \times n$ entries, we can make the following guarantees[13]:

- The TLB can reliably map any $(n-1)$ real-time pages together with any other page. The worst-case collision situation is that these pages collide in all columns; then, these $(n-1)$ pages will be mapped by the TLB using one entry in each of $(n-1)$ columns. Any additional page can be stored somewhere in the remaining column, regardless of how TLB contents are organised.

- With at least one step of reorganisation, the TLB can reliably map up to m real-time pages together with any other page, provided that these m pages are allocated so that they all map to different locations in at least one column of the TLB.

  Assume that up to m real-time pages map to different locations in one column C, and that the TLB has at least two columns. Any additional page a can be mapped by the TLB without overwriting one of these, since:

  – Assume that not all placements for a are occupied by real-time page entries. Then there is at least one placement for a that is not occupied

---

[13]Here, "real-time" and "non-real-time" are primarily to be interpreted as "hard real-time" and "non-real-time *or* soft real-time", respectively; i.e. by real-time pages, we mean pages that must necessarily be mapped by the TLB at all times.

by a real-time page entry, so an entry for a can be written there without overwriting any real-time page entry. This case requires no reorganisation.

– Assume that all placements for a are occupied by real-time page entries. Then any of these real-time page entries can be moved to column C without overwriting a real-time page entry — as all real-time page entries have different placements in column C, this will never overwrite another real-time page entry. This move frees a location where that a can be stored. This case requires one step of reorganisation.

Thus, it will always be possible to store any additional page in the TLB without overwriting any real-time page entry (without any further assumptions regarding the organisation of TLB contents); this will require at most to one step of reorganisation.

• The TLB can reliably map up to $m \cdot (n-1)$ real-time pages together with any other page, if one column C is reserved for non-real-time purposes and the real-time pages are allocated so that they can be concurrently stored in the remaining columns of the TLB.

Any TLB updating is then performed so that C never contains any real-time page entries. Any non-real-time page entry can then always be stored in column C without overwriting any real-time page entry.

For low values of n, this solution places severe restrictions on how TLB contents are organised, so it will likely lead to higher miss rates than the above solutions.

An alternative solution is to add a small extra buffer, that can hold one or a few non-real-time entries, to the TLB. It should be able to map any page, and could then be used as one or a few spare locations in the rare case that a fill for a non-real-time page cannot be satisfied by the main TLB without overwriting a real-time page entry. This buffer could be left unused until it is needed, or be used anyway to reduce conflict misses somewhat.

9.5 Real-time pages and reorganisation

Reorganisation makes replacement more complicated. It might well make replacement more expensive, but for most reasonable cases this should be well compensated for by a reduced miss rate — after all, the cheapest replacement is the replacement that doesn't happen.

In non-reorganising TLB designs, the real-time page entries are fixed. They are not moved around, and they are definitely not overwritten as long as that can be avoided. When reorganisation is introduced, nothing prevents us from moving these entries around, just like any other entries, as long as they are not lost. In a skewed-associative TLB design, two entries sharing a possible placement usually do not have exactly the same placement set (even though the sets overlap), so moving a real-time page entry from one location to another may lead to less collisions not only between real-time pages, but also between real-time and non-real-time pages. It is therefore advantageous to handle the

real-time page entries just like any other entries for all purposes including reorganisation, except that replacements overwriting real-time page entries will be avoided whenever possible. Again, this is a matter of choosing a real-time-conscious replacement strategy and not really an associativity or reorganisation issue.

So what makes real-time-conscious replacement strategies special when we introduce reorganisation? Let us consider real-time and non-real-time page entries as two different cases. Real-time page entries are expected to stay in the TLB until their real-time status is revoked. For real-time tasks, we often demand and expect a TLB miss rate of zero (as compulsory misses can be avoided by performing fills in advance). For non-real-time entries a non-zero miss rate is accepted and expected. From a non-real-time performance point of view, real-time page entries are obstacles. They are getting in the way and forcing us to deviate from our (hopefully) carefully chosen replacement policy, be it LRU or something else.

Simply put, reorganisation means that the TLB locations occupied by real-time page entries are still useful for non-real-time-purposes; as they are moved around, they enable more possible replacement choices when storing non-real-time page entries.

One possible enhancement to real-time-conscious replacement is to allow multiple real-time priorities, the simplest case being one hard and one soft real time level; a process that is only "slightly real-time" could have its pages preferred for keeping, while still allowing them to be thrown out when their locations are needed by higher-priority tasks, or when it is otherwise decided that the locations are better used for other purposes. Such extensions are not further studied here.

9.6 Results of reorganisation with real-time-locked page entries

Real-time page entries are locked in the TLB, in the sense that they are not to be replaced by other entries. We allow them to be reorganised just as any other entries, as long as we keep them in the TLB. This means that they often don't "get in the way" to the same extent as they would without reorganisation. To demonstrate this, a subset of the simulations described earlier in this thesis has been repeated, with the addition that the first n entries stored in a TLB are marked as real-time entries (and therefore never replaced unless there is no other choice)[14]. The value n has been set to a certain percentage of the TLB size. This has been repeated for a number of percentages.

The performance of a TLB with a certain percentage of its entries locked for real-time purposes varies rather much depending on the workload and the percentage of entries that are locked. However, it appears that the collision tendency of skewed-associative models in most cases are much less affected by the presence of arbitrarily locked page entries, except for the 2-way models, which remain sensitive to collisions (although clearly less so than the non-skewed

---

[14]Note that this means that the real-time percentage will not necessarily be fixed for each run — this is especially true for the low-associativity models, notably direct-mapped models. This mostly matters as the results of the direct-mapped reference model should be affected; however, this should disadvantage other models similarly, and have little effect on their relative performance.
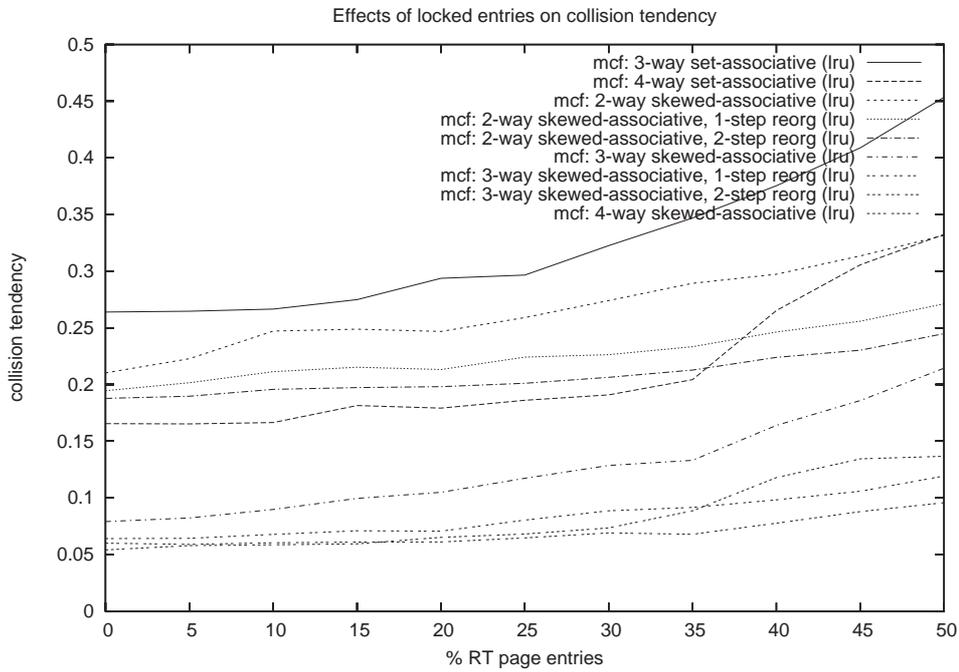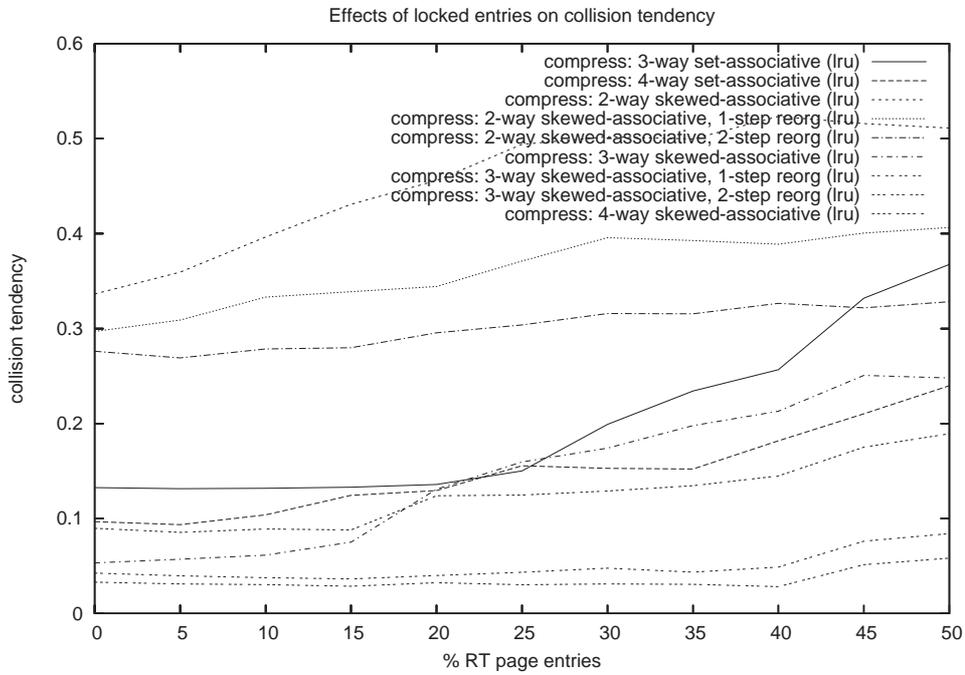
Figure 20: Reorganisation with locked entries: *compress* and *mcf*
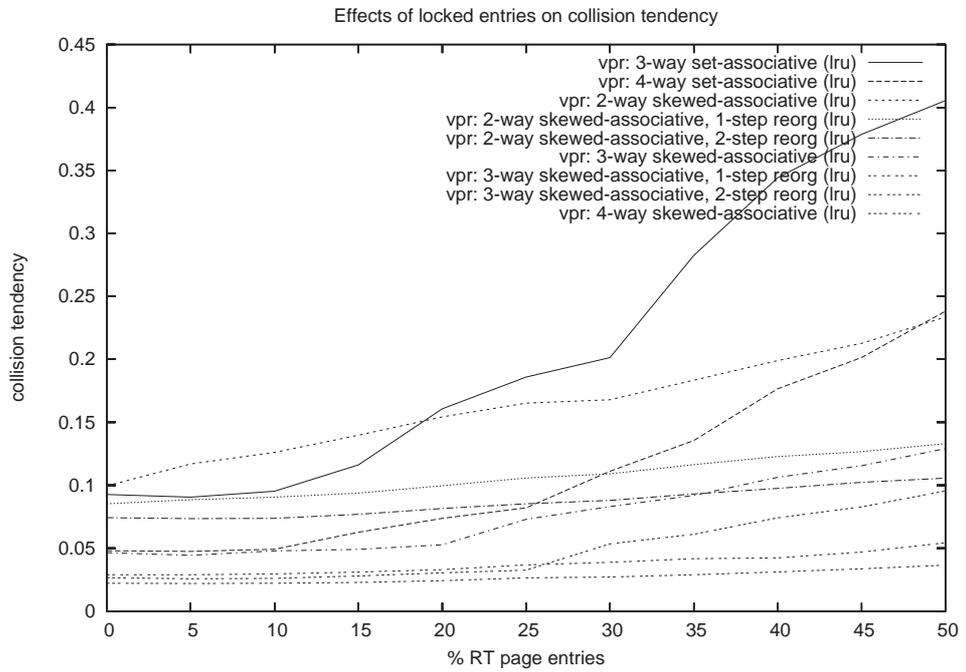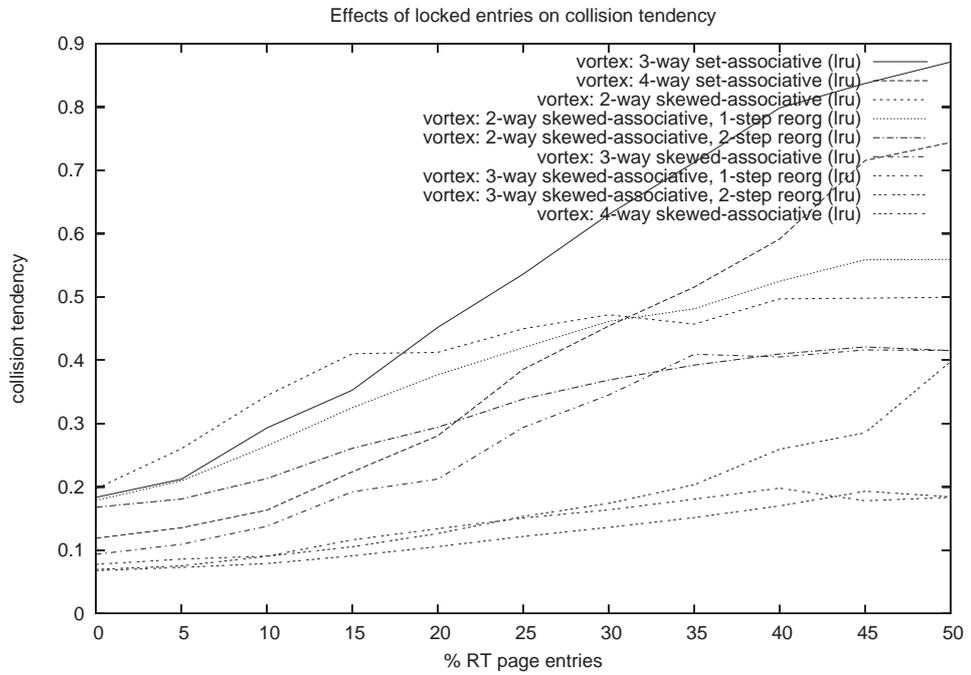
Figure 21: Reorganisation with locked entries: *vortex* and *vpr*

variants). The benefits of reorganisation typically remain even when a large portion of the TLB is occupied by locked entries. A 3-way or 4-way skewed-associative TLB typically has a very stable behaviour with a barely noticeable increase in collision tendency, even when so much as 30% or 40% of the TLB consists of locked entries (see fig .20, 21). With the addition of one or two steps of reorganisation, there is typically hardly any increase at all.

9.7 Performance and energy consumption in embedded applications

The TLB may be responsible for a noticeable part of the energy consumption of a modern processor with virtual memory; Juan, Lang and Navarro [JLN97] mention that in the in the StrongARM, a processor designed for low power applications, 17% of the total power consumption is due to the TLB.

Spjuth, Karlsson and Hagersten [SKH03] argue that reorganising cache models are more energy efficient than non-reorganising models, since a skewed model with reorganisation can achieve virtually the same low miss rates as a set-associative cache of higher associativity, that needs to do a larger number of comparisons in parallel for each lookup. The logic for these comparisons make for a large part of the energy consumption of the cache. This should be just as applicable to TLBs; Juan, Lang and Navarro [JLN97] show that this applies to reasonably sized TLBs.

Embedded applications very often have both real-time and power constraints. This strongly suggest that reorganising skewed-associative designs are very suitable for embedded systems; not only for memory caches, but also for TLBs in embedded systems where virtual memory is needed[15].

10   Conclusions

Skewed associativity is an efficient and low-cost technique to avoid conflicts and thereby decrease miss rates in TLBs. The technique gives good results even when very simple and fast to compute hash functions are used. This thesis shows that by reorganisation, performance can be improved further, in many cases well surpassing that of traditional set-associative models using twice as much parallelism or more; for example, a 3-way skewed-associative TLB with one step of reorganisation is comparable to a 7–10-way set-associative design of the same size.

Replacing traditional high-associativity TLBs with less parallel reorganising skewed-associative designs can thus not only give higher performance while lowering energy consumption through decreased parallelism; it is also cheap in terms of chip area.

Reorganising skewed-associative TLBs can be designed with various degrees of parallelism as well as with various numbers of reorganisation steps. This gives much flexibility in finding a compromise between cost and gains, making the design highly adaptable to different applications and constraints.

---

[15]Most embedded applications are more simple and do not require or use virtual memory. However, some do; these are likely to become more common, as embedded applications keep getting more advanced.

To simplify comparing a multitude of associativity models for a large range of TLB sizes, this thesis presents a methodology for testing and comparing conflict miss avoidance in TLBs. The methodology should also be useful for comparing models of other types of caches.

Reorganising skewed-associative TLBs are also shown to be suitable for real-time applications, notably where real-time and non-real-time tasks coexist in the same system. By permitting reorganisation of real-time and non-real-time page entries alike, conflict misses can still be kept down, giving a good performance for non-real-time tasks even when a large portion of the TLB is occupied by real-time page entries. As real-time constraints are very common in embedded systems, where a low energy consumption often is highly desirable, reorganising skewed-associative TLBs are an attractive choice for these applications.

11 Suggestions for future work

11.1 Further confirming results

The results in this thesis are preliminary in the sense that they do not simulate realistic combinations of associativity models and replacement strategies in more detail; the aim has been to evaluate associativity models and to avoid involving other parameters more than necessary. A good topic for a future paper would be to focus on a few realistic combinations of associativity models, replacement strategies and TLB sizes, and confirm these results, preferably using more and longer traces.

11.2 Asymmetric reorganisation

This thesis focuses on reorganisation between basically equal locations in one TLB. Reorganisation could also be used between different classes of locations in multi-level TLBs and other designs where locations have different speed or associativity properties.

It would be possible to do a variation on the old hash-rehash concept [AHH88] and design a semi-unified TLB (like a semi-unified cache [DS93]) where data entries can be moved to the instruction part of the TLB, and the other way around, on reorganisation. Lookups would then first be done in the "right" part of the TLB (the instruction part for instruction accesses, the data part for data accesses), and then, if it is not found there, in the "wrong" part. Finding an entry in the "wrong" part would still be faster than a TLB miss, and reorganisation could be done so that recently used entries are kept in the "right" part to a large extent. This could be advantageous, since one is no longer limited to a fixed instruction/data TLB reach ratio, and would in most cases still get the advantages of a split TLB; also, there are more possible placements for each address, and thus less conflict vulnerability. Both capacity and conflict misses are therefore likely to decrease, at the cost of a somewhat worse average lookup time.

11.3 More parallelism with less parallelism

Multiported high-associativity TLBs require much power and chip area, since the expensive highly-parallel lookup logic needs to be replicated to handle sev-

eral concurrent lookups. One interesting subject for future research is to devise reorganising skewed-associative designs that efficiently perform concurrent lookups with very little resources.

A   BibTeX entry

The following BibTeX entry is suggested for referring to this thesis in TeX/LaTeX documents:

```
@TechReport{     it:2004-027,
  author         = {Thorild Sel{\'e}n},
  title          = {Reorganisation in the Skewed-Associative {TLB}},
  institution    = "Department of Information Technology, Uppsala University",
  year           = 2004,
  number         = {2004-027},
  month          = sep,
  note           = {M.Sc. thesis.}
}
```

B   Acknowledgements

Thanks to Prof. Erik Hagersten for introducing me to this area of research and for the idea of applying skewed associativity to TLBs. More thanks to him and to Dr. Andreas Ermedahl for much valuable comments and support during my completion of this thesis.

References

[AHH88]   Anant Agarwal, John Hennessy, and Mark Horowitz. Cache performance of operating system and multiprogramming workloads. *ACM Trans. Comput. Syst.*, 6(4):393–431, 1988.

[AKB85]   Cedell A. Alexander, William M. Keshlear, and Faye Briggs. Translation buffer performance in a UNIX environment. *SIGARCH Comput. Archit. News*, 13(5):2–14, 1985.

[AP93]   Anant Agarwal and Stephen D. Pudar. Column-associative caches: a technique for reducing the miss rate of direct-mapped caches. In *Proceedings of the 20th annual international symposium on Computer architecture*, pages 179–190. ACM Press, 1993.

[BS94]   F. Bodin and A. Seznec. Cache organization influence on loop blocking. Technical Report 803, IRISA, Rennes, France, 1994.

[BS95]   François Bodin and André Seznec. Skewed associativity enhances performance predictability. In *Proceedings of the 22nd annual international symposium on Computer architecture*, pages 265–274. ACM Press, 1995.

[CE85]   Douglas W. Clark and Joel S. Emer. Performance of the VAX-11/780 translation buffer: simulation and measurement. *ACM Trans. Comput. Syst.*, 3(1):31–62, 1985.

[CGS97]   David E. Culler, Anoop Gupta, and Jaswinder Pal Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., 1997.

[DGJS93]   Nathalie Drach, Alain Gefflaut, Philippe Joubert, and André Seznec. About Cache Associativity in low-cost shared memory multi-microprocessors. Technical Report 760, IRISA, Rennes, France, 1993.

[DS93]   N. Drach and A. Seznec. Semi-Unified Caches. In *Proceedings of the International Conference on Parallel Processing, St Charles, Illinois*, 1993.

[EJK+96]   Richard J. Eickemeyer, Ross E. Johnson, Steven R. Kunkel, Mark S. Squillante, and Shiafun Liu. Evaluation of multithreaded uniprocessors for commercial application environments. In *Proceedings of the 23rd annual international symposium on Computer architecture*, pages 203–212. ACM Press, 1996.

[GVTP97]   Antonio González, Mateo Valero, Nigel Topham, and Joan M. Parcerisa. Eliminating cache conflict misses through XOR-based placement functions. In *Proceedings of the 11th international conference on Supercomputing*, pages 76–83. ACM Press, 1997.

[HH93]   Jerry Huck and Jim Hays. Architectural support for translation table management in large address space machines. In *Proceedings of the 20th annual international symposium on Computer architecture*, pages 39–50. ACM Press, 1993.

[Hil88]   Mark D. Hill. A Case for Direct-Mapped Caches. *Computer*, 21(12):25–40, 1988.

[HP96]   John L. Hennessy and David A. Patterson. *Computer architecture (2nd ed.): a quantitative approach.* Morgan Kaufmann Publishers Inc., 1996.

[HS89]   M. D. Hill and A. J. Smith. Evaluating Associativity in CPU Caches. *IEEE Trans. Comput.*, 38(12):1612–1630, 1989.

[JLN97]   Toni Juan, Tomas Lang, and Juan J. Navarro. Reducing TLB power requirements. In *Proceedings of the 1997 international symposium on Low power electronics and design*, pages 196–201. ACM Press, 1997.

[JM98]   Bruce L. Jacob and Trevor N. Mudge. A look at several memory management units, TLB-refill mechanisms, and page table organizations. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 295–306. ACM Press, 1998.

[Jou98]   Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache prefetch buffers. In *25 Years ISCA: Retrospectives and Reprints*, pages 388–397, 1998.

[KJLH89]   R. E. Kessler, R. Jooss, A. Lebeck, and M. D. Hill. Inexpensive implementations of set-associativity. In *Proceedings of the 16th annual international symposium on Computer architecture*, pages 131–139. ACM Press, 1989.

[KS02]   Gokul B. Kandiraju and Anand Sivasubramaniam. Characterizing the d-TLB behavior of SPEC CPU2000 benchmarks. In *Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 129–139. ACM Press, 2002.

[MDG+98]   Peter S. Magnusson, Fredrik Dahlgren, Håkan Grahn, Magnus Karlsson, Fredrik Larsson, Fredrik Lundholm, Andreas Moestedt, Jim Nilsson, Per Stenström, and Bengt Werner. SimICS/sun4m: A virtual workstation, 1998.

[Mic01]   Pierre Michaud. A Statistical Model of Skewed-Associativity. 7th International Symposium on High Performance Computer Architecture, Monterrey, Mexico, January 19-24, 2001.

[MKGS97]   S. Manne, A. Klauser, D. Grunwald, and F. Somenzi. Low power TLB Design for High Performance Microprocessors. Univ. of Colorado Technical Report, 1997.

[NUS+93]   David Nagle, Richard Uhlig, Tim Stanley, Stuart Sechrest, Trevor Mudge, and Richard Brown. Design tradeoffs for software-managed TLBs. In *Proceedings of the 20th annual international symposium on Computer architecture*, pages 27–38. ACM Press, 1993.

[SB93]   A. Seznec and F. Bodin. Skewed-associative caches. In *Proceedings of PARLE '93, Lecture Notes in Computer Science*, pages 305–316. Springer-Verlag, 1993.

[SDS00]   Ashley Saulsbury, Fredrik Dahlgren, and Per Stenström. Recency-based TLB preloading. In *Proceedings of the 27th annual international symposium on Computer architecture*, pages 117–127. ACM Press, 2000.

[Sez93]   André Seznec. A case for two-way skewed-associative caches. In *Proceedings of the 20th annual international symposium on Computer architecture*, pages 169–178. ACM Press, 1993.

[Sez97]   André Seznec. A New Case for Skewed-Associativity. Rapport de recherche N° 3208, INRIA, Rennes, France, 1997.

[Sez03]   André Seznec. Concurrent Support of Multiple Page Sizes On a Skewed Associative TLB. (to appear in IEEE Transactions on Computers), 2003.

[SKH03]   Mathias Spjuth, Martin Karlsson, and Erik Hagersten. The Elbow Cache: A Power-Efficient Alternative to Highly Associative Caches. Technical report 2003-046, Department of Information Technology, Uppsala University, Sweden, September 2003.

[Spj02]   Mathias Spjuth. Refinement and Evaluation of the Elbow Cache. Master's thesis, School of Engineering, Uppsala University, Sweden, April 2002.

[Tal95]   Madhusudhan Talluri. *Use of Superpages and Subblocking in the Address Translation Hierarchy*. PhD thesis, University of Wisconsin – Madison, 1995.

[TG99]   Nigel Topham and Antonio González. Randomized Cache Placement for Eliminating Conflicts. *IEEE Trans. Comput.*, 48(2):185–192, 1999.

[TGG97]   Nigel Topham, Antonio González, and José González. The design and performance of a conflict-avoiding cache. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 71–80. IEEE Computer Society, 1997.

[TH94]   Madhusudhan Talluri and Mark D. Hill. Surpassing the tlb performance of superpages with less operating system support. In *Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, pages 171–182. ACM Press, 1994.

[TK86]   Shreekant S Thakkar and Alan E Knowles. A high-performance memory management scheme. *Computer*, 19(5):8–22, 1986.

[TKHP92]   Madhusudhan Talluri, Shing Kong, Mark D. Hill, and David A. Patterson. Tradeoffs in supporting two page sizes. In *Proceedings of the 19th annual international symposium on Computer architecture*, pages 415–424. ACM Press, 1992.

[VLX03]   Xavier Vera, Björn Lisper, and Jingling Xue. Data cache locking for higher program predictability. In *Proceedings of the 2003 ACM SIG-METRICS international conference on Measurement and modeling of computer systems*, pages 272–282. ACM Press, 2003.

[WEG⁺86] D. A. Wood, S. J. Eggers, G. Gibson, M. D. Hill, and J. M. Pendleton. An in-cache address translation mechanism. In *Proceedings of the 13th annual international symposium on Computer architecture*, pages 358–365. IEEE Computer Society Press, 1986.

[WM95]   Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 23(1):20–24, 1995.

[WZ97]   Michael Wu and Willy Zwaenepoel. Improving TLB miss handling with page table pointer caches. Technical Report TR97-296, 16, 1997.

Recent technical reports from the Department of Information Technology

2004-008 Pär Samuelsson and Bengt Carlsson: *An Integrating Linearization Method for Static Input Nonlinearities*

2004-009 Henrik Johansson and Johan Steensland: *A Characterization of a Hybrid and Dynamic Partitioner for SAMR Applications*

2004-010 Neil Ghani, Kidane Yemane, and Björn Victor: *Relationally Staged Computations in Calculi of Mobile Processes*

2004-011 Henrik Björklund, Sven Sandberg, and Sergei Vorobyov: *Randomized Subexponential Algorithms for Infinite Games*

2004-012 Parosh Aziz Abdulla, Johann Deneuz, and Pritha Mahata: *Multi-Clock Timed Networks*

2004-013 Andreas Westling: *Inter-Networking MPLS and SelNet*

2004-014 Pär Samuelson, Björn Halvarsson, and Bengt Carlsson: *Analysis of the Input-Output Couplings in a Wastewater Treatment Plant Model*

2004-015 Magnus Ågren: *Set Variables and Local Search*

2004-016 Maya Neytcheva, Erik Bängtsson, and Björn Lund: *Numerical Solution Methods for Glacial Rebound Models*

2004-017 Parosh Aziz Abdulla and Aletta Nylen: *Better-Structured Transition Systems*

2004-018 Emmanuel Papaioannou, Erik Borälv, Athanasios Demiris, Niklas Johansson, and Nikolaos Ioannidis: *User Interface Design for Multi-platform Interactive Sports Content Broadcasting*

2004-019 Owe Axelsson and Maya Neytcheva: *Eigenvalue Estimates for Preconditioned Saddle Point Matrices*

2004-020 Malin Ljungberg and Krister Åhlander: *Generic Programming Aspects of Symmetry Exploiting Numerical Software*

2004-021 Erik Berg and Erik Hagersten: *Efficient Data-Locality Analysis of Long-Running Applications*

2004-022 Pascal Van Hentenryck, Pierre Flener, Justin Pearson, and Magnus Ågren: *Compositional Derivation of Symmetries for Constraint Satisfaction*

2004-023 Jarmo Rantakokko: *Interactive Learning of Algorithms*

2004-024 Mathias Spjuth, Martin Karlsson, and Erik Hagersten: *Low Power and Conflict Tolerant Cache Design*

2004-025 David Lundberg: *Feasibility Study of WLAN Technology for the Uppsala - Stockholm Commuter Train*

2004-026 David Lundberg: *Ad Hoc Protocol Evaluation and Experiences of Real World Ad Hoc Networking*

2004-027 Thorild Selén: *Reorganisation in the Skewed-Associative TLB*

UPPSALA
UNIVERSITET