A Survey on the Interaction Between Caching, Translation and Protection

Adam Wiggins

UNSW-CSE-TR-0321

August, 2003

disy@cse.unsw.edu.au
http://www.cse.unsw.edu.au/~disy/
Operating Systems and Distributed Systems Group
School of Computer Science and Engineering
The University of New South Wales
UNSW Sydney 2052, Australia

Abstract

Fine-grained hardware protection could deliver significant benefits to software, enabling the implementation of strongly encapsulated light-weight objects, but only if it can be done without slowing down the processor.

In this survey we explore the interaction between the processor's caches and virtual memory in traditional as well as research architectures. We find that while caching and translation mechanisms have received much attention in the literature, hardware protection mechanisms have remained largely neglected, with none of the explored architectures providing truly scalable support for context-sensitive, fine-grained protection.

Based on the insights gained from the survey we outline an approach which facilitates the construction of simple, yet fast, low-power fine-grained protection mechanisms for processor cores.

Contents

1 Introduction

Very fine-grained hardware protection of memory objects will support the implementation of secure componentised systems [DF97], secure sharing of arbitrary data [WCA02], and better fault isolation [CF01, SMLE02]. However, such protection is hard to achieve, as it tends to add expensive associative lookups to the critical path of the processor. The lack of hardware support for fine-grained protection has meant that recent work has concentrated on software techniques, such as type safe languages [Mit96] and proof-carrying code [Nec97].

This survey explores the interaction between the processor caches and virtual memory, critically assessing the suitability of various processor architectures with respect to realisation of fine-grained protection. We find that none of the explored architectures is suitable for providing support for fine-grained protection and sketch out the requirements for a protection architecture, based on the insights gained from the survey, that may fill this void.

In the remainder of this section we briefly outline the organisation of a conventional processor with caches and support for virtual memory, while in Sections 2 and 3 we examine in detail the cache design space and coherence issues respectively. Sections 4 and 5 then explore processor support for the virtual memory features of translation and protection respectively, and finally Section 6 concludes and outlines areas for future work.

1.1 Processor caches

Modern processors are equipped with both caches and support for virtual memory. Processor caches improve application program performance by exploiting memory *locality*. Locality comes in two flavours: *spatial locality*, in which memory addresses near a recently accessed address are likely to be accessed as well; and *temporal locality*, where a recently accessed address is likely to be accessed again in the near future. Caches exploit locality by storing recently referenced blocks of data in fast memory in or near the processor core. To keep its speed high and expense low, the caches on the processor are typically only a few tens of kilobytes in size.

When presented with large multi-megabyte and even multi-gigabyte memories, the coverage of a single processor cache is often insufficient. Therefore one or more extra levels of cache are added, either to the processor or between the processor and main memory. Generally, the further away from the processor's core a cache is, the larger and slower it is. An example of such a hierarchy is depicted in Figure 1. The first-level (L1) cache is in the processor, while the second-level (L2) cache resides between the processor and main memory.

This interaction of faster and smaller memory with progressively slower but larger memory is generally known as the memory hierarchy. At the top of the hierarchy are the processor registers which are the fastest, most expensive and smallest memories, going down the hierarchy are first-level caches, other levels of cache, main memory and finally disk (when virtual memory is present in the system) as depicted in Figure 2. The key property exhibited by the memory hierarchy is that on average a memory accesses approach the speed of the top most level of the hierarchy, while providing as much storage as the lowest level.

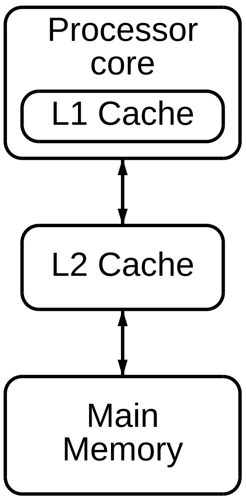

Figure 1: Example multi-level cache organisation.

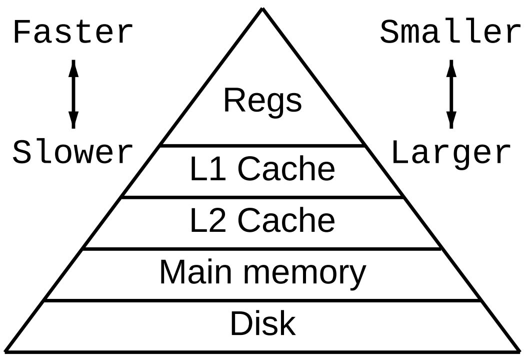

Figure 2: The memory Hierarchy.

1.2 Processor support for virtual memory

The virtualisation of memory enables programs to utilise more memory then is physically present in the system by multiplexing *physical frames*[1] of main memory between a potentially larger set of *virtual pages* of memory. Inactive pages are

[1]A frame is a contiguous block of physical memory some power of two in size, typically 4KB.

typically *swapped out* from main memory into some form of secondary memory, typically a hard-disk drive, until they are once again accessed in the future. An access to a page that is not resident in memory causes an exception, known as a *page fault*. The operating system then allocates a suitable free frame. If no frame is available, a victim page is swapped out. The system then loads (*swaps in*) the accessed page into the target frame.

Virtual memory also allows physical frames to be shared between multiple applications in a secure fashion. For example, the same code might be used by multiple applications, allowing code frames to be shared. However, the system must ensure none of the applications modifies the shared code. This can be achieved by the system marking the shared code pages as *read-only* or even *execute-only*.

Because each memory access must be both translated from a virtual address (VA) to physical address (PA), and have some access rights checked, some kind of hardware support is required. Such hardware support is typically implemented by a memory management unit (MMU) either on the processor or between the processor and main memory.

The MMU, shown in Figure 3, translates virtual to physical addresses and performs protection checks between the processor and main memory. Both the translations and the access rights are stored in a data structure in main memory, know as a *page table*, which is maintained by the operating system. If a virtual page has no translation associated with it in the page table, or if the access rights are insufficient, a page fault exception is raised.

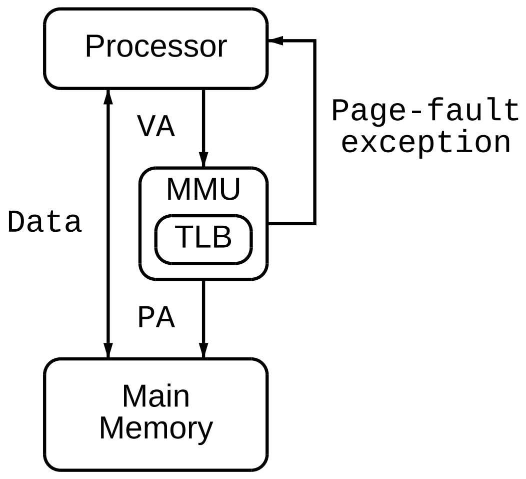

Figure 3: The memory management unit.

A translation and protection check must be performed by the MMU on every memory access by the processor. Each of these translations requires at least one, if not more, additional memory references to look up the page table. To keep the number of additional accesses to main memory by the MMU manageable, the

MMU typically implements a cache of translations, known as a *translation look-aside buffer* (TLB), or *page table cache*. Each TLB entry contains both the translation and protection information of the page it covers in virtual memory.

2 Caching

We begin our investigation into the interaction of caching and virtual memory with the processor cache. A cache is a piece of hardware devoted to storing a limited amount, typically a subset, of information used by a processor for a particular purpose. Caches are characterised by high speed accesses and small size. The larger the cache is, the more it costs in silicon real estate and slower its access speed.

A cache is implemented as an array of fixed-size lines containing some data and a *tag*. The lines are grouped into *sets*, and an *index* is used to select a particular set. Once indexed, the tags are then compared to the address of the current reference, a match indicating a *cache hit*.

Caches can store any type of information used by the processor, varying from instructions and data, to access rights of instructions and data, virtual memory translations or branch prediction tables. In this section we will restrict our attention to instruction and data caches.

Regardless of what is being cached, the goal is the same: To improve the average access latency to the information and thereby allow faster operation of the processor core. A cache's average access latency can be described as

$$H + M \times R$$

where H is the hit access latency[2], M the miss cost, and R the miss rate[3].

2.1 Cache design space

A large number of design choices are available for cache hardware of which Smith [Smi82] provides a thorough, if dated, survey. In our investigation we will look into a subset of the these aspects of cache design, in particular how the cache is addressed, i.e., indexed and tagged. A summary of the design choices is presented in the following sections. Figure 4 illustrates the interaction of the design choices pertinent to the remainder of this paper.

2.1.1 Addressing

The cache address is made up of 3 parts: the index, tag and offset. Firstly, the index selects the set of lines of the cache that will be examined for a hit. Then, the tag is compared with the tag stored in each line of the indexed set, to assert a cache hit

[2]This is usually one to a few processor cycles for first level caches.

[3]While the term *miss rate* is commonly used we are really talking about the probability of a miss.

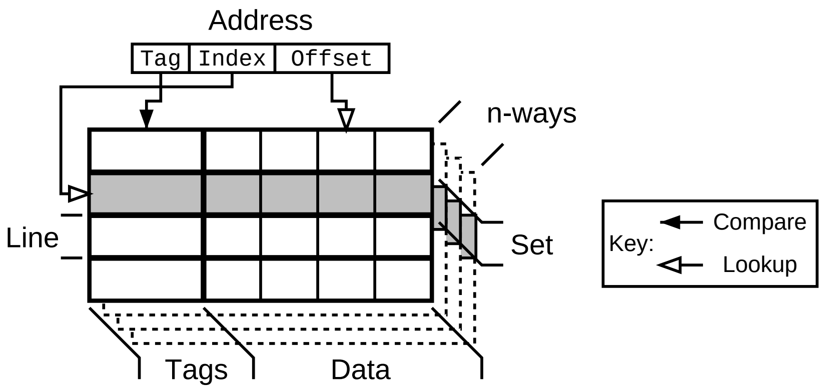

Figure 4: Cache organisation.

or miss. Finally, the *offset* selects the pertinent part of the cache line's data to be accessed.

Although more general hashing schemes are possible, the index, tag and offset are usually substrings of the full address. It should also be noted at this point that the index and tag can be derived, independently, from either the virtual or physical addresses used in the processor.

2.1.2 Set associativity

The set associativity of a cache refers to the number of cache lines in a set whose tags are searched in parallel on a reference. When a single line is checked for a match, the cache is known as a *direct-mapped* cache. At the opposite end of the spectrum, a cache in which every line is checked in parallel for a hit is know as a *fully-associative* cache. The remaining organisations are collectively referred to as n*-way set-associative* caches where n is the number of lines in each set.

Higher-associative designs, in general, exhibit higher power consumption and slower access speeds, compared to direct mapped and lower associativity caches of the same capacity. For this reason fully-associative caches are typically small.

The advantage of higher-associative caches is the fact that their hit rates are inherently higher then those of direct-mapped, and low associative, caches of the same capacity. This is due to the lower *conflict miss* rates of caches with higher associativity. Conflict misses occur when members of the working-set index to the same cache line.

A fully-associative cache has a single set, hence it has no conflict misses. It lacks an index address.

2.1.3 Write-through v. write-back

Because data caches can be updated by processor activity, a choice for data consistency between the cache and main memory has to be made. To keep the data cache and RAM coherent, writes to the cache may also be sent to the memory controller. This organisation is known as a *write-through* cache. While such an organisation maintains strict coherency, it can lead to a performance bottleneck if writes are frequent and the difference in speed between the cache and main memory is large.

To overcome this problem, the *write-back* cache was introduced: Writes only update the cache, avoiding the bottleneck of main memory bandwidth. Main memory is updated when a modified (*dirty*) cache line is evicted from the cache by the replacement policy or via an explicit flush operation. While write-back caches improve cache performance, they lead to inconsistencies with main memory that application software and/or the operating system must be aware of, as we will see in later sections.

2.1.4 Line size

If a single word[4] of data is stored per cache line, the silicon overhead of the tag, and any other cache metadata is quite high, compared to the actual data. In addition, the cache will not take advantage of any spatial locality.

Additionally, line size is influenced by the cost of each memory translation required to fill, and for write-back caches write out, a cache line. The larger the cache line, the longer it takes to transfer between main memory and the cache. The cost of each memory transaction consists of two factors: the constant overhead of setting up the transaction; as well as the per-word access cost. Current RAM technologies facilitate burst reads and writes where the cost of accesses the second, third, etc contiguous word is a fraction of the cost of accessing the initial word.

To reduce the cache overheads for a given cache capacity, caches usually store a collection of contiguously-addressed words. This technique, known as *sub-blocking*, increases the size of each cache line while slightly reducing the size of the tag. Larger line sizes combined with good spatial locality can significantly reduce the number of memory transactions in the system while only fractionally increasing the cost of each memory transaction.

If spatial locality is low, longer line sizes can lead to excessive line fetch overheads where only a fraction of the fetched cache line is actually used. Many modern systems employ a line size of 4–8 words, balancing out spatial locality and burst transactions to main memory.

2.1.5 Replacement policy

When a reference misses the cache, a victim entry must be selected to free up space to load the missing reference into. The selection of the victim entry is known as

[4]Here word refers to the register size of the processor.

the *replacement policy*. Some speciality cache organisations generate an exception on a cache miss, allowing a *software handler* to select the victim and thus leave the replacement policy up to the operating system. The disadvantage of software handlers is that they typically take a longer time to execute and pollute the caches with instructions and data needed to execute the handler.

2.1.6 Read v. read/write allocation

Allocation of a reference to a cache entry can happen on either a read access, or read and write accesses of the data.

2.1.7 Split v. unified caches

Cache organisation for data, protection, or translation can either be *unified*, as in the *von Neumann architecture* for caches, or *split* into separate instruction- and data-stream caches, such as the *modified Harvard architecture* for caches.

Split caches allow for faster processor cores. This is due to the fact that any instruction that accesses the data leads to two memory accesses: one for the instruction word and a second for the data word. By providing separate caches for instructions and data the accesses can occur in parallel.

The disadvantage of split cache implementations is the potential for inconsistencies to arise between the split caches. These inconsistencies occur when data already resident in the instruction cache is modified, for example in self-modifying code[5] or when an executable is first loaded in from disk. Instructions that modify code words will make those modifications through the data cache, creating the potential for an inconsistency. Split caches also incur the power and silicon area overheads of providing two separately addressed caches.

2.2 Cache coherence issues

One of the main areas of concern in cache design is cache coherence. An incoherent cache can lead to incorrect data being accessed or valid memory being overwritten by stale data. In this section we summarise the coherence problems that caches suffer from. Later in Section 2.3 we examine how some coherence issues are only experienced by particular cache designs, while in Section 3 we examine the different approaches put forward by the literature to ensure cache coherence.

2.2.1 Synonyms

Synonyms, also known as aliases, arise when the same data is accessible via several distinct addresses. This situation only occurs in virtually indexed caches, as multiple virtual addresses can refer to the same physical memory.

[5]Self-modifying code is becoming more common due to of *just-in-time* (JIT) and *dynamic* compilation.

Synonyms reduce the effective size of the cache, as well as causing *coherence* issues, since a single piece of data can reside in multiple cache entries. The reduction of the effective size, in turn, results in an increase of so called *capacity misses*, i.e. the misses that result from the size of the working-set being larger than the size of the cache.

Synonyms lead to two forms of coherence problems. The first arises from the simple fact that the same data can reside in more then one cache line. The correct operation of the system may require that future reads see the most recently updated copy of the data. The second problem relates to the coherence of the cache and main memory. It may be additionally required that the most recently updated copy is written back to main memory, while all other copies are invalidated to ensure they do not corrupt valid main memory.

2.2.2 Homonyms

A *homonym* is an address, which, used in different contexts, refers to different data. Homonyms are a problem unique to virtually indexed, virtually tagged caches (see Section 2.3.4) and are the result of the same virtual address being used in different address-spaces, where the virtual-to-physical translations differ.

2.2.3 Multi-cache coherence

When the same data potentially reside in more than one cache, multi-cache coherence can become a problem. For example, a modified Harvard architecture processor or where multiple processors, hence caches, are employed. If the data is updated in one of the caches, the others may potentially cache a stale copy. Alternatively, where write-back caches are employed, main memory contains a stale copy of the data. Other caches may then load this stale copy of the data from main memory. In both cases, stale data maybe used. Either the cache architecture itself or the operating system must take steps to ensure stale copies are either invalidated or updated.

2.2.4 Direct memory accesses

Further problems arise from I/O using *direct memory access* (DMA) with write-back caches. DMA typically by-passes the caches. Since the most up-to-date copy resides in the cache and not in main memory, additional steps must be taken to ensure DMA sees the most recently updated copy.

2.3 Cache indexing and tagging

The focal point of our investigation into cache design is on how caches are indexed and tagged (see Sections 2.1.1 and 2.1.2). For data and instruction caches of a processor, the index and tag can be based on either physical or virtual addresses. The choice of the index and tag addresses are independent, leading to four possible configurations, each of which we shall examine in turn. Cekleov and Dubois [CD97a, CD97b] explored the issues relating to virtually-addressed caches in the context of single- and multi-processor systems. In the following sections we shall revisit their assessment of the strengths and weaknesses of each style of indexing and tagging.

While initially limiting our survey to first-level data and instruction caches, in later sections we will examine caches of protection and translation information.

2.3.1 Physically-indexed, physically-tagged caches

The physically-indexed and -tagged (P/P) cache, commonly known as a *physical cache*, has traditionally been favoured by processor designers because it avoids the problem of synonyms and homonyms.

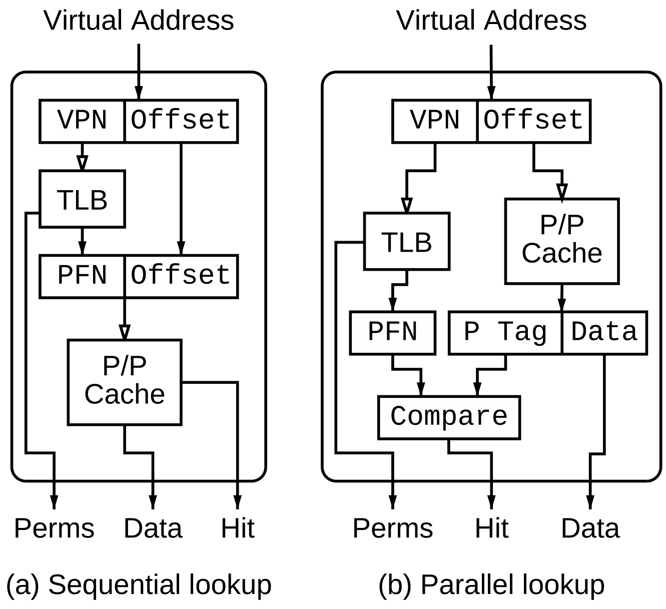

Figure 5: Physically-indexed and -tagged cache.

However, physical caches have performance limitations. Figure 5 shows the two common configurations available for physical caches. Each configuration has its own drawbacks.

In Figure 5 (a), the TLB is first used to obtain the physical address which is then used to index the cache. The problem with such a configuration is that it either limits the processor's clock speed (due to the serial look up of the caches) or requires the TLB and caches to be placed in separate pipeline stages, lengthening the pipeline. Elongation of the processor pipeline adds to the penalties of control flow changes like exceptions and mispredicted branches. This, in turn, can limit the processor's performance.

Figure 5 (b) shows how the TLB and a physical cache can be indexed in parallel, removing any speed or pipelining penalties. In this configuration only the untranslated offset bits of the virtual address can be used in the index. The translated physical frame number (PFN) is then compared against the physical address tag (P Tag) in the cache to validate the access. While this configuration avoids the performance problems of the former, the size of the cache is limited by

$$W \times P$$

where W is the associativity of the cache, and P the minimum page size of the processor core.

The page sizes and cache associativities employed by modern processors might suggest that the limitations of the parallel configuration are insignificant[6]. However, one issue Cekleov and Dubois [CD97a] fail to address is the support for fine-grained protection or translation, down to sizes such as an individual cache line [Lie96]. In such situations, this size limitation becomes significant. For example, assuming a minimum page size of a 64 byte cache line and a set-associativity of 64, the maximum size of the first-level caches is only 4KB.

2.3.2 Virtually-indexed, physically-tagged cache

The virtually-indexed, physically-tagged (V/P) cache has long enjoyed popularity in modern processors as the favoured choice for large, yet fast, first-level caches. Since the V/P cache is virtually indexed, the cache and TLB lookup can occur in parallel, as shown in Figure 6. Parallel lookup facilitates cache speed, and since the index is virtual, the size of the cache is not limited.

However, V/P caches are not without limitations as they subject to the synonym problem. As mentioned in Section 2.2.1, synonyms lead to coherence problems and reduce the effective size of the cache.

In addition to synonyms, mapping coherence is a problem for V/P caches. When a mapping is modified, the physical tag of the cache lines covered by the modified mapping becomes stale. The problem is two-fold: firstly, the old physical frame may be reused at a different virtual address, if a stale cache entry is written back to main memory it will overwrite the valid data in the reused frame. The second mapping coherence problem occurs if the cache entry is not written-back to main memory, leading to the loss of any updates. This results from the fact that the update does not yet reside in main memory and any references in the cache will result in a miss, since the cache's physical tag no longer matches the *physical page number* (PPN) returned by the TLB. These coherence problems are only an issue for write-back caches.

[6]The size limitation really only affects first-level caches. Caches further down the hierarchy are either referenced after the translation step has already occurred, or the increased access cost is acceptable.

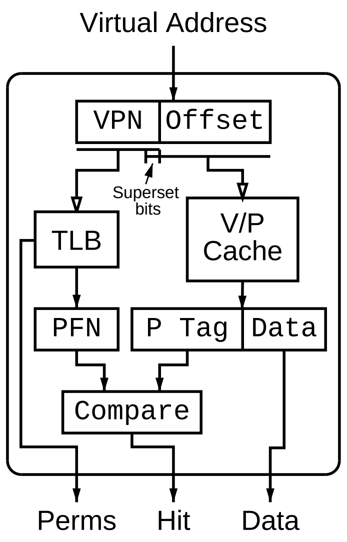

Figure 6: Virtually-indexed, physically-tagged cache.

2.3.3 Physically-indexed, virtually-tagged cache

A quite unusual cache implementation is the physically-indexed, virtually-tagged (P/V) cache employed by the MIPS R6000, see Figure 7. It overcomes the size limitations of a physical cache by employing a new translation cache known as a *TLB slice* [TDF90]. The majority of the cache's indices are made up of the untranslated offset bits while the TLB slice translates a small number of VPN bits into PPN bits making up the remainder of the index.

Since the cache is tagged with the virtual address, a hit can be detected without translating the complete virtual address. Because only a partial translation is needed, the TLB slice can be quite small and fast, impacting only minimally on the processor's cycle time. In particular, because the virtual address is used to validate the access, the TLB slice can be direct-mapped and avoid a validation tag. If the entry is incorrect, the cache's virtual-tag mismatch will indicate a miss.

The original proposal cites two major drawbacks of the TLB slice approach. The first is the potentially low hit rate of the TLB slice. The second is the increased complexity of a cache miss. A cache miss signifies two possibilities:

TLB slice miss: In this case the TLB slice contains the incorrect partial translation, this is in effect a slice miss masquerading as a cache miss. To validate this, the full TLB is queried[7] on a cache miss to provide the full translation. If the slice entry is mismatched, it is updated and the access retried.

Virtual-tag miss: If the TLB slice contains the correct partial translation, the cache's virtual tag missed. This can signify either a real cache miss or a

[7]This access to the main TLB can result in a TLB miss requiring a full TLB reload.

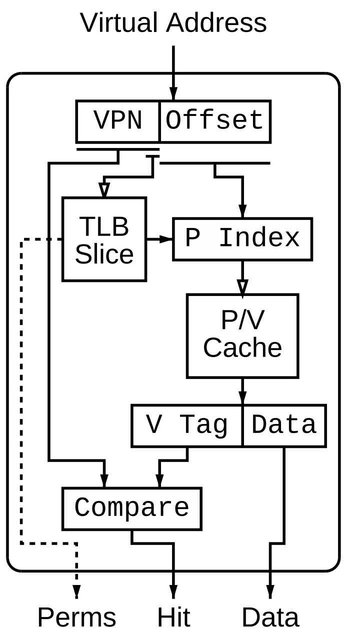

Figure 7: Physically-indexed, virtually-tagged cache.

synonym. To address this problem, the cache also contains a physical address tag which is compared with the full translation from the TLB. A match indicates a hit, in which case the cache's virtual tag is updated, while no match is an indication of a full cache miss.

In their assessment of the TLB slice [TDF90], the authors found that while instruction references showed high TLB slice hit rates, data references suffered from significant misses. The low data hit rate arises from the small, direct-mapped nature of the TLB slice. The authors proposed to use the P/V cache as a second-level cache shielded by a virtual first-level cache to solve the problem.

One weakness of the proposal is its failure to address the issue of how protection is implemented. Two TLB-based approaches could be employed.

Access full TLB in parallel: By accessing the full TLB in parallel to the cache, the permission bits can be retrieved in time to validate the access rights to the cache line. This has the advantage that the physical translation is already available during a cache miss. However, such a design requires the full TLB to be in the processor's critical path, negating of the main benefit of the TLB slice.

Store permissions in TLB slice: At the cost of a significant increase in silicon real estate[8] of the slice, but for similar speed and power costs, the permis-

[8]In the R6000's TLB slice, each entry contains 4 PFN bits. Up to 3 extra bits would be required

sions can be stored in the TLB slice entry itself. This option is highlighted by a dotted line in Figure 7.

As we will see later (Section 5) alternatives to TLB-based protection are also available.

While the P/V cache paired with a TLB slice overcomes the size limitations of a physically indexed cache, it has a number of clear drawbacks: the cost of cache miss handling blows out, requiring more complex miss handler hardware; a physical tag is stored along with the virtual tag, consuming more silicon real estate; and finally, the directly-mapped nature of the TLB slice leads to poor hit rates for data references.

While the above drawbacks are manageable, the issue of protection, particularly fine-grained protection, is not addressed.

2.3.4 Virtually indexed, virtually tagged cache

The virtually-indexed and -tagged (V/V) cache, commonly known as a *virtual cache*, has the unique property of allowing both the indexing and tag comparison of the cache to be completely independent of the TLB (Figure 8).

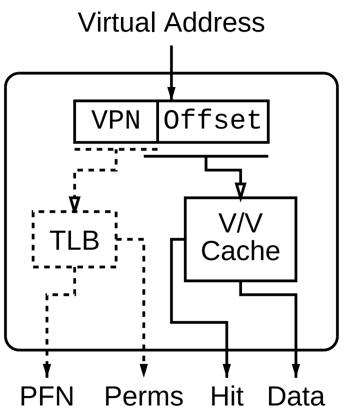

Figure 8: Virtually indexed, virtually tagged cache.

A side effect of this organisation is that TLB lookups are only needed to validate access permissions of the cache reference on a hit. In fact, this can be taken a step further by completely removing the translation step (highlighted by the dotted lines in Figure 8) from the processor's core. This facilitates a single fast and large cache on the critical path of the processor.[9]

While the removal of the TLB from the processor core has advantages (see Section 4) it leaves the problem of validating the access rights of the reference. We

to store permission information in a unified TLB slice, nearly doubling its size.

[9]For now we are ignoring unified caches v. separate instruction/data caches as the choice has little effect on the processor's core.

will examine alternative approaches to caching protection information in Section 5.

Wilkes and Sears [WS92] claim that virtual addresses are typically longer then physical ones leading to longer tags in V/V caches for 64-bit systems. Longer tags increase the silicon area of the cache and its power consumption without increasing its capacity, a serious drawback. It should be noted, however, that this expense comes at the benefit of potentially faster processor architectures.

Virtual caches, like V/P caches, suffer from the synonym problem, and in addition the homonym problem. The homonym problem is particularly nasty due to the fact that, without some hardware support, the only way to effectively deal with homonyms is to flush them from the cache on every address-space context switch.

An advantage of the V/V cache over the V/P cache is that mapping consistency is no longer a problem. This, however, comes at the cost of an additional TLB reference on a cache miss. During a cache miss, if the victim entry needs to be written back to main memory, a TLB lookup must occur to find the physical write-back address. An alternative to the additional TLB lookup is to store the physical address in the cache line, facilitating write back without the additional translation, albeit at the expense of more silicon real-estate.

As Cekleov and Dubois [CD97b] note, these drawbacks have prevented virtual caches from widespread use in modern processors for the data cache. However, they are often the used for instruction caches and have found much attention in the research community.

3 Handling cache coherency

A large number of approaches have been described in the literature for dealing with the coherence issues outlined in Section 2.2. The aim here is not to investigate them thoroughly but to summarise the general approaches and their limitations. Many, but not all, of these are covered in [CD97a, CD97b].

3.1 Synonym avoidance

A number of techniques exist to avoid synonyms altogether. Firstly, synonyms in the cache can be avoided by marking them uncachable. Such a scheme, however, leads to poor performance where the data is accessed frequently.

Another approach is to handle inter-address-space synonyms by flushing the cache on each address-space context switch. While such a scheme has the benefit of dealing with homonyms as well, it fails to handle intra address-space synonyms, and can lead to poor cache performance and high context switch costs for write-back data caches [WH00, WTUH03].

Finally, synonyms can be avoided by various alignment restrictions, like ensuring the superset bits[10] of the VPN translate to the same way for all synonyms of

[10]The superset bits of a V/P cache are those bits of the VPN used to form part of the index address,

any given piece of physical memory. While this somewhat restricts OS memory management, it otherwise exhibits no run-time costs.

3.2 A single global address-space

An effective way of handling synonyms and homonyms in the cache is to simply remove the need for them. This can be achieved by using a single global address-space to address the cache.

Two basic approaches to realising a single address-space addressing of the cache exist, one software based, the other hardware based. The software-based approach is that of the single-address-space operating system (SASOS) [CLFL94, HEV+98]. In a SASOS, all applications run in the same address-space, abolishing homonyms and removing the need for synonyms. SASOSes have yet to gain widespread popularity.

The alternative is to provide some form of segmentation hardware. Since the aim is to provide a single address-space view, to the caches, this segmentation hardware must be on the critical path of the processor, before the cache is indexed. To this end segmentation provides a per-task/process view of the larger single address space used to look up the caches. In the following sections we will examine the different approaches to hardware segmentation mechanisms.

3.2.1 Address-space identifiers

The first and simplest form of segmentation is to prepend the virtual address with an address-space identifier (ASID), producing a unique, global address for the cache tags. On a cache reference the virtual address is tagged with the processor ASID register before being presented to the cache (Figure 9). ASIDs are used by many RISC architectures (MIPS, Alpha, SPARC, etc) because of their simplicity.

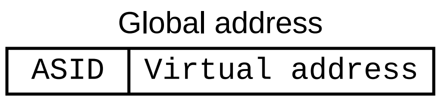

Figure 9: ASID-based global address.

Such a scheme leads to both fast hardware and fast context switches, as the additional cost of saving/restoring the ASID register on a context switch is minimal. Each address space is then a segment of a fixed size which is mapped into the cache's global address-space as shown in Figure 10.

ASIDs handle the homonym problem by transforming it into a synonym problem. This is achieved by taking a context-specific address and transforming it into a unique, global address, by concatenating the ASID with the virtual address. This

see Figure 6.

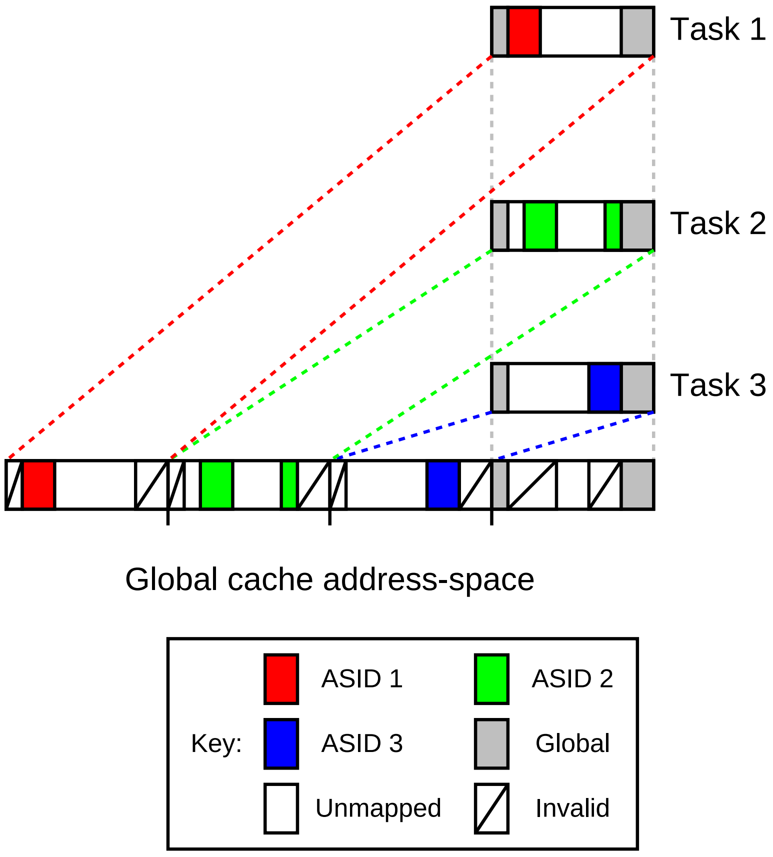

Figure 10: ASID-based segmentation.

form of segmentation, however, does not remove synonyms and also has the unfortunate side effect of turning virtual memory references shared between multiple address-space contexts into synonyms, generating additional coherence issues.

For systems where sharing is insignificant, ASIDs provide an effective solution. However, as soon as heavy sharing occurs between address-spaces, performance suffers. Handling of synonyms remains an issue to be dealt with via an alternative method.

To deal with the issue of sharing, many processors using ASIDs support a coarse-grained form of sharing, where one special ASID indicates that the address is accessible to all address-spaces. While this is useful for operating system constructs, it is not sufficiently general for sharing between user-level applications, without per-context protection.

One approach to overcoming the problem of aliases in ASID-based systems is the common-mask scheme [KT95], where the ASID tag is generalised to be either an address-space identifier or a kind of *shared domain identifier*. The distinction is made via an additional selector bit in the tag. The processor's ASID register is then extended by a shared domain identifier, allowing each process to access a single, unique ASID and one shared domain. While this approach is more general than the plain ASID approach, once again, it is not general enough as only a single shared domain is available.

3.2.2 Segment registers

A more general form of segmentation than ASIDs is the segment register approach employed by such architectures as the PA-RISC [Lee89], Itanium [Int00] and PowerPC [MSSW94]. In this scheme, the most significant n bits of the virtual addresses form an index into a small segment-register file. Each entry expands to an m-bit segment number (Seg #), which is prepended to the remaining bits of the virtual address (segment offset) to form the global address as shown in Figure 11.

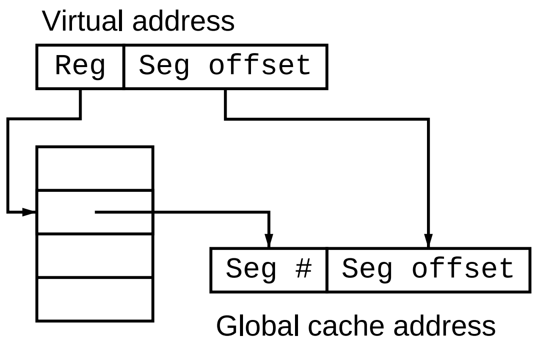

Figure 11: Segment registers.

The advantage of this scheme is that it allows each address space to access up to 2^n equally-sized segments. This can be used to remove both homonyms and synonyms which are at equal offsets from the start of their segments.

The drawbacks, however, should be clear: Firstly the multiple segment registers add additional address-space context to be switched. The tradeoff is: a smaller number of segment registers, hence smaller context restricts the number of uniquely addressable shared segments; on the other hand, increasing the number of segment registers increases both the silicon area of the segment register file and its associated address-space context. Secondly, only segment-aligned synonyms between address-spaces are removed, which severely limits the applicability of this form of segmentation for removing synonyms.

The PA-RISC has 4 segment registers, each either 16- or 32-bits wide to generate a 48- or 64-bit global address, while the Itanium employs 8 *region registers*, each between 18- and 24-bits to provide up to 85-bits of global address space. The PowerPC architecture utilises 16 segment registers, each 24-bits wide to generate a 52-bit global address space. In each of these systems the caches are physically addressed, hence the global addresses are used for TLB lookups only.

3.2.3 Segment tables

More general than a small, fixed set of segment registers is the use of a generalised *segment table*, such as that used by the PowerPC 64 architecture [MSSW94]. A segment table is similar to the page tables found in most modern systems, but typically maps larger regions.

While segment tables can support arbitrary *(base, limit)* address pairs, in practice segment sizes are usually limited to large, fixed-sized powers of two. This greatly simplifies the segmentation hardware.

Similarly to a page table, the segment table needs to be accessed on every memory reference. To avoid the multiple memory accesses required to lookup the segment table entry on each memory reference, the PowerPC 64 architecture employs a segment look-aside buffer (SLB), similar to a TLB, to cache the most recently referenced segments. The SLB can be made valid across all address-space contexts by adding an ASID style tag the SLB.

This form of segmentation allows a much more fine-grained approach to synonym and homonym management, lower address-space context costs and makes wide-spread usage of shared segments practical.

The problem with such an approach, however, is the fact that it simply substitutes the TLB with another cache of similar complexity and coverage, the SLB. This negates any benefit virtual caches have in allowing the TLB to be removed from the critical path.

3.2.4 Thoughts regarding segmentation

As we have seen, hardware segmentation approaches can handle synonyms and homonyms to varying degrees by remapping the virtual address space into a single address space, which is used to address the caches.

The limitation is that, as the techniques provide more general support in handling of synonyms and wide-spread sharing, the hardware involved approaches the complexity of a TLB, adding yet another cache to the processor's critical path. Additionally, only inter address-space synonyms are removed, while synonyms within the same address-space remain an issue.

3.3 Reverse maps

An alternative approach to remapping virtual addresses for removing synonyms is that of *reverse maps*. Reverse maps work by only allowing a single copy of a synonym to reside in the cache at any time. To facilitate this, a directory of some sort has to be maintained on the cache-miss path which keeps track of the active aliases for each synonyms in the cache.

Cekleov and Dubois [CD97a] divide reverse maps into two basic implementing approaches, which are defined below.

3.3.1 Back-pointers

The easiest implementation of reverse maps is in systems with a second-level cache. When a first-level cache miss goes to the second-level cache, the accounting info of the second-level cache line contains the location of the first-level cache entry, if any, that is a synonym of the line that generated the original miss. If such a synonym is present, the miss handling hardware loads the cache line from the first-level synonym, rather than the second-level cache. The old synonym is then invalidated to ensure the uniqueness property in the first-level cache. This extra accounting information is represented by a *back-pointer* added to each second-level cache line. The back-pointer consists of a valid bit and the address of the line in the first-level cache, containing the synonym.

For such a scheme to work, the cache organisation must have the following properties: firstly, the second-level cache must be inclusive, that is, if a line resides in the first-level cache, it must also reside in the second-level cache; secondly, synonyms in the second-level cache must be avoided. This is usually facilitated through the use of a P/P second-level cache.

A bonus of this approach is that if a unified second-level cache is used, as is typically the case, instruction and data cache coherency can be maintained as well. Expanding the back-pointer by a single bit allows it to indicate in which first-level cache the synonym resides.

The only drawback, besides the requirement of a specific kind of second-level cache, is that for systems in which software avoids synonyms and manages instruction/data cache consistency, significant amount of extra hardware is needlessly present.

3.3.2 Dual directories

An alternative to back-pointers is to have a cache dedicated to storing the addresses cached by the first-level cache lines. Such a cache is known as a dual directory. The dual directory is usually a P/P cache, which is ideally fully-associative and has as many entries as there are active synonyms.

The cost of a dual directory can be reduced by reducing its size or associativity. This leads to the possibility of dual-directory misses, the handling of which requires a victim entry to be selected and replaced. This eviction additionally requires the eviction of the first-level cache entry it covers, a process known as *pair eviction*. Pair eviction can lead to the underutilisation of the first-level caches.

While a dual directory might seem to be overkill for uniprocessor systems, when paired with bus snooping protocols, such an approach can provide multi-cache and DMA coherence. It is also worth noting that the dual directory is not on the processors critical path. It is only accessed on each cache miss or bus transactions.

3.3.3 Thoughts regarding reverse maps

Reverse maps employ additional hardware resources to handle synonyms, in general by ensuring only a single alias is active at any one time. In particular, reverse maps can ensure multi-cache and DMA coherence for bus-snooping-based systems. In such systems all bus transactions are snooped by either the second-level cache or all the dual-directories. If the address of the transaction hits in the reverse map, the respective first-level cache entry is either invalidated (in the case of write transactions) or used instead of the copy in main memory and then invalidated (for read transactions).

A side effect of the reverse map approach is that read-only data which may safely reside in multiple caches is forced to reside in, at most, a single cache line. This restriction reduces the access performance of widely shared read-only data. Most bus snooping based systems overcome this problem via *write-update* or *write-invalidate* protocols [Sch94] which allow read-only data to reside in multiple caches, unlike the *access-invalidate* scheme outlined above. Note that only a single alias of a synonym may be active per reverse map.

While reverse maps do not lie on the processor critical path, they are still fairly expensive in terms of silicon area and power consumption. They also fail to deal with homonyms.

3.4 Protection-based approaches

Work with the Mach kernel [WB92] demonstrated that a carefully designed kernel could utilise the processor's virtual memory mechanisms to enforce coherence. Their prototype efficiently managed synonyms as well as mapping and DMA consistency on a processor with virtually-indexed write-back caches.

Similar work [WH00, WTUH03] has demonstrated that such protection-based approaches can, together with careful address-space layout, manage homonyms effectively. This technique attempts to lay out all processes without overlap in the address space, similar to a SASOS, handling any unremovable homonyms with virtual memory protection mechanisms.

3.5 Multi-cache coherence of virtual caches

Many coherent multi-cache systems that employ virtual caches tag each cache line with a physical address, in order to ensure multi-cache coherence as well as potentially handle synonyms. An example of such a system is the SPUR [KEW+85], which went as far as broadcasting both the virtual and physical addresses on the system bus. The problem with such approaches is the increased silicon area of the cache, as well as the expense of increased bus transactions. It also introduces the mapping consistency problem (Section 2.3.2), which V/P caches suffer from [CD97a].

The motivation for such approaches is the P/P nature of the dual directories used for cache coherence. However, a dual directory may itself be implemented as a V/V cache, providing multi-cache coherency based on virtual addresses. Such systems require only virtual addresses to be published on the system bus. The drawback is the dual directory no longer manages synonyms. A less problematic side effect is that each addressable device on the bus must either be mapped to a fixed location in the virtual address space, or have some kind of translation functionality.

3.6 Alternative synonym handling approaches

There is an abundant amount of literature covering hardware techniques to manage synonyms in virtually address caches, for further references see Cekleov and Dubois's survey [CD97a, CD97b].

3.7 Coherence for network-based interconnects

The use of virtual addresses in network interconnected multi-processor, multi-cache architectures is a more recent area of research (see Section 4.2). While this is clearly tied into the future direction of this work, it is beyond the scope of this report and will have to be revisited later.

4 Translation

Section 3 introduced address translation, the process of deriving the physical addresses (PA) used to access main memory, from program-visible virtual addresses (VA). We have already seen that the cache architecture determines where this step occurs in the processor. In this section we will outline the design of conventional translation architectures, namely the TLB, and examine their properties in relation to the cache design issues we have explored. While a detailed analysis of alternative translation architectures is beyond the scope of this investigation, we will outline some of the alternatives to TLB-based translation and attempt to draw some conclusions from this.

4.1 Translation look-aside buffers

Before examining the alternatives, let us take a close look at some of the properties of modern TLB design. Typically, the TLB is on the processor's critical path, translating $VA \rightarrow PA$ in parallel to the first-level cache indexing. On physically tagged cache architectures, the resolving TLB entry is used to validate both the tag and the access rights, while on virtually tagged ones the TLB is simply referenced to validate access rights on a cache hit.

Some key points to note about the design of conventional TLBs are explored in the following sections.

4.1.1 Homonyms

The TLB is a virtual cache, hence it suffers from synonyms and homonyms. Homonyms in the TLB arise from address-space switches. Typically, they are dealt with via some form of hardware segmentation, ASIDs, segment registers, etc. Some architectures, like the x86 [Int02], provide little or no hardware support for handling TLB homonyms, so the operating system must either invalidate the TLB on each context switch, or alternatively, the hardware has to flush them when the register containing the page table pointer is modified. The leads to potential performance problems.

4.1.2 Synonyms and coherence

If a page is shared, multiple TLB entries will be used to map it. These synonyms in the TLB lead to mapping consistency issues. If a shared mapping is unmapped or remapped, multiple TLB entries potentially cover the mapping. All of these may have to be invalidated or updated, for example when a frame is preempted and reused.

Furthermore, TLBs suffer multi-cache coherence issues similar to those of data caches. For example, if an address-space has run on multiple processors and a page is unmapped or remapped, the system must ensure that TLB entries of all processors potentially covering the mapping are invalidated or updated. Typically, hardware does not ensure TLB coherence, so operating system software must ensure it through techniques such as TLB shootdown [BRG$^+$89] where each processor potentially mapping the effected page is interrupted and stalled until all processors have updated their TLBs.

4.1.3 Mixed page-size support

The inadequate coverage of modern TLBs has been highlighted by several studies [CBJ92, HH93, Tal95, KS02]. One approach to improving the coverage of the TLB is to support a number of different page sizes, allowing the TLB to map more physical memory without increasing the number of entries in the TLB [TKHP92].

This issue of support for mixed page sizes is of particular interest, as it highlights one of the main limitations of modern TLB design. Talluri et al. [TKHP92] show that support for more then two page sizes is generally facilitated via the use of fully-associative TLBs. In such organisations, the content-addressable-memory (CAM) cell of the TLB utilises the size mask of the entry to mask out the relevant bits of the virtual address before the tag match is performed.

The limitation of such an organisation is that the TLB's size is restricted due to its fully-associative nature. Increasing the number of entries in the TLB has a detrimental effect on its size, power consumption and speed. To help solve this problem, the TLB either needs to be removed from the the processor core or have its associativity lowered. The latter makes support for multiple page sizes troublesome.

4.2 Off-core TLBs

The first alternative to having the TLB on the processors core, known as an *on-core TLB*, is to move the TLB from the processor to a place further down the memory hierarchy. A particularly interesting approach cited by Cekleov and Dubois [CD97b] is that of in-memory translation [Tel90] in which the TLB is pushed right down to main memory, and virtual addresses are used on the main memory bus.

Such an approach not only removes the TLB from the critical path, easing the constraints on its size, power and speed, it also eliminates the TLB consistency problem, even with DMA. Cekleov and Dubois claim:

> A virtual-address bus combined with virtually addressed memory banks is a very attractive design point for cache-based multiprocessors that has not been fully explored.

Some recent approaches in the literature are even examining the applicability of utilising virtual addressing in network-interconnected, multi-cache systems [QD98, QD01].

4.3 Eliminating the TLB

An alternative approach is to eliminate the TLB altogether. Two methods for eliminating the TLB stand out in the literature, which we will examine briefly.

4.3.1 In-cache translation

The first steps towards eliminating the TLB where made by the SPUR workstation's *in-cache translation* mechanism [WEG+86], in which the processor's first-level cache is used for data, as well as translation information.

In the SPUR, a cache miss is handled by first applying the miss address to a hardware translation walker. This walker then traverses a hierarchical page table in-cache. The walker can generate recursive cache misses up to the root of the page table, whose physical address is stored in a special processor register.

SPUR's in-cache translation mechanism overcomes many of the limitations of conventional TLBs. Because the SPUR provides multi-cache coherency, translation coherency is a side effect; and translation coverage is dynamically traded off with data coverage. Since a cache line is typically much longer then a single translation entry, in-cache translations automatically provides sub-blocking [TH94], where a single entry stores multiple translations.

However, such an approach is not without its drawbacks. For example, without a fully-associative first-level cache, support for mixed page sizes is problematic.

4.3.2 Software managed translation

A particularly novel approach to translation is that of *software-managed address translation* [JM01], in which cache misses raise an exception and are handled by software. The arguments for such an approach:

Replacement policy: The caches replacement policy is totally in control of the operating system, allowing working set profiling and maintenance. Such control can potentially lead to higher hit rates and more effective cache utilisation.

Increased page table flexibility: By minimising the hardware constraints on page table structure, more efficient support for large and sparse address-spaces, super-paging becomes possible. Fine-grained translation down to the granularity of a single cache line also becomes a possibility.

However, such an approach is limited by its impact on the cost and frequency of cache misses. In particular, the cache-miss-exception handler must be fast, and a large second-level cache is required to offset the increased miss penalty with a lower miss rate. Additionally, some mechanism must be available to pin part of the exception handling code into the cache.

4.4 Thoughts regarding translation

All of the above approaches to removing the TLB from the processor core employ virtual caches. However, such systems leave the issue of validating access permissions for first-level cache references unresolved. The SPUR, for example, utilised four segment registers to restrict access, while Jacob and Mudge's software translation scheme use PowerPC segments in a similar way. These methods facilitate only coarse-grained protection.

These schemes will work well if we can de-couple protection and translation. The challenge is then to provide some kind of access rights cache that exhibits more effective coverage than an on-core TLB, while enabling more fine-grained protection.

5 Protection

We conclude our survey with an examination of the different approaches to protection in architectures where translation has been removed from the processor core. Our aim is to discover an approach which is cheaper, smaller and faster than a conventional TLB, while providing better coverage and more flexible protection.

5.1 In-cache access rights

In their study of software-managed translation, Jacob and Mudge [JM01] proposed to provide protection down to a single cache line by storing the protection bits in the cache line itself.

While an attractive idea, such a scheme suffers from two major problems. Firstly, protection updates are potentially very expensive. As a protection update covers from zero to all cache lines, each cache line must be accessed and updated.

The second issue relates to fine-grained per-context protection. Protection bits in the cache provide no mechanism for sharing a cache line between separate address spaces with different access rights. As mentioned earlier, Jacob and Mudge provided per-address-space protection through the use of the PowerPC segmentation mechanisms. Such an approach, while allowing fine-grained protection on a global basis, still provides only course-grained protection on a per-context basis.

5.2 Segmentation-based protection

Access rights can be supplied by hardware segmentation, as we have seen from some of the above systems. However, the same problems identified in Section 3.2 come into play, basically that segmentation-based approaches either lacks flexibility or performance. Segmentation-based access rights would either be too course-grained or would result in hardware of similar complexity and expense to that of a conventional TLB.

5.3 Protection look-aside buffer

In order to facilitate the use of virtual-cache architectures with support for per-page and per-domain access rights, Koldinger el al. [KCE92] put forward the proposal of a protection look-aside buffer (PLB). The design of the PLB is identical to that of a conventional TLB, except that the physical-address part of each PLB entry is dropped. This leads to cache organisation as depicted in Figure 12. The PLB was put forward as a suitable protection architecture for processors aimed at SASOSes.

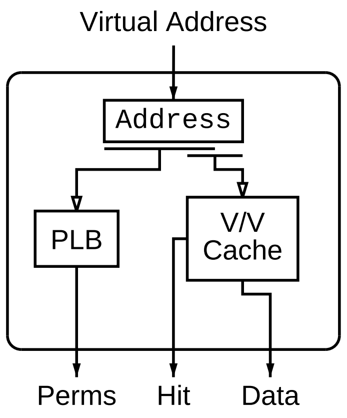

Figure 12: PLB-based cache organisation.

Such an approach has several advantages over a conventional TLB-based architecture. Firstly, the use of a PLB completely decouples protection from translation, allowing the employment of an arbitrarily small protection page size at the cost of reduced coverage. The selection of the minimum page size for protection is completely independent from the translation techniques employed by the system.

The PLB, like its cousin, the TLB, can also increase its coverage by providing support for mixed protection-page sizes. In addition, migration of virtual address-space regions to different protection page sizes avoids the problems associated with migrating to different translation page size. In particular, migrating from a smaller to a larger page does not require the allocation of large contiguous areas of physical memory and resulting copying of data.

The main point to note at this stage is that the PLB will suffer similar coverage problems as a conventional TLB. Mixed page sizes will require full associativity, limiting the PLB's size, while supporting a single size imposes a hard limit on the granularity of protection, requiring a hardware decision to be made for the granularity v. coverage tradeoff.

Some other advantages cited by Koldinger's proposal include the reduced silicon of the PLB v. a TLB of the same coverage, as well as support for shared memory. Wilkes and Sears provide a critical qualitative comparison of the PLB approach to the protection architecture employed by HP's PA-RISC [WS92]. While this report presents some questionable comparisons, their comments regarding the comparative sizes of virtually-addressed v. physically-addressed tags merit attention.

Although the PLB itself requires a smaller silicon area then an equivalent TLB, the use of a virtual cache will result typically in larger cache tags which has a detrimental effect the silicon area, power consumption and speed of the cache.

While this is qualitatively clear, only quantitative simulation results can truly evaluate its effects. For example, modern processors support larger and larger physical address spaces, reducing the effects of virtual v. physical tags for the cache.

5.3.1 Range PLB

An interesting extension to the PLB design is that of the *range PLB* (RPLB) [SM99], which, in addition to access rights, provides protection domain rights, facilitating potentially light-weight protection domain context switches. The inherent complexity of the RPLB raises serious doubts about the possibility of designing an RPLB suitable for a processor with a very high clock rate, and Skousen and Miller's simulations do not alleviate this fear.

5.4 Protection keys

It seems fairly clear that Intel's new Itanium [Int00] architecture's *protection-keys* are based on Wilkes and Sears's critique of the PLB v. the HP PA-RISC protection architecture [WS92]. The organisation of the Itanium's caches is outlined in Figure 13.

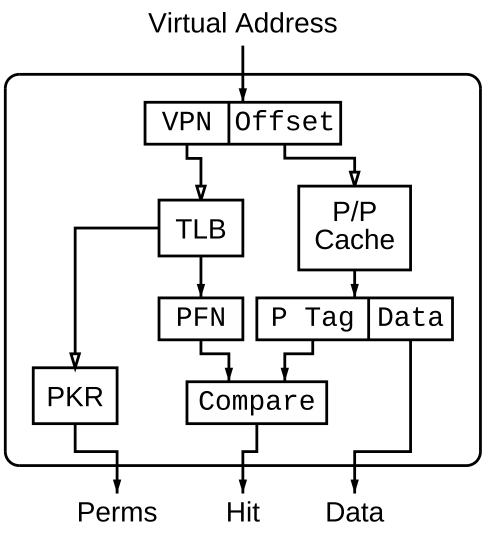

Figure 13: Itanium cache organisation.

In Itanium[11], TLB entries are tagged with a protection key which is used to index into a set of *protection-key registers* (PKR). The PKR is a software-managed, fully-associative cache with no context-specific tags. Of particular note is its small size relative to the number of protection keys, only 16 entries on the first generation Itanium-I processors, compared to the 2^{18} supported protection-keys.

Because the PKR is indexed after a TLB lookup, the TLB and PKR lookups are serialised, requiring the accumulative time of both lookups to be achievable in a single clock cycle or for them to be pipelined. On a hit, the PKR entries access rights are AND-ed with the access rights stored in the TLB entry. Because the PKR entry only conveys access rights, the cache lookup is not dependent on the PKR lookup. Invalid accesses can then be aborted in the pipeline stage after the relevant memory access stage.

The small size of the PKR could be influenced by the following design choices: firstly, the choice of a fully-associative cache inherently limits its size and speed; secondly, the lack of context-specific tags means the PKR must be invalidated or saved on every context switch, leading to additional overheads. Neither of these are inherent limitations of the protection key approach.

The benefits of Itanium's protection keys lie in the fact that they facilitate a partial de-coupling of protection and translation, while maintaining a small tag size. This approach has a few drawbacks, some general to the approach, others specific to this particular implementation.

Because the protection keys tag TLB entries, the TLB is still on-core, leading to the problems we have discussed previously. Some additional problems also exist

[11]The size and associativity of the first-level caches on the Itanium-1/2 indicate that they are P/P.

which we will discuss.

While the protection keys partially decouple protection and translation, there is still a strong link between them, in particular, the minimum protection granularity is that of a single TLB entry. This is obviously not suitable for fine-grained protection.

In addition, no hardware support is provided for retagging TLB entries with new protection-keys when a protection object is split up, although software techniques can achieve this at the cost of more complex key management.

All of this comes at the cost of yet another cache on the processor core!

5.5 Thoughts regarding protection

Wilkes and Sears [WS92] make an interesting statement that has formed much of the motivation of this exploration.

> The intellectual attractiveness of associating per-page memory access rights with the process is undeniable: it is a very simple, straightforward model. Unfortunately, it is potentially expensive to implement, and the real issues arise in determining what to compromise in implementing an efficient approximation to it.

The problem is that all the protection architectures presented here make architecture-level compromises. None provide an architecture which allows operating system policies to control the various trade-offs. While *policy-free* architectures are impractical, it is never-the-less a worthy aim to try to push policy decisions from the hardware into the system software.

If we focus our attention on the PLB and Itanium protection-key approaches, we can highlight these architectural-level compromises.

The PLB, while de-coupling protection from translation, results in hardware of similar power, size and speed constraints as that of a conventional TLB. The de-coupling, however, presents the hardware designer with the tradeoff of protection granularity v. coverage.

The Itanium protection-key approach presents a different compromise, one of protection-object coverage v. context-switch costs. Additionally, this approach fails to deliver true de-coupling of protection and translation or fine-grained protection, because protection-keys are still tied to the (coarse-grained) TLB.

6 Conclusions and future work

This study illustrates some of the many interactions between caches of data, translation and protection information. In particular it has highlighted that virtual caches, combined with a fast and flexible protection cache and an off-core translation scheme, are an attractive basis for faster and lower-power processor cores with support for fine-grained protection.

Out of the architectures surveyed only the PLB-based and protection-key-based ones approach approach the properties desirable for fine-grained protection. However, neither approach would appear to scale well, as the PLB involves the same trade-offs as a TLB, while protection keys are coupled to the TLB, keeping the TLB on-core.

If we look at the PLB and protection-key architectures, we find a mix of the desired properties. Hence, we define a basis for an alternative protection architecture incorporating aspects from both, see Figure 14. In this organisation we associate the *name* of the object being protected with the cacheline that it resides in. This name is then is then concatenated with a context identifier before being used to index a protection-key cache (PKC), where the access rights are stored.

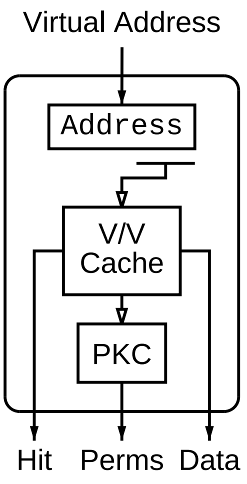

Figure 14: Alternative protection architecture.

The PKC architecture allows per-context protection attributes to be associated with data down to the granularity of a single cacheline. Unlike the PLB, the PKC need not be fully associative, as mixed page-sizes are not required. Hence, the PKC should scale better then a PLB.

A number of questions remain to be answered though. While the PKC may provide the the key to fast cores, where and how does $VA \rightarrow objectname$ translation occur? How can sub-cacheline object names be stored efficiently in the caches? Given the ability to remove the TLB from the core, what address translation method is most appropriate to replace it? Can synonyms and homonyms be effectively managed or even removed by software, or is some form of hardware support preferential? And how can the virtual address space form the basis of cache coherence across network-based interconnects?

Finally, how can an operating system kernel most effectively exploit and export these hardware mechanisms, allowing performance tradeoffs to be made by high-level user policies?

Some of these questions are currently being explored in work to provide fine-grained protection through a PKC-based mechanism [WWTH03]. Future work will aim at bringing these protection techniques together with new coherence techniques for virtually-addressed caches in multiprocessor systems, exploring the new applications of fast and flexible protection made possible by such architectures.

References

[BRG$^+$89] David L. Black, Richard F. Rashid, David B. Golub, Charles R. Hill, and Robert V. Baron. Translation lookaside buffer consistency: a software approach. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 113–122, 1989.

[CBJ92] J. Bradley Chen, Anita Borg, and Norman P. Jouppi. A simulation based study of TLB performance. In *Proceedings of the 19th International Symposium on Computer Architecture (ISCA)*. ACM, 1992.

[CD97a] M. Cekleov and M. Dubois. Virtual-address caches part 1: Problems and solutions in uniprocessors. *IEEE Micro*, pages 64–71, September 1997.

[CD97b] M. Cekleov and M. Dubois. Virtual-address caches part 2: Multiprocessor issues. *IEEE Micro*, pages 69–74, November 1997.

[CF01] George Candea and Armando Fox. Recursive restartability: Turning the reboot sledgehammer into a scalpel. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS)*, pages 125–130, 2001.

[CLFL94] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and protection in a single-address-space operating system. *ACM Transactions on Computer Systems*, 12:271–307, 1994.

[DF97] Drew Dean and Edward W. Felten. Secure mobile code: Where do we go from here? In *DARPA Workshop on Foundations for Secure Mobile Code*, Monterey, CA, USA, March 1997.

[HEV$^+$98] Gernot Heiser, Kevin Elphinstone, Jerry Vochteloo, Stephen Russell, and Jochen Liedtke. The Mungi single-address-space operating system. *Software: Practice and Experience*, 28(9):901–928, July 1998.

[HH93] Jerry Huck and Jim Hays. Architectural support for translation table management in large address space machines. In *Proceedings of the 20th International Symposium on Computer Architecture (ISCA)*, pages 39–50. ACM, 1993.

[Int00] Intel Corp. *Itanium Architecture Software Developer's Manual*, February 2000. URL http://developer.intel.com/design/itanium/family.

[Int02] Intel Corp. *IA-32 Architecture Software Developer's Manual*, 2002. URL http://developer.intel.com/design/pentium4/manuals.

[JM01] Bruce Jacob and Trevor Mudge. Uniprocessor virtual memory without TLBs. *IEEE Transactions on Computers*, 50:482–499, 2001.

[KCE92] Eric J. Koldinger, Jeffrey S. Chase, and Susan J. Eggers. Architectural support for single-address-space operating systems. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 175–86, 1992.

[KEW$^+$85] Randy H. Katz, Susan J. Eggers, David A. Wood, C. L. Perkins, and R. G. Sheldon. Implementing a cache consistency protocol. In *Proceedings of the 12th International Symposium on Computer Architecture (ISCA)*, pages 276–283, 1985.

[KS02] Gokul B. Kandiraju and Anand Sivasubramaniam. Characterizing the d-TLB behavior of SPEC CPU2000 benchmarks. In *Proceedings of the ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2002.

[KT95] Yousef A. Khalidi and Madhusudhan Talluri. Improving the address translation performance of widely shared pages. Technical Report TR-95-38, Sun Microsystems Laboratories, Mountain View CA, February 1995.

[Lee89] Ruby B. Lee. Precision architecture. *IEEE Computer*, 22(1):78–91, January 1989.

[Lie96] Jochen Liedtke. *On the Realization Of Huge Sparsely-Occupied and Fine-Grained Address Spaces*. Oldenbourg, Munich, Germany, 1996.

[Mit96] John C. Mitchell. *Foundations of Programming Languages*. MIT Press, 1996.

[MSSW94] Cathy May, Ed Silha, Rick Simpson, and Hank Warren, editors. *The PowerPC Architecture: A Specification for a New Family of RISC Processors*. Morgan Kaufmann, 1994.

[Nec97] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119, January 1997.

[QD98] Xiaogang Qiu and Michel Dubois. Options for dynamic address translation in COMAs. In *Proceedings of the 25th International Symposium on Computer Architecture (ISCA)*, pages 214–225, 1998.

[QD01] Xiaogang Qiu and Michel Dubois. Towards virtually-addressed memory hierarchies. In *IEEE Symposium on High-Performance Computer Architecture (HPCA)*, pages 51–62, January 2001.

[Sch94] Curt Schimmel. *UNIX Systems for Modern Architectures*. Addison Wesley, 1994.

[SM99] Alan Skousen and Donald Miller. Using a distributed single address space operating system to support modern cluster computing. In *Hawaii International Conference on System Sciences (HICSS)*, 1999.

[Smi82] Alan Jay Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, 1982.

[SMLE02] Michael M. Swift, Steven Marting, Henry M. Levy, and Susan G. Eggers. Nooks: An architecture for reliable device drivers. In *Proceedings of the 10th SIGOPS European Workshop*, pages 101–107, St Emilion, France, September 2002.

[Tal95] Madhusudhan Talluri. *Use of Superpages and Subblocking in the Address Translation Hierarchy*. Phd thesis, University of Wisconsin-Madison Computer Sciences, 1995. Technical Report #1277.

[TDF90] George Taylor, Peter Davies, and Michael Farmwald. The TLB slice - a low-cost high-speed address translation mechanism. In *Proceedings of the 17th International Symposium on Computer Architecture (ISCA)*, pages 355–363. ACM Press, 1990.

[Tel90] Patricia J. Teller. Translation-lookaside buffer consistency. *IEEE Transactions on Computers*, 23:26–36, 1990.

[TH94] Madhusudhan Talluri and Mark D. Hill. Surpassing the TLB performance of superpages with less operating system support. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 171–182, San Jose, CA, USA 1994.

[TKHP92] Madhusudhan Talluri, Shing Kong, Mark D. Hill, and David A. Patterson. Tradeoffs in supporting two page sizes. In *Proceedings of the 19th International Symposium on Computer Architecture (ISCA)*. ACM, 1992.

[WB92] Bob Wheeler and Brian N. Bershad. Consistency management for virtually indexed caches. In *Proceedings of the 5th International*

Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 124–136. ACM, 1992.

[WCA02] Emmett Witchel, Josh Cates, and Krste Asanović. Mondrian memory protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2002.

[WEG$^+$86] David A. Wood, Susan J. Eggers, Garth Gibson, Mark D. Hill, Joan M. Pendleton, Scott A. Ritchie, George S. Taylor, Randy H. Katz, and David A. Patterson. An in-cache address translation mechanism. In *Proceedings of the 13th International Symposium on Computer Architecture (ISCA)*, pages 358–365, 1986.

[WH00] Adam Wiggins and Gernot Heiser. Fast address-space switching on the StrongARM SA-1100 processor. In *Proceedings of the 5th Australasian Computer Architecture Conference (ACAC)*, pages 97–104, Canberra, Australia, January 2000. IEEE CS Press.

[WS92] John Wilkes and Bart Sears. A comparison of protection lookaside buffers and the PA-RISC protection architecture. Technical Report HPL-92-55, HP Labs, Palo Alto, CA, USA, March 1992.

[WTUH03] Adam Wiggins, Harvey Tuch, Volkmar Uhlig, and Gernot Heiser. Implementation of fast address-space switching and TLB sharing on the StrongARM processor. In *8th Asia-Pacific Computer Systems Architecture Conference (ACSAC)*, Aizu-Wakamatsu City, Japan, September 2003. Springer Verlag.

[WWTH03] Adam Wiggins, Simon Winwood, Harvey Tuch, and Gernot Heiser. Legba: Fast hardware support for fine-grained protection. In *8th Asia-Pacific Computer Systems Architecture Conference (ACSAC)*, Aizu-Wakamatsu City, Japan, September 2003. Springer Verlag.