

Abstract

A report for COMP9221 on the memory hierarchy, focusing on cache implementation. Topics covered include a memory hierarchy overview, cache design overview, cache design considerations and hit and miss latency reduction strategies.

Memory Hierarchy From a Software Perspective

Ian Wienand

April 13, 2006

Contents

1	Memory Hierarchy Overview	2
2	Cache Design	4
2.1	Overview	4
2.2	Cache Organisation	5
2.2.1	Associativity	5
2.2.2	Tags and Indexes	5
2.2.3	Modifying cache parameters	6
2.2.4	Replacement Policy	7
2.2.5	Write Policy	7
2.2.6	Inclusive and Exclusive caches	7
2.2.7	Cache coherency in multi-processors	7
2.3	Cache addressing	7
2.3.1	Physically-addressed cache	8
2.3.2	Virtually-indexed, Physically-tagged	8
2.3.3	Virtually-addressed cache	9
2.3.4	Superset bits	10
2.3.5	Itanium2 - Prevalidated Cache design	12
2.3.6	Trace caches	13
3	Cache Performance Considerations	15
3.1	Miss Penalties	15
3.1.1	Types of misses	15
3.2	Techniques for miss penalty reduction	15
3.2.1	Size and associativity considerations	15
3.2.2	Critical Word First	16
3.2.3	Victim Caches	17
3.2.4	Way Prediction	17
3.2.5	Prefetching	17
A	Cache Hitter	20
A.1	Cache Hitting Program	20
A.2	GCC	20
A.3	ICC	21

Chapter 1

Memory Hierarchy Overview

The speed at which data can move into the processor is a major determining factor over system performance. Unfortunately the faster memory gets the greater the costs in terms of transistor count, complexity, power usage, heat output and (of course) money.

Computer code exhibits two forms of locality

- *Spatial locality* ensures that data within blocks is likely to be referenced together.
- *Temporal locality* ensures that data that is used recently will likely be used again shortly.

These two forms of locality mean that we can gain great benefits by implementing as much quickly accessible memory storing small blocks of relevant information as practically possible. We can then create a hierarchy of cheaper (and hence slower) memories, each storing more information. This is illustrated in Figure 1.1.

The smallest and fastest memory in the processor is the registers, implemented within the CPU core and pipeline. Registers are a very limited resource.

From here we move to *cache* memory; small super-fast memory that keeps blocks of relevant information available for the processor to load into registers quickly. The following paper investigates issues surrounding cache memory. It assumes a familiarity with general virtual memory concepts.

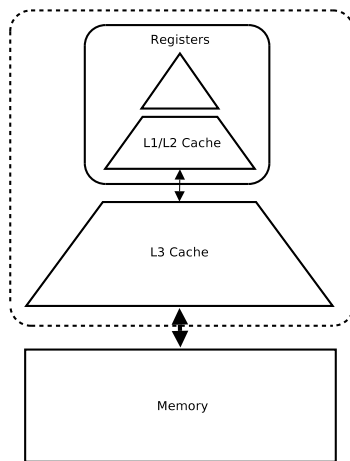


Figure 1.1: The memory hierarchy moves from registers down to main memory. Each level is larger but slower to access. The hierarchy exploits locality to improve overall performance.

Chapter 2

Cache Design

2.1 Overview

The processor cache is second in the memory hierarchy to registers. Processors have a cache hierarchy going down from the smallest, fastest L1 to larger but slower L2 and, for some architectures, a much larger and again slower L3 cache (which in some cases, such as the POWER processor, is connected over a specialised bus).

L1 caches are generally further split into instruction caches and data, known as the “Harvard Architecture” after the relay based Harvard Mark-1 computer. Split caches help to reduce pipeline bottlenecks as earlier pipeline stages reference the instruction cache and later stages the data cache. Apart from reducing contention for a shared resource, providing separate caches for instructions also allows for alternate implementations which may take advantage of the nature of instruction streaming; they are read-only so do not need extensive multi-porting nor need to handle sub-block reads.

Just as virtual memory systems operate on memory in quanta of pages or frames, the hardware cache operates on *cache lines* (sometimes referred to as *blocks*) of a given size. Line sizes vary for each architecture, but are usually within 8B – 128B for a data cache and the same if not a little larger for instruction caches.

Evaluating Montectio Split L2 cache Montectio, the upcoming (as of 2006) Itanium2 processor has a unique large split L2 cache, with a 1MiB L2 I-cache and a 256KiB L2 D-cache. It retains a 16KiB L1 I-cache as per its predecessor.

To understand why the designers might have implemented such a unique L2 design, it is important to consider a number of factors. Firstly is the size of instructions that reside within the cache. In Table 2.1 we compare the instruction size and cache size for POWER5 and Itanium2 processors.

We can see the Itanium2’s smaller L1 I-cache is further penalised by a larger instruction size. Despite the growth in the size of the L2 I-cache, hit latency is maintained at 7 cycles (same as the predecessors unified L2 cache) thanks to the reduced complexity of an instruction cache.

Secondly, we refer to studies that have shown commercial applications (categorised by high I-cache miss rates due to large working sets) are improved by aggressive instruction pre-fetching, but with a unified L2 cache this pre-fetching has a consequence of more L2 cache data misses.

One study implemented a scheme to reduce combined L2 pollution by pre-fetching instructions into a separate buffer which could move the line into L2 very quickly after a hit. We would consider the results seen from this pollution reduction scheme would be somewhat equivalent to results seen from a separate

Processor	Cache Size	Instruction Size	Instructions per KiB	Instructions held in cache
POWER5	64KiB L1D	32 bits	256	16384
McKinley	16KiB L1D	128 bits	192	$\approx 2457^a$
Montectio	1MiB L2D	(3 instructions/bundle)		≈ 157286

Table 2.1: Cache size v. instruction size for POWER5 and Itanium2. McKinley is the present (as of 2006) version of Itanium2, whilst Montectio is upcoming. We can see that the larger instruction size of Itanium2 reduces instructions per KiB of cache, increasing the penalty of a smaller cache.[3].

^aBoth figures assume 20% of instructions are no-ops

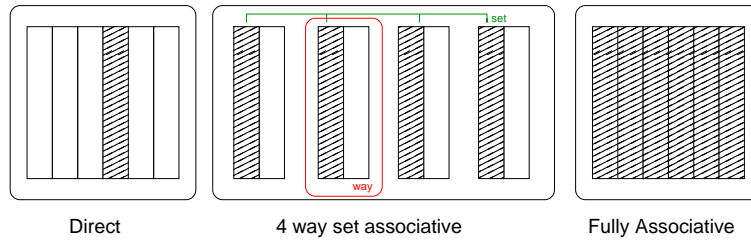


Figure 2.1: A given cache line may find a valid home in one of the shaded entries.

L2 I-cache, which will not pollute the L2 D-cache at all. With less pollution in the combined L2 they saw an increase of 3%-9% over various polluting pre-fetching schemes with various commercial workloads [12]. Quoted but unsubstantiated figures from Intel say there is up to 7% performance improvement from the modified Montectio hierarchy[9], which is consistent.

2.2 Cache Organisation

During normal operation the processor is constantly asking the cache for particular lines, so the cache needs some way to very quickly find if the entry is present or not. If a given cache line can exist anywhere within the cache, the entire cache needs to be searched every time a reference is made to determine a hit or a miss. To keep searching fast this is done in parallel in the cache hardware, but searching every entry is generally far too expensive to implement for a reasonable sized cache.

Thus the cache can be made simpler by enforcing where a particular line must live. This is a trade-off; if the processor can only store a particular line in one location and another line comes along that must also be stored at that location the two will have to fight for the cache entry, even if others are free.

Thus we can generalise on three ways which caches organise where lines are placed within them (see also Figure 2.1).

2.2.1 Associativity

- *Direct mapped* caches will allow a cache line to exist only in a single entry in the cache. This is the simplest to implement in hardware, but may cause overlapping entries to fight for single entries even when others are free.
- *Fully Associative* caches will allow a cache line to exist in any entry of the cache. This avoids the problem with conflicts above, since any entry is available for use. But it is very expensive to implement in hardware.
- *Set Associative* caches allow a cache line to exist in some subset of entries within the cache. The cache is divided into even compartments called *ways*, and a particular line could be located in any way. Thus an *n-way* set associative cache will allow a cache line to exist in any entry of a set sized $total_blocks \bmod n$. This is a relaxation of the direct mapped cache, allowing up to n entries that might share the same location to still reside in the cache together.

Both the direct mapped and fully associative organisations can be considered special cases of a set associative cache.

A one-way set associative cache means that the cache is essentially undivided, so a given line can only exist in one location and it is hence analogous to a direct mapped cache. Conversely, an n -way associative cache with n total entries means a single set makes up the entire cache, and hence allows a line to exist anywhere in the cache and thus is analogous to a fully associative cache.

2.2.2 Tags and Indexes

Caches keep a directory of what lines are in the cache. The cache directory and data may be separate, such as in the case of the POWER5 processor which has an on-core L3 directory, but accessing the data requires traversing the L3 bus off-core. This facilitates quicker hit/miss processing without the other costs of keeping the entire cache on-core.

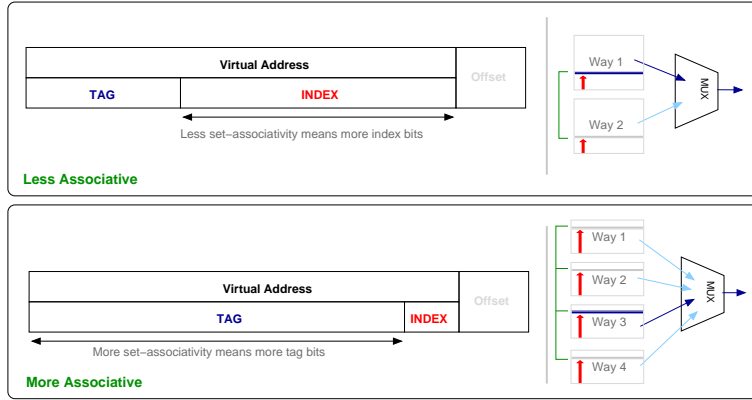


Figure 2.2: Tags need to be checked in parallel to keep latency times low; more tag bits (i.e. less set associativity) requires more complex hardware to achieve this. Alternatively more set associativity means less tags, but the processor now needs hardware to multiplex the output of the many sets, which can also add latency.

We mentioned that in a direct mapped cache, a particular cache line can only go in one location within the cache. This location is the *index* and is simply an offset into the cache directory location that should be checked.

In a set associative cache this is relaxed to be one location within a way, and the entry may exist at this offset in any of the n ways. For a fully associative cache there is no index, as the entry may exist anywhere within the cache.

To distinguish between two addresses that may have the same index and hence map to the same location in the directory, each directory entry is given a *tag* which is sufficient to uniquely identify it. In a direct mapped cache, the tag comparison is very quick as only one location is checked. If the tag matches, then there is a hit, otherwise a miss.

For a set associative cache, each of the ways need to be checked in parallel. The difficulty with this scheme is that the result from each way needs to be multiplexed into a final result that reflects a hit or a miss (2.2; more ways means more data to multiplex, which can increase latency times. This is further exacerbated in a fully associative cache which must multiplex the output of checking every line.

Tag and index bit sizing

We can see the number of bits required for the index as per equation 2.1.

$$index_bits = \log_2 \left(\frac{cache_size}{\frac{associativity}{line_size}} \right) \quad (2.1)$$

Since *cache_size* divided by *line_size* gives the total number of line size entries in the cache, we further divide by *associativity* allowing us enough bits to index into a single set. Each set then checks that index against the tag, in parallel, and the final result is multiplexed together.

2.2.3 Modifying cache parameters

From Figure 2.2 and Equation 2.1 we can see that increasing associativity dedicates more bits to the index and hence less bits to the tag. Increasing associativity involves more on chip resources, and since more sets also means more places for the processor to check for a given tag, the cost to locate a tag within the sets increases as processor has a more difficult task multiplexing the results from the many possible sets.

Conversely less associativity means larger tag sizes. As each cache tag needs to be checked by the CPU to determine if the data is present, the cache tags are normally searched in parallel to minimise hit latency. The trade-off is that parallel searching becomes more expensive in terms of hardware and power consumption for larger tags.

2.2.4 Replacement Policy

When the cache is full, an entry needs to be selected to be removed. Generally, the replacement policy is one of random, least recently used (LRU) or first in, first out (which approximates LRU with less overhead).

LRU generally provides the best performance (e.g. the lowest hits), however for larger cache sizes (256KiB), LRU and random replacement provide approximately the same performance. For cache sizes lower than this FIFO tends to outperform random, with LRU still the most effective policy [5].

2.2.5 Write Policy

When data is written into the cache, a policy decision needs to be made about how that value will be reflected in memory. A *write-through* policy has data updated in the cache and written into the lower level memory simultaneously.

A *write-back* policy writes the updated values only into the cache, then periodically flushes the values back to the lower level memory. The *dirty bit* provides an optimisation for a write-back cache, creating a quick and easy to check flag on a line to see if it needs to be written back to memory or not.

2.2.6 Inclusive and Exclusive caches

If an entry exists in both a higher-level and lower-level cache at the same time, we say the higher-level cache is *inclusive*. Alternatively, if the higher-level cache having a line removes the possibility of a lower level cache having that line, we say it is *exclusive*.

We examine some of the trade-offs below.

2.2.7 Cache coherency in multi-processors

All modern processors need to work in multi-processor environments, which introduce cache coherency issues. Each processor must ensure that its cache is up to date with every other processor, or undefined behaviour can occur.

Common cache coherency protocols use *snooping* to maintain coherency in the system. This is where processors watch for any memory traffic relating to updates of cache lines they hold. If the line is updated, it must be invalidated so the data is re-fetched (certain schemes, such as MOESI, attempt to improve the scalability of this paradigm). Snooping protocols have an effect on how architects might design their cache.

We can see from Table 2.2 that L1 caches are usually *inclusive*, that is all data in the L1 cache also resides in the L2 cache. In a multiprocessor system, an inclusive L1 cache means that only the L2 cache need snoop memory traffic to maintain coherency, since any changes in L2 will be guaranteed to be reflected by L1. This reduces the complexity of the L1 and de-couples it from the snooping process allowing it to be faster.

We can also see that most all modern high-end (e.g. not targeted at embedded) processors have a write-through policy for the L1 cache, and a write-back policy for the lower level caches. There are several reasons for this.

Since in this class of processors L2 caches are almost exclusively on-chip and generally quite fast, the penalties from having L1 write-through are not the major consideration. Further, since L1 sizes are small, pools of written data unlikely to be read in the future could cause pollution of the limited L1 resource.

Additionally, a write-through L1 does not have to be concerned if it has outstanding dirty data, hence can pass the extra coherency logic to the L2 (which, as we mentioned, already has a larger part to play in cache coherency).

2.3 Cache addressing

The address presented to the cache is most always a virtual address; only in special cases will the processor be running in physical mode.

Aliasing Since the cache is passed a virtual address, there is clearly the potential that two virtual addresses may actually refer to the same underlying physical page. When two virtual addresses refer to the same physical page we say that they are *synonyms* (in language, a word having the same meaning as another word) or are *aliases* for a physical page.

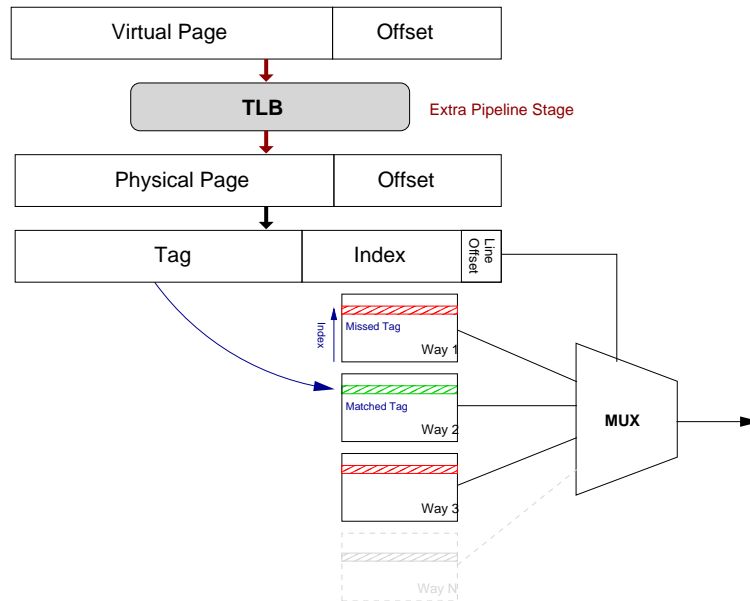


Figure 2.3: In a physically-addressed cache there is an extra step where the TLB converts the virtual address to a physical one. This is usually an extra pipeline stage, with associated penalties such as increase pipeline flush costs and branch mis-prediction costs.

In the best case, synonyms have the potential to reduce the effective size of the cache due to their redundant nature. However of more importance is *coherence* issues that arise from duplicate entries.

Reads clearly need to see the most up-to-date version of data to maintain coherence, and aliases make this problematic. A dirty cache line could be outstanding in a write buffer, meaning that without special checks another aliased address would see a stale copy. The situation becomes more complex in a multiprocessing environment where multiple caches need to maintain coherency between themselves.

A related problem are *homonyms* (in language, a word with the same sound but a different meaning). This is the opposite situation, when the same virtual address in two different addresses spaces refers to different underlying physical pages.

Different cache implementations will need to be aware of, and deal with these problems. Below we examine how the cache can take this virtual address and find a correct cache line for it.

2.3.1 Physically-addressed cache

In a physically addressed cache design the virtual address is translated by the TLB before being requested from the cache (Figure 2.3). This clearly avoids both synonym and homonym problems since virtual addresses are already translated to the canonical physical page for the cache to look for.

The disadvantage of this scheme is the extra latency from requiring the TLB lookup to be complete before requesting the line from the cache. This latency can be hidden with extra processor pipeline stages, however this has disadvantages such as more expensive pipeline flush costs and branch mis-prediction costs, along with power, heat and complexity drawbacks.

Note that for lower (L2 and beyond) levels of cache the TLB overhead is not considered, as the latency for the TLB lookup can be hidden in the increased time to access the lower level caches.

2.3.2 Virtually-indexed, Physically-tagged

As illustrated in Figure 2.4, the lower untranslated bits of the virtual address can be used as the index bits for the cache, allowing the cache to begin index selection within the ways before the TLB lookup completes and returns the tag. This avoids the overhead of waiting for the TLB to complete, giving the cache a “head start” in selecting the index.

However the maximum possible index bits, and thus the way size, are limited to the page size of the architecture. To increase the size of the cache either greater associativity is required (i.e. more sets to

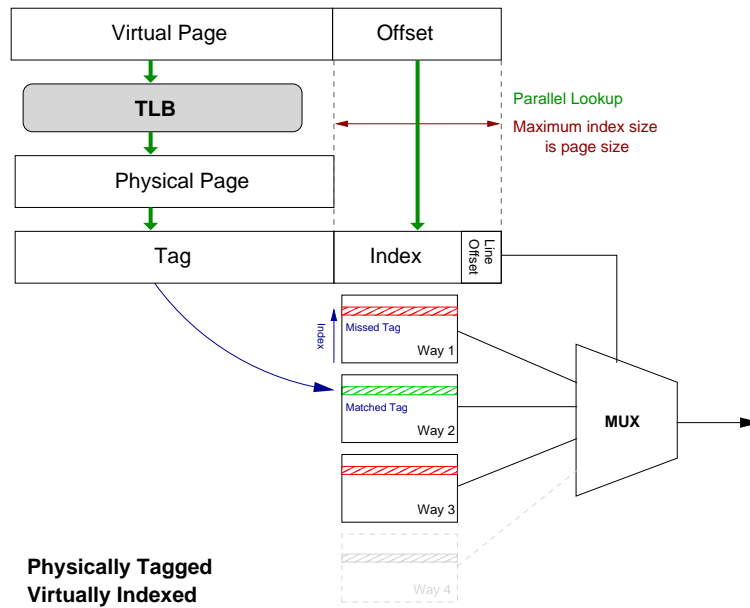


Figure 2.4: In a virtually indexed, physically tagged cache the TLB lookup can happen in parallel with the index selection. The disadvantage is that the index size is limited to the untranslated virtual address bits.

choose from) or a larger system page size. Both have drawbacks: greater associativity causes problems as discussed 2.2 and greater page size can cause greater internal fragmentation.

This scheme removes any synonym and homonym related problems as per the pure physically addressed cache since the operating system will only share on page boundaries (i.e. two processes can share at minimum one page, no lower) and the tags are translated by the TLB.

Due to the limitations of way size being at most the page size and latency considerations with more associative caches, a physically addressed L1 will have to be very small.

2.3.3 Virtually-addressed cache

In an effort to remove the extra translation step incurred with the physically addressed design (as per Figure 2.3), there is the possibility of a purely virtually tagged cache.

The trade off for faster access times to the cache is that you must deal with both synonym and homonym consistency issues.

Synonyms

The synonym problem occurs because two separate virtual addresses map the same physical page. If the cache tag is purely a virtual address then we lose the condition that the tag uniquely identifies a cache line entry, since two virtual addresses (and hence tags) could easily differ but refer to the same underlying cache line.

This scheme has the potential to have redundant data within the cache; two lines may have different tags but contain the same data. This is suitable for read only data, but if one of the duplicated lines is written there must be guarantees that the other copies will be invalidated. The simplest strategy to avoid this is to not allow writes to the cache.

If we were to decide to allow writes, we need to be sure that no other line in the cache is an alias for the address we are writing. This means either doing a translation for every cache line entry, an unappealing prospect, or simply flushing the whole cache. Flushing increases the *cold start* problem where possibly heavily used lines are ejected to maintain consistency.

Whilst write flushing is expensive for a D-cache, this limitation is not so bad for an I-cache implementation since it can be made non-writable. Despite self modifying code being an exceptional case, it must be dealt with.

If the I-cache is non-writable all writes go to the D-cache. To allow for self-modifying code, if the write modifies code the programmer is responsible for ensuring the D-cache and I-cache are made coherent

(D-cache line is flushed to memory and I-cache invalidated as to be reloaded) before executing the code (e.g. the `imb` PAL call on Alpha and `fc.i` instruction on Itanium).

Homonyms

Switching address spaces with a purely virtually tagged cache leads to homonym problems, as the same tag might now refer to different data. One simple approach to solving this problem is again flushing the cache on every context switch, although there are other approaches.

Rather than run each process in its own address space (a *hierarchical model*) a *global model* modifies the system to use a single large(er) address space[13]. We avoid both synonyms and homonyms since no virtual address is used twice.

Single Address Space Operating Systems A single address space operating system (SASOS) avoids aliasing problems by never sharing virtual addresses between processes but rather giving all objects their own place in a very large address space. This can also lead to other interesting properties such as transparent distributed systems and interesting persistence tricks [4]. Objects to be shared are shared through their canonical virtual address, which is the same for all processes.

A similar approach for a traditional operating system is assigning a region of virtual memory to be shared between all processes. This wastes address space as a this region must be dedicated to sharing – in use or not. However it provides a more conceptually simple approach. For example, Windows 95 implements a shared 1GiB shared region between 2–3GiB for all processes, and a similar approach was also taken by OS/2 [6].

ASIDs and Segmentation An *address space identifier* (ASID) can be prepended to the virtual address of each processes address space to create a larger global address space. The disadvantage is that addresses spaces wishing to share virtual addresses now need multiple TLB entries.

A similar approach employed by processors such as POWER, PA-RISC and Itanium is the segment-register approach. Below we examine the Itanium implementation.

On Itanium, the top three bits of a virtual address are an index into one of 8 (e.g. 2^3) *region registers*. Each region register holds a architecturally defined 18-24 bit *region id* (Itanium2 implements only the minimum 18 bits) which is prepended to virtual addresses [10].

This approach allows sharing of address space regions, should the operating system request it via giving two processes the same region id for a particular region.

TLB considerations

In a traditional system, the TLB provides protection information which needs to be referenced to ensure the cache hit is allowed. The goal of a pure virtual cache is to avoid TLB overhead, but protection must be maintained. There are several alternative schemes available to enforce protection without the direct intervention of a TLB such as segmentation, capabilities, storing protection data in the cache tag, protection look-a-side buffers and page-group models of protection (associating protection to a group of pages to be thought of as an “object” as per Itanium2 protection keys) [15]. At this stage, none of these techniques has wide-spread adoption.

2.3.4 Superset bits

We mentioned the problem with physically addressed caches is that the way size is limited to the base page size, and increasing associativity is expensive. A purely virtual cache has penalties that make it unsuitable for a D-cache implementation.

As a way around these problems, just some of the translated bits of virtual address can be used in the index. This allows the way size to be larger than the page size, but introduces some aliasing problems. We refer to these shared bits as *superset* bits[15].

The situation is illustrated in Figure 2.5. The full virtual page is translated to a physical address, meaning that tags are a unique identifier for a line. However there is the possibility of aliasing with the shared index bits. The situation may arise where the same tag is present at multiple indexes within the cache. Each n shared bits between the virtual address and offset introduces 2^n possible aliased index locations. For small n this is manageable, as we see below.

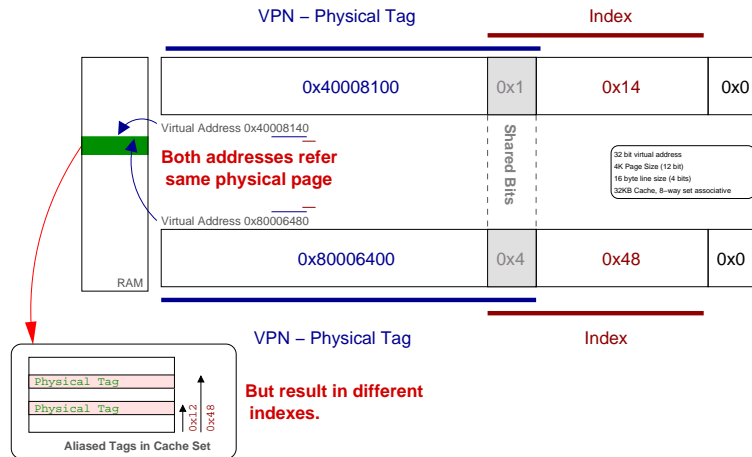


Figure 2.5: An example of a synonym problem. By using some of the VPN bits as an index into the cache set we are exposed to aliasing. Aliasing can occur whenever the shared bits between the VPN and index (illustrated in grey) are not the same value, since two VPNs may be translated into the same physical address but will have different index values. To the operating system these refer to the same physical page, but as far as the cache is concerned these addresses are two different pages!

Forcing Alignment

Synonyms occur when different virtual addresses share the same physical page. In a physically addressed cache no other entry is a synonym, in a purely virtual cache every entry in the cache may be a synonym. When there are superset bits, any entry where the superset bits differ may be an synonym.

Thus the operating system can avoid synonym problems by ensuring that virtual addresses that share physical pages always have the same superset bits. To phrase this differently, the index bits must be guaranteed to be the same for virtual addresses which alias the same page. The trade-off is that shared data must now be aligned on larger boundaries.

In the example from Figure 2.5 an 8 way set-associative cache of 32KiB with 16 byte lines is being used on a system with 4KiB pages. Each way contains 4096 lines, thus requires 12 bits to index. The 16 byte line requires another 4 bits to index, leaving only $12 - 4 = 8$ bits from from the page as an index. Thus we need to overlap an extra 4 bits from the virtual page number to use as the way index. If two addresses were to share a physical page without having a different index, they would need to ensure that the extra 4 superset bits are the same in each address. This means that the addresses must be aligned on a bigger boundary, in this case $12 + 4$ bits or a 64KiB boundaries.

The fact that arbitrary virtual addresses can not share an underlying physical page does not turn out to be a large constraint for applications, who largely do not care about the alignment of shared data (they want to share it, but don't really care about the virtual address it is located at).

The operating system, however, utilises aliasing to implement *copy on write* schemes for shared data. As the application may not even be aware that copy on write is being implemented it may not obey alignment rules.

Whilst the underlying operating system can attempt to choose correctly aliased addresses as much as possible, there will be cases where this will not be possible so schemes are required to ensure correctness.

Hardware Anti-aliasing

The hardware can be aware of its limitations and check that addresses are not aliased in the cache.

MIPS R10000 Implementation The MIPS R10000, released in 1995, has a 32KiB two way set-associative cache with a 32 byte line size. Equation 2.1 tells us we require 9 bits (to index 512 entries), plus 5 bits for each byte of a 32 byte line, giving a total of 14 bits taken up by indexing. However, the MIPS R4000 supports a base page size of 4KB, meaning the untranslated offset portion of a virtual address is only 12 bits long ($2^{12} = 4096$).

As we can see in Figure 2.5 sharing part of the translated address as an index allows the possibility of synonyms within the cache; that is entries with the same physical tag at different indexes. This situation

may arise when the overlapped bits do not hold the same value. The R10000 handles this by making the L2 cache inclusive of the L1 cache and storing the superset bits in the L2 cache tag [16].

For two aliased virtual addresses VA1 and VA2, either can make it into the L1 cache first. However, when the second, aliased address is translated it will index into a different entry and cause an L1 miss. This will be intercepted by the L2 cache, which will get a tag **miss** since the L2 tag is increased to have the extra bits. At this point the regular conflict miss logic takes over, ejecting the existing L1 entry. This aliased addresses are rare, this method provides a good trade off between correctness and the hardware limitations of cache design.

Alpha 21264 The Alpha 21264 has a 64KB two-way set associative data cache with a 64 byte line size. Equation 2.1 tells us we require 9 bits for index, plus 6 bits for each byte of a 64 byte line size, giving a total of 15 bits. With 8KB pages, which take 13 bits this leaves 2 bits of overlap, or four possible locations aliased locations within the cache (XXX: HP bug, says 8? pg. 445).

At this point the exact mechanism the 21264 uses remains unclear. Some suggest that on a miss the other sets are checked to see if they also map to the physical address, and are then invalidated [5] whilst the architecture does not go into detail other than to say that only one address is kept in the cache at any one time [2].

Reverse Maps

Reverse maps work by keeping a directory of synonyms within the cache, and ensuring that two entices never overlap. Two schemes are called *back pointers* and *dual directories* (covered in depth elsewhere [14]).

2.3.5 Itanium2 - Prevalidated Cache design

Itanium implements an innovative scheme called a “prevalidated tag”. This design attempts to remove latency penalties that occur from having to a physically tagged cache, without the disadvantages of a virtually tagged cache.

In a traditional cache, the “critical path” for a cache hit requires an underlying sequence of four operations

1. A virtual tag is presented to the TLB, which converts this to a location in the TLB memory where the the physical tag can be found.
2. The physical tag is retrieved from the TLB memory
3. This physical tag is compared to find a match within the cache.
4. The match on this tag then allows the processor to select the correct data line from the cache.

A prevalidated cache [8, 1] attempts to remove the overhead of retrieving the physical tag from TLB memory by tying the L1 cache and TLB very closely together. Rather than the cache being tagged with the physical tag, it is tagged with a TLB location.

By ensuring that the cache way size is small enough that it does not need more index bits than provided by the minimum page size (4KiB), we can ensure that the index bits will never alias and hence index selection can start early.

To keep the processor logic simple (and hence, fast) a *one-hot* (also called *unary*) encoding is used for the tag. This trades information density for speed by having a bit field vector where only one bit is ever set (e.g. to represent 10 states, 10 bits are required). The upside is that hit detection logic is extremely simple, simply consisting of a logical AND of vectors (see Figure 2.6).

A prevalidated cache can be tuned for very low latency, but will place some restrictions on the architecture.

The TLB tied to the prevalidated cache tags needs to be very fast and hence is very small, and one way to do this is to not keep translations for writable data. Stores necessarily require finding a physical address (the process we are trying to avoid) also have the potential to cause thrashing in the small L1 TLB.

Thus store logic can be moved to a slower, larger TLB. This is a suitable tradeoff, as latencies when writing are greater and so the prevalidation benefits are less. However, additional logic in a lower level TLB needs to be integrated to handle write translations.

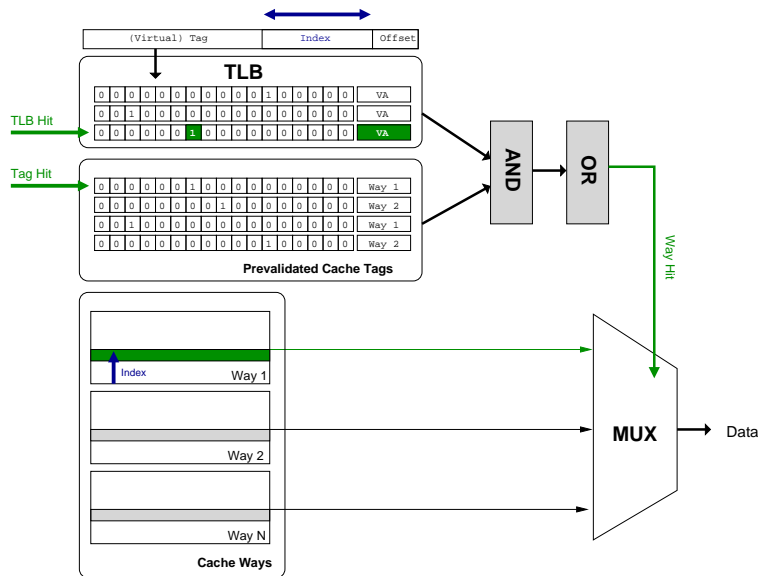


Figure 2.6: Prevalidated tags are a unique cache organisation scheme for the Itanium processor. The TLB stores a “one hot” vector for each entry. The cache is tagged by these vectors, facilitating a quick and simple way hit detection path.

Secondly is the dis-parity between the minimum size of a page and the size of a cache line. With a 4K page and a large 128 byte cache line, invalidating one TLB entry will invalidate 32 cache lines. This could cause significant cache thrashing. Invalidation may also be more difficult as the cache does not contain a tag giving the physical location; only a TLB reference.

2.3.6 Trace caches

Instruction streams do not map well to the block/line paradigm, as they constantly branch (and rarely on block boundaries!). Branch instructions typically come along every 5 or 10 instructions, even a 64-byte block can store up to 20 x86 instructions, leaving considerable waste. In an effort to overcome this, a *trace cache*[11] is an instruction cache that stores instructions across branch boundaries in a contiguous block.

This provides for much greater density in the instruction cache, at the expense of much more complicated caching logic.

The trace cache stores instructions once retired from the processor pipeline. As instructions are

Feature	Alpha 21264 (EV6)	Itanium2	Niagara	Power5	Pentium 4 EE (3.73Ghz)
L1D Size	64 KiB	256 KiB	8 KiB	32 KiB	16 KiB (Trace)
L1D Associativity	2 way	4 way	4 way	4 way	4 way
L1D Store Policy	Write-Through	Write-Through		Write-Through	
L1D Line Size	32 bytes	64 bytes	16 bytes	128 bytes	64 bytes
L1I Size	64KiB	16KiB	16KiB	64KiB	12K μ ops
L1I Associativity	2-way (predicted)		Direct Mapped	2-way	
L1I Line Size	32 bytes	64 bytes	32 bytes	128 bytes	
L1 Latency	2 cycle	1 cycle			4 cycles
L2 Size	1MiB-16MiB Off-Chip	256KiB	3MiB (shared)	1.9MiB	2MiB
L2 Line Size	64 bytes	128 bytes	64 bytes		64 bytes
L2 Mapping	Direct	8-way set	12-way set	10-way set	8-way set
L2 Write Policy	Write-Back	Write-back	Write-back	Write-Back	Write-back
L2 Latency	12 cycles	5 cycles	21 cycles	12 cycles	9-20 cycles
L3 Size		3MiB+		36MiB (off chip)	
L3 Line Size		128 bytes		256 bytes	
L3 Associativity		12-way		12-way	
L3 Write Policy		Write-back		Write-back	
L3 Latency		14 cycles		123 cycles	

Table 2.2: A small survey of cache properties of some common high-end processors

retired, they are put into a trace cache “line”. The trace cache follows branches, so unused instructions are omitted from the cache. The trace cache index is based on the program counter and a branch prediction index. As the processor executes, if at a particular point there is a hit from the current program counter and current branch prediction registers, the data is retrieved from the trace cache.

Since the data is indexed by both program counter and branch prediction counters, there can be duplicated data in the cache as common program counter start points taking different branches are kept separately, possibly increasing capacity misses.

Chapter 3

Cache Performance Considerations

3.1 Miss Penalties

There is an inevitable latency involved when requested data is not in the cache and needs to be retrieved from lower in the memory hierarchy. By minimising this time we can increase system performance.

Previously we have been examining ways to reduce the amount of time taken when data is in the cache. Below we discuss dealing with misses in the cache.

We have already examined a major component of miss penalty reduction in common usage – the multi-level cache. The larger L2 cache is slower, but is still faster than going further down the memory hierarchy. It should be self-evident that whenever we impose a faster upper layer within the memory hierarchy our overall performance will increase.

A “rule of thumb” for cache implementation is that the L1 cache speed influences the maximum clock speed of the processor, whilst the lower level caches influences the miss penalties of the L1 cache [5]. We examined an example of the importance of this relationship in Section 2.3.5 with the Itanium2’s L1 high speed prevalidated cache design.

Larger L2 and L3 caches tend to get many fewer hits than misses. Thus techniques for reducing miss latency, such as larger block sizes and higher associativity are more important in the outer cache levels.

3.1.1 Types of misses

Misses in the cache can be broadly divided into four different types.

1. *Compulsory* misses occur since the data has to move into the cache at some point.
2. *Capacity* misses arise when the working set outgrows the cache size and the processor misses on data that has been ejected from the cache.
3. *Conflict* misses occur in direct-mapped or set-associative caches when the processor misses on data which has been ejected from a full set
4. *Coherency* misses occur in a multi-processor system when a cache line must be invalidated due to another processor updating it.

3.2 Techniques for miss penalty reduction

3.2.1 Size and associativity considerations

Compulsory hit rates are unavoidable, and are generally small for a long running program. One method to decrease compulsory hits is to increase the line size; this brings advantages from spatial locality. However, if the cache size remains the same increased block size may lead to increased capacity misses and internal fragmentation may lead to wasted space and capacity misses. The miss penalty will also be higher as more data needs to be moved between lower layers and the cache, although in a low-latency, high bandwidth situation this may be more appropriate. (XXX HP bug: page 427 doesn’t list 1K cache, 428).

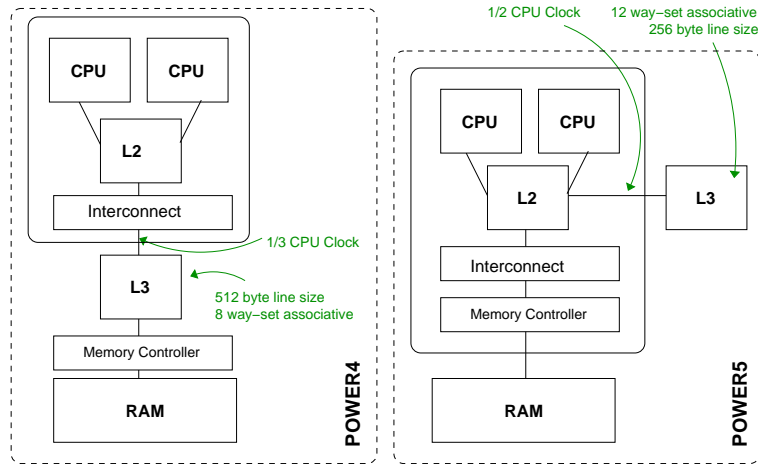


Figure 3.1: A view of the evolution of the POWER memory hierarchy [7].

Clearly the easiest way around capacity misses is to increase the size of the cache. This is of course not without implications; a larger cache will reduce capacity misses issues but may increase conflict misses or possibly result in larger line sizes. As on-chip real estate is expensive, some processors implement an off-chip cache. For example the Alpha 21264 implemented an off-chip L2 cache ranging from 1MB through 16MB.

Conflict misses give cache designers more room to move in implementation terms. Increasing associativity of caches allow more placement choices for blocks, so a more associative cache will reduce conflict misses compared to a less (or worse, direct mapped cache).

Studies have shown that for caches up to around a 512K an 8-way associative cache provides only little benefit over a fully associative cache [5]. Looking at associativity levels from some modern processor designs suggests a slightly higher associativity for larger (megabyte and greater) is required.

Size and associativity in the POWER processor

POWER is IBM's flagship processor. Figure 3.1 shows the evolution of the POWER memory hierarchy from the POWER4 to POWER5.

The POWER4 has a 32MB off chip L3 cache, connected via a dedicated bus as a buffer to main memory. It is organised as a 8-way set associative cache, and has a relatively long latency of a 123 cycles. To contrast, the POWER5 moves the memory controller inside the processor package and connects the 36MB, 12-way set associative L3 cache directly to the L2 cache via a "back side bus", reducing latency to 87 cycles [3]. In both cases the L3 cache is a victim cache for the L2 cache (i.e. stores lines kicked out of the L2 cache), hence the L2 cache is not inclusive. This suggests the associativity need not be closely tied.

Line size was also decreased from 512 bytes to 256 bytes. We can theorise that the lower spatial locality from a smaller line size is amortised by the lower latency, and hence has a lower miss penalty.

3.2.2 Critical Word First

Latest Itanium2 processors claim a memory bus bandwidth of 8.5GiB/s, and run at a clock speed of 1.5Ghz. Equation 3.1 shows that with a 128 byte cache line, even ignoring initial latency in asserting bus lines, etc, it still takes a considerable 21 processor cycles to move a 128 byte block from main memory.

$$\frac{128}{\left(\frac{1}{1500000000}\right) * (8.5 * 1024 * 1024 * 1024)} = 21.03 \quad (3.1)$$

The maximum value a register requires is 8 bytes (64 bits). If the processor brings in this word to the CPU without waiting for the entire cache line to be available, it is called *critical word first*. In the example above, the processor can get started after the initial latency period plus the few cycles it takes to move the first 8 bytes into the cache. If the processor brings in the line in sequential order, but returns the word when it is available it is called *early restart*. In that case the processor must still wait, but if the required word is at the start of the block it will not incur the penalty of the whole read.

This reduces miss penalty for access to random blocks within a cache line. The improvement for streaming data is less, as the initial latency is a large part of the overall transfer time, which can not be avoided. Despite this, some of the cache latency can be hidden if the word is returned early (somewhat similar to prefetching, see below).

3.2.3 Victim Caches

When an entry is removed from the cache, we say it is a *victim*. Studies have shown that recently victimised pages are often quickly used again, so a small, fast, fully associative buffer for recent victims can considerably improve performance.

A victim cache allows you to get close to LRU performance with a FIFO set organisation since the penalty for getting a line back from the victim cache is low. For example, if the “first in” entry is the most used but is ejected, it should be retrieved to the back of the FIFO from the victim cache quickly and be ready for re-use. If the entry really is not used, it will fall out of the victim cache. This way you get the benefit of less logic required to maintain true LRU, but LRU like performance.

3.2.4 Way Prediction

Way prediction keeps extra bits in the cache to predict which set the cache line will be in. The multiplexer can be therefore set early to select the block from the predicted set whilst the tag comparison is being done. If the tag matches, latency is reduced. However, if the way is mis-predicted the process must be “recycled” and complete in the traditional way. The Alpha 21264 uses way prediction in it’s two-way set-associative I-cache; when correct hit latency is 1 cycle and when not correct hit latency is 3 cycles, and is correctly predicted 85% of the time [5].

3.2.5 Prefetching

Non-blocking caches

With a pipelined processor, the processor can continue fetching instructions whilst a cache miss is in progress. If a cache miss can satisfy hits whilst it is waiting for data to complete a miss it is called a *non-blocking cache*.

Prefetching

Prefetching refers to bringing information into the cache before the processor requests it. Prefetching works with a pipelined processor by hiding cache latency; as the processor is continuing with other work the cache subsystem can be bringing in data it predicts it will require soon.

Often prefetching is done by hardware; for example when the I-cache misses two blocks (rather than just the requested block) are fetched; one into the I-cache and one into an *instruction stream buffer*. The buffer is like a reverse victim cache, and is checked before requesting the data from lower levels.

Software can also issue prefetch requests if it is aware that it will need some data in the near future.

Prefetching Example

We can analyse how prefetching is a powerful way to hide latency using the Itanium2 and its powerful performance monitoring facilities. Appendix 3.2.5 shows a code listing for a cache stressing program. It first initialises then walks a large array with a cache line stride.

In Table 3.2.5 we show results from analysis of versions compiled with the GNU compiler (GCC) and the Intel Compiler (ICC), the latter scheduling prefetching while walking the array.

We can see that the total number of cache misses close to the range we expect for the Itanium2 64 byte L1D cache line for both programs (i.e. $\frac{200000000}{64} = 12,500,000$); this is as we expect since they are ultimately walking the same data running on the same machine.

We then analyse the `BE_EXE_BUBBLE_GRALL` event to see where backend stalls are holding up the pipeline; here we see a drastic difference between the prefetching ICC and the non-prefetching GCC. To be sure, we check that these stalls are not coming from register to register dependencies (e.g. data hazards). The architecture manuals tell us that `BE_EXE_BUBBLE_GRALL - BE_EXE_BUBBLE_GRGR` is the stalls introduced by waiting for the memory subsystem to deliver data.

In fact, on the 900Mhz system this was run on we can see that we approximately made up for the entire runtime difference with these stalls ($\frac{737891717-129629838}{900000000} = 0.675s \approx 1.507 - 0.824 = 0.683s$).

Metric	ICC	GCC	Description
Run Time (seconds)	0.824	1.507	Program execution time
L1D_READ_MISSES_ALL	12,514,832	12,505,200	L1 Data Cache Read Misses
BE_EXE_BUBBLE_GRALL	129,629,838	737,891,717	Full Pipe Bubbles in Main Pipe due to Execution Unit Stalls
BE_EXE_BUBBLE_GRGR	0	0	Back-end was stalled by exe due to GR/GR dependency

As we can see in the extract below, the prefetching is done by the `lfetch.nt1` instruction in the second bundle of the modulo scheduled loop (which rotates both general and predicate registers for you on the `br.ctop` instruction). This instruction requests the upcoming data be moved into the L2 and L3 cache.

```

4000000000000940:      [MII] (p16) ld4 r32=[r3],64
4000000000000946:              (p17) add r34=r35,r33
400000000000094c:              nop.i 0x0
4000000000000950:      [MMB] (p16) lfetch.nt1 [r2],64
4000000000000956:              nop.m 0x0
400000000000095c:              br.ctop.sptk.few 4000000000000940 <walk+0x80>;

```

Since both bundles can be executed by the processor at once, the prefetch gives us a “head start” on having the data ready for the next iteration of the loop.

Bibliography

- [1] D.E. Bradley, P. Mahoney, and B. Stackhouse. The 16kb single-cycle-read-access cache on a next-generation 64b itanium microprocessor. In *Proceedings of the International Solid State Circuits Conference*. IEEE, February 2002.
- [2] Compaq. *Alpha 21264 Microprocessor Data Sheet*, 1999.
- [3] Paul DeMone. Sizing up the super heavyweights. *RealWorldTech*, September 2004. ”<http://h21007.www2.hp.com/dspp/files/unprotected/Itanium/sizingsuperheavys.pdf>”.
- [4] Gernot Heiser, Kevin Elphinstone, Jerry Vochtelo, Stephen Russell, and Jochen Liedtke. The Mungi single-address-space operating system. *Software Practice and Experience*, 28(9):901–928, 1998.
- [5] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 3rd edition, 2003.
- [6] Bruce L Jacob. Segmented addressing solves the virtual cache synonym problem. Technical report, University of Maryland, December 1997. UMD-SCA-97-01.
- [7] Ron Kalla, Balaram Sinharoy, and Joel M. Tandler. IBM Power5 chip: A dual-core multithreaded processor. 24(2):40–47, March/April 2004.
- [8] Terry L Lyon. Method and apparatus for updating and invalidating store data. US Patent 6920531, 2005. Assignee: Hewlett-Packard Development Company, L.P., Houston, TX(US); filed Nov 4, 2003.
- [9] C. McNairy and R. Bhatia. Montecito: a dual-core, dual-thread itanium processor. *IEEE Micro*, 25(2):10–20, 2005.
- [10] David Mosberger and Stéphane Eranian. *IA-64 Linux Kernel: Design and Implementation*. Prentice Hall, 2002.
- [11] Eric Rotenberg, Steve Bennett, and James E. Smith. Trace cache: A low latency approach to high bandwidth instruction fetching. In *International Symposium on Microarchitecture*, pages 24–35, 1996.
- [12] Lawrence Spracklen, Yuan Chou, and Santosh G. Abraham. Effective instruction prefetching in chip multiprocessors for modern commercial applications. *IEEE Int. Symp. High-Performance Computer Architecture*, pages 225–236, 2005.
- [13] 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 System (ASPLOS)*, volume 27, pages 124–136, New York, NY, 1992. ACM Press.
- [14] Adam Wiggins. A survey on the interaction between caching, translation and protection. Technical Report UNSW-CSE-TR-0321, August 2003.
- [15] Adam Gordon Wiggins. *Voodoo: Towards scalable, fine-grained memory access control*. PhD thesis, UNSW, 2006. Draft Version.
- [16] K. C. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, 16(2):28–40, April 1996.

Appendix A

Cache Hitter

A.1 Cache Hitting Program

```
#include <stdio.h>
#include <unistd.h>

#define N 200000000

int array[N];

int walk(void)
{
    int i,sum;

    for (i=0; i < N; i+=16)
        sum += array[i];
    return sum;
}

void init(void)
{
    int i;

    int skip = getpagesize() / sizeof(int);

    for(i=0; i < N; i+=skip)
        array[i] = i;
}

int main(void)
{
    init();
    return walk();
}
```

A.2 GCC

```
4000000000000720 <walk>:
4000000000000720:      [MMI]      nop.m 0x0
4000000000000726:      addl r15=160,r1
400000000000072c:      mov.i r2=ar.lc
4000000000000730:      [MLX]      nop.m 0x0
4000000000000736:      movl r14=0xbebc1f;;
400000000000073c:
```

```

4000000000000740:      [MMI]      nop.m 0x0
4000000000000746:      nop.m 0x0
400000000000074c:      mov.i ar.lc=r14;;
4000000000000750:      [MMB]      nop.m 0x0
4000000000000756:      ld4 r14=[r15],64
400000000000075c:      nop.b 0x0;;
4000000000000760:      [MIB]      nop.m 0x0
4000000000000766:      add r8=r8,r14
400000000000076c:      br.cloop.sptk.few 4000000000000750 <walk+0x30>;;
4000000000000770:      [MIB]      nop.m 0x0
4000000000000776:      mov.i ar.lc=r2
400000000000077c:      br.ret.sptk.many b0;;

```

A.3 ICC

Note the use of the `lfetch.nt1` to prefetch cache lines.

```

40000000000008c0 <walk>:
40000000000008c0:      [MMI]      alloc r19=ar.pfs,8,8,8
40000000000008c6:      addl r17=64,r1
40000000000008cc:      mov r15=-1
40000000000008d0:      [MLX]      mov r18=0
40000000000008d6:      movl r16=0xbcbc20;;
40000000000008dc:
40000000000008e0:      [MII]      ld8 r3=[r17]
40000000000008e6:      mov r20=pr
40000000000008ec:      adds r14=1,r15
40000000000008f0:      [MMI]      mov r34=0;;
40000000000008f6:      adds r2=2304,r3
40000000000008fc:      mov.i r24=ar.lc
4000000000000900:      [MII]      nop.m 0x0
4000000000000906:      sxt4 r11=r14;;
400000000000090c:      mov.i ar.ec=2
4000000000000910:      [MMI]      add r10=r16,r11;;
4000000000000916:      adds r9=-1,r10
400000000000091c:      mov pr.rot=0x10000
4000000000000920:      [BBB]      nop.b 0x0
4000000000000926:      nop.b 0x0
400000000000092c:      nop.b 0x0;;
4000000000000930:      [MII]      nop.m 0x0
4000000000000936:      sxt4 r8=r9;;
400000000000093c:      mov.i ar.lc=r8;;
4000000000000940:      [MII] (p16) ld4 r32=[r3],64
4000000000000946:      (p17) add r34=r35,r33
400000000000094c:      nop.i 0x0
4000000000000950:      [MMB] (p16) lfetch.nt1 [r2],64
4000000000000956:      nop.m 0x0
400000000000095c:      br.ctop.sptk.few 4000000000000940 <walk+0x80>;;
4000000000000960:      [MII]      mov r8=r35
4000000000000966:      mov.i ar.lc=r24;;
400000000000096c:      mov pr=r20,0xfffffffffff003e
4000000000000970:      [MIB]      nop.m 0x0
4000000000000976:      nop.i 0x0
400000000000097c:      br.ret.sptk.many b0;;

```