The state of the art in

Cache Memories and Real-Time Systems

MRTC Technical Report 01/37

Filip Sebek

Department of Computer Engineering
Mälardalen University
Västerås, Sweden

Presented 2001-09-28, revision 2nd October 2001

Abstract

The first methods to bound execution time in computer systems with cache memories were presented in the late eighties — twenty years after the first cache memories designed. Today, fifteen years later, methods has been developed to bound execution time with cache memories . . . that were state-of-the-art twenty years ago.

This report presents cache memories and real-time from the very basics to the state-of-the-art of cache memory design, methods to use cache memories in real-time systems and the limitations of current technology. Methods to handle intrinsic and extrinsic behavior on instruction and data caches will be presented and discussed, but also close issues like pipelining, DMA and other unpredictable hardware components will be briefly presented.

No method is today able to automatically calculate a safe and tight Worst-Case Execution Time (WCET_C) for *any* arbitrary program that runs on a modern high-performance system — there are always cases where the method will cross into problems. Many of the methods can although give very tight WCET_C or reduce the related problems under specified circumstances.

Mälardalen Real-Time Research Centre
Department of Computer Engineering
Mälardalen University
Västerås, Sweden

This document was written in the text editor Emacs 20.7.1, typesetted in LaTeX, spell and grammar checked in Microsoft Word 2000 and compiled with MikTeX 2.1. Figures were created by `gnuplot 3.7.1`, `gpic` by B.W. Kernighan and `dot` from AT&T Bell labs.

Contents

Chapter 1

Introduction to cache memories

When the gap between CPU and memory speed has increased as long as computers has been made by semiconductors (Figure 1.1), the idea of a memory hierarchy with cache memories that increase performance substantially, is quite common knowledge. The knowledge about how cache memories actually works is although not that common even if it has been described in many text books [HP96] [Ekl94] that has been used in undergraduate courses. This report will give a brief summary on how cache memories works to be complete.

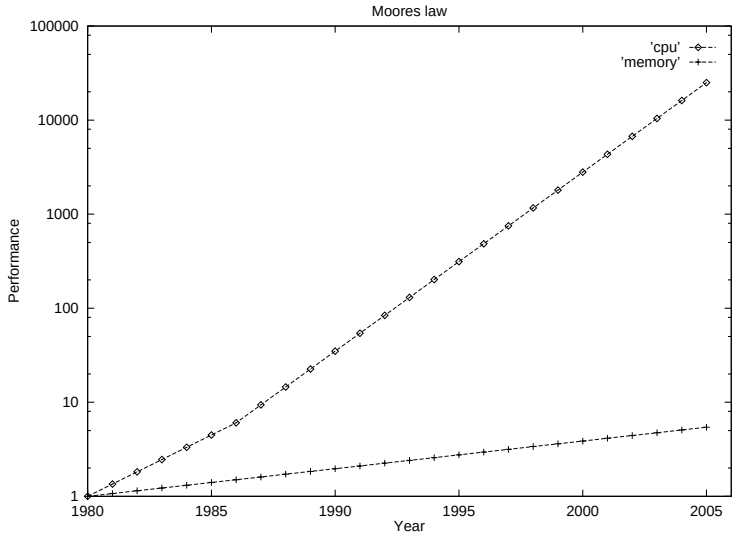

Figure 1.1: Moore's law{indexMoore's law shows that technology enhancements has been put into performance on the CPUs and size on memory. The performance gap widens . . . From [HP96]

1.1 Locality

One fundament of cache memories is *locality* that either can be temporal or spatial.

- *Temporal locality* (also called *locality in time*) concerns time. If a program use an address the chance is bigger to use it in the near future than an arbitrary other address.

- *Spatial locality* (also called *locality in space*) states that items that are close to each other in address space tend to be referred close in time too.

The statements above builds on the fact that instructions are formed in sequences and loops, and that data is often allocated as stacks, strings, vectors and matrices. Sometimes also *sequential locality* is mentioned when discussing locality; memory locations are accessed in order in some types of data accesses as they are for instance in instructions streams and disk accesses. Sequential locality is often defined as a subset of the spatial locality definition.
With the spatial behavior of programs in mind, much sense would be to load more data from the underlying hierarchic memory at once to take advantage of greater bandwidth and long latency that characterizes primary memory. This chunk of data is called *cache line (line)* and is the smallest piece of data a cache handles. There physical location in the cache memory where the line is stored is called a *cache block (block)*. In most cases no distinction is however made between "block" and "line" and are often referred as equivalents.

1.2 Cache basics

1.2.1 Placement

A cache memory can be viewed as a hardware implementation of a hash table where the primary memory address is the key. A straight forward method is to have just one distinct place for a block in the cache — *direct mapped* . The mapping is usually

$$(Block\ address)\ MOD\ (Number\ of\ blocks\ in\ cache)$$

that can be illustrated with the following example:
The cache size is 512 blocks and a cache block is 16 bytes (see Figure 1.2).. In this case the user is interested to access all bytes independently which means we have to use $\log_2 16 = 4$ bits (more realistic is to read 16, 32 or 64 bit-words at once since 8-bits machines are rare now days). If the address to the primary memory is $\$012345ab_{HEX}$ then the 4 least significant bits can be ignored since they are used to address a byte in the block.

$$\$012345a_{HEX}\ MOD\ 512_{DEC} = 145_{HEX} = 325_{DEC}$$

The performance of a direct mapped cache will be analyzed in section 1.4, but as it will be shown that there are some drawbacks of the direct mapped concept, also two other common implementations must be presented:

- If the block can be found and placed anywhere in the cache, the cache memory is said to be *fully associative*.

- A cache memory that can place a block in a set of places is called *set-associative cache memory* . The cache is then organized in *ways* (sometimes also called *bank*) where 2,4 or 8 ways are the most common. This means that a direct mapped cache is a 1-way set associative cache, a fully associative cache memory have one set and that a direct mapped cache with 128 sets is equal in data size with a 4-way set associative cache memory with 32 sets if the block size is the same. To calculate where a which set of ways a block can be placed the formula must be modified to

  $$(Block\ address)\ MOD\ (Number\ of\ sets\ in\ cache)$$

  For an illustration of the set associative cache memory see (Figure 1.3).

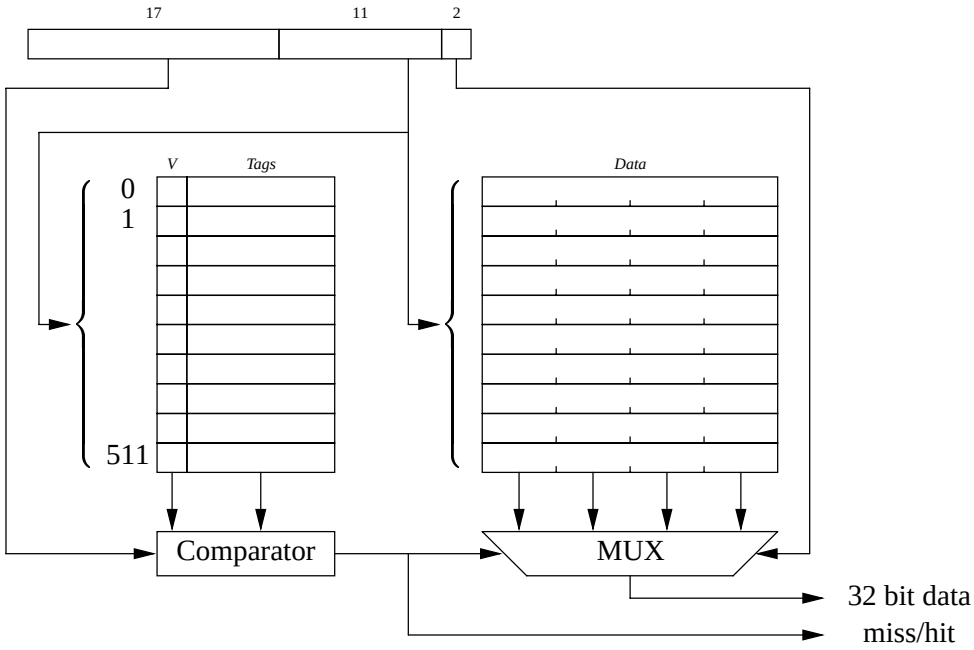

Figure 1.2: Example of a 8 kB direct mapped cache memory with 512 sets and a block size of 16 bytes. The CPU always reads 4 bytes when addressing and is able to address 1 GB (2^{30}) of primary memory address space.

1.2.2 Probing

To find out if a block is in the cache two pieces of information are necessary; a *valid-* or *invalid-flag* and a *tag* for each block.
A tag can be any bit-pattern as long as it is unique in the set. An easy solution is to let the tag be the primary address, but since the least significant bits that address in the block and map into the sets would be redundant information, these bits can be ignored.
When computer system starts, the cache memory is reseted (also called *flushed*) and all blocks are marked as invalid. When a set is addressed in the cache probes (compares) all valid tags in the set and if a comparison is equal in one of the ways we get a *hit* and the data is transferred in the fast manner the cache is supposed to work like. Invalid tags will never result in a hit. If all tags in the set are unequal to the comparing one, the cache memory indicates a *miss* and the correct block must be loaded from the underlying memory (primary memory, level-2 cache etcetera). The new loaded block will be marked as valid.

Cache misses can occur for three reasons – also mention as the “three-C” [HP96].

- Compulsory – the line is not in the cache since the associated blocks are empty (invalid). These misses are also called *initial* or *transient*.
- Conflict – the line is not in the cache and all blocks associated to the set are being used (valid) by other lines.

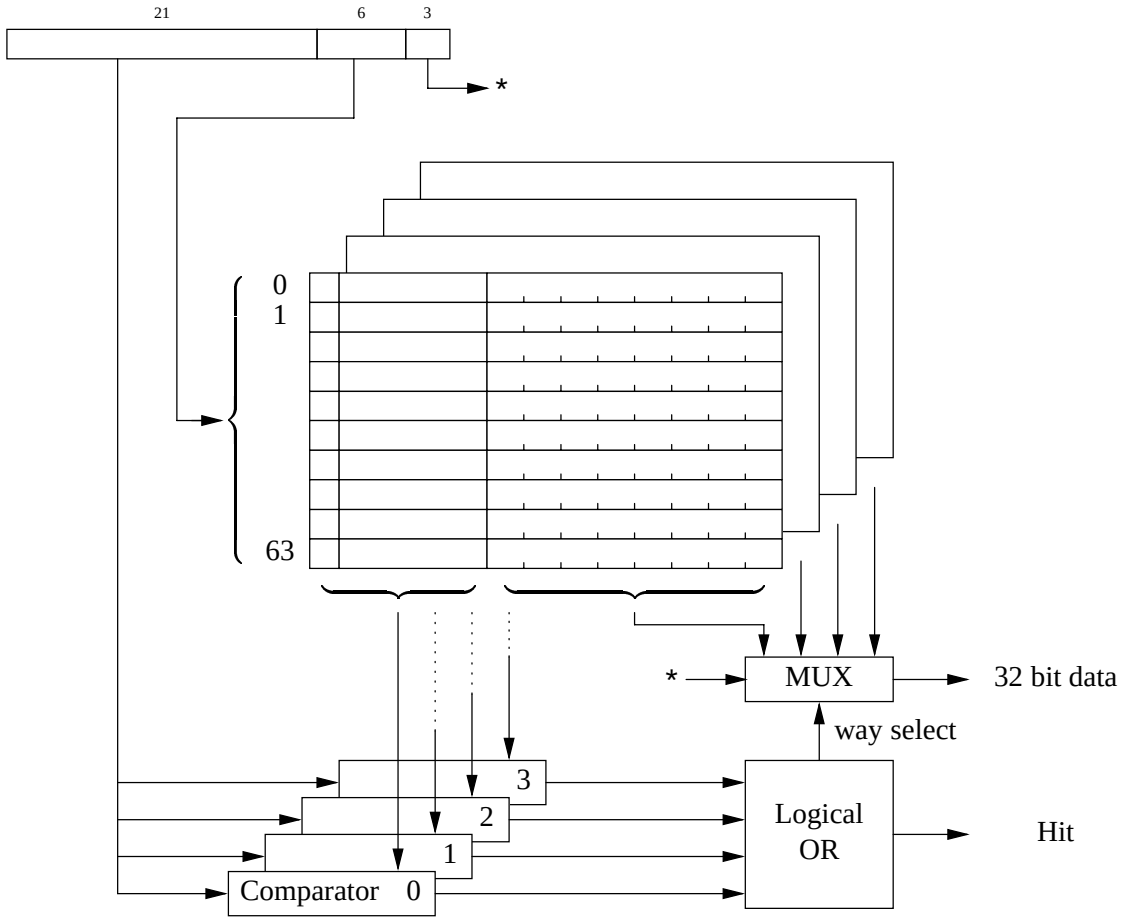

Figure 1.3: Example of a 4-way set-associative cache memory with 64 sets and a block size of 32 bytes. The data size of the cache is $4\ ways \times 64\ sets \times 32\ bytes/block = 8192\ bytes$

- Capacity – the complete cache memory is full by other lines than the requested. In nearly all cases only a full associative cache memory can be "full" in this kind of way.

1.2.3 Writing

Cache memories that can hold data must be able to handle writing/store operations. Writing on a hit in the cache can be done in two different manners:

- *Write-through* which means that any writing to the cache is also made to the lower memory level.
- *Copy-back* or *write-back* writes in the cache only and writes to the lower memory only when the block is replaced.

The advantage with write-through is that the underlying memory level is coherent which simplifies multiprocessor systems. Copy-back's major pro is the reduction of bus traffic that is both faster and reduces the average latency time for other bus masters. To solve inconsistency with lower memory

levels, a flag called *dirty* or *D* can be added. This means that besides the tag and the valid-flag and, each block also has to have a D-flag where the block can be dirty (modified) or clean (unmodified). Writing on a miss has also two strategies:

- *Write allocate*. The block is loaded into the cache followed by a write-hit-action that is described above. When a miss occurs in the cache with write allocate and the block is dirty, the block in the cache is first written to the lower level memory and then the new loaded block replaces the old one. Clean blocks can always be directly overwritten which means that the elapsed time on a miss can differ. (See also Figure 1.4)
- *No-write allocate*. The block is only modified in the lower level — not in the cache. The idea behind this is that a program or function ends its operation with storing a result. This result will not be used so a replacement with a valid block could be a bad idea. The concept works well in these kinds of programs with write-through caches.

Case	$time_{CPU}$	
read hit	t_{A1}	
WB write hit	t_{A1}	
WT write hit	t_{A2}	
WT read miss	$t_{A1} + t_{A2}B/I$	
WB read miss, clean block	$t_{A1} + t_{A2}B/I$	
WB read miss, dirty block	$t_{A1} + t_{A2}B/I + t_{A2}B/I$	
	write allocate	no write allocate
WT write miss	$t_{A2}B/I + t_{A2}$	t_{A2}
WB write miss, clean block	$t_{A1} + t_{A2}B/I$	t_{A2}
WB write miss, dirty block	$t_{A1} + t_{A2}B/I + t_{A2}B/I$	t_{A2}

Figure 1.4: **Access time with different cache implementations, states and actions.** t_{A1} is access time for the cache, t_{A2} is access time for the next lower memory level, *B* stands for *block size* and *I* for *data bus width*. The cache is attached in a look through manner (see chapter1.3.2)

1.2.4 Replacement

If one of the blocks must be replaced, due to a cache miss, one must be chosen. Ideally it should be the one that won't be used in the (near) future but unfortunately an oracle isn't possible to implement, so other algorithms has to be used:

- *Random* — replace an arbitrary block in the set.
- *Least Recently Used - LRU* — log all accesses and replace the block that has been used least.
- *Other* ideas that are bad in some kind of sense or very hard to implement like:
  - *First In First Out - FIFO* is a commonly used algorithm in other application but it has nothing to do with locality. Only because a block was read a long time ago, doesn't mean that it hasn't been used much.

– *Least Frequently Used - LFU* might seem to be a good idea. The flaw with LRU is that it is hard to implement. In this case also the time must be logged and to decide what block that has been used least a complex calculation has to be done. Do not forget that a fast cache memory has to do everything within a single clock cycle. Another way to solve the time costly calculation would be to use (large) pre-calculated tables.

  A simplified version is to just associate a counter to each block in the set and then increase the counter for each access. The block in the set that has the lowest counter will be exchanged on a miss and the blocks counter will be reset to zero. The disadvantage with this simplification is that words that have been recently loaded into the cache have a low count and that words that has been heavily used in a tight but now exited loop very hardly will be exchanged. To compensate for this effect the cache controller could decrease all counters periodically but then one must choose the size of the period. All these problems can surely be solved but to a cost of a complex and large hardware solution.

Except for the random-case and direct mapped caches (that has no choice), each set has to log and store access information to maintain the selection algorithm. Together with the valid- and the eventual dirty-bit, the log-bits makes the set's so called *status-bits*.
Pseudo-varieties are commonly used with all algorithms. A pseudo-random strategy is good enough to react randomly, but it simplifies debugging since any state can be reproduced. On a highly associative cache many bits must be used to keep track of the accesses in a LRU based implementation. To maintain and compare bits is a complex task that takes both space and time. A pseudo-LRU implementation reduces both space and time. See the case studies in this report for example.
The random algorithm works surprisingly well. Only some few percent worse hit-ratio affects the performance compared with LRU, but since the miss penalty can be rather high this difference in execution time can be much greater than that.

1.3 Implementative perspectives

Even if all the basic parameters that has been described in the previous section, also other concepts and ideas must be carefully chosen. The author has drawn the line here between basics and the rest since the cache actually works when all these four aspects has been considered.

1.3.1 Partitioning the cache organization

The cache memory can be unified to contain both instructions and data after von Neumann's "stored program concept" — also called Princeton architecture. In contrast to this data and instructions can be divided into two physically separated cache memories; one for data and one for instructions — also called Harvard architecture. The advantage of a unified cache memory is that the computer will be more general since it can be efficient in both data and instruction intensive programs. A split cache structure demands the constructor to choose the absolute size of the both memories, which in many application cases will not use the both memories full size. An instruction intensive application, like a database program, will not use much data cache and a scientific or mathematical application with vectors, matrices and narrow loops will fill up the data cache and not use the instruction cache. Such partitioning is not necessary in a unified cache memory. The really big advantage of the Harvard architecture is the bandwidth that will be doubled and reduce structural hazards in instruction pipelines. Many measurements has been done by several researchers and both architectures can show better

performance than the other in special cases. It has also been showed that the best solution is very application dependable [Smi82].
The (cache) memory can also be divided into other constellations such as giving each program, process or task a part of the cache to guarantee all running programs to have at least a minimum of cache space. The compiler can do that by logically place parts of the program in different address spaces that doesn't interfere the same blocks, or by hardwiring the hardware in the same way. In real-time applications this concept can also make the system more predictable which will be discussed in the Real-Time chapter (page 31) in this report.

1.3.2 Look through & look aside

Cache memories can be attached to a system in two ways:

- Look through or serialized (fig. 1.5); between the CPU core and the memory which will isolate the CPU from the memory which will simplify bus arbitration and reduce bus traffic. No addressing on the bus is performed on read-hits, and write-actions can also be reduced. The average access-time can be calculated with:

$$AverageAccessTime = HitTime + MissRate \times MissPenalty \tag{1.1}$$

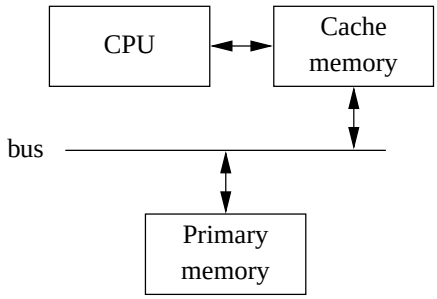

Figure 1.5: A look-through model

- Look aside or in parallel (fig. 1.6); All memory accesses goes to the main memory directly but in parallel the cache makes a tag match and if it is a hit, the cache interrupts the memory access and deliver the data to the CPU. The cache can be an own component that easily can be attached, detached or replaced and all cache misses will be handled faster since the memory access and tag compare is done in parallel.

$$AverageAccessTime = HitRate \times HitTime + MissRate \times MissPenalty \tag{1.2}$$

  In reality this solution will in fact become a disadvantage since caches and CPU:s today works at very much higher speed than the bus. This means that the penalty to evaluate a miss or hit is just a small fraction of the memory access and added to that all memory accesses goes out on the slow system bus with all the delay and increased traffic, the look-aside-concept is a more and more rare choice.

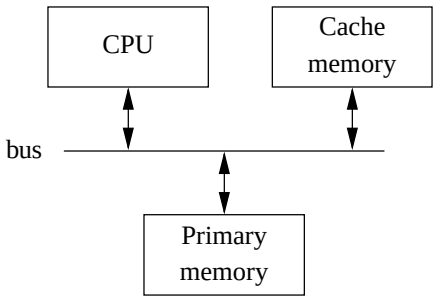

Figure 1.6: A look-aside model

1.4 Performance

A cache memory can simply be described as a small fast memory and its main purpose is to make programs run faster. The first question to ask is then *"How much faster?"*. Since caches concerns purely and nothing else but performance (which in the other hand virtually every implementative issue also concerns, many performance studies has already been done [PHH89] [Smi82]. Performance measurement can either be done by hardware on a working system or by simulation and concerning cache memories, simulation is the far most used since hardware configuration is more difficult to change than a parameter to software based simulator driven by memory access traces (*trace-driven simulation*) derived from different type of applications. Measuring on a running system is complicated but will give a true measurement since it will all code running at the computer (including operating system etc.), but in many cases a small amount of extra code must be included to make the measurement. A third way is to estimate the performance analytically [AHH89] with a cache model and by representing cache properties that affect performance. Comparing to trace-driven simulation, the analytical method can save lots of simulation time.
The speedup of an application that can be gained by a cache memory is depending at the application's structure and the cache memory's internal structure [Smi82]. A cache will gain more if the application has a "high deree of locality". Measuring locality is discussed in section1.4.3, The best metrics to use when measuring performance is *execution time*, which in cache memory cases is reduced to *average access time*. Miss-ratio, bandwidth, hit-time etc. are all insufficient measurements if stated alone since execution time is the only thing that really matters — especially in real-time systems.

1.4.1 Size

As seen in equation (1.1), an important parameter in performance aspects, *hit-ratio* is one key to successful cache memories. Large cache size increase hit-ratio is the first conclusion to state. Large caches are on the other hand more expensive and slower. How much slower depends on which parameter the computer architect tunes.

Blocksize

Choosing larger blocks is the easiest way and the method that generates least hardware overhead area to increase the cache size. More bits will be needed to point out the correct word in the block and the TLB and the memory area for storing tags will be some bits smaller since fewer bits will generate the tag.

As seen in table 1.4 the B/I factor is an important factor when calculating penalties. To have a low penalty the block size should be as large as the bus width or at least as close as possible. If the block is smaller it is a waste of bus bandwidth, but on the other hand large blocks are costly to load into the cache and very large blocks will as seen in figure 1.7 result in lower hit-ratio even if the cache is larger. The explanation is that the probability to access all the parts of the block becomes lower — as the principals of locality states. The new loaded large block will push some other data out that probably also could be used. The large blocks "pollute" each other and we call the point where hit-ratio decreases the *pollution-point*.

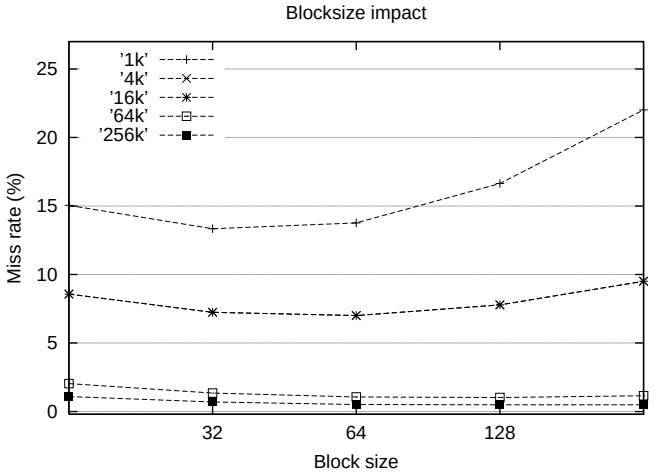

Figure 1.7: The block size impact on miss ratio. Observe the "pollution-point" for each cache size. From [HP96]

Number of sets

Increasing the number of sets is also an easy task. The number of significant bits that decodes which set of blocks that is pointed out has to be increased but also the area to store the increased amount of tags, LRU-information and other statusbits.

Associativity — number of ways

The most expensive way to increase cache memory size is to make it more associative. This means that there are more alternatives to place the block into the cache memory hence it will be utilized better and less space will be unused. This also means more administration for the replacement algorithm and more bits has to be compared by the tag matching unit since the tag field also will grow. Large associative cache memories are therefore slower and larger (more expensive) than low associative or direct mapped caches.

1.4.2 Replacement algorithms

The efficiency of the replacement algorithm in a set-associative cache memory can also be measured or calculated. A very advanced replacement algorithm can take lots of time to calculate which leads to poor access time on writes and replacements. On the other hand a poor algorithm leads to wrong

replacement, which punish the overall performance with a poor hit ratio. In reality it has been shown that the hit ratio differs very little between different replacement algorithms on caches with a small number of ways like two or four. On a simulator used in undergraduate education, three different replacement algorithms can be chosen. On a four-way set-association the same program gave the following result;

Replacement algorithm	Hit ratio
Random	78,6%
FIFO	79,1%
LRU	81,1%

One should of course never forget that hit-ratio is very application specific depending on locality of code and data. On the other hand miss penalty is very costly which leads to the conclusion that execution time will increase more rapidly than miss ratio. Two percent change in hit ratio can in some cases increase performance with ten percent on a modern processor.
What kind of replacement algorithm that is used depends today on the associativity since that is the most important factor regarding cache memory access time. The more associativity, the more work to do during a cache access.

Associativity	Replacement algorithm	Example
-4	LRU	Intel Pentium
8	Pseudo LRU	Power PC
16-	FIFO	Strong-ARM

The largest impact of replacement is code and data locality, and this leads to the conclusion that a cache optimizing compiler is generally a better choice to increase performance than an advanced replacement algorithm.

1.4.3 Locality

The “degree” of locality can be determined by for instance Grimsrud’s analytical methods based on stride and distance or a simpler model with just measuring “stack distances”[BE98]. There is on the other hand no generally accepted model of program behavior, which makes it difficult to define metrics for such properties.

1.5 Evolution – theory and practice

In reality no modern cache memory is that simple as described in the previous sections. By studying program locality and cache behavior, many enhancements have been implemented to the cache, but also memory organization in general to reduce penalties and increase hit-ratio.

1.5.1 Sector cache memories

Sector (also called sub-block) caches differ from regular cache memories by allowing more than one block to be associated with each address tag. A block that is associated to a tag is called a *sector* and the unit that is loaded or stored on a cache miss is called a *subsector* or *subblock* (See Figure 1.8).

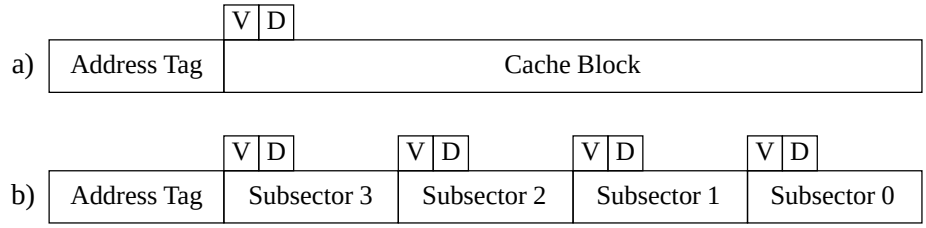

Figure 1.8: A schematic illustration of the difference between a normal cache block frame (a) and a sector frame (b) in a unified or data sector cache. 'V' stands for validbit and 'D' for dirtybit.

Each subsector must have a corresponding valid bit and, if it is containing data and copy-back strategy is used, also a dirty bit.
The advantage of this approach is that less data space for storing tags is required which means that overhead costs and space can be reduced. The second benefit is that dividing a block into sub blocks reduces the penalty of a cache miss if all memory accesses take equally long time. Unfortunately primary memories work in burst modes, which means that bandwidth is increased, but with the same latency. The block should in this case be equal to the data size of the burst to exploit this feature to the maximum. The next disadvantage is the price for reduced flexibility of mapping subsectors into the cache — generally meaning increasing miss ratio.
Other benefits has been discovered by using sector caches such as reduction of bus traffic since each block transfer is small which also reduces miss latency. On the other hand a sector is replaced in most cases before all its subsectors are occupied with data, which can be interpreted as bad utilization of cache space. The second drawback is when a sector is replaced; all of its subsector must be replaced (or invalidated) even if only one subsector is or will be fetched and used. In [RS99] a pool of subsectors is suggested where subsectors are allocated dynamically to reduce the negative parts of sectorizing cache memories.

1.5.2 Multiple levels of caches

A larger cache memory has a better condition to keep old data that will be reused much later. To just make a cache memory bigger introduces however some problems:

- A large chip is more expensive to manufacture since the yield is lower when the die-area is larger [Ekl99].
- A large chip is slower than a small since the speed of light a limiting factor in this case.
- To interconnect many small cheap ICs makes distances between the gates even longer (slower) and PCB-space is even more expensive, so that solution is out of the question.

These facts makes it clear that there are no simple solutions which makes the *"golden middle way"* a reasonable path. A cache memory that is so big that it can hold the majority of the most frequently used functions but much faster than a regular memory can be connected between the fast cache memory and the primary memory. It can of course not be as fast as the level one (L1) cache but it will boost up performance substantially. A level two (L2) cache is normally 10-30 times larger than the L1-cache and is 2-10 times slower. If a L1-cache is for example 32kB and can be accessed in 1 clock cycle, the L2 should be 512kB and the access time can be expected to be 3 clock cycles. (See Figure1.9)

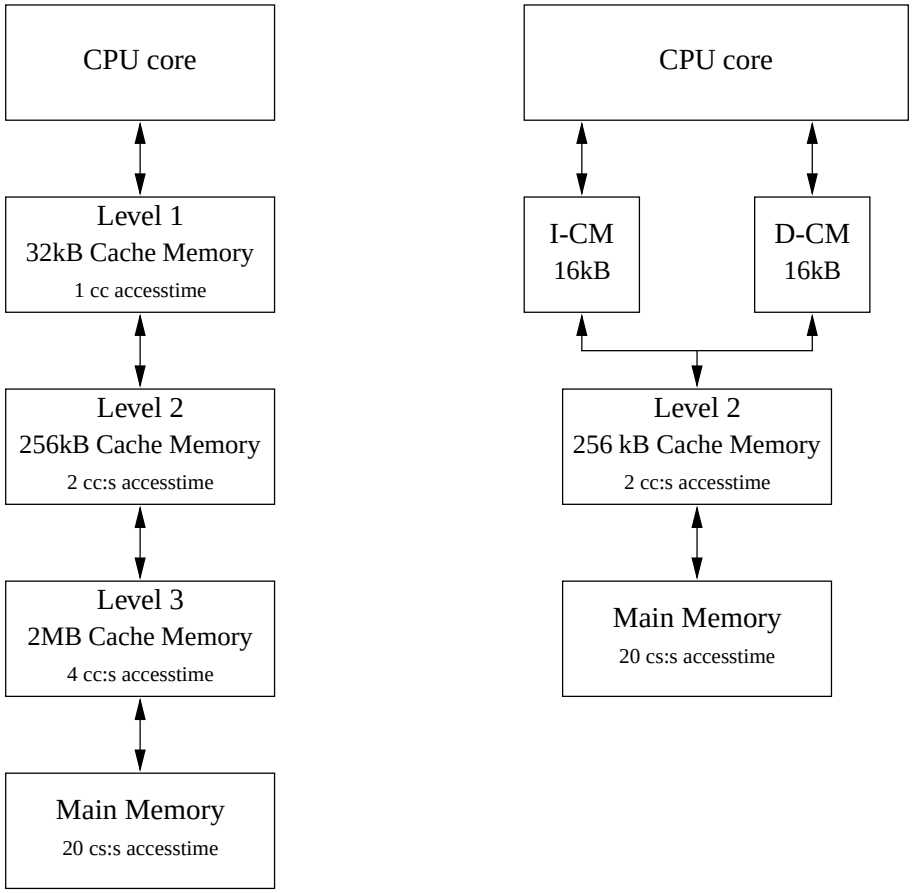

Figure 1.9: Multiple levels of caches. The model to the right has a partitioned level-1 cache. Note: "cc" = clock cycle

1.5.3 Small fully associative caches in co-operation

By adding a very small fully associative cache in the ordinary cache system, conflict misses can be reduced — especially direct mapped cache memories since the concept increases "over-all associativity" [Jou90].

Miss caching

A *miss cache* is a small fully associative cache memory that contains only two-five cache blocks and is located between the first and second level of the caches. Since it is so small it can easily fit on the same chip as the CPU core and level one cache memory. When a miss occurs, data is return to both the cache memory and the miss cache under it. Replacement policy can be any but the LRU is the most common. When the cache memory is comparing a tag, the miss cache looks up all its entries as well. If a hit occurs only in the miss cache, the cache can reload the data next clock cycle and in that case save time since the long off-chip penalty will be replaced.
Especially data conflict misses are removed (compared with instructions) since data tends to be smaller in space than function calls (procedures for instance). The miss cache is to small to contain the complete outswapped function so it will help very little. On the other hand comparing two data strings that

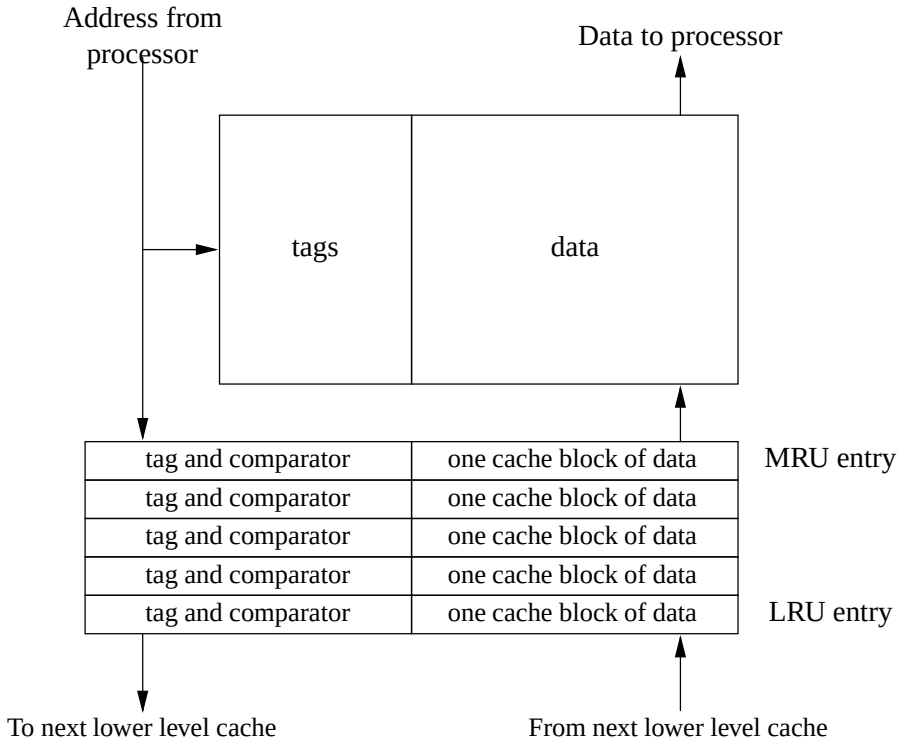

Figure 1.10: A miss cache organization

are thrashing each other will not fill the miss cache and can therefore take greater advantage of it.

Victim caching

On cache misses a miss cache loads new data from the lower level and puts it into both the miss cache and the regular cache. This is of course waste of expensive cache memory (especially the miss cache). A better way to utilize the space is to store the outswapped cache block into the fully associative cache and in this case no duplicates are necessary. The outswapped cache block is also called the victim block which naturally makes the fully associative cache be called a *victim cache*.
A victim cache is an enhancement of the miss cache so it reduces the conflict misses always better. Depending on the application, a four-entry victim cache removed 20-95% of the conflict misses in a 4kB direct-mapped cache.

1.5.4 Inexpensive set-associativity

Increasing associativity is common way to reduce miss rate but to the price of larger and more expensive cache directory and a slightly slower access/hit time. Some ideas how to make direct mapped caches act like set-associative caches will be presented in the following subsections.

Sequential associativity

Instead of having n parallel tag comparators in a n-way set-associative cache a direct mapped cache could sequentially search through n tags forward from the origin point. This *naive* approach will make

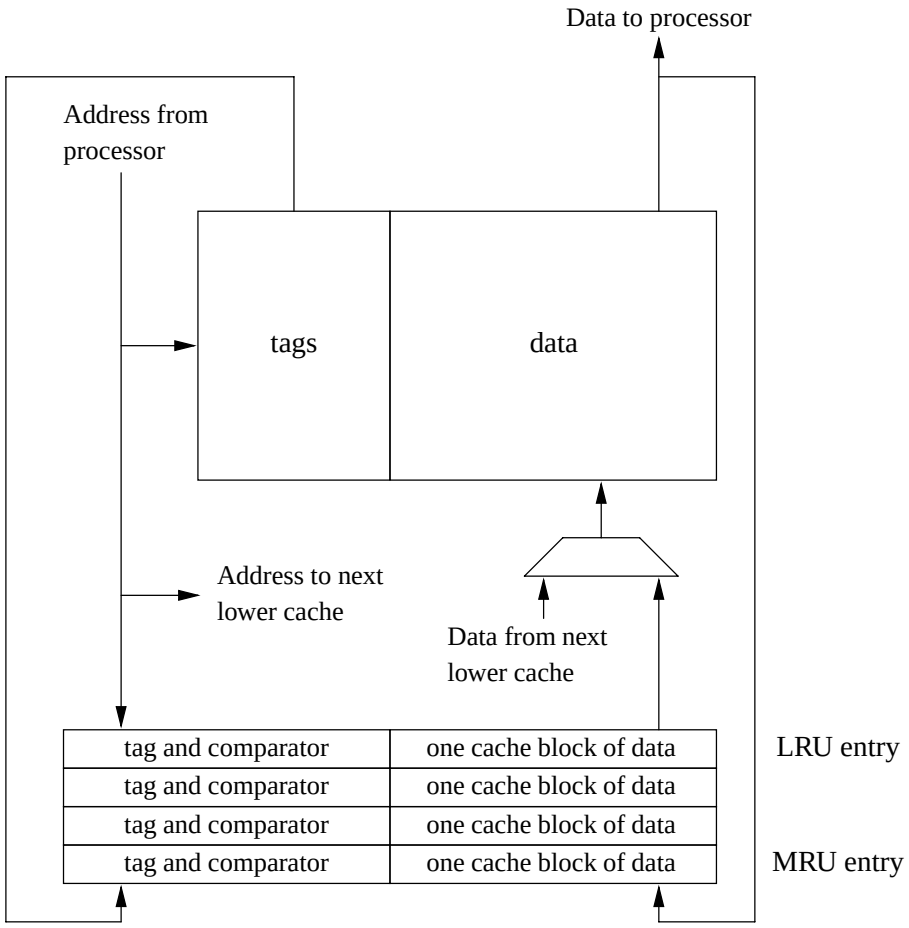

Figure 1.11: A victim cache organization

the average number of probes $\frac{(n-1)}{2} + 1$ [KJLH89] which i rather costly compared to just one parallel probe in the set-associative organization. One improvement of the concept is to order the stored tags so the most probable will be probed first. Such an order could be placing the *Most Recently Used (MRU)* tag first which is reasonable due to temporal locality. On a hit in the cache the order is updated. Instead of swapping the tags and blocks around which is rather costly in time, a special field informs in which order the tags should be probed. An LRU-replacement algorithm can also use this field. A second improvement of the sequential search is to make the probing in two steps. In step one a fraction of tag bits are probed and only the one that passes the first test will be examined further.
Any of the mentioned implementation will be less costly than a true set-associative but to the price of a longer average hit time.

Pseudo-associativity

By making a cache memory *pseudo-associative* also called *column-associative* [AP93] an associativity can be achieved. When a miss occurs in the cache memory an alternate hash function (in this section called *"rehash function"*) is called to probe a second set of tags. The rehash function can be implemented real simple by just for instance inverting the most significant bit in the look-up address. If a hit occurs it's called a second-hit or pseduo-hit. This means that the cache has two hit times; a regular

and a pseudo-hit time. A dilemma can now occur; the (most) used block can be in the alternative pseudo-way, which would lead to worse performance. To reduce this effect each hit in the pseudo-block generates a swap with the first-block, which at average should be better — stated by temporal locality — choice than living with the dilemma. The cost for this "FIFO-solution" is that the swap also costs some time in average access time.

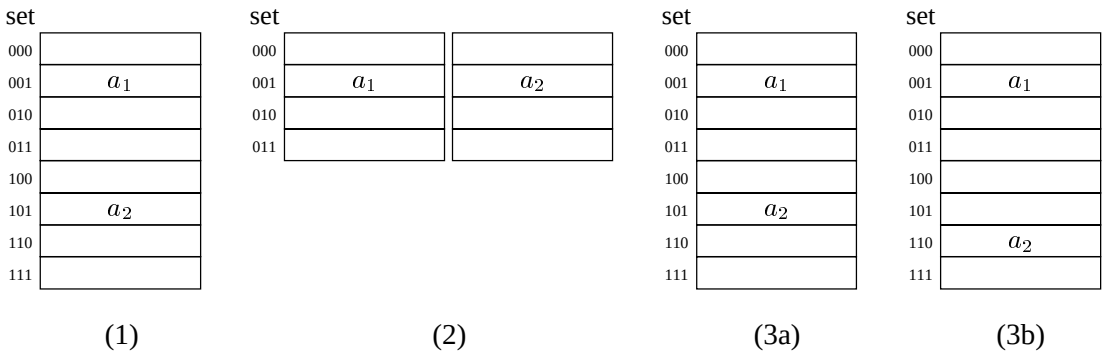

Figure 1.12: (1) Direct mapped cache (2) 2-way set associative cache (3a) Pseudo associative cache – most significant bit is flipped in the rehash function (3b) Pseudo associative cache – all bits are flipped in the rehash function

The use of a rehash function may result in a potential problem. A bit flipped address must never been mistaken by an unaltered address since a block in the cache must have a one-to-one relation with a primary memory location. A hazard could change the semantics or data values of a program. The problem can however easily be solved by adding information to the tag about it's placement condition. A bit that states if the bits are flipped or merging the unaltered most significant bits to the tag are two solutions. A third solution is to merge the hash functions index-bits to the tag (figure1.13).

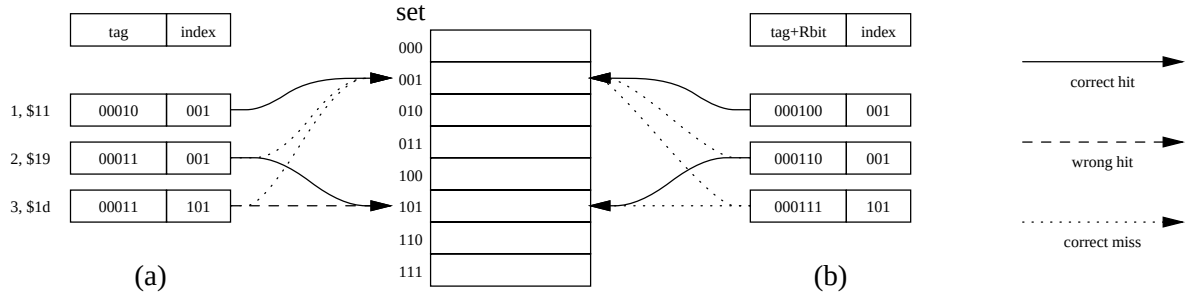

Figure 1.13: Example of potential hazardous behavior. The pseudo associative cache has 8 blocks which needs 3 bits to address and the rest of an 8-bit address will make the tag (5 bits). Scenario: the address \$11 will target at index 001 and \$19 will after a miss go to the pseudo block 101. When \$1d enters the scene aiming at 101 (a) will fault, and (b) with an additional merged index bits will cope the situation correctly.

A direct mapped pseudo-associative cache memory with *one* rehash function will give almost the same hit ratio as a two-way set-associative cache memory. A pseudo-associative cache memory can of course be equipped with more than one rehash function and hereby make it more associative but also increase the amount of hit-times.

Even if associativity can be achieved by a low cost, it generates the problem with variable hit times — something that's not too thrilling when designing pipelined implementations or calculating execution times in real-time applications.

1.5.5 Skewed association

To map an address (line) into a set of blocks in the cache memory has the disadvantage that the hashing function that points out the set of blocks will make some lines to compete for the same set — a *"hot-spot"*. A skewed association [Sez93] uses different hashing functions for each way of blocks in the cache which reduces the amount of hot-spots and increases hitratio.
The placement algoritm as presented in 1.2.1 must in this organization be extended with one or more skewing functions. A good skewing function must spread over a large number of lines in other blocks and it must be simple to compute to not extend the hit time of the cache. Such properties can be achieved by using a perfect shuffle[Sto93] of the address. By XORing some address bits; new "addresses" will be generated that can be used to point in the cache memory.
Example: A 2-way *set*-associative cache-organization with 64 blocks/way and 64 bytes/block each way must be addressed with 6 bits; in this case bits $\{a_{11} - a_6\}$ are dedicated to point out the sets. To build a similar cache with a skewed approach some bits can be XORed with each other to shuffled abit more. Bits $\{a_{11} - a_6\}$ will be replaced with (observe the order) $\{(a_7 \oplus a_{13}), (a_9 \oplus a_{15}), (a_{11} \oplus a_{17}), a_6, a_5, a_{10}\}$ in way 1 and $\{(a_6 \oplus a_{12}), (a_5 \oplus a_{14}), (a_{10} \oplus a_{16}), a_7, a_9, a_{11}\}$ in way 2. Another example is illustrated in fig 1.14.

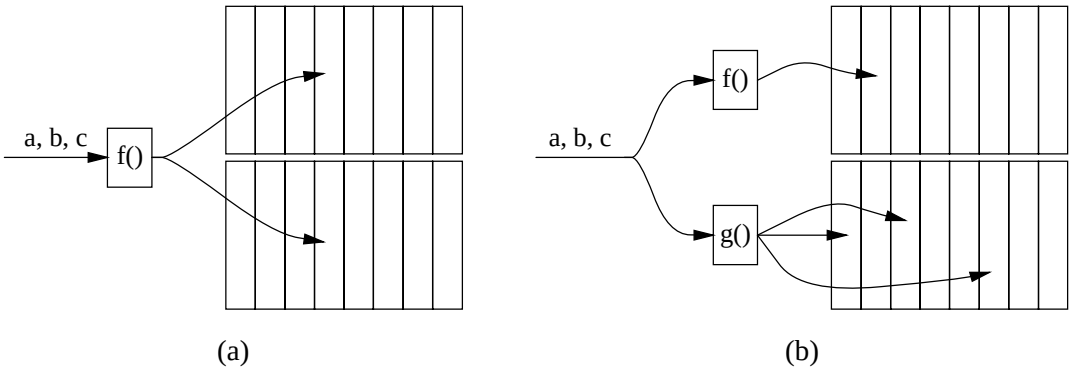

Figure 1.14: (a) is a 2-way set-associative cache that is accessed with the three addresses $\{a, b, c\} \exists f() \rightarrow \{f(a) = f(b) = f(c)\}$ all will correspond to and compete for two blocks in the same set. (b) has one skewed function to each way $(f(), g())$ which has the property $\{f(a) = f(b) = f(c) \neq g(a) \neq g(b) \neq g(c)\}$

A replacement algorithm other than a random or pseudo-random is less than a nontrivial task since the set of blocks isn't fixed — which also is the key concept of skewed association. A true LRU implementation requires a time stamp for *every* block in the cache which is costly and hard to handle. By asserting a Recently Used (RU) bit associated to a block on an access, periodically reset and a replacement policy prioritized by different conditions a pseudo-LRU behavior is implemented also referred as *Not Recently Used Not Recently Written (NRUNRW)*. [BEW98] propose to store a "partial time stamp" to each block by using some high-order bits in a clock. On a miss the time stamps is compared (modulo subtraction that also handles clock wrap around) with current time and the block with the highest value will be replaced.

Skewed association gives better performance than a regular set-associative cache; a 4-way skewed-associative cache can outperform a 16-way set-associative cache [BS97]. The cache memory's speedup can be more predictable estimated[Sez97] — at least in single threaded applications.

1.5.6 Instruction fetching and comparing tags simultaneously

Most instructions has to be managed in the same way — they are fetched, decoded, executed and finally the result is written somewhere. Instead of doing this as a four-state-machine the different parts of the CPU could be divided and connected as an assembly line. If the machine is broken into four (equal) pieces, this means that each part makes it work in a fourth of the original time and the clock rate can be quadrupled. Since the pipeline's parts can work simultaneously the processor will produce a result each clock tick and the performance is quadrupled! The speedup of pipelining is proportional to the *depth* (number of stages) of the pipeline. See also section2.5.2 for more details about instruction pipelining.
Instruction pipelining is an efficient way to increase CPU performance — not without problems but they can all be solved more or less.
To keep up with today's high clock rate, cache memories are a must and a common instruction pipelined structure contains an Instruction Fetch and a Memory stage. These stages are connected to an instruction respectively a data cache memory. But a pipeline is never faster than the slowest stage and since those stages that are connected to the memories are often the slowest, the complete pipeline will suffer. To deal with the problem, the stages are divided into substages and the rate can continue to be high.
The MIPS R4000 has divided its' instruction fetch stage into three sub-stages; Instruction Fetch – First Half (IF), Instruction Fetch – Second Half (IS) and Register Fetch (RF) [Hei94].

- IF — A branch logic selects an instruction address and the instruction cache fetch begins. Simultaneously an address translation of the virtual-to-physical address begins in the instruction translation look-aside buffer (ITLB).
- IS — The instruction is fetched and the virtual-to-physical address translation is completed.
- RF — Instruction decoding and cache tag probing is performed. If there are dependencies that cannot be solved or a cache miss is detected, the three first stages in the pipeline are frozen until the dependence has broken or the correct cache block has been loaded from the main memory. Any required operands are fetched from the register file.

The memory stage is also divided into three stages and works in a similar way. What makes it possible to fetch the instruction from the cache without tag comparisment is only possible in this construction when the cache is direct mapped — which it is in the R4000. As seen in the performance section a direct mapped cache has lower hit-ratios than set-associative solutions but the architecture wins with higher clock rate and faster hit-time.
A set-associative cache could be used, but then the identification and register fetch hardware had to be multiplied followed by multiplexors controlled by the duplicated cache tag probers.

1.5.7 Trace cache

Modern CPU:s fetches multiple instruction to a superscalar pipeline. Branches in the code will however cause stalls and decrease performance, since the execution path isn't determined until the branch

instruction reaches the execution unit. The problem can be solved by branch prediction that relays on repetitive branch behavior (see section 2.5.2). This simple solution can however only cope with a fetch of a size of a basic block — multiple branches in a single fetch cannot be handled. If a fetch occurs over two cache lines, memory bank conflicts will cause a delay and a fetch each clock cycle will not be possible.
These two problems can be solved if the basic blocks were stored contignous in a buffer — these instruction streams are called *traces* whereas the memory store is called a *trace cache*. An implementation is described in [RBS96]. The trace caches performance increasement rely on

- temporal locality — a referred trace will probably be fetched again
- branch behaviour — the branches that were taken will probabluy be taken in the same manner again

Spatial locality is however not to be considered since the trace "block" doesn't contain items that are stored closed to the referred. The trace cache only speculates in branch behaviour, not in close objects. The key concept of a trace cache is to to deliver "a high bandwidth instruction fecthing with low latency".
The following example will illustrate the concept of a trace cache and also show the difference with an ordinary instruction cache. Ponder two instruction streams of basic blocks {ABCDE} and {FEFGH} where the blocks are located in the memory as in figure 1.15.

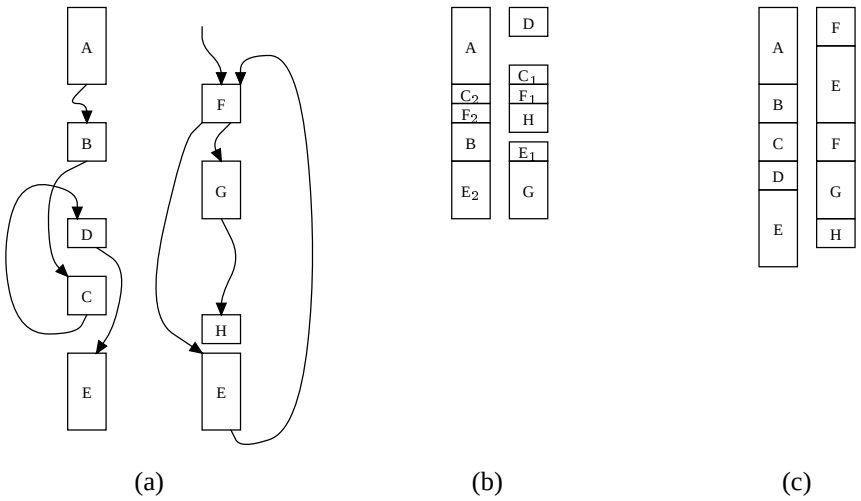

Figure 1.15: Code allocation in (a) primary memory, (b) 2-way set-associative instruction cache, (c) trace cache.

Observe that items can be duplicated in the trace cache. Fetches can cross instruction cache block boundaries and by this increase the risk of a bank conflict with a delay as a result. The trace cache has shown 15-35% performance increase compared with a system with a regular instruction cache [RBS99].
The Intel Pentium 4 was the first commercial CPU that was equipped with a trace cache [Int01]. Its size is 12kB and stores decoded micro-operations to feed the execution units directly.

1.5.8 Write-Buffers and pipelined writing

When the cache is writing to the lower memory level, often the cache memory (and CPU) is locked and idle until the complete write procedure is completed — which is costly and results in reduced performance. The write-though strategy will of course suffer more than the copy-back concept. A solution to this problem can be a *write buffer* that is an independent unit of the cache memory. Instead of waiting till the write is completed to the lower memory level, the write is left to the buffer to be completed and the CPU may continue. If another write is performed before the write buffer is finished, the CPU will have to wait. A solution to this problem is to connect more buffers in a FIFO-queue. *Data merging* in the FIFO is a method to reduce system bus traffic and the size of the queue (amount of buffer space). When two writes to the same address space are queued in the FIFO the first (old and "wrong" value will be merged and replaced by the new correct one. For example if all elements in a vector are being summarized this will generate a burst of writes to the sum-variable (if the code is bad written/compiled since it should use a register instead). With data merging only the complete sum — the last write — will be written to the primary memory.

Write buffers gives also the possibility to give *priority to read misses over writes*. When the CPU fetches (reads) new data on misses, it would be starved (no instructions to execute) if this read stream would be interrupted by a write. The fact that data traffic comes very often in bursts (e.g. on context switches) makes waiting with the write till a less dense data traffic occurs sense.
This feature might although introduce some problems. What happens when data in the write buffer is so old that the original data in the cache has been replaced and a miss occurs? The lower memory hasn't the correct value since it still hasn't been written even though it "should" have been done due to the semantics of the code. This situation is called *load hazard* and [SC97] presents four different solutions to a FIFO-organized write buffer when such an access occurs;

- Flush-full — Write all values in the buffer to the lower memory and reload it to the cache. Buffer will be empty (flushed).
- Flush-partial — Write all values in front of the queue till the requested value to the lower memory and reload it to the cache. The last values in the FIFO will remain in the buffer (if the critical word wasn't the last – then the buffer will be empty.
- Flush-item-only — Only the requested value will be written to the lower memory and reloaded to the cache. The rest of the values will remain in the buffer.
- Read-from-WB — The requested value will be written directly to the requesting unit. The queue, cache, and lower memory will not be changed. In this case the write buffer will act like a supplemental but temporal cache memory.

1.5.9 Early restart

When a cache access ends up in a miss, the cache memory loads a new block after an eventual write-back if the block was dirty. When the block is on place the requested word in the block — hereby called *critical word* — is forwarded further to the CPU. This means that the CPU is blocked during the miss and to reduce the penalty a good idea would be to forward the critical word to the CPU as soon as it arrives to the cache. The rest of the block can after this continue to be loaded into the cache memory. This solution is called *early restart* and saves time especially when the critical word is at the beginning of the block. It also means that the miss penalty will vary depending on which of the word

in the block that is being wanted — a not to tasteful property in real-time WCET-calculation. When considering performance this concept will always gain speedup to a very low cost.

1.5.10 Critical word first

To enhance this method and also give a fixed miss penalty time is to let the cache memory ask for the *critical word first*. This means that the cache memory doesn't request for a block but for a specific word and the lower memory have to send that first before it sends the rest of block. The "rest" is normally sent "wrap-around"; that is the contiguous words after the critical word is sent and when it reaches the block end it restarts at the beginning of it till it reaches the word before the critical. Example: If the critical word in a four-word block is the third one, the words are sent in the order {3,4,1,2}. The method is therefore even mentioned as *wrap-around fill cache* [WMH+97].
The lower memory have to justify it's burst-mode to be able to handle this new feature. The performance gain varies depending on primary memory latency, block and word size.
An illustrative example can maybe give a hint of the speedup.
The cache block is 32 bytes, the word size of the CPU is 32 bits (=4 bytes) and the data bus width is 64 bits. This means that a cache block transfer takes 4 bus reads. If the latency time to the primary memory is 50ns with a cycle time of 20ns when accessing different memory banks, a system *without* the enhancement a cache miss costs $50 + 3 \times 20 = 110ns$. Using the *critical word first*-concept the penalty will be reduced to 50ns which is more than twice as fast.
Both *early restart* and *critical word first* gives a speedup at the first cache miss, but if the next access (in the next CPU-clock cycle) is in the same block a new penalty will arise since the block hasn't been completely loaded yet. This and the fact of a more complicated low-level memory behavior requires a rather complex implementative solution but is manageable and today most new high-performance microprocessors are equipped with the feature *critical word first*.

1.5.11 Non-Blocking Cache Memories

A cache miss will make the cache memory to go into a waiting state till it has received the correct new block. Ponder the following section of code;

```
LOAD R1,R0($12)   # data is not in the cache
LOAD R2,R0($56)   # data is in the cache
```

If two consecutive instructions both read some data where the second is but the first isn't in the cache, the cache memory will in the first instruction enter a wait state — which normally means blocked. An instruction pipelined or superscalar[1] CPU with "data dependency checking" will detect the independency and take precautions like stalling, register renaming or forward data to avoid the hazard. In this specific case the two loads are independent and doesn't belong to the same block. A *non-blocking* (also referred as *lock-up free* cache memory will detect a miss and start to load a new block, but it will still be able to serve the CPU or upper memory hierarchy with other information — such as the next instruction's request. The first instruction's request is placed in a special buffer and a special engine that is independent of the rest of the cache structure will take care of this task. How many tasks a cache can take care of depends of how many buffers and machines that has been implemented. When all buffers are full the cache memory will although enter a blocked state. The Intel Pentium Pro-CPU can buffer up to 4 misses and still work. The amount of buffers needed depends on performance requirement,

[1] a CPU that is available to issue two or more instructions simultaneously

chip space available, miss penalty, system clock frequency and the application's highest frequency of load and store instructions in a piece of code.

1.6 Prefetching

Even if "early restart" and "critical word first" can reduce penalty, the best performance will be gained by excluding all or at least some of the penalties. This can be achieved by *prefetching* blocks before the processor actually is requesting for them. The prefetching must however be done not to early since they then can be replaced by other blocks, but of course not too late either since there will be a delay with a performance loss as an result. The prefetching can be done directly into the cache or into a special buffer.

1.6.1 Prefetching with software

When entering a new section of data or instructions the cache has to be filled with these blocks, which of course will reduce performance. By having special prefetch instructions in the instruction set, these instructions can be inserted several instructions ahead and by this reduce the cost of context change. Prefetching with instructions makes only sense if the cache memory is non-blocking (see section 1.5.11) – otherwise the prefetch instruction would stall the processor.
Certain data areas and calculations in for instance matrices will not take advantage of the temporal locality since they only make one access in each position of the matrix and after this will go further in the code. This kind of code will not take advantage of the cache and locality concept, and is forced to use prefetching to reach high performance.

Example;

```
for(i=0; i<1000; i=i+1)
  for(j=0; j<1000; j=j+1)
    sum=sum+a[i][j];
```

The above program will not take advantage over temporal locality. The spatial locality will at least make it possible to not decrease performance, but the performance achieved could be done with a CPU without any data cache memory what so ever. If the cache block contains 4 integers, every fourth access in 'a' will end up with a miss and a cache block fill.
A prefetch of data will hide latency to the price of some extra instruction execution. Since memory latency increases and execution time decreases this price will however be rather small;

Example;

```
for(i=0; i<1000; i=i+1)
  for(j=0; j<1000; j=j+1)
  {
    if(j % BLOCKSIZE == 0)
      prefetch(a[i][j+LATENCY]);   /* prefetch block in advance */
    sum=sum+a[i][j];
  }
```

The first accesses in 'a' will miss and the last prefetches in the loop will gain nothing, but all the other accesses will take advantage of the prefetching. The higher proportional latency a cache miss yields (compared with cache hit), the larger LATENCY-factor must be used.

By prefetching and *loop unrolling*, miss penalty can be reduced to zero. The loop unrolling has the effect that more code will take more time to execute which gives the possibility to prefetch simultaneously as executing some other code. Loop unrolling is also beneficial to reduce bubbles in instruction pipelining since the amount of branches will be reduced. In a tight loop, the administration of it (an increasing counter, a compare and a branch) will consume a large part of the execution time and by making more "real work" in the loop, a third gain of loop unrolling can be stated. The longer latency for the fetching the more unrolling is needed, but conceptually the result can be something like;

```
for(i=0; i<1000; i=i+1)
{
  prefetch(a[i][j]);       /* prefetch first block in the column */
  for(j=0; j<1000; j=j+4)
  {
    prefetch(a[i][j+4]);   /* a 4-word-block in advance in the row */
    sum=sum+a[i][j+0];     /* unroll the code twice */
    sum=sum+a[i][j+1];
    sum=sum+a[i][j+2];
    sum=sum+a[i][j+3];
  }
}
```

Where the prefetch-instruction exactly must be inserted in advance has to do with the processors execution pace and the memory's access time (latency). The faster CPU and slower memory access, the more ahead a prefetch must be made which in this case can be achieved by more unrolling of the loop.

1.6.2 Prefetching with hardware

Prefetching can be accomplished by letting special hardware in the CPU try to predict what block is to be used in the near future. Spatial locality can give such a hint; if a miss occurs in the cache, not only the missed block is fetched from the lower memory but also some of the consecutive blocks. In the Alpha AXP 21064 on instruction misses the requested block is loaded into the cache and the prefetched block is directed to an instruction stream buffer. This kind of prefetching relies on unused bus and memory bandwidth which can with this approach be better utilized.

1.7 Software design

1.7.1 Compiler optimizations

Performance can be enhanced by better hardware (for instance cache memories) but also by designing the software to better fit the hardware. A program that instantly makes caches to thrash data mustn't necessarily mean that the cache has to be more associative or larger. By just slightly justify the program "miracles" can happen to performance. The easiest way to prove that is by the following small example;

```
int a[1000][1000];
...
for(i=0; i<1000; i=i+1)
  for(j=0; j<1000; j=j+1)
    sum=sum+a[i][j];
```

```
...
for(j=0; j<1000; j=j+1)
  for(i=0; i<1000; i=i+1)
    sum=sum+a[i][j];
```

In the above code examples the only difference is that the left one is adding column by column and the right is performing the same procedure but in rows instead. Ponder a system where an integer requires 4 bytes of memory space and the cache block is 64 bytes large in an 32kB data cache. The memory for the matrix `a[][]` is allocated in rows by the compiler, and consecutive addressing in columns will always generate misses since the data cache can't hold all 1000 blocks in memory (64 kB) at once. Addressing the matrix the row-way will take advantage of the spatial locality and generate 15 hits for each miss. With prefetching hit-ratio can, as shown in 1.6.1, increase even more.
Another way to increase performance by data caches is to exploit the locality by declare variables that will be used close in time close to each other (in sequence) to take advantage of the spatial locality (a real smart compiler will however rather use registers than the cache).
Other kinds of code optimization for a cache memory is *data merging*; data that will be used "simultaneously" can take advantage of spatial locality by for instance grouping the data in structures;

```
int a_v[8192],b_v[8192];
...
for(i=0; i<1000; i=i+1)
  sum=a_v[i]+b_v[i];
```

If the data cache is 8kB, the summation will cause instant thrashing and reduce performance to lower than a system without a cache(!). A suggested solution in this case could be;

```
typedef struct{int a;
               int b;}pairs;
pairs vector[8192];
...
for(i=0; i<1000; i=i+1)
  sum=vector[i].a+vector[i].b;
```

The competition of the same cacheblock is avoided and spatial locality is exploited. A good rule is not to allocate data in sizes that are even multiples of the cache size. It is better to allocate a little bit more (preferably a prime number) and by this avoid direct thrashing.

1.7.2 Code placement

High hit ratios can be achieved by placing code in such manner that it doesn't interfere with other parts of the code that is used frequently. The interference turn up when two or more sections of the code maps to the same part of the cache.
The behavior of a program can be described in a *Control Flow Graph* (CFG) where each node correspond to a basic block and directed edges represents to a a control dependency between basic blocks. Assume two functions with CFGs as described in figure 1.16 and that b_4 calls b_6. With a typical input data the code will be executed in the following sequence;

$$(b_0, b_2, b_3, b_4, b_6, b_7, b_8, b_3, b_4, \ldots b_3, b_5)$$

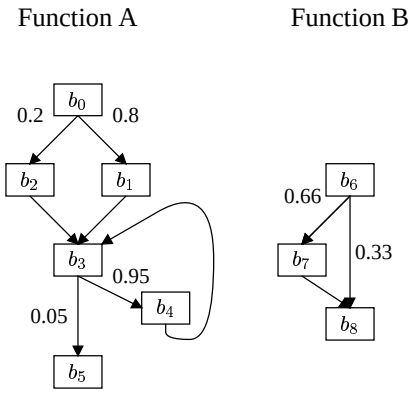

Figure 1.16: Weigthed control flow graph. The weight reflects the probability to choose a specific execution path

It is more probable that basic block b_2 will be executed after b_0 than b_1 will. In this case b_2 should be placed after b_0 because the chance is higher that the two basic blocks are on the same cache line. The risk is also lower that b_0 and b_2 will map (and compete) for the same cache lines since they are in sequence. Due to the same reason b_4 should be placed before b_3, b_5 after b_3 etc. which will render the following basic block placement;

$$b_0, b_2, b_1, b_4, b_3, b_5, b_6, b_7, b_8$$

In [TY96] an optimizing method is described to select code layout into main memory to achieve higher hit ratios. The method uses an ILP-approach and experimental results showed 35-45% reduction of miss ratio.

1.8 Case studies on single CPU systems (no RT aspects)

1.8.1 Intel Pentium III

Overview

The Intel Pentium III is a high-performance 32-bit processor that is meant to be used in personal computers, multimedia applications, and games. It was released 1999 and it's main structure is still very close to the Pentium Pro released in 1995. It is x86-compatible and the instruction set has been increased with multimedia instructions such as MMX and streamed SIMD. Aggressive branch speculation and out-of-order execution will make performance non-deterministic; even for a specific code sequence since it may depend on a state the machine was in when that code sequence was entered.
The first level cache is divided into a 16kB instruction cache and a 16kB data cache. The second level cache is unified, direct mapped and 512kB, but in Q1 2000 an enhanced E-version of the L2 cache was released — a smaller but more advanced L2 cache called ATC (advanced transfer cache). A MESI snooping protocol supports cache coherence in multiprocessor systems.
The L1 I-cache supplies the fetch unit with blocks that are decoded and put in a reorder buffer (called "instruction pool"). The execution unit i capable to issue three instruction each clock cycle from the instruction pool and finally the retire unit keeps the instruction till the semantic order of the instructions is "secured". All data in this subsection is retrieved from [Int99] and [Int01].

The first level cache

The first level of caches is divided into an instruction and a data part (Harvard architecture). The instruction cache is four-way set-associative organized in 256 sets of 32-byte-blocks. Replacement strategy is LRU. The data cache is less associative — 2 ways — and since the block size is also here 32 bytes 512 sets are needed. Both caches are dual-ported, use LRU replacement strategy, and write policy for the data cache is write-through.

The second level cache

The second level cache comes in two versions; a “discrete” and an “ATC” (also called ’E’) version, where the latter is more advanced and was released in Q1 2000. The L2-cache is non-blocking and can handle up to four accesses simultaneously.

- The discrete L2-cache i 512kB, 4-way set-associative and use a block size of 64 bytes. It is resided with standard L2-components (e.g. Burst pipeline Synchronous static RAM (BSRAM) technology) on another die than the CPU-core but in the CPU-capsule. It is connected look-through the L1-cache on a dedicated “back-side” 64-bit data bus to the CPU-core and runs at half the CPU-core speed.
- Advanced Transfer Cache (ATC) is 256kB in 8 ways and it resides on the same die as the CPU-core and L1-cache. The 256bit data bus runs at the CPU-core speed which means that the peak throughput has been 8-folded. ATC also incorporates 4 write-back buffers, 6 fill buffers and 8 bus queue entries to reduce latency on high bus activity. The cache blocks are increased to 128 bytes but sectorized where each subsector is 64 bytes.

1.8.2 Motorola Power PC 750

Overview

The MPC750 (also advertised as “G3”) is a 32-bit processor meant to be used in personal computers (e.g. Apple Macintosh) and high-end embedded systems (e.g. Region Processors in Ericsson AXE-routing system) which requires a processor with most available features such as out-of-order execution, branch prediction, superscalar pipelines with multiple execution units and an advanced cache system to keep the pace. Low power consumption and price are areas that might be less considered.
The first level cache memory is split into a data, an instruction part of each 32kB organized in 128 sets, 8 ways, and 32 bytes block size. A second level cache of up to 1MB with an access time of only 1,5 clock cycle can be attached to the CPU. To this, read and write buffers decrease latencies, and a MEI snooping protocol maintains cache coherency when multiple bus masters (e.g. DMA, other processors) reside on the system bus.

First level cache memory

The level-1 cache memory is divided into a instruction and a datapart and both parts look much the same except of course for write-capabilies in the datapart. Each block is 32 bytes containing 8 words and all blocks are organized in 128 sets and eight ways. The CPU-core is fed from a six-entry instruction queue that is connected to the instruction cache via four word wide databus. The data cache is connected to the core via a 2 word wide databus. Both caches are non-blocking to be able to maintain the core’s multiple execution units that can hold up to six instructions simultaneously. Replacement strategy is pseudo-LRU which is a fast, high-performance, and very close a true LRU implementation.

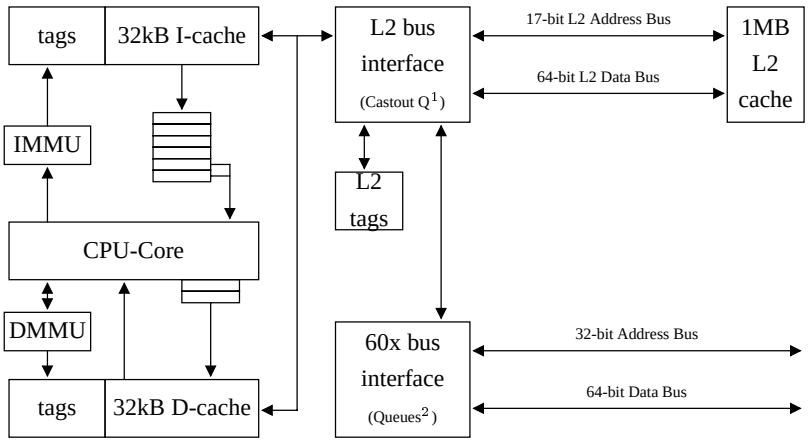

Figure 1.17: Block diagram of the MPC750 cache memory system. The L2 Bus interface is equipped with a 4-entry castout queue[1] to hide write latency during simultaneous reads to the L2 and the 60x bus interface is in the same fashion equipped[2] with one instruction fetch buffer, two castout buffers and one data load buffer.

Second level cache memory

The level-2 cache is two-way set associative and can either be none, 256kB, 512kB or 1024kB. The L2 cache tag memory can hold 2×4096 tags and is located on the CPU-core die for faster performance. The data is organized in sectors and is located on an other die of SRAM memory. A sector is one L1-block (32 bytes = 8 words).

total size	sets used	sectors / L2-block	L2-address bus used
256kB	2048	2 = 64 bytes	14-0
512kB	4096	2 = 64 bytes	15-0
1024kB	4096	4 = 128 bytes	16-0

Replacement policy is LRU that is maintained by a single bit at each set in the tag memory. Write strategy can be set by software to either write-through or copy-back. Hit time depends on the frequency of the L2 bus that can be set to {1, 1.5, 2, 2.5, 3} times CPU-core clock frequency. Which speed that can be set depends on CPU-core speed, SRAM-technology, and write-policy (pipelined/flow-through). On L2 cache misses the critical word first will be forwarded to the L1 cache simultaneously when it reaches the L2 cache. The first level cache is always a subset of the secondary cache memory.

Read/write buffers

To boost up performance a numerous different buffers and queues are available to reduce bus activity penalties, latency etc. Both the CPU-core (L1-cache) and the L2 cache bus controller are equipped with *castout queues* that can hold writes/stores to the memory if other bus activity is performed to reduce the amount of blocked executing units. When the queues are full after a burst of writes the unit will however be blocked until the writes are performed to the lower hierarchical memory. The 60x-bus interface unit has also been equipped with one instruction fetch buffer, one data load buffer and a two-entry castout queue.

Software control

Each cache block has four bits called WIMG that specifies some memory characteristics that can be set by the operating system with the
tt mtspr-instruction. The WIMG attributes controls the following properties of the cache memory;

- Write-through (W) – sets write-policy to be either write-through (1) or copy-back (0)
- Caching-inhibited (I) – turns on the cache (0=on, 1=off)
- Memory coherency (M) – ensures that memory coherency is maintained (1)
- Guarded memory (G) – prevents out-of-order loading and prefetching (1)

If self-modified code is used, data (instructions) is written to the data cache and not to the instruction cache. A special sync-instruction-sequence can update the instruction cache with the data cache modification. The snooper that maintains the MEI-protocol has special instructions and the caches can be locked, but will still be coherent by letting the snooper modify the MEI-status bits. Single blocks but also the complete cache can be invalidated (flushed) by single instructions.

Hit and miss ratio for data and instructions in both of the L1 and L2 cache memories can be monitored by using special *performance monitor* registers. This monitor can also register events regarding snooping, branch prediction, TLB miss ratio etc. and can be valuable tool to tune in applications for best performance.

Pseudo-LRU replacement algorithm

Administrating a real LRU-algorithm on a 8-way set-associative cache memory is difficult. In a single clock cycle up to 7 comparisments and counter increments must be performed which of course is hard to handle when running at 500MHz or more and also rather costly in space to maintain. The PPC 7x-series handle the replacement algorithm with only 7 bits/set of blocks, one ROM-table to set those bit on memory accesses and one table to interpret them on cache misses. The bits, their selection, and updating are cleverly chosen so the set's state is very close to a true LRU algorithm.
This solution is fast, small, behavior reproducible, easy to understand and implement.

1.8.3 StrongARM SA-1110

Overview

In February 1995, Digital Equipment Corporation and ARM Ltd announced a development and license agreement that enabled Digital Semiconductor to develop and market a family of low-power microprocessors based on the ARM 32-bit RISC architecture. On May 17, 1998, Intel Corporation acquired Digital's StrongARM which today means that StrongARM is a product developed by Intel and ARM. The Intel StrongARM SA-1110 Microprocessor (SA-1110) is a highly integrated communications micro controller that incorporates a 32-bit StrongARM RISC processor core, system support logic, multiple communication channels, an LCD controller, a memory and PCMCIA controller, and general-purpose I/O ports. It's meant to be used in performance demanding, portable and embedded applications. When running at 206 MHz it only consumes < 400mW at normal mode and that could be lowered when using power-management. The SA-1110 incorporates a 32-bit StrongARM RISC processor capable of running at up to 206 MHz. The SA-1110 is equipped with instruction and data cache,

Current access to	update the PLRU bits to						
	b0	b1	b2	b3	b4	b5	b6
W0	1	1	x	1	x	x	x
W1	1	1	x	0	x	x	x
W2	1	0	x	x	1	x	x
W3	1	0	x	x	0	x	x
W4	0	x	1	x	x	1	x
W5	0	x	1	x	x	0	x
W6	0	x	0	x	x	x	1
W7	0	x	0	x	x	x	0

x = does not change

PLRU bits status						Block to be replaced
b0	0	b1	0	b3	0	W0
	0		0		1	W1
	0		1	b4	0	W2
	0		1		1	W3
	1	b2	0	b5	0	W4
	1		0		1	W5
	1		1	b6	0	W6
	1		1		1	W7

Figure 1.18: Pseudo-LRU tables. The left table shows, which bits are to be set on a hit or replacement. The right table shows which block is to be replaced. Example: a set's PLRU bits are 1011001 and on a hit in W4 the PLRU bits would have the new value **0**0**1**1**0**1. If the next access generates a miss W1 is to be replaced due to the bold marked bits **001**1011 — and don't forget to update those bits with 11x0xxx.

memory-management unit (MMU), a victim cache (called "minicache") and read/write buffers. All data material from [Hei94].

- 32-way set-associative caches
  - 16 kB instruction cache
  - 8 kB write-back data cache
- Write buffer with 8-entry, between 1 and 16 bytes each
- Read buffer 4-entry, 1, 4, or 8 words
- 512 byte mini data cache
- all cache memories and buffers can be turned off, invalidated, pushed or drained by software.

Instruction cache

The instruction cache (I-cache) is 16 kB divided in 32-byte (8 words) blocks and 32-way set-associativity, which will leave us with 16 sets of blocks. On misses blocks are replaced in a round-robin (FIFO) algorithm. The instruction cache isn't able to detect writes in the data cache or the write buffers which means that coherency and consistency must be maintain by the programmer. The programmer is able to push out values from the cache and also drain the write buffers.

Data caches

This version of the StrongARM has a smaller data cache (D-cache) — 16kB has been decreased to 8kB, but it looks much the same as the I-cache: 32 bytes blocks and 32-way set-association. Write strategy is copy-back and replacement policy is round-robin. When reset each set of blocks resets it's way-pointer to zero. To compensate for the poor replacement algorithm a 16 block minicache

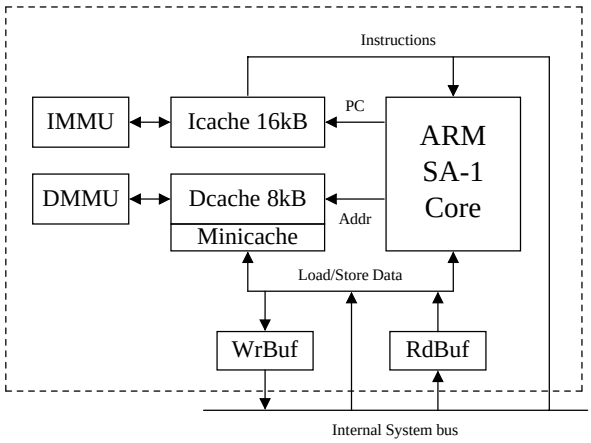

Figure 1.19: Part of the StrongARM-1110 block diagram

is connected look-aside to the main data cache. The minicache is 2-way set-associative and use an LRU-replacement strategy. A data block can never be duplicated in both of the D-caches.
Both D-caches use the virtual address generated by the processor and allocate only on loads (write misses never allocate in the cache). Each line entry contains the physical address of the line and two dirty bits. The dirty bits indicate the status of the first and the second halves of the line. When a store hits in the D-caches, the dirty bit associated with it is set. When a line is evicted from the D-caches, the dirty bits are used to decide if all, half, or none of the line will be written back to memory using the physical address stored with the line. The concept is described in a previous section as sector caches (section 1.5.1), but in this case only the write-back is sectorized — not the loading. The D-caches are always reloaded with a complete block (8 words) at a time.

Write Buffers

To reduce penalties on cache misses on writes a write buffer has been attached to the system.
If the write buffer is enabled and the processor performs a write to a bufferable and cacheable location (these properties are set in the MMU), and the data is in one of the caches, then the data is written to that cache, and the cache line is marked dirty. If a write to a bufferable area misses in both data caches, the data is placed in the write buffer and the CPU continues execution. The write buffer performs the external write sometime later. If a write is performed and the write buffer is full, then the processor is stalled until there is sufficient space in the buffer. No write buffer merging is allowed in the SA-1110 except during store multiples.

Read Buffers

The SA-1110 is equipped with a four 32-byte read buffers controlled by software and makes it possible to increase performance of critical loop code by prefetching read-only data. By prefetching data to the read buffers, memory latency and cache misses can be reduced since the loading of a read buffer doesn't lock the CPU-core. Observe that these buffers cannot be used to prefetch instructions.

Chapter 2

Real-Time and unpredictable hardware

2.1 Introduction to real-time

A more general and flexible machine — the computer, has in the last decades replaced automatic control and traditional controlling system in industrial applications.
It can not only calculate positions, flows and take precautions to regulate processes, but it can also be connected to other controllers (computers) databases, administrative systems and give not only the man on the working floor information about how the temperature is in a nuclear power plant core, but also provide estimations about how long the uranium will last to the buying office or the boss with an economic report if the system will be driven in the way it actually does. The computer has in many areas out competed PLCs and manual gears due to its flexibility, accuracy, and speed. As mentioned a automatic control system, or real-time system (RTS) as they now days are called, can be a nuclear power plant, but also an air bag in a car, power control in a microwave oven, a robot painting automobiles or a telephone switching device. In all these systems time and timing is crucial. The response time cannot be acceded or the system will fail.
To control a process with a computer (real time) system is a achieved by to following steps (in order)

1. Observe the process – *sample* the values from a probe
2. Decide what to do – execute some instructions to calculate new positions etc.
3. Actuate – Control the process by giving a device (motor, relay etc.) a signal

This approach raises two new questions; how often does a sample have to be performed and how fast must each sample be computed to give a correct control signal? If the sample isn't performed often enough, the system will have a very rough and "jumpy" view of the world. A higher sample rate will give a smoother view, but also more data to handle. The sampling process should therefore not be performed more often than necessary. The second issue about the computation (and maybe even physical) delay is maybe even more important. A fast response time can maybe be achieved by a small and simple calculation, but a more sophisticated algorithm with better precision takes longer time to calculate. If the response time is too long the system will not work and maybe cause errors and injuries. The system is controlling a world from the past and not "real time".
The answer to both questions is that the environment where the system interacts must be specified and from this specifications one can for instance get answers like that the sample rate must be no less than 500Hz and worst-case response time must no more than 10 milliseconds.

A job must always be finished until *deadline* but in many application the job mustn't be finished before a certain time either. In a *hard* RTS all jobs must be finished in this window of time but in *soft* only a certain percentage of the jobs must do this. Hard RTS are typically those who control something that mustn't fail or something or someone will be hurt or damaged — for instance air plains. Soft RTS RTS are those that can tolerate misses to some extent and will regulate it with decreased quality of the result — for instance a telephone switch station. The specifications of the system describes if it is a hard or soft real-time system.

2.2 Real-time and scheduling

The simplest form of an RTS is a single program running at a computer with no other tasks to handle. More complex systems are however easier to construct with several tasks (processes, threads) running simultaneously in a time-sharing environment which also utilize system resources better. A simple operating system gives every task an equal slice of the time and runs the jobs in a round-robin order. An operating system that is specialized for real time systems can utilize performance even more; giving the tasks priorities, allow preemption (interrupt a low-priority work by a high-priority and then resume), use more advanced scheduling algorithms etc.
Scheduling can be performed in advance (static) or at run-time (dynamic). A static schedule can theoretically utilize system performance to 100% but only a few applications can be designed to do that. A schedule done in advance is easy to proof that it will meet time constraints. Systems that are event-driven can in many cases utilize system resources if they are scheduled at run-time; if certain events occur only sporadically a static schedule must have allocated this time for each

2.2.1 Static scheduling algorithm example: Rate monotonic

The principle for the rate monotonic scheduling algorithm is to give the task with the shortest period time gets the highest priority and the tasks with the longest period time will consequently get the lowest priority. All tasks time constraints are defined by period time $T_i$and execution time C_i. Each tasks load or utilization of the system can then be calculated as $U = C_i/T_i$ and for n tasks the utilization is $U = \sum_{i=1}^{n} \frac{C_i}{T_i}$. Lui and Layland[LL73] found that a task set with n tasks where

1. all tasks are periodical,
2. no task must exceed it's deadline that is the period time,
3. all tasks are independent of each other and must never wait for a message or synchronization,

is schedulable if

$$U = \sum_{i=1}^{n} \frac{C_i + \gamma_i}{T_i} \leq n(s^{1/n} - 1) \tag{2.1}$$

The γ_i stands for the time it takes for the operating system to make a context switch[1] (sometimes even called *overhead time*). The condition leads to a maximum 69% utilization of the system when the number of tasks is approaching infinity.
Liu and Layland's equation is however pessimistic since there are several cases where the task set is schedulable even if the utilization is more than 69%. The condition is therefore sufficient but not necessary.

[1] In equation 2.2 γ_i has been evolved to separate operating system issues and cache related issues, thus $\gamma \rightarrow 2\delta + \gamma$

2.2.2 Dynamic scheduling algorithm example: Earliest deadline

Earliest deadline is a dynamic scheduling algorithm that dynamically can change the priority of the tasks during run-time. The key concept is to assign the task with shortest time until deadline the highest priority. This leads to the possibility of higher system utilization than for instance the rate monotonic algorithm. The main disadvantage of earliest deadline is that if a task misses it's deadline, the execution order is undefined.

2.3 Execution Time Analysis

Real-Time systems rely on correct timing. All scheduling needs timing data and real-time means in this case *real time*. To be able to schedule a task set, each task must be provided with the *calculated worst-case execution time* WCET_C in for instance milliseconds.
To really know how long a program sequence takes to execute, it's almost never as easy as to take a stopwatch and measure the time it takes to execute the section of the program. The response time of a program depends not only of the critical code sequence that must be performed but also on computer performance, workload, condition, state etc. — an analysis must be performed.
The execution time of a program is depending of the following factors[Gus00]

The input data — The value of the data may steer which execution path or how many iteration a loop will make. The timing of the data is also important; data comes in bursts or irregularly

The behavior of the program (as defined by the program source code) — The design of the program (tasks and operating system) can be short, long, data intensive, interrupt-driven, polling, short or long time-slices, task layout on multiprocessors ...

The compiler (the translation from source to machine code) — Optimizing the code by loop unrolling, use CPU registers instead of main memory etc. can affect execution time substantially.

The hardware — Number of processors, processor type, bus bandwidth, memory latency, cache memory and instruction pipelines states how fast the machine instructions will be executed.

All these factors affects the WCET, but also the Best Case (BCET) that can be equally important (e.g. a too early shot of a air bag in a car can be fatal for the driver). Execution time analysis is preferably performed statically, which means that it is *calculated* (index C) and overestimated from *actual* (index A) since the analysis must take a very pessimistic approach and include for instance infeasible execution paths and maximal iterations of all loops. Figure2.1 illustrates the basic execution time measures.

2.3.1 Software analysis

To test all possible execution paths in a program is not a realistic way to find out the WCET_A . A simple example is a sequence of code with 35 if-statements, which means that there are 2^{35} execution paths. If each test takes 1 millisecond to perform, the search for WCET_A would last for more than one year. Another example is to test a function *foo(x,y,z)* where *x,y* and *z* are 16-bit integers. To test all possibilities, one would need $2^{16} \times 2^{16} * 2 \times 2^{16} \approx 3 \times 10^{14}$ test cases or with a millisecond test over 9000 years. A more analytical and smart method is therefore necessary. A crucial fact of computer software is that analysis and testing is hard to perform since traditional mathematic and scientific methods are (almost) unusable on computer software. Thane[Tha00] states two key accounts:

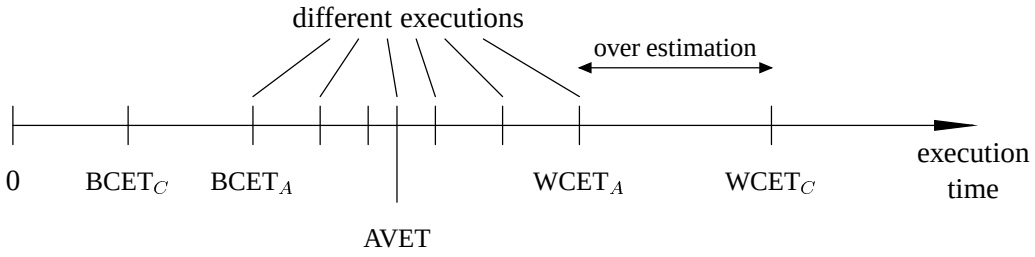

Figure 2.1: Basic execution time measures (from [Gus00])

- They have discontinuous behavior.
- Software lacks physical attributes like e.g., mass, inertia, size and lacks structure or function related attributes like e.g. strength, density and form

The only physical attribute that can be modeled and measured is *time*.

2.4 Cache memories in RTS

To calculate a tight WCET$_C$ in a system with cache memories is very tricky since the contents of the cache memory relies on the programs former execution path. The execution path is on the other hand depending (to some extent) on the cache contents! This section will give a very brief description about cache issues in real-time systems and a more deep survey will be presented in chapter3.
A naive approach to calculate WCET$_C$ would be to handle all memory accesses as misses but that would end up with a WCET$_C$ that would be many times worse than a system without a cache, which will be shown in the next example. In that case it would better to turn of the cache, which have been the case for safety critical hard real-time systems.
There are two categories of block swap-outs [AHH89];

- intrinsic (inter-task) behavior depends of the tasks internal design and execution path. Two functions or data areas in the task may compete for the same cache space and increasing the cache size and/or associativity can reduce the effects.
- extrinsic (intra-task) behavior depends of the environment and the others task's inter-task behavior. At context-switch (preemption) the cache contents will be more or less displaced by the new running task. This performance loss is also called *cache related preemption delay*. In [BN94] the preemption's impact on WCET$_C$ is defined as:

$$WCET'_C = WCET_C + 2\delta + \gamma \tag{2.2}$$

, where $WCET'_C$ is the cache affected WCET$_C$, δ is the execution time for the operating system to make a context-switch (two are needed for a preemption) and γ symbolize the maximum cache related cost by a preemption[2]. The effects can be reduced by *partitioning* the cache and assign tasks to private partitions of the cache.

[2]Observe that γ_i in equation 2.1 is symbolized as δ in this equation

Even if both kind of swap-outs can be reduced, the real-time aspect remains as long as hit-ratio is less than 100% Several studies how the cache performance can be predicted will be presented later in this chapter.

2.4.1 Write-back or write-through?

Write-back strategy gives in the average case better performance than write-through. In execution time analysis write-back strategy will lead into looser WCET$_C$ prediction since it is harder to analyze. Each write in a write-through takes the same time but write-back has two different execution times depending if the data is dirty or not. See figure 1.4 for equations.

2.4.2 Better and worse performance with cache memory = loose WCET$_C$

Performance speedup gained by cache memories relies totally on program behavior and locality. Programs with poor locality will thus gain less speedup. A greater problem is that systems with cache memories can give *worse* performance than without which will be illustrated with the following example that copy eight words from memory section A to B.

	Label	Instr.	Operands	ET	Block
1	.data				
2	DataA:	DC.L	2048		B1,B2
3	DataB:	DC.L	2048		B3,B4
4					
5	.code				
6		MOVEI.W	D0,#8	IF+3	B5
7		MOVEA.L	A1,DataA	IF+3	
8		MOVEA.L	A2,DataB	IF+3	
9	Loop:	MOVE.L	(A1+),(A2+)	IF+3+DF+DF	
10		DBRA	D0,Loop	IF+3	B6

Assume that data and instruction are stored in 32-bit words. A memory access costs 10 clock cycles and beside the instruction fetching and data access(es) 3 additional cycles are needed to execute an instruction. The program starts with 3 initial instructions and the loop that copies data consists of two instructions that makes two data accesses each revolution. This makes 19 instruction fetches and 16 data accesses.

- The execution time (ET) without a cache would take $19(10+3)+16(10) = 407$ cc.
- BCET on a system with a cache memory would occur if no thrashing occurred and only initial misses to fill the cache would occur. Assume that a cache hit takes one cycle and a miss four accesses (40 cc). The first instruction block B5 with the instructions 6–9 would generate one miss and three hits. The loop would generate two data misses every fourth revolution (in this case 4 misses and 12 hits) and on the instruction side one initial instruction miss in B6 but then seven hits; Instructions: 2 misses, 17 hits and data: 4 misses, 12 hits; BCET $= 2(40+3)+17(1+3)+4(40)+12(1) = 326$ cc.

  The poor performance speedup depends of the fact the copied data isn't reused (yet).

- $WCET_A$ on a system with cache occurs if almost all data and instruction thrashes each other which could be the case with a unified 2kB direct mapped cache. All data will thrash each other including the instructions. B1, B3 and B5 is competing for the same block and B2, B4 and B6 does the same. This will lead to 25 misses and 10 hits; $WCET_A$ =1067 cc. If extrinsic misses also occur we'll get 100% miss = 1457 cc

Even if cache memories makes life more difficult to live for the real-time society, Kirk comes to the following thoughtful conclusion in [Kir88];

> *"...real-time systems are often characterized by predefined task sets. For this reason, program behavior is available at system configuration time, and can be used to enhance predictability and sometimes the performance of a cache ..."*

This can also be interpreted that cache memories are more suitable for real-time programs than other types of programs due to it predefinition of work and cyclic behavior! Kirk also states;

> *"...it is possible to improve the hit ratio for a given task, while at the same time guarantee a certain minimal hit count. The guaranteed minimal hit count is then used to reduce the worst case execution time."*

Instead of turning off the cache to be safe but under-utilize the processor, the cache memory can – if correctly used – make unschedulable task sets due to lack of performance to schedulable.

2.5 Other unpredictable hardware and hardware issues in Real-Time systems

Before going further and describe suggested solutions of the problems with cache memories in real-time systems, it's worth mention some related and similar problems in the area of real-time, execution time analysis and unpredictable hardware. This section isn't meant to cover the field completely since this thesis' main focus is on cache memories — it will show some examples and show question similarities with cache memories.

2.5.1 Translation look aside buffers and virtual memory

A large memory makes it possible to run large and more programs simultaneously. *Virtual memory* where parts that doesn't fit into primary memory can be stored at a magnetic media instead is an economic way to make a large memory. It will conceptually work like the cache and primary memory with the principle of locality as a "grant" for high performance. Virtual memory is divided into segments or blocks with a size of 4-64kB. A table points at blocks in the memory and the disk and software maintains writes (copy-back) and exchange algorithm (almost always fully associative). One tricky part is that this table that keeps records of all blocks is large and also is stored in the virtual memory that leads to the problem that each memory access must be translated twice. To speed up this translation the most recent translation are stored in a cache-like memory buffer — a *translation look-aside buffer* or TLB.
Virtual memory can also be used in diskless systems and all the applications can fit into the primary memory. One reason to this is special hardware requirements where certain addresses are mapped to special hardware. An other reason is the extended possibility to store programs, functions and data at

memory addresses where they won't interfere with other data in the cache — *software partitioning*. See chapter 3.2.2 for a more detailed description.
A miss in the primary memory leads to a data swap from and to the disk, which can takes millions of clock cycles with a very large variation. The access time depends of distance of the requested sector on the disk and the reading head. Best case is when the sector is just to be passed and worst case is if it passed right after the request. This variation and the variation of the translation time make execution time analysis very hard to cope with. Virtual memory is therefore not very common in real-time systems — it is much easier to equip the system with a large primary memory. In very small or embedded systems, hard disks themselves are also rare due to their physical and mechanical weaknesses.

2.5.2 Instruction pipelining

A traditional CPU executes instructions by first fetching an instruction, decode it, fetch operands and data from memory, execute, write back the result and then restart the sequence by fetching a new instruction. This approach is easy to implement by using micro code to control the internal micro-architecture and is also easy to specialize and upgrade the instruction set. Since memory technology couldn't feed the CPU with instructions and data in such high pace as the CPU could handle, and this gap was widened, an easy way to increase performance is to get more done at each instruction. Complex and huge instructions sets was a way to bridge the gap between memory access speed and CPU execution pace — the Complex Instruction Set Computer (CISC) was a solution and a fact.
By dividing each logical phase of the execution of an instruction into physical stages, a performance improvement equal to the number of stages can be achieved if all stages take the same amount of time. When a stage is completed with its task it will feed the next stage — like workers building cars at an assembly line.
With cache memories the problem with feeding the CPU-pipeline with a new instruction at each clock cycle is almost eliminated.

Pipelining conflicts

A problem with instruction pipelining is the presence of different hazards that can occur. There are three kinds;

- data hazard – an instructions operand isn't updated since it is still in the pipeline handled by a close previous instruction.

  ```
  ADDI R1,R2,#5          # R1 <- R2+5
  ADD  R5,R1,R3          # R5 <- R1+R3
  ```

  This can be solved with several techniques [HP96];

  - *stall* the ADD instruction and wait until ADDI is completed
  - *forwarding* data values through multiplexors as soon as R1 is calculated
  - using *score boarding* or *Tomasulo's algorithm*

- structural hazard. A resource is used and a new instruction needs it to complete its task. A data cache miss can stop other LOAD, STORE but maybe even other instructions to pass the

MEM-stage of a pipeline. A complex floating-point instruction takes more than a clock cycle to complete. Solutions

  - *stall* until the previous instruction has completed its task in the stage — inefficient
  - *duplicate* the hardware resource so more instructions can be handled at the same time — expensive and may under-utilize the resources
  - make a *bypass* so instructions that doesn't have to use that particular stage of the pipeline can sneak around — may on the other hand cause *"anti-conflicts"*

- control hazard. A conditional or unconditional jump instruction is fetched and the instruction fetch stage feeds the pipeline with (possibly) wrong instructions. The target address (new Program Counter value) is not yet calculated and the condition is maybe even undecidable due to data hazard.

  - *stall* until the correct value of the program counter is calculated.
  - letting a specific number of instructions after the branch instruction always execute and perform a *delayed branch*. These instructions must be independent of an eventual condition in the branch. The compiler preferably handles this reorganization of code. The tricky part is to find enough independent instructions that can be stuffed behind the branch and if that doesn't succeed NOP (NO Operation) instructions must be used.
  - Calculate new PC and test the condition in an earlier stage to reduce penalty.

  By using prediction units that memorize previous jumps, the stalls affected by a control hazard can be reduced and performance increased in most cases. This means that branches in modern hardware in a real-time angle is even more trickier; the unit that memorize previous jumps is mostly organized like a cache; this means that there is a certain probability that the branch isn't logged in the memory with less probability to make a correct guess, and please observe *probability*.

  Interrupts and preemption are on the other hand hard to predict and will always end with a flushed pipeline.

Pipelining and cache memories

The concept of pipelining is an efficient way to increase performance but the presence of hazards, stalls and miss prediction of branches, execution time may vary and give a looser WCET_C bound. In a system with both cache memory and pipeline these hazards may hide each other and give an even looser WCET_C -estimation. Executing three instructions in a pipeline takes under ideal circumstances only three clock cycles (plus some initial instructions to fill up the pipeline). The following code with independent instructions (in two separate cache blocks);

```
STORE R0(1000),R2 # instruction cache miss and data hit (3 cycles penalty)
LOAD R3,R5(2000) # instruction cache hit and data miss (3 cycles penalty)
LOAD R4,R5(2004) # instruction cache hit and data hit
```

. . . will however have less luck . . .

Stages	1	2	3	4	5	6	7	8	9	10	11	12	13
I Fetch	ST	ST	ST	ST	L1	L2							
I Decode					ST	L1	L2						
Execute						ST	L1	L2	L2	L2	L2		
Memory							ST	L1	L1	L1	L1	L2	
Write back												L1	L2

These instruction would need 13 cycles to execute but if they would be in the following order;

```
LOAD R3,R5(2000) # instruction cache hit and data miss (3 cycles penalty)
STORE R0(1000),R2 # instruction cache miss and data hit (3 cycles penalty)
LOAD R4,R5(2004) # instruction cache hit and data hit
```

. . . they would need . . .

Stages	1	2	3	4	5	6	7	8	9	10	11	12	13
I Fetch	L1	ST	ST	ST	ST	L2							
I Decode		L1				ST	L2						
Execute			L1				ST	L2					
Memory				L1	L1	L1	L1	ST	L2				
Write back								L1		L2			

. . . only 10 cycles to execute. The instruction cache miss by ST is hidden by the L1-miss (or vice versa) and gain 3 cycles. This example shows that if penalties (or individual instruction's execution times) are added together, the WCET$_C$ -analyze will be even looser. Further reading about pipelining and real-time aspects can be done in for instance [HAM+99], [LS99b] or [LBJ+95].

2.5.3 Direct Memory Access - DMA

A Direct Memory Access (DMA) controller can transfer data between a hard disk or an other I/O device and the primary memory with reduced CPU involvement (figure 2.2). The CPU is able during such a transfer to concurrently execute instructions. The presence of a DMA arises also cache coherency problems that make a snooper and a snooping protocol necessary.
In a real-time system a DMA introduces new problems since the bus might be taken by the DMA when the CPU needs it with a delayed execution time as a result. A safe approach of such a problem would assume that $WCET_{total} = \sum (WCET_{CPU-operations}) + \sum (WCET_{DMA-operations})$, but this over pessimistic assumption results in a loose WCET$_C$. CPU clock cycles are either categorized as bus-activity (B), such as instruction fetching or data load/write, or as execution cycles (E).
Beside a pure and faster data transfer, the DMA is able to "steal" clock cycles during the execution phase of an instruction and utilize the bus even more. The stolen E-cycles can also be described as parallel work that will be considered as serialized in a safe WCET$_C$ calculation. All the E-cycles can although not be used since bus arbitration (bus master transfer time — BMT) takes some time and delay the switch.

Huang and Liu describe in [HL95] an algorithm to calculate a tighter WCET$_C$ than the described safe approach; For each type of instruction in the instruction set the useful DMA stealth cycles are defined. The next step is to sum up all E-cycles and subtract them from the DMA clock cycle need during the instruction sequence. If the result is positive $WCET_{total} = \sum (WCET_{CPU-operations}) + \sum (WCET_{DMA-"leftover"})$ and if it is negative (all DMA activities are done in parallel) $WCET_{total} =$

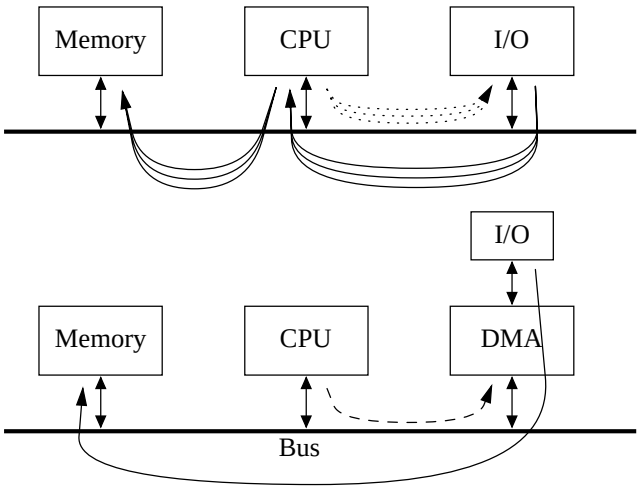

Figure 2.2: An I/O operation in a system without and with a DMA controller

$\sum(WCET_{CPU-operations})$. The calculated $WCET_C$ was compared with simulated results on a MC68332 that ran a benchmark set. Their algorithm achieved for instance a 39% improvement in accuracy of the prediction on a matrix multiplication on a fully utilized I/O bus.
The method is however not applicable on systems without instruction pipelining and cache memories. Instruction pipelining fetches new instruction during the execution phase that will give no time left for other bus masters. A cache memory could on the other hand solve this problem by hiding the CPU activities from the bus, but then the suggested model would not be applicable at all.

In [HLH96] the method is extended to include direct mapped instruction cache memories (but not pipelining). Their approach is an extension of a method developed by Li, Malik, and Wolfe [LMW95] (se also section 3.1.1 in this work). In a system without cache all instructions starts and some ends with B-cycles. This means that no cycle stealing is possible between instructions. In cached system on the other hand, instructions may hit in the cache and cause E-cycles only — even between two cache blocks if they are present in the cache. The method doesn't handle set-associative caches or data caches. Nor does it handle scheduling and preemption issues.

2.5.4 High priority hardware interferences

Park and Shaw observe and measure the real-time effects of interfering hardware in [PS91]. A way to cope with these interferences is to also schedule their activity among the applications' tasks.

Interrupts

In real-time systems, time is of great essence and must be at hand to synchronize processes and maintain scheduling. Park and Shaw measured the effects caused by periodical timers with an oscilloscope on a Motorola 68010-system. Timers normally generate interrupts to the processor that increase a clock counter by a step. The interrupt handling and the instruction execution are a load and delay in the system.

Dynamic Memory refreshment

Dynamic RAM consists of a transistor gate and a capacitor where the capacitor stores the 0 or 1. Due to the capacitors self-drain it must be refreshed periodically every 13 microseconds. This memory refreshment is done completely in hardware and can be considered as a high-prioritized task that can delay any software instruction. Park and Shaw measured this cyclic task to a processor slowdown by 0–6.7 percent with an average of 5 percent.

Chapter 3

Analysis on cache memories in real-time systems

3.1 Intrinsic interference

3.1.1 Integer Linear Programming (ILP) methods

From a program graph and additional constrains (eg. manual annotations indication infeasible paths, maximum iterations in a loop etc), WCET_C can be calculated with *Integer Linear Programming* (ILP) optimization. The use of ILP as an analytical method has numerous advantages;

1. the graph construction can be based on many different levels of code; object, source etc., which makes the method general.
2. It supports control-flow breaking constructs like `return` and `goto`-statements
3. Manual annotations are supported which simplifies an estimation of a tight WCET_C
4. Except for the WCET_C , also other information about the program behavior will be available.

Cinderella - ILP

Basic method. Li and Malik[LM95] from Princeton University identified two main components of WCET_C analysis:

- *Program path analysis* — determines the sequence of instructions to be executed in for instance the worst-case scenario.
- *Microarchitecture modeling* — how long time it takes to execute the scenario in the actual hardware.

Instead of searching through all possible execution paths, the method analytically determines the worst-case scenario by *implicit path enumeration* by converting the problem to sets of integer linear problems with manual annotations. When the ILP are solved the number of executing instructions are defined and by the assumption that each instruction takes a constant time to execute, the total WCET_C can be computed by

$$WCET_C = \sum_{i=1}^{N} c_i x_i \tag{3.1}$$

where x_i is the instruction count[1] of a basic block[2] B_i, c_i the constant execution time of the basic block and N is the number of basic blocks in the program.

Method extension to include an instruction cache. To include instruction cache memories the method was extended[LMW99] by dividing the basic block into *line-blocks* or simply *l-blocks*[3] that corresponds to each line in the cache. An *l-block* is defined as *"a contiguous sequence of code within the same basic block is mapped to the same cache set in the instruction cache memory"*. Figure3.1 (from [LMW99]) illustrates a *Control Flow Graph* (CFG) with three basic blocks and these basic blocks' mapping into l-blocks into a cache with the size of four sets. Two l-blocks that are mapped to the same cache set will conflict with each other (in the example basic blocks B_1 and B_3 will conflict), but in case where the basic block boundary doesn't align with the cache line boundary two line-blocks mustn't conflict. Those are called *nonconflicting l-blocks* and an example is $B_{1,3}$ and $B_{2,1}$ shown in figure 3.1.

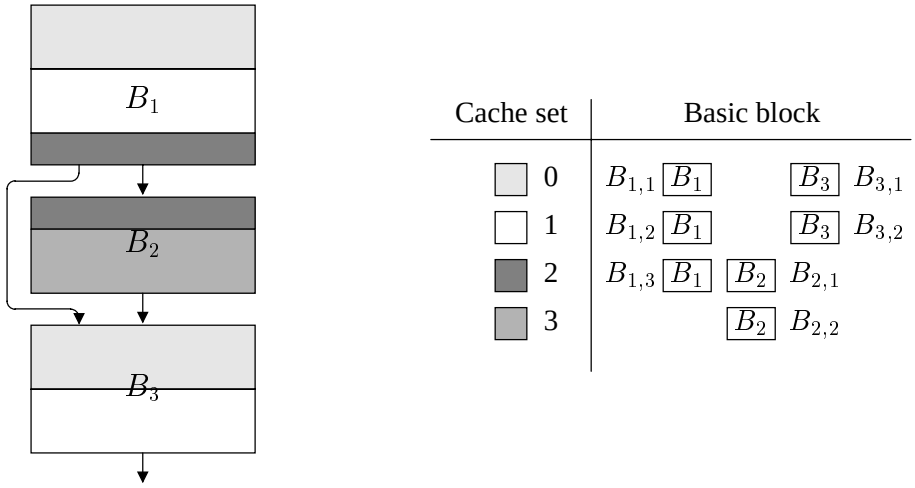

Figure 3.1: The above example shows the relationship between l-blocks and basic blocks.

By adopting new cache related terms into the analysis, the formula 3.1 must split and consist of one *hit-term* and one *miss-term*.

$$WCET_C = \sum_{i=1}^{N} \sum_{j=1}^{n_i} (c_{i,j}^{hit} x_{i,j}^{hit} + c_{i,j}^{miss} x_{i,j}^{miss}) \tag{3.2}$$

[1]The *instruction count* is by Li and Malik defined as the number of executed instructions within a limited space of a program

[2]A *basic block* is a linear sequence of instructions without halt or possibility of branching except at the end

[3]Li and Malik have chosen to call the smallest cache item for *(cache) line* and not *(cache) block* to reduce semantic confusion with *basic block*

After identification of l-blocks and their relationship according to conflicts a cache conflict graph (CCG) is constructed for each cache set that contains conflicting l-blocks. During WCET_C analysis, counters sum the hits and misses for l-blocks in the basic block and is then later used to calculate WCET_C for the complete program.

Modeling and results. All analysis methods are implemented in a tool called `cinderella`, which estimates WCET_C running on an 20MHz Intel i960KB processor with 128kB main memory and several I/O peripherals. The tool reads executable code, constructs CFGs, CCGs, and asks the user to provide loop bounds whereas a WCET_C bound can be computed. To solve ILP problems, `lp_solve` by Michel Berkelaar is used.
The results from the method have given close results to actual WCET_A — especially for small programs.

Ottosson and Sjödin

In [OS97], Ottosson and Sjödin extend the *Implicit Path Enumeration Technique (IPET*[LM95] (described in section 3.1.1) to also include pipelining and cache memories. The cache is modeled with non-linear arithmetic and logical formulas, which are available for a finite domain constraint solver. The model can describe a unified set-associative cache since instructions and data is handled in the same way. The method is also feasible to model instruction pipelining, but it is very compute intensive and suffers from performance problems — especially when large programs or loosely constrained references are given as an input.

3.1.2 Static analysis with graph coloring

One powerful method to optimize register allocation for or instance compilers builds on a color mapping technique by Chow and Hennessy[CH90]. By identifying variables with overlapping lifetimes, the optimization can be performed with a method analogous to the problem of graph coloring, which is a colored graph where no adjacent nodes have the same color[4]. For register allocation, a node represents a variable and variables with overlapping lifetimes are connected by an edge. Each color symbolize user registers and their method assigns colors to *"live ranges"* (that corresponds to the variables lifetime) in the order of priorities that depends on anticipated performance speedup. The remaining variables that are uncolored (maybe even uncolorable) must share registers and therefore leverages are split and new graphs created.

Jai Rawat

A master thesis from Iowa state university by Jai Rawat[Raw93] describes a static analysis method to find intrinsic misses in a direct mapped data cache memory.
Rawat's technique for static analysis of cache behavior is similar to Chow and Hennessy's register allocation. The (data) cache block contains variables from where live ranges are specified. These "variable live ranges" specifies the block's live range. Splitting of live ranges correspond to replacement of the block in the cache. The analysis algorithm is as follows;

[4]That an arbitrary map with distinct areas (for instance countries with shared borders) can be established with four colors or less, has been a well known fact for several hundred years. Mathematicians have tried to prove this statement mathematically, but not until 1976, a mathematical proof (yet very discussed since it was non-analytically performed) was found. To determine the minimum amount of colors to a graph of arbitrary dimension is a NP-hard problem.

1. Live ranges on variables and memory locations are calculated
2. Variables with highly intersecting live ranges are grouped in sizes of cache blocks. The block's live ranges are determined.
3. An interference graph for each cache set where edges bind variables to the same cache block (node) and which live range is equal is constructed.
4. Live ranges are split in case of a possible cache conflict.
5. The amount of cache misses is estimated through summing up all frequencies of all live ranges at all memory blocks that are used in the program.

The results of the estimated performance were compared with simulations. Five small test programs were simulated on a DINERO cache simulator with a DLX[HP96] instruction set as a target. The test programs had a hit ratio between 59–78%. All estimations were safe and overestimated the miss ratio (high miss ratio yields longer execution time) to 5–37%. By identifying the live ranges, variables could be regrouped to fit into the cache in a more beneficial way. Hit ratio could in one case by this rearrangement be increased from 49% to 94%.

Discussion. There are many limitations in this work; it handles only data on write-allocate with random replace and write-back strategy. It doesn't support function calls, dynamic memory management, and global variables. All loops must iterate 10 wraps (no more, no less). All analysis is on source-code thus compiler optimizations will not be taken into account. A cache block that has been classified as a missing one is always a miss — even in a tight loop. Yet, the work is only a master thesis and further research could possibly solve many of the mentioned limitations. One reason why no one has picked up this "loose end" to complete the work is that the method will only handle scalar variables (not arrays or dynamic memory) and they are mostly handled as registers by optimizing compilers.

3.1.3 Abstract Interpretation

Abstract interpretation is a formal method that analyzes a program statically and gives safe information about the execution properties of the program. It formalizes the idea that the semantics can be more or less precise according to the considered level of abstraction. The approach renders approximations that might include incomputable results, represented as "I don't know". The result is however, even if an approximation, always safe and is never underestimated.
The main advantages with abstract interpretation [CC77] are:

- it yields run-time properties through static analysis
- with the suitable choice of abstractions, the analysis will terminate, even for non-terminating programs
- since the calculations are made on simple abstractions, the calculations are typically much faster than the concrete executions
- the results are always provable safe

The definition is "*An abstract interpretation of a program P is a pair of* $(\mathcal{D}_P, F_P)$ *such that* $\mathcal{D}_P$ *is a complete lattice and* F_P *is monotone.*"

Alt, Ferdinand, Martin, and Wilhelm

A group at Universität des Saarlandes in Germany use abstract interpretation to predict set-associative instruction and data cache behavior. The analysis is performed at single program hence only intrinsic cache behavior is studied and predicted. Their method categorizes basic blocks into *may* and *must* be in the cache which derives two safe states; *always hit* and *never hit* from which the analysis defines a WCET_C for each basic block [FMWA99]. Loops and iteration can however render a very pessimistic WCET_C since the basic block in loop can be classified as "may" (never hit). This might be true in the first iteration but since the instructions and possibly also data is reused each iteration, the estimated WCET_C would be looser for each iteration. To cope with this problem, loops are *unrolled* one step to "initialize" the cache before entering the actual loop. The initializing block can be classified as "may" but the basic block in the loop will be classified as "must" (always hit). Loops are treated (in the analysis) as procedures to be able to use existing methods of interprocedural analysis such as the *callstring approach* and an own method called *Virtual Inlining and Virtual Unrolling*. Figure3.2 illustrates an example of a loop transformed to a recursive procedure call.

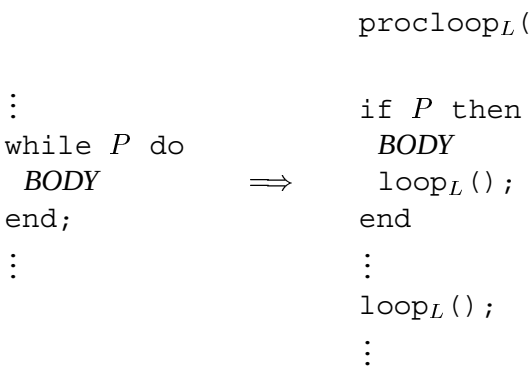

Figure 3.2: Transformation of a loop to a recursive equivalent

The cache analysis technique is implemented into PAG (Program Analyzer Generator)[AM95] that takes the control flow graph and cache design description as an input. The static cache analysis together with the program path analysis generates WCET_C , BCET_C , cache behavior, and other data. The analysis can take several minutes for a small test program. No special input of a skilled user is required to tune the analysis for acceptable performance.

Theiling and Ferdinand

This work continues the previous by combining ILP with abstract interpretation[TF98]. The cache block analysis is extended to, beside *must* and *may*, also have a *persistent* classification. This is possible when a first execution of the block may result in a hit or miss in the cache, but all consecutive references results in hits. The *may* categorization is leads to very pessimistic WCET_C -estimations in loops since those normally starts initially with misses at the first iteration but when instructions are loaded, the cache will yield hits. This categorization is very similar to the Florida research group (described in 3.1.4) as the *first miss*.
The analysis is performed roughly in the following steps:

1. Blocks are categorized as always hit, always miss, persistent or not classified by join functions controlled by abstract interpretation rules.

2. Loops are transformed to recursive equivalents.
3. The program path is described as ILP problems and constraints are automatically generated from the control flow graph.
4. A cost function computes how many CPU cycles the program needs to execute.

This work is implemented into `PAG` and `lp_solve` (by Michel Berkelaar) is used to the solve ILP problems. The results have been compared with the "Florida research group" by using parts of the same test suit as they did. On optimized code the accuracy of the predicted analysis was between 0.5-8% from the traced WCET_A . Code that hasn't been optimized by the compiler could be over estimated by over 50%

3.1.4 Data flow analysis

Florida research

The approach of these researchers from Florida state university is to produce timing data from data-flow analysis through a *static cache simulation* and was first published in [MWH94]. To analyze the behavior in a direct mapped instruction cache the following steps were made;

1. A control-flow graph is constructed out of a compiled C-program
2. The graph is analyzed to detect possible instructions that compete for the same location in the cache memory.
3. Each instruction is categorized by its behavior.

For an overview behold figure 3.3. At this stage recursion and indirect calls could not be analyzed.

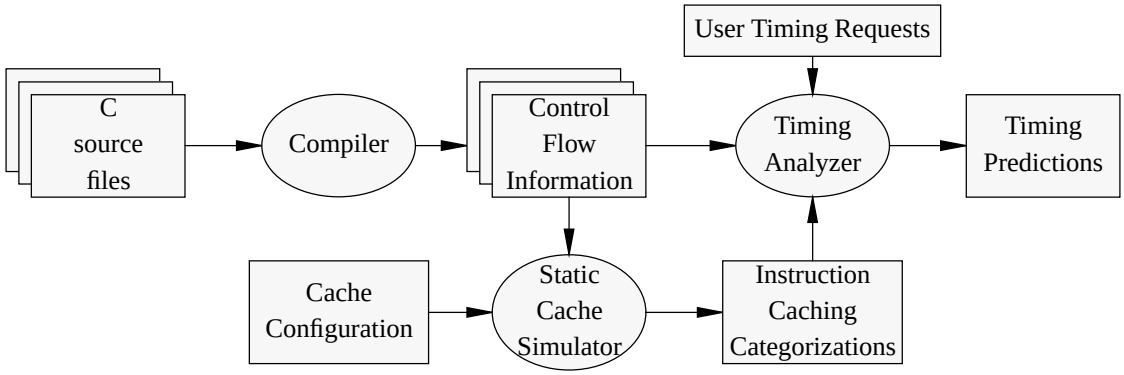

Figure 3.3: Overview of the flow to bound instruction cache and pipeline performance.

The method categorize the caching behavior of each instruction into one of the following four categories;

- **Always Hit** (ah) – the instruction is always in the cache. After a miss the consecutive instructions within the block will also be loaded and those will ride on the principles of spatial locality but also guaranteed in the cache (if preemptions, traps to OS and interrupts aren't allowed).

- **Always Miss** (am) – the instruction is never in the cache
- **First Miss** (fm) – at the first access of the instruction, the cache will generate a miss, but all sequential accesses will generate hits. This is a common situation in for instance loops (illustrates also temporal locality), where the first iteration renders a miss but when the instruction is loaded into the cache it will generate hits.
- **First Hit**[5] (fh) – the first access is a hit but all consecutive accesses will be a miss. This situation can occur in for instance the following scenario; A piece of code with the instructions $\{i_1, i_2, \ldots, i_{n-1}, i_n\}$ will be executed where i_2 to i_n are the border instructions to a loop. If the first and last instructions are members of a cache block that compete for the same spot in the cache, i_1 will be categorized as *always miss* (if it's the first instruction in a cache block, otherwise *first hit*), i_2 as *first hit*.

After the cache simulation the *timing analysis* is performed to bound WCET_C . The tool interactively asks the user to assign each loop the maximum amount of iterations that the compiler couldn't automatically determine. In the next step the analyzer constructs a *timing analysis tree* and the worst-case cache performance is estimated for each loop in the tree. After these steps the user can request timing information about parts, functions or loops in the program.

In [Whi97, WMH+97] direct mapped data cache memories were included into the model. The approach works on optimized code and exploits both temporal and spatial locality. Limitations in the method still concern recursion and indirect calls. The four instruction cache categories can be used for scalar data references but was clumsy to handle arrays, vectors, and strings and an extension of the categorization was needed. The new fifth state **Calculated** (c) has a counter associated and keeps a record over the maximum number of misses that could occur at each loop level in which the data reference is nested.
Example; Assume a large data cache memory that could hold four words in each block. Scalar variables will reside in registers to make the example clearer. To sum 50 elements in an array can be accomplished with `sum += a[i];` which would render a sequence of misses (m) and hits (h);

```
m h h h m h h h m h h h ...m h h h m h
```

This data reference will be categorized as

```
c 13
```

If the next statement after this loop was a *new* loop that accessed `a[i]` again and the cache was big enough to hold all those previous values, this line would be categorized as *always hit*. The method yields an average 30% tighter WCET_C than an analysis without data cache issues.

In [WMH+99] a timing analysis method on *wrap-around fill caches* (also called *critical word first*) is presented. The method detects also pipeline-hidden misses to bound WCET_C even tighter and the results gave an average tighter WCET_C from 1.38 times WCET_A to 1.21 (14%)

The method was later extended to also include pipelining [HWH95, HAM+99] (see also section 2.5.2) and instruction set-associative cache memories[WMH+97].

[5] Actually in [MWH94] the fourth state was **Conflict** – the state of the instruction is undeterminable since the instruction might compete of the space in the cache with another instruction. All instructions that cannot be categorized by the three others will be assigned to this cache state. The handling of this state was however the same as *always miss*. In later work this state was exchanged by *first hit* to bound WCET_C tighter.

Those extensions results in a overestimation of 1.32 times (32%) on average on a benchmark set compared to a conservative disabling of the cache that renders 9.25 times overestimation (= underutilization) of WCET_A .

Discussion The method has been criticized to have a to conservative view of the categorization model in the simulator[LBJ+95]. Assume that the blocks B1 and B2 (but no other blocks) compete for the same space in the cache in the following code;

```
for(i=0;i<N;i++) {
   ...
   S1:      (B1)
   ...
   for(j=0;j<M;j++) {
      ...
      S2:      (B2)
      ...
   }
}
```

In this case only the first reference in S2 will cause a cache miss and all other will be hits. The cache simulation will however not categorize this block as *first miss* but as *conflict* or *always miss* with a loose WCET_C as an result.

Sung-Soo Lim *et al*

To not suffer from the pessimistic WCET_C as mentioned under discussion in section 3.1.4 these Korean researchers have proposed an alternative method to bound WCET_C with respect of instruction caches in [LML+94]. A small and simplified example will open the explanation of the approach.
Assume the access sequence of cache blocks $\{b_2, b_3, b_2, b_4\}$ on a direct mapped cache of a size of two blocks. The cache contents before the sequence is unknown. Even addresses $(\{b_2, b_4\})$ will be mapped to block 1 and odd $(\{b_1, b_3\})$ to block 0. The flow can be seen in figure 3.4.

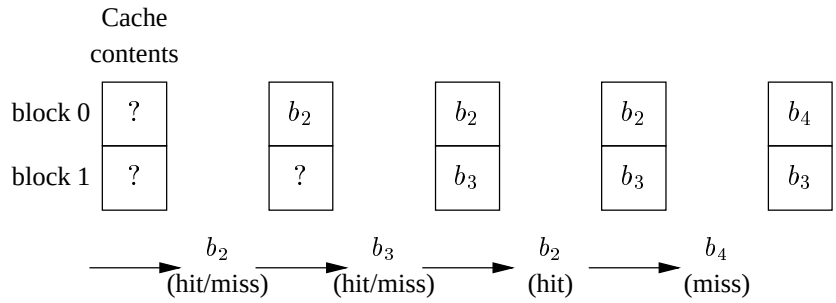

Figure 3.4: A sequence of cache memory accesses

The method to track the cache state and from this calculate WCET_C is to store timing information and the state of the cache of each block access in a structure (“atomic object”);

```
struct timing_information {
   block_address first_reference[NO_OF_BLOCKS];
   block_address last_reference[NO_OF_BLOCKS];
   time t;
}
```

The `first_reference` stores the blocks for which hits and misses depends on the cache contents before the sequence is entered and `last_reference` is assigned the view of the cache when the sequence ends after `t` clock cycles. In the shown example the structure will be $\{\{b_2, b_3\}, \{b_4, b_3\}, 38\}$ On selections (`if (`*exp*`) then` S_1 `else` S_2) those timing information structures are *concatenated* and *unionized* in such a way that the new

$$c = \{c_{exp} \oplus c_1\} \cup \{c_{exp} \oplus c_2\}$$

where $\oplus$ is a special concatenation algorithm[6], c_x is a timing information structure which is associated with the instruction sequence S_x. Sets of atomic elements (such as the one derived from the selection) can be simplified by safely removing elements in the set that have smaller WCET_C than others if the block states are equivalent. Others can be reduced by pruning (also a kind of concatenation). Sequences of code and function calls can be pruned and while-loops are handled with the three conservative assumptions that (1) cache-contents is invalid upon entrance, (2) the loop leaves invalid cache contents, and (3) each loop iteration benefits only from the instruction blocks that were loaded into the cache by the immediately preceding loop iteration.
To handle set-associativity the methods must be extended which the authors to the paper claims is an easy matter to implement. To handle data caches a special hardware support is needed to decode a special *"allocate-bit"*. This bit controls whether the memory block fetched on a miss will be loaded into the cache. Loads and stores to memory locations that can be determined statically will have this bit set and other load/stores will have this bit cleared by the compiler. The array-elements in the timing information structure will ignore clear bit operations and handle them as misses.

The method was in [LBJ+95] extended to also include pipelining issues and the approach use concatenate and union methods that are similar to the cache methods.

Discussion The method will handle nested loops as described in a less conservative way than the "Florida research approach", but nevertheless it has some limitations with loops. A (nested) loop that has a selection inside will be very conservative handled since the method will only take advantage of the previous iteration's accesses. If the execution path is altering during each iteration

```
i=0;
while(i<N) {
  if(i % 2 == 0) {
    ...
  }
  else {
    ...
  }
  i++;
}
```

[6] see Figure 5 in [LML+94] for exact semantics of $\oplus$

the method will yield a loose $WCET_C$. Handling dynamic data cache references as always miss is a too defensive approach to bound $WCET_C$ tightly.

Sung-Kwan Kim *et al*

In [KMH96] Kim *et al* propose two different techniques that extends Lim's analysis (page50) to yield a tighter $WCET_C$ regarding data caching. C-G Lee (page 61 and also others use a similar defensive approach on data caches — all dynamic references will conservatively be considered as misses.

Reduce misclassified load/store instructions Since all dynamic references in some methods (for instance Lim (section 3.1.4) or Lee (section 3.2.3)) are considered as misses, it is very important that the categorization of static/dynamic references doesn't overestimate the number of dynamic references. It is common for RISC load/store architectures to only supply a very limited number of addressing modes. Data addresses are in those cases computed by adding a base address to a displacement. Static memory references are located in the local stack (pointed from stack pointer – `sp`) or in a global memory area (pointed from global pointer `gp`) and the base address is in those cases either `gp` or `sp`. This must however not always be true. The base address is still static if it is inherited from `gp` or `sp`. Such an access is illustrated in the following two versions of a code example;

```
...                              ...
addiu R15,sp,#16                 addiu R15,sp,#0
...                              ...
loadw R24,0(R15)                 loadw R24,16(R15)
...                              ...

(a)                              (b)
```

An analysis method will in case if it has a undetailed view of the code, class the `loadw` access in (a) as dynamic only because the base address differs from `sp` and `gp`. The semantically same code in (b) would on the other hand be correctly classified as static. The proposed technique tries to derive `sp` or `gp` from loads and stores by a data-flow analysis so the access can be handled as in the (b)-example.

Reduce $WCET_C$ overestimation on load/store instructions The suggested method is feasible in loops with data arrays and vectors as work sets. Instead of assuming that all accesses are misses, a region of references by each load/store instruction is determined. All blocks in "last reference block set" that corresponds to the loop will be invalidated. The next step is to find an upper bound of distinct memory locations referenced by the set of load/store instructions in the loop nest.

Discussion Even if this is one of the more promising approaches to bound $WCET_C$ on data caches it is still very pessimistic since it cannot handle dynamic load/stores and adding to this also assume that all those accesses will lead to a double miss since a the miss can possibly be followed by a write-back on replacement. Blocks that are only loaded and never written to will not suffer from this assumption, and all accesses are not misses. It might be obvious but for the sake of science it has been proved in for instance [LS99a].

3.1.5 Reducing and approximative approaches

Nilsen and Rygg

Most proposals to compute WCET analysis on modern processors only approaches one or very few of the issues regarding caches, pipelining, DMA etc. One of the very first proposals that described a complete framework to handle both software and hardware issues is described in [NR95]. Pipelining issues are simulated and cache performance is predicted with a live range method described in section7. To ease the prediction a subset of C/C++ is used for programming and analyzed by a special tool called *C Path Finder* (`cpf`). The tool analyzes the control flow on source-level[7] and determines the worst case execution path based on manual annotations in the source code. No experimental evaluations were however made, but the authors claimed that *. . . it seems unlikely that . . . the cache analyzer can predict more than 50% of the actual hits for realistic workload.*

Liu and Lee

In [LL94] the approach is to estimate an exact WCET_C through extensive search in CFG-trees. To reduce search time, the proposal suggests a *"hybrid approach"* that with a number of approximations reduce the search tree. The trade-off between the analysis complexity and tightness of the results is also discussed.

Lundqvist and Stenström

By combining path and timing analyses for programs, Lundqvist and Stenström has developed a method based on *cycle-level symbolic execution* [LS99b]. The method is feasible on modern high-performance processors since it includes cache, superscalar pipelining, and dynamic execution issues. Instead of using manual annotations to bound the number of iterations in loops, the method use automatic path analysis. Infeasible paths are detected and excluded from the analysis. The approach is to analyze all paths in the program and to handle the exponentially growth of test cases, a *path merge strategy* is employed: Each time two simulated paths meet in the program they are merged into one. The crucial point by this handling is to choose the worst-case execution path.
Variable values can either be discrete numerical or *unknown*. By this simple approach WCET_C can be exactly estimated. Results from experimental evaluation showed that six out of seven benchmark programs $\text{WCET}_C = \text{WCET}_A$.

In [LS99c] an improvement to handle data caching is proposed. The approach is to distinguish predictable from unpredictable data cache behavior for data structures. Experimental results show that up to 84% hit ratio in a data cache is achievable on a benchmark program.

Discussion No other method has managed to integrate automatic path analysis and detailed timing analysis into the same method. To handle pipelining, instruction and data caching simultaneously creates a good environment and possibility to bound a tight WCET_C . The approach is simple, but in some cases it seems to be too simple since it cannot handle data domains or sets of data values (as [Gus00] do). For a loop where the number of iterations depends on unknown input data[8], the method will fail to predict WCET_C .

[7] Each expression is treated as straight-line code. Optimized object code cannot be analyzed.
[8] Unknown data can be exemplified with temperature values from a sensor that might be anything between 0–100

Altenbernd and Stappert

Straight-line code without loops and recursive functions that is for instance found in automatically generated code for real-time systems. The *Program Timing Analyzer (PTA)* covers both low and high-level aspects in the analysis. The method presented in [SA97], covers unified and split data and instruction caches, and pipelining. The system is automated which means that manual annotations are unnecessary. An overview of the timing analysis is illustrated in figure 3.5

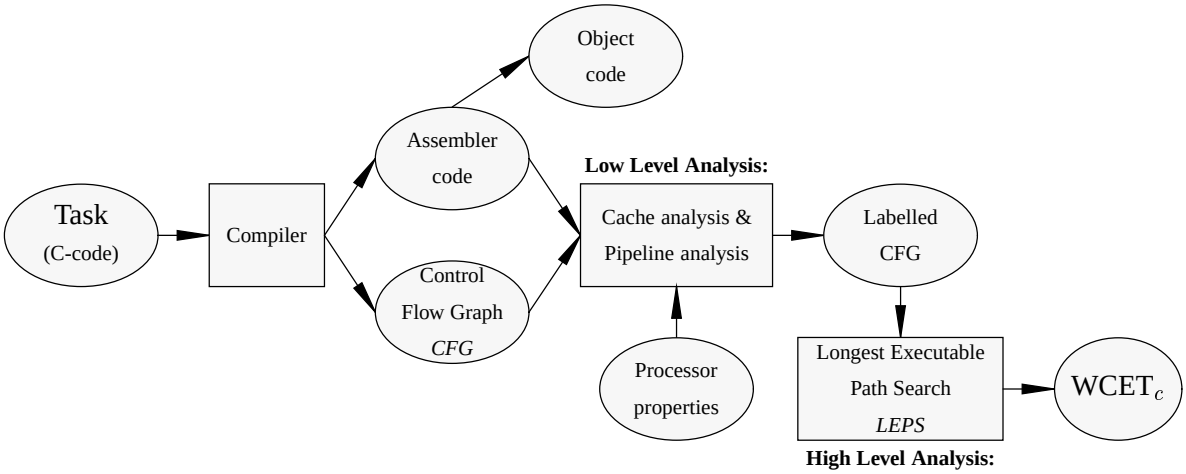

Figure 3.5: PTA system overview

The code supported by the system is a C-subset. PTA searches for the longest execution path in the control flow. Observe that the approach of straight-line code will assume that an instruction that has been accessed will never be accessed again which simplifies the constraints of cache behavior.

The experimental results on four test tasks shows that the frequency of overestimated instruction and data cache misses are between 0–14% and the overestimation on WCET$_C$ is 4–13%. The target system was a PowerPC and the measurement was performed with the built-in performance monitoring facilities. The analysis time was about 2–18% of the total compile time. A naive approach yields 6–8 times overestimation of WCET$_C$.

3.2 Extrinsic interference

Extrinsic (inter-task) interference in the cache can be described in different ways, but Basumallick and Nilsen's approach[BN94] by defining it as an delay to the pre-empted task is the most common and will hereby be used. Cache related preemption delay (CRPD) is a side effect of extrinsic (inter-task) occurrences. The CRPD was measured in [MB91] to 500 micro seconds in UNIX-system with round-robin scheduling by trace driven simulation. This value is although getting a bit old...
The fact that it is a delay makes it more common to discuss *Worst Case Response Time – WCRT*$_c$ in these situations than about WCET$_C$. C-G Lee shows in [LHM$^+$97] that since the cache refill time increases relatively to CPU speed, the CRPD takes a proportionally large percentage in the WCRT of a task.

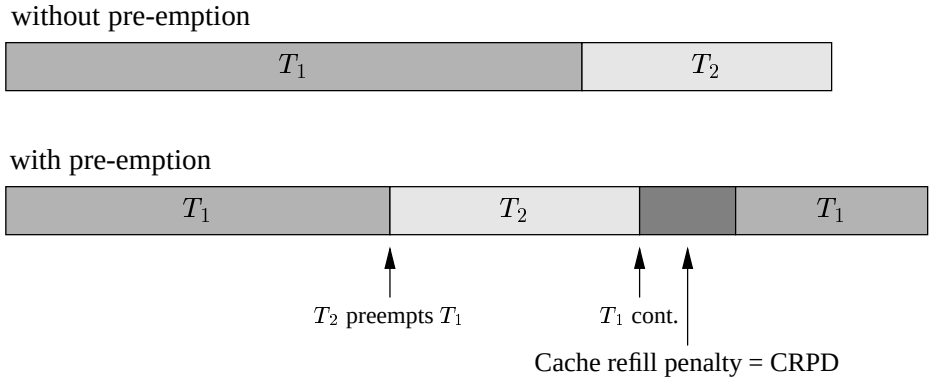

Figure 3.6: The cache related pre-emption delay is a cost

In [BMWSM$^+$96b, BMWSM96a] Busquets-Mataix *et al* identify five different approaches to to assess the refill penalty γ after a context-switch.

1. The time to refill the entire cache.
2. The time to refill the lines misplaced by the preempting task.
3. The time to refill the lines used by the preempted task
4. The time to refill the maximum number of useful lines that the preempted task may hold in the cache at the worst case instant a preemption may arise. Useful lines are those that are likely to be used again.
5. The time to refill the intersection of lines between the preempting and preempted tasks.

The chosen assess approach depends of the analysis approach and its' limitations.
Besides the delay, a cache refill burst is effect costly which is an important parameter when constructing mobile devices that run on batteries [ABR01] . In those cases the CRPD should be eliminated or at least reduced.

This section will be organized as follows; first it will present cache partitioning as a method to avoid extrinsic cache behavior and then some methods to compute the (worst case) CRPD.

3.2.1 Partitioning by hardware

David B. Kirk

Kirk describes in [Kir88] how to divide a full associative instruction cache memory into a static part that is preloaded at context switch and a regular LRU-part. The key concept of the idea is to guarantee a certain hit-ratio since some blocks are fixed resulting a certain amount of hits. If a block should be in the static or the LRU-part can be chosen statically at compile-time or dynamically at run-time by keeping track of how often some blocks are touched. How the preload should be done and what time it could take is however not explained.
In [Kir89] Kirk suggests a partitioning scheme called "SMART" (Strategic Memory Allocation for Real-Time) with a shared pool of blocks and a number of private segments owned by tasks. A private segment will eliminate extrinsic cache interference and reduce cache related pre-emption delay (CRPD) at context switch.

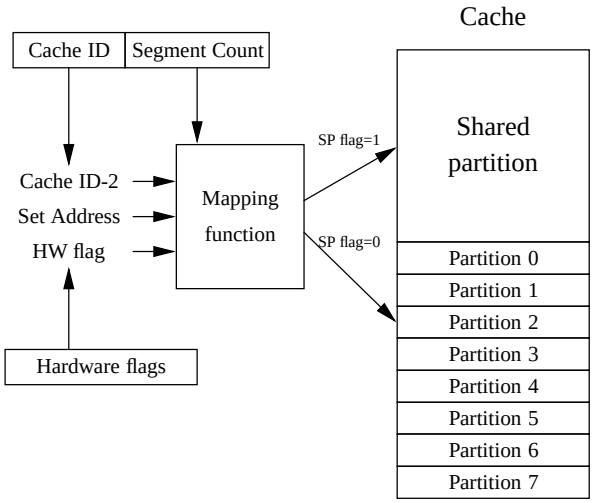

Figure 3.7: SMART cache design

To choose size of the segments and how many segments should be owned by which task an algorithm is presented in [KSS91].

In [KS90] Kirk and Strosnider describe an implementation of SMART on a MIPS R3000 CPU with an 16kB direct mapped instruction cache. The extra hardware needed to handle the partitioning render a 15% performance loss on a 25MHz processor.

Henk Muller *et al*

A group at University of Bristol describes in [MMIP98] a partitioned direct mapped data cache with a modified hash-function. The hash-function needs besides the memory address also additional information such as partition start pointer and partition size to point out the correct partition.
To be efficient, the compiler must be able to analyze access patterns and find out strides on data in loops to know how much data must be shifted in memory to allocate the partition efficiently. In this

case a shift parameter is used in the hash-function. The information is automatically derived from the program without any user intervention.

Discussion

The partitioning of a cache by additional hardware's major drawback *is* the need of a specially designed cache hardware which maybe was hard to accomplish when Kirk's papers were written, but might be easier today with FPGA technology [Xil01]. Data coherency must also be maintained since data structures can be duplicated in more than one partition. It can for instance be solved in the same manner as in a system with multiple bus masters — with a cache coherency protocol and a (cache internal) snooper.

3.2.2 Partitioning by software

Andrew Wolfe

Inspired by Kirk's proposition, Wolfe suggests in [Wol93] to partition instructions to memory addresses so they map into the cache to reduce extrinsic cache interference. The major advantage of this approach is that no extra hardware is needed which is a less expensive solution but also a faster since more hardware will end with slower hardware due to longer signal paths. Wolfe shows with some trivial examples how and where tasks should be located to cause as less interference as possible.
Wolfe suggests a method to segmentize the memory by altering the decoding of memory addresses. Instead of decoding the address to (in order) tag, set, and offset, the address field can be interpreted as set, tag, and offset. In the traditional decoding of a cache address all parts of the memory will map the cache contiguously. The altered version will map the contiguously addresses to a segment of the cache (vice versa so to say). Spatial locality will in this case not been exploited so a hybrid version where a part of the set-bits also are used the traditional way will solve the problem. See figure 3.8. (The method can also be implemented in hardware.) By this approach each task can be assigned a segment of the memory and will not interfere with other tasks in other segments.

Frank Mueller

In [Mue95] Frank Mueller automates Wolfe's ideas by letting the compiler and linker assign tasks to addresses and solve the puzzle by assembling the code in the right order. Instruction partitioning is solved by non-linear control-flow transformations and the data partitioning use transformations of data references. Large tasks may not fit entirely into it's own partition (and not a mapping segment of the memory since they are of equal size), so if the tasks code must map to it's own part of the cache, the code must be at such a place in the memory so it will map to the correct part of the cache. Splitting code and connect them with unconditional jumps and global data is supported by the method. See figure 3.9 where a task that can't suffer from CRPD has its own partition of the cache and all other tasks share the rest of the cache. The privileged task will nevertheless still have intrinsic cache misses. A task's assignment to a certain percentage of the cache will block the same amount of the main memory; if task is assigned to a partition of a size of 40% of the complete cache, there will be only 60% of the main memory left to other tasks even if the task only uses 2% of main memory — the rest will be wasted. (This waste can however be avoided by letting the operating system control the memory management [LHH97].)
Tasks at the same priority level have however a special relationship. They will never preempt each other and will therefore be scheduled in a FIFO-manner. The consequence of having several tasks

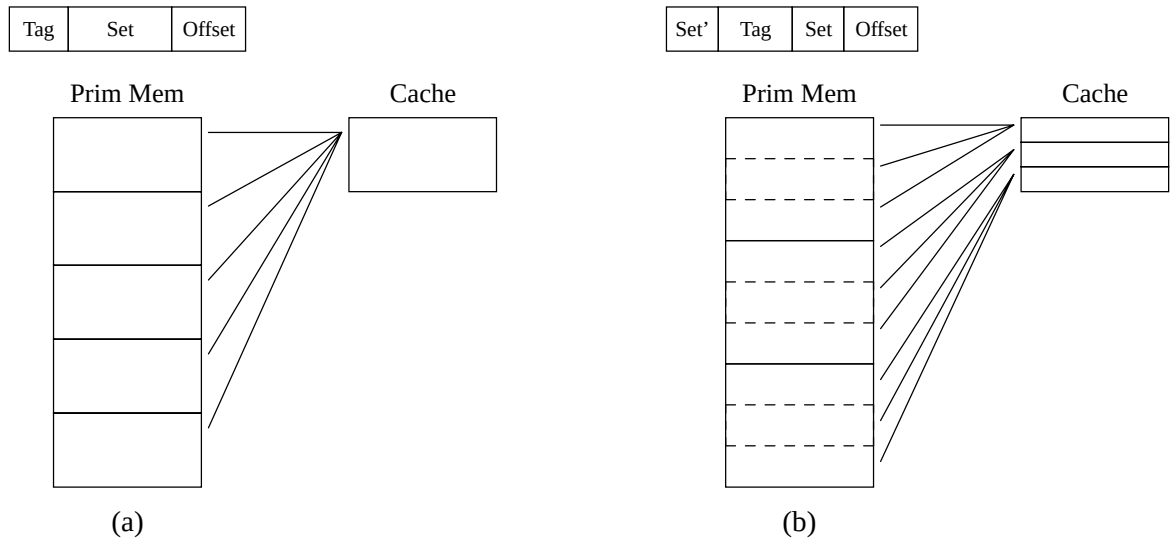

Figure 3.8: In (a) cache addresses are traditionally address spaced decoded. In (b) a hybrid version of address spaced and contiguous segments is presented.

of equal priority sharing the same partition is an increase of "relative internal" interference (lower hit-ratio), which is a consequence of having more code in a (small) cache.
Each tasks code and dataarea is compiled separatly with cache partitioning information as in input. All files are then piped further to the linker that allocates the code into the correct partitions.

Code partitioning is carried out by splitting the code into portions of size(s) that the cache partition information file provides. Each portion is terminated by an unconditional jump to next portion. A cache partition will be of equal size as the portions. Portions may be padded by NOP-code to fill up the partition if necessary.
Conditional and unconditional jumps between code portions (so called *remote transfers*) must be justified, but internal jumps can be left as is. Indirect jumps and conditional branches may have difficulties to be performed remotely since the number of bits to set the offset might be too few. In those cases jump tables with absolute addresses or a local short jump to an unconditional jump might solve the problem. Procedure calls and returns from them might have to be located in the same portion since most architectures use indirect implementations. Traps to the operating system should normally be unaffected.
The code transformation requires, as mentioned, in some cases additional code which leads to increasing code space.

Data partitioning is more trickier than code since data can be located as *global*,*local (stack)* and *dynamical (heap)*. Large global data structures must be split and indexing of arrays justified. The next code example will illustrate how such splitting and code transformation might look like. Observe that the example is in high-level-source for better understanding, but in reality this should be implemented in the back-end of the compiler.

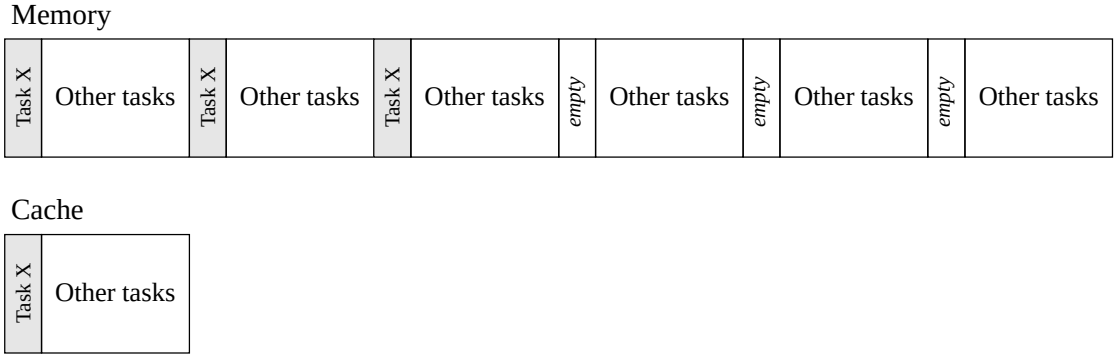

Figure 3.9: Task X has it own part of the cache. Observe that even if the task's size is no more than three segments, the space in the empty spaces must remain empty since a use of those addresses would interfere with the privileged task.

```
/* Original code */
int i, sum, a[1000];
...
for(i=0;i<1000;i++)
  sum = sum+a[i];
```

```
/* Indexing function */
...
for(i=0;i<1000;i++)
  sum = sum+a[f(i)];
...
int f(int i) {
  return(i/PS)*CS + i%PS;
}
```

```
/* Counter Manipulation */
int max_i,max=1000;
max_i=(max/PS)+CS+max%PS;
...
for(i=0;i<max_i;i++) {
  sum = sum+a[i];
  if(i%PS == 0)
    i = i+CS-PS;
}
```

The original code contains a large array that is accessed linearly, but since this part might be to large for a single portion (size PS) the accesses will be non-linear. If all portions are of the same size, the cache (of size CS) will contain $\frac{PS}{CS}$ partitions. One solution is to hide the non-linearity with a function that calculates the real positions of the data, but the disadvantage of this solution is that the calculation and function cost execution time. Another solution is to manipulate the indexing variables directly, but this might give side effects to other variables that might use them for other purposes.
Local data is stored into the stack, which leads to a split of the stack and stack pointer manipulation. Each task must exchange the regular stack pointer with an offset and partition to handle the pop and pushes. Dynamic allocation on the heap can be supported as long as the memory need is less than a cache partition size. If the allocation need extends the partition size, data can be scattered in multiple partitions just like global data.

3.2.3 Analysis methods

Basumallick and Nilsen

In [BN94] an approach to include CRPD in Rate Monotonic Analysis by adding γ as a term in to the scheduling. Inheriting the CRPD of tasks with higher priorities will accumulate this CRPD-term. Instead of assuming that the complete cache must be refilled at preemption, the method estimates how large the fraction of the cache that must be refilled by preemption. The concept is best explained with an example with three tasks;

Task	priority	cache use
A	high	1/8
B	middle	1/4
C	low	1/1

Assume that A intersects C with 1/16 of the cache and 1/16 with both B&C. If C is preempted by B that is preempted by A follows the question how much of the cache has to be refilled when C resumes its execution?
A: 1/4 of the cache will be altered by B and 1/8 by A. Since A and B intersect in the cache, C will not have to suffer the complete A-use but only half of its use (1/16) since the rest is "shadowed" by B. The cache refill will be $1/16 + 1/4 = 5/16 \approx 31\%$ of the complete cache.

Discussion The approach is very pessimistic since the approach doesn't take into account that pieces of the tasks could be shared and not altered.

José V. Busquets-Mataix and Juan J. Serrano-Martín *et al*

These Spanish researchers have studied cache related costs during context-switches in pre-emptive real-time systems. Their approach is to use Basumallick and Kelvin's equation 2.2, but with the approximation that the only cost during a context switch is the refill penalty. Thus will formula be simplified to

$$WCET'_C = WCET_C + \gamma \tag{3.3}$$

which means that Liu and Layland's formula (see equation 2.1) can be used without any change or mix-up with the symbols. In [BMSM95] they have compared a system with Rate Monotonic scheduling with independent tasks in a partitioned instruction cache (PART) and a rate monotonic scheduling that included cache related pre-emption delay (CRMA). The pre-emptive scheduling algorithm has fixed-priority, which means that tasks don't change priorities due to for instance scheduling efficiency or breaking deadlock situations. With a synthetic benchmark as a load they could for instance control number of tasks, frequency, and utilization load and came to the following (summarized) conclusions regarding γ;

- Cache size is important for both CRMA and PART. Since the complete cache is considered as empty after a context switch, the refill time is proportional to γ
- The MIPS factor is important for PART, while it can be overlooked for CRMA. The higher MIPS rate the processor can achieve, the higher instruction fetch rate and higher intrinsic interference comparatively to extrinsic one.
- A high number of tasks is more affecting factor for PART than for CRMA.
- Cache partitioning performs better as cache size (and the partitions) increases.
- Workload factors influence more than hardware factors.

In [BMWSM$^+$96b, BMWSM96a] the comparison is extended to also include a cached version of Response Time Analysis (CRTA). RTA is a more precise method than RMA since it deals with more information, which theoretically leads to a higher utilization capacity. The research showed the not too

surprising conclusion that the CRTA also leads to a higher utilization than RTA, RMA, and CRMA. In some cases a partitioned cache can still lead to a better utilization than the cached versions of the scheduling algorithms.

In [BMWSM97] a hybrid solution of CRTA and partitioning of instruction caches is proposed to take advantage of the best of both worlds. The partitioning can by any of a hardware or software solution. In contrast to Kirk's SMART-solution in hardware (see section 3.2.1), all tasks are cached in the hybrid partitioning. On the other hand the cache partitioned aren't private but shared by several tasks. To avoid inherited CRPD, the best solution is to provide only one shared partition. Private partitions should be assigned to high priority task and tasks with high frequency to reduce the high number of indirect cache interference.
Simulated evaluations show that the utilization of the hybrid solution always is equal to or better than CRTA, regardless of cache sizes, number of tasks or the frequency of them. The PART-scheme has (as also shown in former work) better utilization at workloads with many tasks and where a large cache is available.

Chang-Gun Lee *et al*

Busquets-Mataix *et al*, Basumallick and Nilsen, and many others approach to bound CRPD is to assume that each cache block that has been replaced by the preempting task has to be copied back when the control is given back to the preempted task. This assumption must be considered as pessimistic since it is very possible that the replaced memory block will not be longer needed after the preemption. These Korean researchers approach to bound a tighter WCET_C on CRPD is to identify and only use *useful cache blocks* into the calculation. Their approach is performed in two steps;

1. **Per-task analysis**: Each task is statically analyzed to determine the preemption cost at each execution point by estimating the number of *useful* blocks in the cache. In the example illustrated in figure 3.10 the cache during time t_0 contains memory blocks number $\{0, 5, 6, 3\}$ reside. The

time	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8
access event	-	4	5	6	7	0	0	1	2
action	-	m	h	h	m	m	h	m	m
cache block 0	0	**4**	4	4	4	**0**	**0**	0	0
1	5	5	**5**	5	5	5	5	**1**	1
2	6	6	6	**6**	6	6	6	6	**2**
3	3	3	3	3	**7**	7	7	7	7
useful blocks	{5,6}	{5,6}	{6}	{}	{}	{0}	{}	{}	{}

Figure 3.10: A memory access scenario from which usefull cache blocks has been derived.

technique to determine the amount of useful cache blocks is based on data flow analysis over the control flow graph (CFG) of a program. One array stores the blocks that are reachable (*reaching memory blocks* – RMB) and another stores *live memory blocks* (LMB) at each execution point. From this material the amount of useful blocks at each execution point can be determined with an iterative method. The worst-case preemption scenario turns up when a task is preempted at the point with the largest amount of useful cache blocks since the cost will be largest when the task resumes its execution. The second worst scenario is a point with the second largest amount

of useful blocks etc. and all those scenarios are stored in a *preemption cost table*. Each task has a corresponding vector ($f_{i,j}$) where the j:th entry reflects the point with the j-th largest preemption cost.

2. **Preemption delay analysis**: A linear programming technique is used to compute γ[9] by summing up the product of number of preemption of a task during a time period with the number of useful cache blocks at an execution point (taken from the preemption cost table). More formally

$$PC_i(R_i^k) = \sum_{j=1}^{i} \sum_{l=1}^{\infty} g_{j,l} f_{j,l} \tag{3.4}$$

where $g_{j,l}$ is the number of invocations of task j that is preempted at least l times during a given response time R_i^k. It is however not possible to determine exactly which $g_{j,l}$ combination that gives the worst case CRPD to a task.

By defining constraints that among other information yield a maximum amount of preemptions by a single invocation of a task, an integer linear programming (ILP) problem will be formulated and a safe $g_{j,l}$ can be derived.

The model has been verified by experimental results on a 20MHz R3000 RISC CPU with an instruction cache of 16kB (the data cache was turned off). The workload was four different programs that were intentionally located in the main memory to compete for the same area in the cache. The proposed technique gave a 60% tighter CRPD estimation than the best of previous approaches.
To handle data caches, memory references must be categorized as *static* or *dynamic* load/store instructions. All static block accesses can be handled in the same way as instructions but since the method states that each instruction is in a specific block, the dynamic data areas will be conservatively considered as misses.

In [LHM+97] the method is simplified in "step 1" to only use the largest number of useful blocks (f_j) for the task j into the analysis instead of grading the scenarios (since this approach had very small impact on the result). Hence equation 3.4 will be simplified to

$$PC_i(R_i) = \sum_{j=2}^{i} g_j f_j \tag{3.5}$$

More important; the analysis has been extended to not assume that all useful cache blocks are replaced at preemption, but to only select useful blocks in task that *intersect with cache blocks of the preempting tasks*. This novel approach will bound the worst case CRPD tighter since it will only select useful blocks that may have been replaced by the preempting task. This means for instance that if the complete program will fit into the cache and the tasks don't map to the same locations in the cache, the CRPD will be computed (correctly) as zero. The method will however still not be exact in all other cases since there are several (safe) approximations in the analysis.
The extension can very briefly be described as a new table called *augmented preemption cost table* is constructed and added, and used as an input to the previous constrains that has been extended.
The experimental results has been performed at the same machine as the previous work and the workload has been four tasks that has been mapped to the cache in three different ways; completely, partly, and non-intersecting. The latter will of course yield no CRPD and will stand as a reference to calculate the CRPD at the other task-sets. The four tasks were all of different sizes and running at different period. The load of the system the CRPD generates depends of how large the refill penalty is.

[9] by Lee *et al* referred as $PC_i(R_i^k)$

refill penalty	worst case CRPD load
10 cc	1%
25 cc	2%
50 cc	5%
100 cc	9%
250 cc	18%

This work branched further by Sheayun Lee presented in section 3.2.4.

3.2.4 Cache-sensitive scheduling algorithms

Sheayun Lee *et al*

Since the method proposed by Chang-Gun Lee *et al* (presented in section 3.2.3) analyze all execution points to determine the number of useful cache blocks, it is at hand to find out execution points where the minimum number of useful blocks are reachable. Those points are the best opportunities to make a context switch in respect of CRPD. Such a method is presented and evaluated in [LLL+98] and is called *Limited Preemptible Scheduling* (LPS). Limiting preemption points increase the *blocking time* suffered by higher priority tasks. The risk is that if a task is blocked to long, it might potentially miss its deadline.
The *response time* of a task is the time that starts from where a task is permitted to start till it has actuated. Two main issues delays the task from execute code and finally actuate; *blocking delay* – tasks with higher priority executes and force the task to wait, and *pre-emption delay* – the cost to make a context switch by the operating system and CRPD.
In other words, the CRPD can be reduced to the price of a longer blocking delay on low priority task and possibly also a longer WCRT (Worst Case Response Time). High priority tasks will by this yield a lower CRPD and by this also a lower WCRT.

Experimental results show that LPS can increase schedulable utilization by more than 10% and save processor time by up to 44% as compared with a traditional fully preemptible scheduling scheme.

3.2.5 Abstract interpretation approach

In [KT98], Kästner and Thesing propose a static method to schedule a task set and incorporates CRPD. The method can handle *non preemptable* scheduling. The algorithm can detect if a task set is schedulable but not directly influence the scheduling decisions.

3.2.6 WCET measurement with analysis support

To test and measure *all* execution paths to determine WCET_A isn't a realistic approach, but if only local WCET-paths were chosen, the number of test would decrease dramatically.

Petters and Färber (*et al*)

This German research group has implemented a tightly coupled multi-processor system on a PCI-bus with MIPS4600 and dual Intel Pentium II/III CPU-high performance units (HPU). The system also

contain a Real-Time Unit[10] , Configurable I/O Processor (CIOP) with FPGA and dual ported RAM. The system goes under the name REAR.
In [PF99], Petters and Färber describe a measurement approach that starts with an automatically compiler generated Control Flow Graph (CFG). The CFG reflects the optimized code and will after an analysis that cleans infeasible paths and skips non WCET-paths yield a *reduced CFG*. The reduced CFG will be manually partitioned into measurement blocks to reduce complexity and hereby decrease the time of the measuring phase. The last step before the actual measurement is to insert probes (procedure calls `sttrace(id)`) into the measurement blocks that will provide the CIOP with time-stamped id-tags.
The CIOP use the dual ported RAM as a FIFO-buffer to push the data to an external host for postmonitoring. Each measurement starts with an initialization of the cache and pipeline and all paths in the reduced CFG of the measurement block are executed. The data cache flush might be a gain since dirty data is backwritten so a replacement will be achieved with less penalty so a compensating penalty is added to the execution time. Under normal conditions this flushing will however lead to an overestimation of WCET_C . All measurement block's highest WCET are summed up and a safety margin is added to cover DRAM refreshment. The method can be used on HPUs as well as the RTU.
The operating system will cause context-switches, which means that the switch time and the cache related effects also must be included to the tasks' WCET_C . All RTU-service calls have an WCET_C so also this can easily be measured and added. CRPD is calculated from the task size added with global data since this is assumed as lost at preemption. If the size exceeds the cache size, the maximum cache size is chosen.

Discussion The method in the paper isn't performed exactly in order as described. The testing is performed through manipulation of the object code to select execution paths and then a measurement is performed. The next step is to evaluate the result, make a new manipulation, possibly skip a code section, and start new measurements. This is performed *iteratively* and to reduce complexity and manual work, the measurement is performed from "good", "safe", low-complex points in the code at loops, function calls etc. However, the method needs lots of hands-on and knowledge of how the software is constructed to give safe and a fast WCET_C . It has also been criticized for being safe enough to be used in *hard* real-time systems. The methods attraction is that very complex systems can be analyzed with a very tight WCET_C bound result.

3.2.7 Task layout to predict performance (and minimize misses)

Task sets, which may not be schedulable when the layout of tasks in main memory is arbitrarily chosen, might become schedulable provided the layout minimizes the CRPD. In some sense, task layout can be considered as an advanced form of (software) partitioning. Another approach is to only permit some (prioritized) tasks to reside in the cache and deny other tasks to use the cache.
The concept can easily be shown by an example of non-partitioned code and a modified placement of code that is more "cache-friendly". In figure 3.11(a) a piece of code with two tasks is located in memory. Unfortunately the two tasks map partly to the same parts of the cache with extrinsic cache interference as an result. In (b) the pieces are carefully assigned addresses so no interference is possible.
The method can of course even be used to reduce intrinsic cache interference by for instance assigning a frequent function to special parts of the cache.

[10] A dedicated Intel Pentium II CPU with a local SRAM that runs the operating system

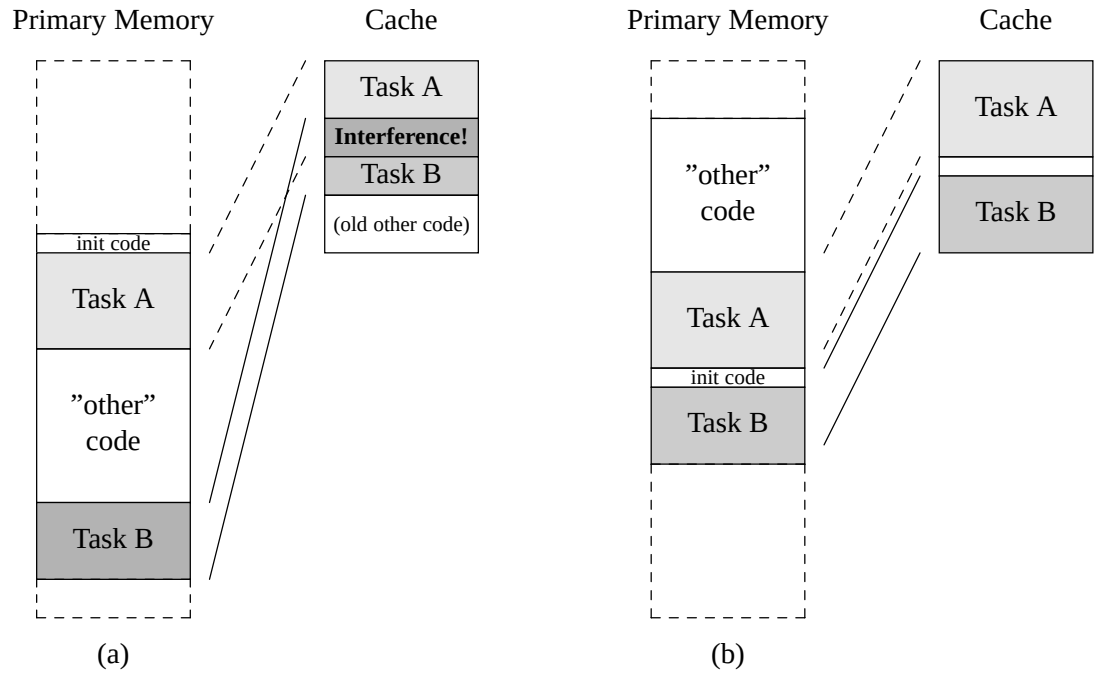

Figure 3.11: In (a) code parts intersect in the cache, but in (b) they won't due to carefully chosen address-assignment.

Datta *et al*

In [DCB+00] and [DCB+01] a proposal to a technique to find out an optimal layout which minimizes the WCRT of *one* task and satisfy the deadline to all other tasks. The method uses ILP formulations and the experimental results show that the method renders better performance than arbitrary chosen layouts – with a few exceptions. The method is only feasible on direct mapped instruction caches with task sets containing periodic tasks only.

Tomiyama and Dutt

The approach in [TD00] is similar to [TY96] described briefly in section 1.7.2. This paper includes a real-time perspective and determines the execution path that uses the maximum number of cache blocks. This is a better approach than assuming that the longest execution path automatically will yield the maximum CRPD. A simple example of that is illustrated in figure 3.12.
Experimental results show that this ILP-approach provides up to 69% tighter bounds on CRPD than a conservative assumption where a refill is of the size of the preempting task(s).

Lin and Liou

Instead of letting all tasks share the cache, Lin and Liou propose in [LL91] to disable the cache to all tasks but the most privileged or frequently used, and they must not be larger than they would all fit into

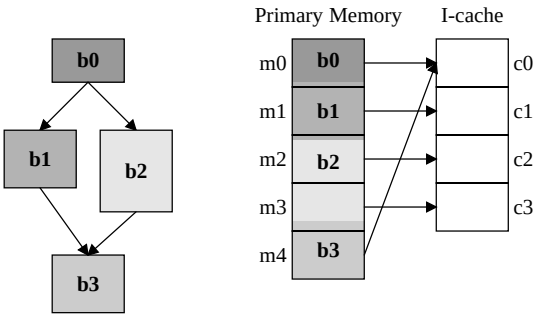

Figure 3.12: the execution path through b1 will use four cacheblocks but the longer execution path through b2 will only use three.

the cache. Extrinsic cache misses and this would also eliminate CRPD. The authors claim that a task that doesn't fit into the cache will yield the same WCET_C as a system without a cache. In [AMWH94] it is however shown that this approach is unnecessary defensive and that the worst case performance is better for a cached system than for an equal with a (for some or all tasks) disabled cache.

3.3 Special designed processors and architectures

Instead of trying to analyze or predicting every state a program could put the cache into, reducing the possibilities by a hardware or software framework will force programmers to choose another way and hereby make a safer implementation. This method will of course cost in for instance execution time, flexibility or forbid some general solutions, which might be a reasonable price to get a tight WCET_C bound. This section will show some suggested methods to such avoiding approaches.

3.3.1 MACS

The MACS (Multiple Active Context System) approach, suggested by Cogswell and Segall in [CS91] is to avoid caches by using memory banks. The execution of tasks is performed on a single shared pipeline and all tasks are running at "task level parallelism" with instructions. Instead of making context switch at a regular millisecond bases, the pipeline is fed with a new instruction alternating from each task. This means that if N tasks are running at the system, a new instruction will be issued from this task every N clock tick. To exploit the pipelining to the maximum there must be at least as many tasks as stages in the pipeline — otherwise bubbles with NOP-instructions must be inserted. Each task must have its own register file, set of condition flags and program counter in the processor that follows the tasks instruction in the pipeline. By this approach all pipeline hazards are avoided. Instead of fast cache memories, instructions are stored in memory banks that are used in an interleaved manner. The number of banks that must be used can be calculated by dividing the latency of a memory bank by the longest pipeline-stage's cycle time. The higher clock frequency or longer memory latency, the more memory banks will be needed. See figure 3.13 for an overview.
All problems seem to be solved; no pipeline hazards, no cache misses, and one instruction is executed each clock cycle. One catch is how to handle dynamic data or data structures that are larger than a single block.

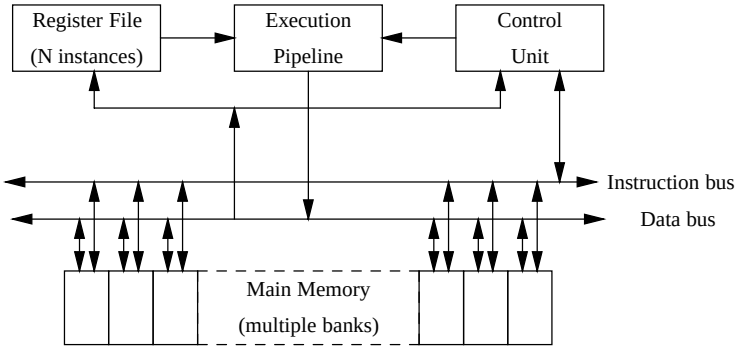

Figure 3.13: Overview of MACS

3.3.2 A hardware real-time co-processor — the Real-Time Unit

Johan Stärner suggested in [Stä98] to prefetch the complete thread or task into a local memory (or cache) before the actual context switch is performed. By this approach not only the extrinsic cache interference can be avoided, but also the complete cache can be used by a single task with a higher performance than a scattered partitioned (small) cache as a result. This can be accomplished by a special co-processor (Real Time Unit – RTU) that among other services also schedules tasks in a system by controlling the CPU and its' registers[SAFL96].
The RTU sends an interrupt request when it wants to switch tasks to the CPU. The CPU starts a subroutine that writes all current user register values to the RTU and then reads the corresponding tasks register values and sets the program counter to the new task. This performs the context-switch. See figure 3.14 for an overview of the proposal.

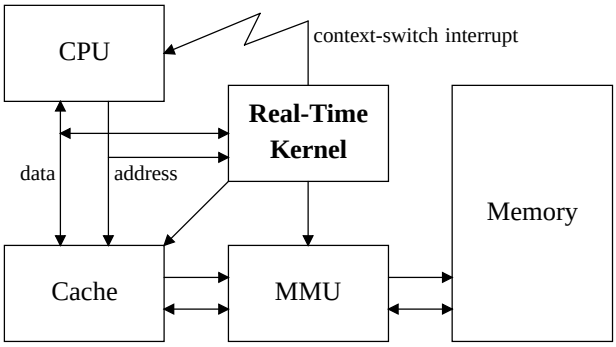

Figure 3.14: Overview of a system where the RTU controls the prefetching before a context-switch

The method is utilizable straigth-forward on instructions, but to prefetch data some kind of help from the operating system is needed.
The disadvantage of the approach is that a special co-processor is needed and a cache memory that can be controlled by an external device. DMA, floating point units and MMU were on the other hand also co-processors but are today accepted as regular (and even necessary) components in a computer

system. The RTU has been implemented in an FPGA but also as an ASIC.

3.4 Summary

A table of all described methods that handle cache issues in real-time systems summarizes this chapter. Even if it is not possible to see what kind of approach that is used or the mentioned limitations, it can although give a rough picture of what areas that are more solved than others.

	year	Extrinsic	Intrinsic	Instructions	Direct mapped	Set-associative	Data	Direct mapped	Set-associative	Automated	L2	Pipelining	Superscalare
Alt, Ferdinand *et al*	95–99		√	√	√	√	√	√	√	√			
Altenbernd & Stappert	1997		√	√	√	√	√	√	√	√		√	√
Basumallick & Nilsen	1994	√	√										
Busquet-Mataix *et al*	94–97	√		√	(√)	(√)							
"Florida research"	94–99		√	√	√	√	√	√		√		√	
Cogswell & Segall	1991	(√)	(√)	(√)			(√)					√	
Datta *et al*	2000	√		√	√								
Kirk	88–91	√					√			(√)			
Kim *et al*	1996		√				√						
Kästner & Thesing	1998	√		√	√	√				√			
Lee C-G *et al*	96–98	√		√	√	(√)	(√)						
Lee S *et al*	1998	√											
Li, Malik & Wolfe	95–99		√	√	√	√							
Lim S-S *et al*	94–95		√	√	√	(√)	(√)						
Lin & Liou	1991	√	√	√									
Liu & Lee	1994		√	√	√								
Lundqvist & Stenström	1999		√	√	√		√			√		√	√
Mueller	1995	√		√			√			√			
Muller*et al*	1998	√					√			(√)			
Nilsen & Rygg	1995	√	√							√		√	
Ottosson & Sjödin	1997		√	√	√	√	√	√	√			√	
Petters & Färber	1999	√	√	√	√	√	√	√	√	(√)	√	√	√
Rawat	1993		√				√	√	√				
Stärner	1998	√		√						(√)			
Theiling & Ferdinand	98–00		√	√	√	√	√	√	√	√			
Tomiyama & Dutt	2000	√		√	√								
Wolfe	1993	√		√									

Chapter 4

Conclusions

This chapter summarizes and discusses the state of the art in the area of real-time systems and with focus on cache memories.

4.1 Summary

4.1.1 Modern cache memories . . .

Cache memories today are not just fast small simple memories, but complex components with features like prefetching, write buffers, data pipelining, victim caching, fast replacement algorithms etc. to hide main memory *latency*. This report explains how cache memories work from the very beginning; show different methods how to enhance caches with features, and how they finally are designed in some modern microprocessors.

4.1.2 . . . in real-time systems

Computer processors of today are mainly constructed for high performance in the average case — not the worst case. A hard real-time system must never exceed its deadline and that is why modern processors seldom are found in those processes, and if they are installed, most of the features like caching and pipelining is disabled.
This report present several methods to include cache memories into worst-case performance analysis in real-time systems. No method is today able to calculate a safe and tight Worst-Case Execution Time (WCET_C [1]) for any arbitrary program that runs on a modern high-performance system.

4.1.3 Extrinsic behavior

Extrinsic cache behavior is less studied than intrinsic since many researchers find the problem solved by *partitioning* the cache and assign tasks to dedicated parts of the cache. The problem might be avoided, but to the price of lower hit ratios and decreased performance since the cache will be smaller for the task. A few major proposals have been made to incorporate caches into the scheduling but only instruction caching has been successful. The impact of cache related preemption delay (CRPD) is getting more notified since the cache refill time increases relatively to CPU speed, and the CRPD takes a proportionally large percentage in the WCRT of a task.

[1] In preemptive multi-tasking systems it is more common to discuss Worst-Case Response Time (WCET_C) but the difficulties are the same as WCET_C .

4.1.4 Intrinsic behavior

Instruction caching

Several methods with different approaches model instruction caching quite well — especially when the cache is direct mapped and the software is single threaded without interrupts or preemptions. Theoretically all execution paths can be searched yielding an exact WCET_C . Searching through all paths is although not practically feasible even for a small program but with design limitations and model approximations a tight WCET_C can be calculated. Statically non-preemptive scheduling can be analyzed in the same manner since the execution of the execution order is static.
Very few methods are fully automated and needs manual annotations in the code or user interaction during the analysis to bound WCET_C . Some of the methods cannot handle some of these programming constructs: recursion, switches, and gotos.

Data caching

It seems clear that instruction caching is much easier to analyze than data caching and there are several reasons for that;

- spatial locality is inheritably higher for instructions than for data. Instructions come in sequences while data only can take advantage of this on vectors and strings.
- data might be altered by other devices such as other CPU:s and DMA. Coherency must be maintained.
- Memory mapped I/O must never be cached and the analysis must be aware of which addresses that might be cached or not
- Writing data can be done in several ways; emphwrite-through and emphwrite-back can be performed *write allocate* or *no write allocate*
- Data can be stored either in either global, local or dynamic space. All three methods use different techniques and thus must be modeled differently.
- Addresses to variables or data structures can change even if the data is the same.
- Since data might be allocated on an arbitrary space on the heap when dynamic memory is used, thrashing, mapping, updating and replacement is difficult to predict.
- Consistency must be maintained on multi-processor systems.

4.2 Open areas and future work

Even if research has been ongoing for twenty years to solve caching in real-time systems, very much work is still left to be examined. The following list is just some suggestions of open areas with real-time and cache aspects (no particular order).

- Caching dynamic memory in real-time systems.
- CRPD on multi-level caches.

- CRPD on data caches.
- Flexible (run-time) hardware partitioning.
- Cache write buffers and real-time systems.
- Low-power consuming cache memories.
- Compiler supported code/data optimization by prefetching to cache.
- . . . and more.

Solving any of those problems would be worth a PhD-title. . . . The author of this paper has although aimed his future work on how to measure, control and reduce the cache related preemption delay on modern processors.

Bibliography

[ABR01] Andrea Acquaviva, Luca Benini, and Bruno Riccó. Energy characterization of embedded real-time operating systems. In *Proceedings of Parallel Architectures and Compilation Techniques*, Barcelona, Spain, September 2001.

[AHH89] Anant Agarwal, Mark Horowitz, and John Hennessy. An analytical cache model. *ACM Theory of Computing Systems*, 7(2):184–215, May 1989.

[AM95] Martin Alt and Florian Martin. Generation of efficient interprocedural analyzers with PAG. In *Static Analysis Symposium*, pages 33–50, 1995.

[AMWH94] Robert D. Arnold, Frank Mueller, David B. Whalley, and Marion G. Harmon. Bounding worst-case instruction cache performance. In *Proceedings of the IEEE Real-Time Systems Symposium 1994*, pages 172–181, December 1994.

[AP93] Anant Agarwal and S. Pudar. Column-associative caches: a technique for reducing the miss rate for direct-mapped caches. In *Proceedings of the 20th International Symposium on Computer Architecture*, San Diego, CA, May 1993.

[BE98] Mark Brehob and Richard Enbody. An analytical model of locality and caching. Technical report, Dept. of CSaE – Michigan State University, 1998.

[BEW98] Mark Brehob, Richard Enbody, and Nick Wade. Analysis and replacement for skew-associative caches. Technical report, Dept. of CSaE – Michigan State University, 1998.

[BMSM95] José V. Busquets-Mataix and Juan J. Serrano-Martín. The impact of extrinsic cache performance on predictability of real-time systems. In *Proceedings of Workshop on Real-Time Computing Systems and Applications*, October 1995.

[BMWSM96a] José V. Busquets-Mataix, Andy Wellings, and Juan J. Serrano-Martín. Adding instruction cache effect to an exact schedulability analysis of preemptive real time systems. In *Proceedings of 9th Euromicro Workshop on Real-Time Systems*, Tokyo, Japan, June 1996.

[BMWSM$^+$96b] José V. Busquets-Mataix, Andy Wellings, Juan J. Serrano-Martín, Rafael Ors-Carot, and Pedro Gil. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *IEEE Real-Time Technology and Applications Symposium (RTAS '96)*, pages 204–213, Washington - Brussels - Tokyo, June 1996. IEEE Computer Society Press.

[BMWSM97] José V. Busquets-Mataix, Andy Wellings, and Juan J. Serrano-Martín. Hybrid instruction cache partitioning for preemptive real-time systems. In *Proceedings of EuroMicro Workshop on Real-Time Systems*, pages 56–63, June 1997.

[BN94] Swagato Basumallick and Kelvin D. Nilsen. Cache Issues in Real-Time Systems. In *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, June 1994.

[BS97] Francois Bodin and Andre Seznec. Skewed associativity improves program performance and enhances predictability. *IEEE transactions on Computers*, 46(5):530–544, May 1997.

[CC77] P. Cousout and R. Cousout. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximations of fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, January 1977.

[CH90] Fred C. Chow and John L. Hennessy. The priority-based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems*, 12(4):501–536, October 1990.

[CS91] Bryce Cogswell and Zary Segall. Macs – a predictable architecture for real time systems, 1991.

[DCB+00] Anupam Datta, Sidharth Choudhury, Anupam Basu, Hiroyuki Tomiyama, and Nikil D. Dutt. Task layout generation to minimize cache miss penalty for preemptive real time tasks: An ilp approach. In *Proceedings of 9th Workshop on Synthesis and System Integration of Mixed Technologies (SASIMI 2000)*, pages 202–208, April 2000.

[DCB+01] Anupam Datta, Sidharth Choudhury, Anupam Basu, Hiroyuki Tomiyama, and Nikil D. Dutt. Satisfying timing constraints of preemptive real-time tasks through task layout technique. In *Proceedings of VLSI Design*, pages 97–102, 2001.

[Ekl94] Sven Eklund. *Avancerad Datorarkitektur*. Studentlitteratur, Sweden, 1994.

[Ekl99] Sven Eklund. *Modern Mikroprocessorarkitektur*. Studentlitteratur, Sweden, 1999.

[FMWA99] Christian Ferdinand, Florian Martin, Reinhard Wilhelm, and Martin Alt. Cache behavior prediction by abstract interpretation. *Science of Computer Programming*, 35(2–3):163–189, November 1999.

[Gus00] Jan Gustafsson. *Analyzing Execution-Time of Object-Oriented Programs Using Abstract Interpretation*. Doctorial thesis, Uppsala University and Mälardalen University, Västerås, Sweden, May 2000.

[HAM+99] Christopher A. Healy, Robert D. Arnold, Frank Mueller, David B. Whalley, and Marion G. Harmon. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, 48(1):53–70, January 1999.

[Hei94] Joe Heinrich. *MIPS R4000 Microprocessor UserŠs Manual Second Edition*, 1994.

[HL95] Tai-Yi Huang and Jane W.-S. Liu. Predicting the worst-case execution time of the concurrent execution of instructions and cycle-stealing DMA I/O operations. *ACM SIGPLAN Notices*, 30(11):1–6, 1995.

[HLH96] Tai-Yi Huang, Jane W.-S. Liu, and David Hull. A method for bounding the effect of DMA I/O interference on program execution time. In *Proceedings of the RTSS'96*, pages 275–287, Washington D.C., USA, December 1996.

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, second edition, 1996.

[HWH95] Christopher A. Healy, David B. Whalley, and Marion G. Harmon. Integrating the timing analysis of pipelining and instruction caching. In *Proceedings of IEEE Real-Time Systems Symposium 1995*, pages 288–297, December 1995.

[Int99] Intel Corporation. *Intel Architecture Software Developer's Manual*, 1999.

[Int01] Intel Corporation. *Intel Pentium 4 Processor Optimization – Reference Manual*, 2001.

[Jou90] Norman P. Jouppi. Improving direct-mapped cache performance by addition of a small fully-associative cache and prefetch buffers. In *The 17th ISCA*, pages 364–373, May 1990.

[Kir88] David B. Kirk. Process dependent static cache partitioning for real-time systems. In *Proceedings of the 1988 IEEE Real-Time Systems Symposium*, pages 181–190. IEEE Computer Society Press, 1988.

[Kir89] David B. Kirk. SMART (Strategic Memory Allocation for Real-Time) Cache Design. In IEEE Computer Society Press, editor, *Proceedings of the Real-Time Systems Symposium - 1989*, pages 229–239, Santa Monica, California, USA, December 1989. IEEE Computer Society Press.

[KJLH89] R. E. Kessler, R. Jooss, A. Lebeck, and M. D. Hill. Inexpensive implementations of set-associativity. In Michael Yoeli and Gabriel Silberman, editors, *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 131–139, Jerusalem, Israel, June 1989. IEEE Computer Society Press.

[KMH96] Sung-Kwan Kim, Sang Lyul Min, and Rhan Ha. Efficient worst case timing analysis of data caching. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, pages 230–240, Brookline, Massachusetts, USA, June 10–12, 1996.

[KS90] David B. Kirk and Jay K. Strosnider. SMART (strategic memory allocation for real-time) cache design using the MIPS R3000. In IEEE Computer Society Press, editor, *Proceedings of the Real-Time Systems Symposium - 1990*, pages 322–330, Lake Buena Vista, Florida, USA, December 1990. IEEE Computer Society Press.

[KSS91] David B. Kirk, Jay K. Strosnider, and John E. Sasinowski. Allocating SMART cache segments for schedulability. In *Proceedings from EUROMICRO '91 Workshop on Real Time Systems*. IEEE Computer Society Press, 1991.

[KT98] Daniel Kästner and Stephan Thesing. Cache sensitive pre-run scheduling. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, volume 1474, pages 131–145. Springer, 1998.

[LBJ+95] Sung-Soo Lim, Young Hyun Bae, Gyu Tae Jang, Byung-Do Rhee, Sang Lyul Min, Chang Yun Park, Heonshik Shin, Kunsoo Park, Soo-Mook Moon, and Chong Sang Kim. An accurate worst case timing analysis for RISC processors. *IEEE Transactions on Software Engineering*, 21(7):593–604, July 1995. Best Papers of the Real-Time Systems Symposium, 1994.

[LHH97] Jochen Liedtke, Hermann Hartig, and Michael Hohmuth. OS-controlled cache predictability for real-time systems. In *Proceedings of the Third IEEE Real-Time Technology and Applications Symposium (RTAS '97)*, pages 213–227, Washington - Brussels - Tokyo, June 1997. IEEE.

[LHM+97] Chang-Gun Lee, Joosun Hahn, Sang Lyul Min, Rhan Ha, Seongsoo Hong, Chang Yun Park, Minsuk Lee, and Chong Sang Kim. Enhanced analysis of cache-related preemption delay in fixed-priority preemptive scheduling. In *Proceedings of the 18th Real-Time System Symposium*, pages 187–198, San Francisco, USA, December 3–5, 1997. IEEE Computer Society Press.

[LL73] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[LL91] T.H. Lin and W.S Liou. Using cache to improve task scheduling in hard real-time systems. In *Proceedings of the IEEE Workshop on Architecture support for Real-Time System*, pages 81–85, December 1991.

[LL94] Jyh-Charn Liu and Hung-Ju Lee. Deterministic upperbounds of the worst-case execution times of cached programs. In *Proceedings of the 15:th IEEE Real-Time Systems Symposium*, pages 182–191, 1994.

[LLL+98] Sheayun Lee, Chang-Gun Lee, Minsuk Lee, Sang Lyul Min, and Chong Sang Kim. Limited preemptible scheduling to embrace cache memory in real-time systems. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 51–64, June 1998.

[LM95] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the 32nd ACM/IEEE Conference on Design Automation (DAC'95)*, page 6, San Francisco, CA, USA, June 12–16, 1995.

[LML+94] Sung-Soo Lim, Sang Lyul Min, Minsuk Lee, Chang Park, Heonshik Shin, and Chong Sang Kim. An accurate instruction cache analysis technique for real-time systems. In *Proceedings of the Workshop on Architectures for Real-time Applications*, April 1994.

[LMW95] Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *Proceedings of the 16th Real Time System Symposium*, pages 298–307, December 1995.

[LMW99] Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Performance estimation of embedded software with instruction cache modeling. *ACM Transactions on Design Automation of Electronic Systems*, 4(3):257–279, July 1999.

[LS99a] Thomas Lundqvist and Per Stenström. Emperical bounds on data caching in high-performance real-time systems. Technical Report 99-4, Department of Computer Engineering, Chalmers, Göteborg, Sweden, April 1999.

[LS99b] Thomas Lundqvist and Per Stenström. An integrated path and timing analysis method based on cycle-level symbolic execution. *Journal of Real-Time Systems*, pages 183–207, November 1999. Special Issue on Timing Validation.

[LS99c] Thomas Lundqvist and Per Stenström. A method to improve the estimated worst-case performance of data caching. In *Proceedings of the 6th RTCSA*, December 1999.

[MB91] Jeffrey C. Mogul and Anita Borg. The effect of context switches on cache performance. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 75–84, Santa Clara, CA, USA, April 1991.

[MMIP98] Henk Muller, David May, James Irwin, and Dan Page. Novel caches for predictable computing. Technical Report CSTR-98-011, Department of Computer Science, University of Bristol, October 1998.

[Mue95] Frank Mueller. Compiler support for software-based cache partitioning. In *ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems*, La Jolla, CA, USA, June 1995.

[MWH94] Frank Mueller, David B. Whalley, and Marion G. Harmon. Predicting instruction cache behavior. In *ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems*, June 1994.

[NR95] Kelvin D. Nilsen and Bernt Rygg. Worst-case execution time analysis on modern processors. *ACM SIGPLAN Notices*, 30(11):20–30, 1995.

[OS97] Greger Ottosson and Mikael Sjödin. Worst-case execution time analysis for modern hardware architectures. In *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Real-Time Systems (LCT-RTS'97)*, 1997.

[PF99] Stefan M. Petters and Georg Färber. Making worst case execution time analysis for hard real-time tasks on state of the art processors feasible. In *Proceedings of the 6th Real-Time Computing Systems and Applications RTCSA*, Hong-Kong, December 13–15, 1999. IEEE Computer Society.

[PHH89] Steven Przybylski, Mark Horowitz, and John Hennessy. Performance tradeoffs in cache design. *Proceedings of the 15th Annual Int. Symposium on Computer Architecture*, pages 290–298, June 1989.

[PS91] Chang Yun Park and Alan C. Shaw. Experiments with a program timing tool based on source-level timing schema. *IEEE Computer*, 24(5):48–57, May 1991.

[Raw93] Jai Rawat. Static analysis of cache performance for real-time programming. Master thesis TR93-19, Iowa State University of Science and Technology, November 17, 1993.

[RBS96] Eric Rotenberg, Steve Bennett, and James E. Smith. Trace cache: A low latency approach to high bandwidth instruction fetching. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 24–34, Paris, France, December 2–4, 1996. IEEE Computer Society TC-MICRO and ACM SIGMICRO.

[RBS99] Eric Rotenberg, Steve Bennett, and James E. Smith. A trace cache microarchitecture and evaluation. *IEEE Transactions on Computers*, 48(2):111–120, 1999.

[RS99] Jeffrey B. Rothman and Alan J. Smith. The pool of subsectors cache design. Technical Report CSD-99-1035, University of California, Berkeley, January 6, 1999.

[SA97] Friedhelm Stappert and Peter Altenbernd. Complete worst-case execution time analysis of straight-line hard real-time programs. Technical Report 27/97, C-Lab, Paderborn, Germany, December 9 1997.

[SAFL96] Johan Stärner, Joakim Adomat, Johan Furunäs, and Lennart Lindh. Real-time scheduling co-processor in hardware for single and multiprocessor systems. In *Proceedings of the 22:nd Euromicro conference*, 1996.

[SC97] Kevin Skadron and Douglas W. Clark. Design issues and tradeoffs for write buffers. In *Proceedings of the Third International Symposium on High Performance Computer Architecture (HPCA '97)*, pages 144–155, Los Alamitos, Ca., USA, February 1997. IEEE Computer Society Press.

[Sez93] Andre Seznec. A case for two-way skewed-associative caches. In Lubomir Bic, editor, *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 169–178, San Diego, CA, May 1993. IEEE Computer Society Press.

[Sez97] Andre Seznec. A new case for skewed-associativity. Technical Report RR-3208, Inria, Institut National de Recherche en Informatique et en Automatique, 1997.

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

[Stä98] Johan Stärner. Controlling cache behavior to improve predictability in real-time systems. In *Proceedings of 10th Euromicro Workshop on real-time systems*, June 1998.

[Sto93] Harald S. Stone. *High Performance Computer Architecture 3rd ed*. Addison-Wesley, 1993.

[TD00] Hiroyuki Tomiyama and Nikil Dutt. Program path analysis to bound cache-related preemption delay in preemptive real-time systems. In *Proceedings of 8th International Workshop on Hardware/Software Codesign (CODES 2000)*, pages 67–71, May 2000.

[TF98] Henrik Theiling and Christian Ferdinand. Combining Abstract Interpretation and ILP for Microarchitecture Modelling and Program Path Analysis. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 144–153, Madrid, Spain, December 1998.

[Tha00] Henrik Thane. *Monitoring, Testing and Debugging of Distributed Real-Time Systems*. Doctorial thesis, Royal Institute of Technology, KTH and Mälardalen University, Stockholm and Västerås, Sweden, May 2000.

[TY96] Hiroyuki Tomiyama and Hiroto Yasuura. Optimal code placement of embedded software for instruction caches, March 1996.

[Whi97] Randall T. White. *Bounding Worst-Case Data Cache Performance*. PhD thesis, Florida State University, April 1, 1997.

[WMH+97] Randall T. White, Frank Mueller, Christopher A. Healy, Frank Mueller, David B. Whalley, and Marion G. Harmon. Timing analysis for data caches and set-associative caches. In *Proceedings of the Third IEEE Real-Time Technology and Applications Symposium (RTAS '97)*, pages 192–202, Washington - Brussels - Tokyo, June 1997. IEEE.

[WMH+99] Randall T. White, Frank Mueller, Christopher A. Healy, David B. Whalley, and Marion G. Harmon. Timing analysis for data and wrap-around fill caches. *Real-Time Systems*, 17(2–3):209–233, 1999.

[Wol93] Andrew Wolfe. Software-based cache partitioning for real time applications. In *Proceedings of the 3rd International Workshop on Responsive Computer Systems*, September 1993.

[Xil01] Xilinx corporation. *Xilinx web site http://www.xilinx.xom*, 2001.

Index

Chance is irrelevant — we will succeed.

– Seven of nine