

CHEMNITZ UNIVERSITY
OF TECHNOLOGY

Faculty of Electrical Engineering &
Information Technology

Chair of Process Automation

Diploma Thesis

Instruction Timing Analysis
for Linux/x86-based
Embedded and Desktop Systems

Author: Tobias John

Supervisors: Prof. Peter Protzel

Dr. Robert Baumgartl

Date of Submission: 22th September 2005

John, Tobias

Instruction Timing Analysis for Linux/x86-based Embedded and Desktop Systems

Diploma Thesis, Faculty of Electrical Engineering & Information Technology

Chemnitz University of Technology, 2005

Declaration of Authorship

I hereby declare that the whole of this diploma thesis is my own work, except where explicitly stated otherwise in the text or in the bibliography.

This work is submitted to Chemnitz University of Technology as a requirement for being awarded a diploma in Electrical Engineering - Automation Engineering.

I declare that it has not been submitted in whole, or in part, for any other degree.

Chemnitz, 22th September 2005

Tobias John

Conceptual formulation¹

Objective:

Real-time aspects are increasingly relevant in standard PC environments. At the same time x86-based processors are used more often in embedded systems.

For the construction of real-time systems standard techniques are available, which unfortunately reduce the efficiency of the systems. Therefore specific system parameters often do without a guaranty, and overdimensioned hardware is used instead.

The intention of this work is to obtain quantitative statements about the chronological behaviour of current x86 based processor architectures under the Linux operating system.

Relevant aspects are as follows:

- * comparison of types Pentium 4 and AMD Elan SC 410
- * outline and abstract of micro-benchmarks which address certain units (ALU, FPU, MMX, SSE) specifically
- * assessment of typical operating systems services (often used system calls) on both architectures
- * assessment of given real-time applications (for example current multi media codes)

This work will be ministered by chair of Process Automation of the faculty of Electrical Engineering & Information Technology (Prof. Protzel) and by the juniorprofessorship Real-time Systems of the computing faculty (Dr. Baumgartl).

¹Appendix A holds a copy of the original (german) conceptual formulation.

Abstract

As the conceptual formulation expresses this work should compare x86 based general purpose processors with embedded ones intended to be used in real-time systems. The focus was directed especially the jitter of execution times.

However to analyse the execution of instructions on a microprocessor in a way that the completion is one time as fast as possible and the other time as slow as in the worst case, requires that the architecture is known in detail.

It is necessary to know which aspects, and in which order, influence the execution flow.

This information is often not published, not detailed enough or even wrong².

As a consequence the first step has to be the analysis of the underlying hardware.

That this step would take me most of the time could not be known in advance and therefore the results of this work are a collection of microbenchmarks to explore the caching and branch prediction architecture to obtain the missing information.

Nevertheless, the gained results are interpreted in a way that best and worst case in execution timing are stressed.

²E.g. [LH] and [Sea00] declare that the PII uses a strict LRU replacement strategy, although a pseudo technique is actually applied to L1!

Acknowledgements

Although this work has been created by myself, several friends have had great influence in it, that is why I want to mention those:

I would like to thank Jette & Tim for proofreading, my family for the patience they had with me, Chris for remembering me on the duties of a student and for always being open to my questions. Special thanks goes to Brigitte for wakening the last energy resources and to Carli, she showed me that there *is* something worth fighting for.

Contents

1	Introduction	1
2	Background knowledge	2
2.1	Hardware	2
2.2	Performance monitoring	2
2.3	Branch prediction	3
2.4	CPU caches	5
2.4.1	INTELs Cache Architecture	8
3	Analysis concepts	10
3.1	Performance Monitoring - IA-32 architecture	10
3.1.1	P6 family	10
3.1.2	PIV, Xeon	11
3.2	Branch prediction	13
3.2.1	Branch History (shift-)Register (BHR)	13
3.2.2	Branch Target Buffer (BTB)	15
3.3	Caching	20
3.3.1	Adjacent cache line prefetch - PIV	20
3.3.2	Caching strategy of the L1 cache	20
3.3.3	Replacement strategy	22
4	Worst case on caching	29
4.1	Cache flooding I	29
4.1.1	Double Purge Scenario	29
4.1.2	Cache flooder – as described in [LH]	30
4.2	Cache flooding II	33
4.2.1	The algorithm	33
4.2.2	Conditions on the cache architecture	36
4.3	Filling vs. Flooding	37
5	Results	39
5.1	Branch prediction	39
5.1.1	BHR - Branch History Register	39
5.1.2	BTB - Branch Target Buffer	39
5.2	Caching	44
5.2.1	Adjacent cache line prefetch - PIV	44
5.2.2	L1 - caching strategy	44
5.2.3	Replacement strategy	45
5.3	Cache flooding	50
6	Conclusions	53
A	Conceptual formulation	55

1 Introduction

Real-time aspects are becoming more important in standard desktop PC environments and x86 based processors are being utilized in embedded systems more often. While these processors were not created for use in hard real time systems, they are fast and inexpensive and can be used if it is possible to determine the worst case execution time.

Information on CPU caches (L1, L2) and branch prediction architecture is necessary to simulate best and worst cases in execution timing, but is often not detailed enough and sometimes not published at all. This document describes how the underlying hardware can be analysed to obtain this information.

This document is structured as follows:

The following section (sec. 2) gives background information to the covered topics: performance monitoring, branch prediction and caching. With this general knowledge it should be no problem to understand the exploration techniques presented in section 3. These pages cover the ideas behind the benchmarks and how they have to be implemented. Section 4 describes the worst case on caching. Two possibilities are explained on how to achieve it, whereas the first one is based on [LH] and the second has been developed by myself.

These theoretical concepts have been implemented and tested on Intel Pentium II, III and IV processor, belonging to the architectures P6 and Netburst. The results obtained are presented in section 5.

Finally summary and a list of open questions and remaining problems (sec. 6) follows.

2 Background knowledge

2.1 Hardware

The tests described in the following sections were executed on different Intel architectures: P6 (Pentium II, III) and Netburst (Pentium IV).

The PII is a "Klamath" at 233 MHz with a 66 MHz system bus frequency and a 512 KB second level cache running at half the core clock.

The PIII, from the same family of processors, has a CPU frequency of 500 MHz, a system bus rated at 100 MHz and an L2 cache of the same size as the PII that runs at half core speed, too. Its Codename is "Katmai". Both processors have the MMX instruction set but only the PIII utilizes SSE.

The representative of the Netburst microarchitecture, the PIV, is a "Northood" processor. It does *not* feature Hyper Threading, has a clock frequency of 2.66 GHz and a 512 KB big second level cache.

2.2 Performance monitoring

If the execution time of a program varies, this parameter is not usable for making comparisons or assertions anymore. Therefore counting the events corresponding to the analysed topic as cache or branch behaviour is the only way to draw exact conclusions.

Many processors therefore provide model specific registers (MSR) that serve this purpose, the so called performance monitoring registers. There exist several utilities to access these registers as **VTune** for the Intel processors, **PAPI**, **OProfile** and many others.

VTune is limited to Intel processors and to some special Linux distributions only.

PAPI, the Performance API, is an interface to the performance counter hardware for different platforms. It mainly consists of a library that needs to be linked to your software, wherefore it cannot be used in kernel modules.

OProfile is a profiler under GNU GPL, that supports even analysis of kernel modules. As the name suggests it is a profiler that is not called directly to read the counters. Instead it is called through an interrupt released by an overflow of a performance counter.

Under some circumstances it is unavoidable to work in kernel mode. For example if physical addresses are needed (→ cache flooder) or if you intent to analyse a RTAI module. That is why libraries as PAPI cannot be used. Profiling software is either too inexact or produces too much overhead (if the counter is setup to overflow at a minimal count) wherefore I decided to implement my own functions to access the performance monitoring hardware - for the time being, limited to the Pentium II/III, IV.

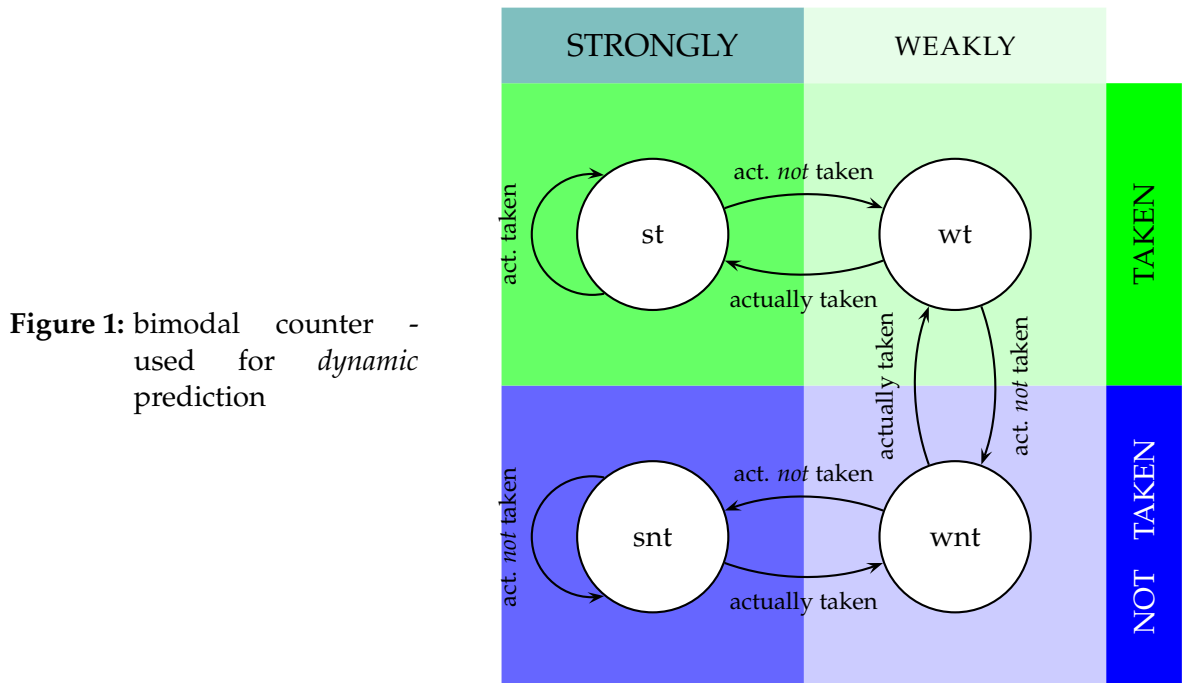
[Int04] describes how to configure the MSRs to count certain events. Some helpful functions as setting up registers and starting, stopping and resetting counters were written as inline assembly macros in a header file. So it is possible to count for example cache misses and branch (mis)predictions within a kernel module without installing additional software or patching the kernel (as it is necessary for PAPI). Direct manipulation of the MSRs is allowed in privilege level 0 only, so our performance monitoring macros are limited to kernel code, but because our intention was to analyse kernel modules (they are in conjunction with RTAI the simplest possibility to run code without interruption) this does not face a problem.

2.3 Branch prediction

In order to get more instructions completed faster, modern microprocessors are deeply pipelined. That means that instructions do not wait for the previous ones to complete before their execution begins. A problem with this approach arises, due to conditional branches. If a conditional branch is encountered and the result of the condition has not yet been calculated, the microprocessor does not know whether to take the branch or not. The applied solution is branch prediction - the processor decides whether to take the branch or not and starts executing the instructions at the predicted branch target. Finally when the result of the branch condition is known it is obvious whether the branch has been predicted correctly or not. In the latter case the already executed instructions of the wrong (mispredicted) path have to be thrown out (flushing the pipeline) which is particularly expensive with deeply pipelined processors. The delay for a mispredicted branch is usually equivalent to the pipeline depth.

There are two main types of branch prediction: static and dynamic one.

Static prediction assumes that the majority of backwards pointing branches occur in the context of repetitive loops, where the condition is used to determine whether the loop is to be repeated or not. Therefore backward branches are predicted to be "taken", whereas forward pointing branches are predicted "not to be taken".



The ability to dynamically predict the direction and the target of branches is based on the branch instruction's linear address, using the branch target buffer (BTB). If there is no valid entry in the BTB for the recent branch then static prediction will be used to decide which path to take.

A widely used scheme is the following:

There is a branch history register (BHR) with a width of N_H bits that stores the outcomes of the last N_H conditional branches. Either there is one global BHR, based on the correlations between subsequent branches in the *whole program flow* or several local ones that are based on the correlation between subsequent executions of the *same* branch. Some bits of the BHR together with the branch instruction's address index a table of n -bit saturating counters (usually $n = 2 \Rightarrow$ *strongly taken* (st), *weakly taken* (wt), *weakly not taken* (wnt), *strongly not taken* (snt)) that are updated when a jump condition is evaluated and predict the branch outcome.

Figure 1 shows such a bimodal counter and figure 2 is a scheme of the described architecture. [Sto01], [MMK04]

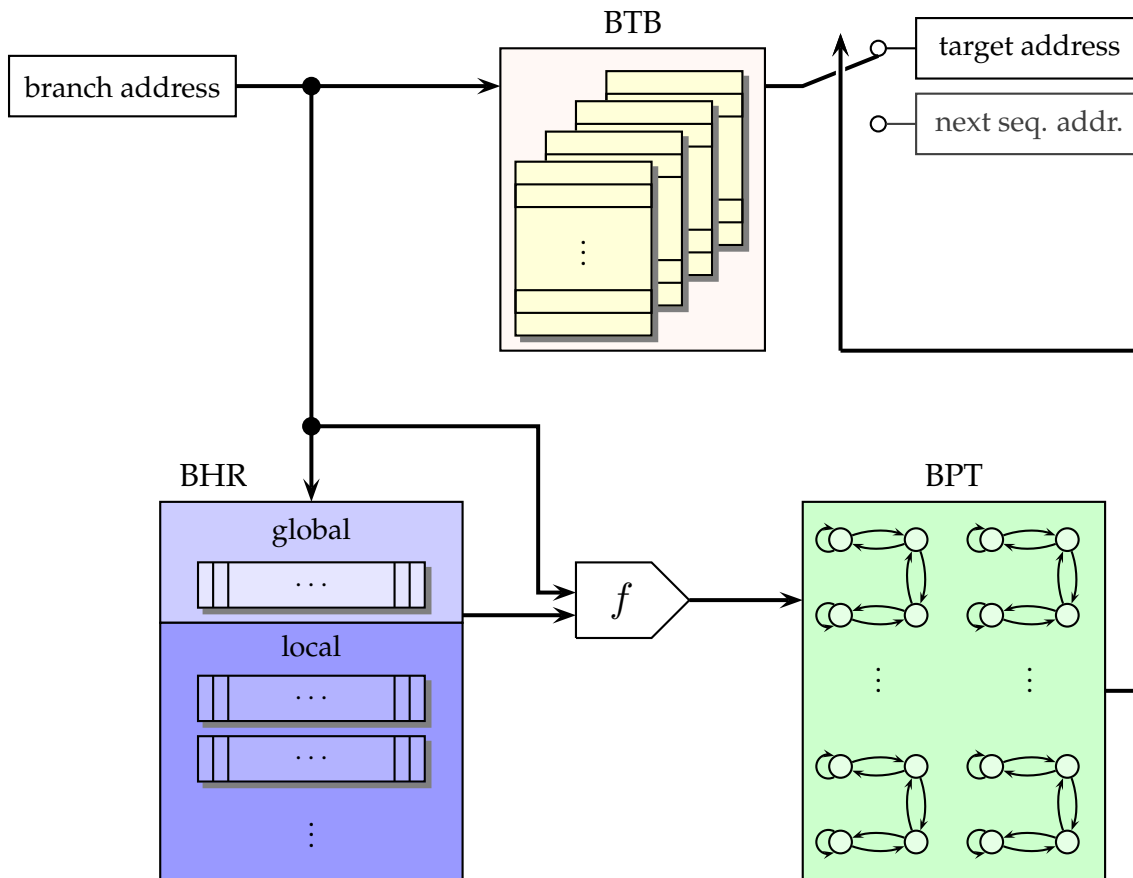


Figure 2: usual branch prediction architecture

Unfortunately, some processor manufacturers provide almost no information on the exact predictor implementation, although there are several advises on how to optimize your code (→ [And]).

2.4 CPU caches

Most common CPU cache architectures use several levels of set-associative caches to reduce the number of cycles the CPU has to wait for data from memory. A line is the smallest unit that can be transferred between a cache and main memory. If a line can be stored in any place in a cache it is called fully associative as opposed to direct mapped, where each line can only be stored in a specific place. Set associativity is the compromise where a line can be cached in W different locations. Those W lines form a set and because the address of the data within one line can be stored in any of that W places the address only indexes the set and not the line within the set. The appropriate line within a set is found through comparison.

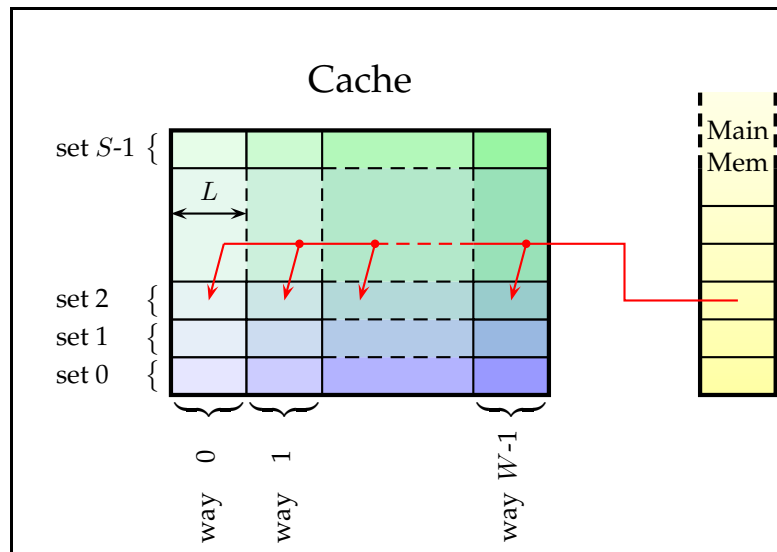


Figure 3: simplified structure of a cache

If data, that is accessed by the CPU, is found in a cache, it is called a cache *hit*, otherwise a *miss*. Several strategies exist to decide what to do with an accessed line. The most common are:

Table 1: common caching strategies

line in cache?	strategy	description
hit	write through	the cache is updated and the next level of memory too (either another (slower) cache or main mem.)
	write back	only the cache is updated (the line is marked "dirty" and written back later)
miss	write allocate	the next level of memory is updated and the line is fetched into the cache
	write no-allocate	the next level of memory is updated the line is <i>not</i> fetched

When data is going to be cached and no free space is available (That does not mean that the whole cache is filled! Because a line can be stored in only W places of *one* set it is of no help if there are other free sets available.) another line has to be purged out of the cache - which one, the replacement policy decides. Based on [Mil04] the most common replacement policies are Random, Round-Robin, LRU (Least Recently Used) and pLRU (pseudo LRU). Among the pLRU algorithms are pLRUt (pseudo LRU tree based) and pLRUm (pseudo LRU

based on MRU (Most Recently Used) bits).

Strict LRU is quite costly and complex because the whole history of the W cache lines in a set have to be saved and updated on an access. Therefore the pseudo LRU mechanisms try to reduce the number of bits needed to store the pLRU information and the time to manage them, through approximation of the “real” algorithm.

The tree based pLRU policy (pLRUt) uses a binary tree to point to the *assumed* least recently used cache line. When accessing a line and thereby making it the *most* recent one, all the tree bits that lay on the path to that line are updated to point to the opposing direction ($0 \leftrightarrow 1$). The left column of figure 4 shows an example to a 4-way cache that utilises the pLRUt strategy. To keep it simple, only one set of the cache is shown and the tree bits keep ‘r’/‘l’ for ‘right’/‘left’ instead of ‘0’/‘1’. The black arrows denote the path down the tree that points to the (pseudo) least recently used cache line. The contents of a cache line are marked through lowercase alphabetic characters and bold letters symbolize modified or updated entries. Each of the three pictures in a column of figure 4 shows the cache line and the pLRU bits *after* the given instruction (eg. read(b)) has been executed. The red boxed bits are those that have been updated.

With respect to the tree bits in step 0 of figure 4 the third cache line is the least recently used. When data b is read, which is already in the cache, all tree bits that are on the path down to that line have to be updated to point to the opposite direction. In step 2 a *new* line is read and another entry has to be freed to store k in the cache. Because the tree bits point to the second way (data c) this line is replaced and after updating the tree, data d is the LRU line.

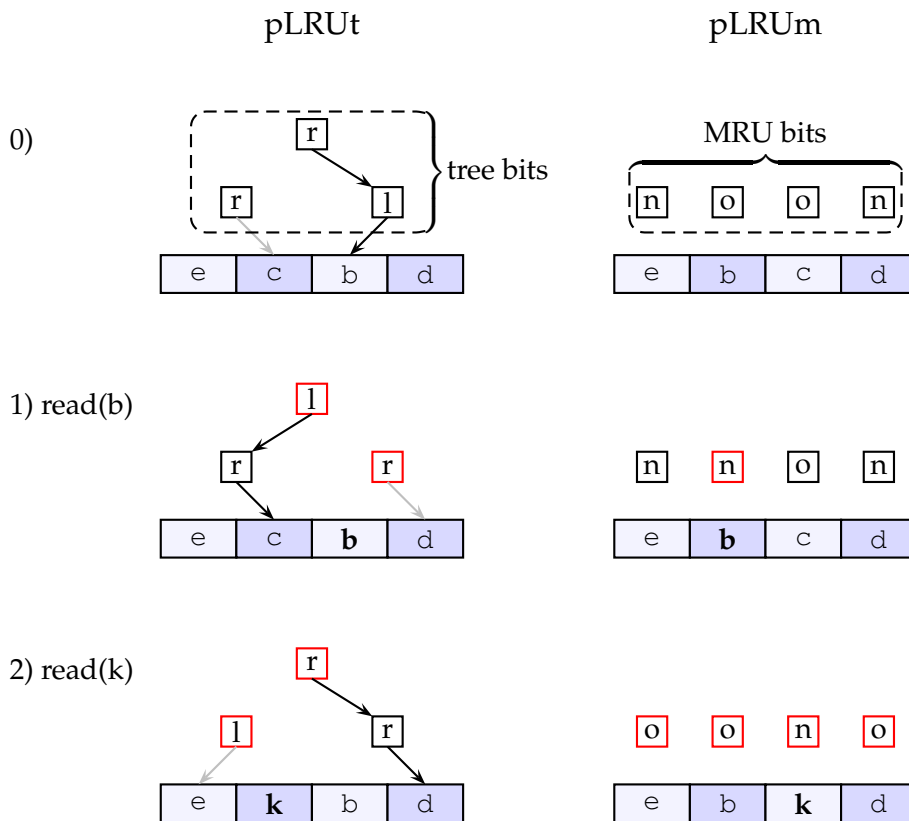


Figure 4: tree based LRU and MRU based LRU replacement policies

Another approximation of the LRU policy is the MRU bits based (pLRU_m) method. Every line in a set has a MRU bit that shows whether that line has recently been used ('n': new) or not ('o': old).

An accessed line is marked 'new' and only 'old' ones are replaced. If the last 'old' line of a set is replaced and marked as 'new' all other $S - 1$ MRU bits are updated to 'old' otherwise all bits would mark their corresponding lines as 'new' and none could be replaced. Step 2 in figure 4 shows that speciality. The other steps are self-explanatory.

Necessary information to describe a cache is:

name	data	parameter	
associativity (the number of ways)	2^{w_i}	w_i	$i = \{1, 2\}$
cache size	2^{s_i}	s_i	and refers to L1, L2
line length	2^l	l	

It is common, although not necessary, that the line length is identical for both cache levels, therefore this document covers the case where $l_1 = l_2 = l$.

A set-associative cache has 2^{s-w-l} entries:

$$\frac{\text{size}}{\text{associativity} \cdot \text{linelength}} = \frac{2^s}{2^w \cdot 2^l} = 2^{s-w-l}$$

That means it has an address width $a_i = s_i - w_i - l$.

The least significant l bits of an address are used to determine the corresponding byte within a cache line and the following a_1 bits are used to index the corresponding set of L1.

When data is fetched from memory always a whole cache line is read. That is why the right-most l bits are of no importance in indexing a cache. Often *reduced* addresses are used to ease understanding and so the least significant l bits are neglected.

An example for a 2-way cache of 32 B size and 4 B line size is shown in figure 5:

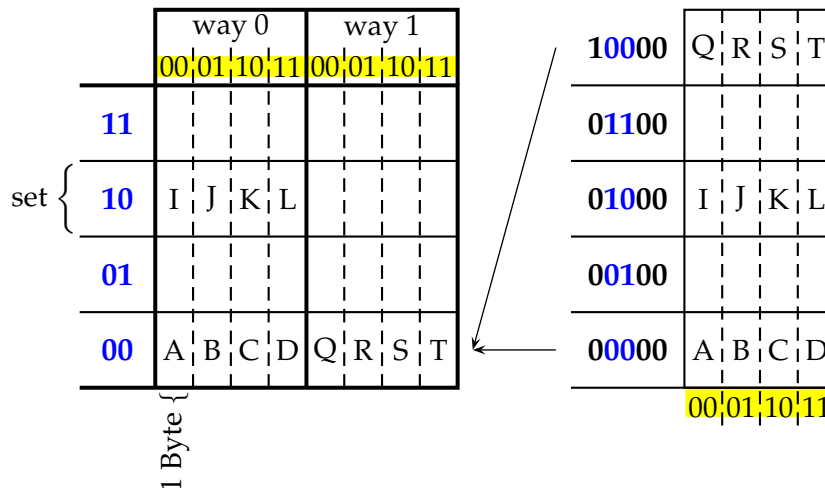


Figure 5: example of a 2-way cache

missing or even are incorrect, e.g. the statement that the L1-Cache of the PII/III uses a LRU replacement strategy (→ section 3.3.3).

A general description of the caching architecture and their configuration gives Intel's System Programming Guide ([Int04]).

The Netburst microarchitecture ([H⁺01]) of the P4 features a 128 B sectored cache that fetches 2 adjacent cache lines on a miss from memory and a hardware prefetcher that monitors access patterns and prefetches data automatically.

Both features can be disabled through the IA32_MISC_ENABLE MSR (bit 9 and/or 19 on address 0x1a0). When enabled *one* cache miss initiates *two* 64 B memory reads, to fill *two* adjacent cache lines (sector based read), however writes are always line based and only write the modified 64 B back into main memory.

[Int] states that L1 uses a write through policy and that "All caches use a pseudo-LRU replacement algorithm."

Yet *which* pLRU algorithm is not mentioned!

3 Analysis concepts

3.1 Performance Monitoring - IA-32 architecture

Model Specific Registers (MSRs) can be read and written in privilege level 0 only, using the `rdmsr`, `wrmsr` instructions, where registers `EDX:EAX` hold the content that is either read from or written to the MSR addressed by `ECX`.

The Time Stamp Counter, available since the Pentium processor, is incremented every CPU cycle and can be read using the `rdtsc` instruction. Here again, the 64 bit content is available in `EDX:EAX`.

“The RDTSC instruction is not serializing or ordered with other instructions. Thus, it does not necessarily wait until all previous instructions have been executed before reading the counter.” ([Int04, p. 15-26])

The instructions `rdmsr`, `wrmsr` however, are serializing and can be executed *before* the TSC is read.

3.1.1 P6 family

The P6 family - to which the PII, PIII belong - utilizes two 40 bit counter with corresponding event select and controlling registers:

Table 4: performance monitoring MSRs - P6 family

Name	Address	Meaning
PerfEvtSel0	0x186	event selection MSR 0
PerfEvtSel1	0x187	event selection MSR 1
PerfCtr0	0xC1	counter 0
PerfCtr1	0xC2	counter 1

Because the event selection registers are 32 bit wide, it is enough to modify only `EAX` when writing to these MSRs:

```
// set the low part of PerfEvtSel MSR to val
#define set_esr(msr, val)\
    __asm__ __volatile__(\
        "xorl %%edx, %%edx\n\t"\
        "wrmsr"\
        :\
        : "c" (msr), "a" (val)\
        : "edx")
```

The counters can be started and stopped by setting / clearing the `ENABLE` flag in the `PerfEvtSel0` register:

```
// start counting
#define start_counting()\
    __asm__ __volatile__(\
        "movl $0x186, %%ecx\n\t"\
        "rdmsr\n\t"\
        "bts $22, %%eax\n\t"\
        "wrmsr"\
        :\
        :\
        : "eax", "ecx", "edx")
```

```
// stop counting
#define stop_counting()\
    __asm__ __volatile__(\
        "movl $0x186, %%ecx\n\t"\
        "rdmsr\n\t"\
        "btr $22, %%eax\n\t"\
        "wrmsr"\
        :\
        :\
        : "eax", "ecx", "edx")
```

Appendix A.3 of [Int04] lists performance monitoring events of the P6 family. These are simple to configure and are not mentioned here.

3.1.2 PIV, Xeon

The Pentium IV features 18 performance counters and configuration registers. Table 15-2 on page 15-28 of [Int04] shows the association of counters, CCCRs (Counter Configuration Control Reg.) and ESCRs (Event Selection Control Reg.) and appendix A.1 gives information on countable events.

Table 5 lists only a few of them.

The columns ESCR, CCCR, Counter show the addresses of the MSRs, whereas EvSel and CSel give the values that have to be in the EVENTSELECT field of the ESCR and in the ESCR-SELECT field of the CCCR register.

If several CCCR and counter addresses are given, then one of them has to be chosen.

The configuration and control registers are 64 bit wide, however the upper 32 bit are reserved, so that only the lower part (EAX) is modified. Bit 16 and 17 shall always be set, so this is done when setting up a CCCR:

```
// set the low part of CounterConfigurationControl MSR to val
// set bit 16,17 (must be set)
#define set_cccr(cccr, val)\
    __asm__ __volatile__(\
        "xorl %%edx, %%edx\n\t"\
        "orl $(0b11<<16), %%eax\n\t"\
        "wrmsr"\
        : \
        : "c" (cccr), "a" (val)\
        : "edx")
```

Counters are started and stopped through bit 12 of the corresponding CCCR.

Counting non-sleep CPU cycles

Chapter 15.10.9 of [Int04] describes how to count non-sleep clock ticks

“Non-Sleep Clockticks - Measures clock cycles in which the specified physical processor is not in a sleep mode or in a power-saving state.”

1. select one of the 18 counters and its corresponding ESCR, CCCR (→ table 15-2 of [Int04])
2. set EvSel to anything other than no event: 0x01
3. enable threshold comparison: set compare bit in CCCR (bit 18)
4. set threshold (bit 20-23) to 15 and set complement flag in CCCR (bit 19)

```
// cnt. non-sleep cycles
#define CYCLE_ESCR 0x3a6
#define CYCLE_CCCR 0x368
#define CYCLE_CNTR 0x308
// ---- set up counting of non-sleep cycles ----
set_escr(CYCLE_ESCR, ESCR_OS | (0x01<<ESCR_EVS_SHIFT));
set_cccr(CYCLE_CCCR, CCCR_CMP | CCCR_CPL | 0xf<<CCCR_THRESH_SHIFT);
```

Table 5: PIV - MSR configuration of selected performance monitoring events

event	ESCR	CCCR	Counter	Event Select	ESCR Select	Event Mask
predicted / mispred. branches		0x36c	0x30c	0x06	0x05	<div style="border: 1px solid black; display: inline-block; padding: 2px;"> 12 11 10 9 TM TP NM NP </div> T : Taken N : Not-Taken P : Predicted M : Mispredicted
	0x3cc	0x36d	0x30d			
		0x370	0x310			
		0x36e	0x30e			
	0x3cd	0x36f	0x30f			
	0x371	0x311				
cache misses		0x36c	0x30c	0x09	0x05	MSR 0xef1: set bits 24, 0 (L1 miss), 1 (L2 miss) MSR 0x3f2: set bit 0
	0x3cc	0x36d	0x30d			
		0x370	0x310			
		0x36e	0x30e			
	0x3cd	0x36f	0x30f			
	0x371	0x311				
µops retired		0x36c	0x30c	0x01	0x04	Bit 0: bogus Bit 1: non-bogus
	0x3b8	0x36d	0x30d			
		0x370	0x310			
		0x36e	0x30e			
	0x3b9	0x36f	0x30f			
	0x371	0x311				
FSB data activity	0x3a2	0x360	0x300	0x17	0x06	0: drive data onto bus 1: read data 2: other processors reset bits 3,4,5
		0x361	0x301			
	0x3a3	0x362	0x302			
		0x363	0x303			
IOQ allocation	0x3a2	0x360	0x300	0x03	0x06	0-4: 0b00001 5: read 6: write 7:UC 8:WC 9:WT 10:WP 11:WB 13: own 14: other proc. 14: prefetch
		0x361	0x301			
	0x3a3	0x362	0x302			
		0x363	0x303			

3.2 Branch prediction

[MMK04] describes possibilities to determine the organization of branch predictors, e.g. to explore the width N_H of the BHR and the number of bits that are used to index the branch target buffer (BTB).

With this information it is easy to achieve both, the best and worst case in a branch prediction benchmark.

The strategies presented in [MMK04] were adapted to run as RTAI kernel modules and were extended by a test whether a local or a global history component is used, which seemed to be more simple and easy to understand than the one given in the paper.

As well, the algorithms were extended to take the cache behaviour into account (\rightarrow sec. 5.1.2, p. 40).

3.2.1 Branch History (shift-)Register (BHR)

If the BHR has a width of N_H bits it can store the last N_H outcomes (T: Taken / N: Not taken). Any further branches will override previous ones and the index function will select a wrong entry in the BTB - a misprediction is likely to occur. The outcomes of a branch that is taken only every mod 'th iteration will fit in the BHR as long as $mod < mod^*$ and almost no mispredictions will be counted. Yet, for $mod \geq mod^*$ the MPR (MisPrediction Ratio) will raise, because every mod 'th branch is mispredicted.

```

                                movl $ITER, %ecx    # number of iterations (outer loop)
                                movl $MOD, %ebx    # modulo parameter
again:    xorl %edx, %edx    # clear edx
                                movl %ecx, %eax
                                divl %ebx          # eax=(int)(eax/ebx), edx=modulo(eax,ebx)
                                testl %edx, %edx
                                jz 10              # "spy" branch
                                clc                # do sth.
10:      decl %ecx
                                jnz again

```

Figure 7: BHR benchmark

At this point it has to be distinguished between local and global BHRs. For a local component the number of history bits is $N_H = mod^* - 2$, because the history refers to the "spy branch" only but a global register saves the outcomes of the surrounding loop too, so $N_H = 2 \cdot (mod^* - 2)$. The reason for subtracting 2 is given in an example with a global BHR of width $N_H = 6$:

$N_H = 6$ means that the last 6 outcomes can be saved and because it is a *global* register only every second bit of the register is reserved for the spy branch (the other one is for the outer loop \rightarrow `jnz again`), therefore a history pattern of length 3 should fit in the register and one of length 4 should not fit and cause mispredictions:

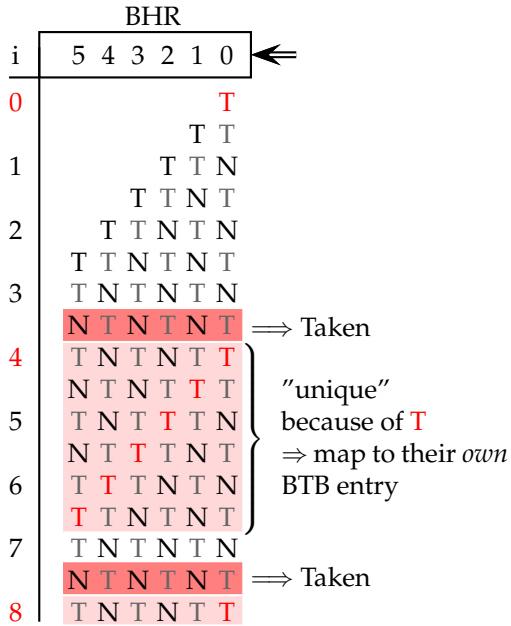


Figure 8: BHR pattern of length 4

As it can be seen in figure 8, a history pattern of length 4 can still be predicted correctly because the address of the spy branch instruction together with the history register content is unique for every taken spy branch and therefore refers to the same BTB index.

However a pattern of length

$$\frac{1}{2} N_H + 2 = mod^* = 5$$

causes every 5th branch to be mispredicted, due to a non-unique BHR content for the taken spy branches.

Through a variation of mod it is possible to find mod^* which is either an indication to a local $mod^* - 2$ bit BHR or a global $2 \cdot (mod^* - 2)$ bit BHR.

The modulo parameter mod can be given as `mod=<num>` option when loading the kernel module that executes the given code above (fig. 7).

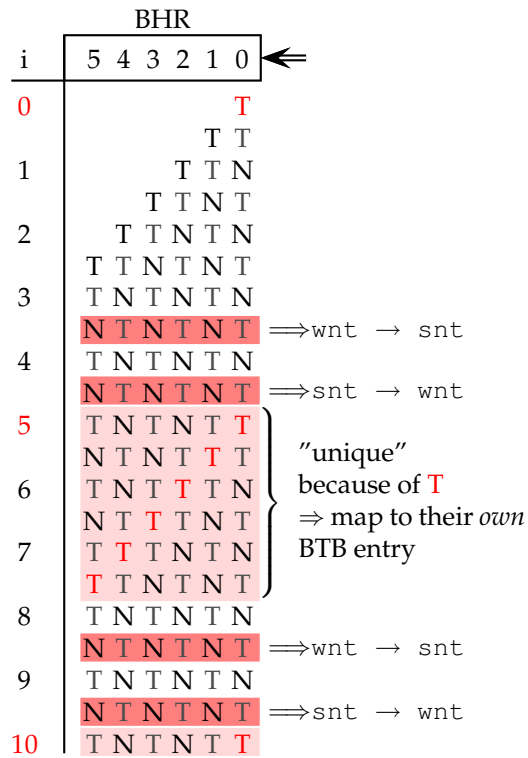


Figure 9: BHR pattern of length 5

To verify which kind of register - local or global - is used, I decided to execute a test with *two* of these modulo-branches, with the same modulo parameter.

If local history registers are used, then both branches have their own and the MPR will raise at the same modulo parameter as in the test with just one "spy" branch. However, if a global register is used, then both branches have to share it and influence each other, wherefore the MPR raises at a *lower* modulo parameter (at a shorter history pattern).

3.2.2 Branch Target Buffer (BTB)

When describing how to analyse the branch prediction architecture, [MMK04] assumes the number of BTB entries is known. Yet it is not difficult to obtain that information through some tests based on the algorithm explained in the following section, therefore the hints on how to measure N_{BTB} are given afterwards (p. 19).

As described when introducing caches (sec. 2.4, p. 5), addresses are split up into different parts to indicate which bits are used for what purpose. Because only one buffer is available, only one part, namely the z part is used to address it and the y and x portions are combined to y . The following figure shows an address used to index the BTB:

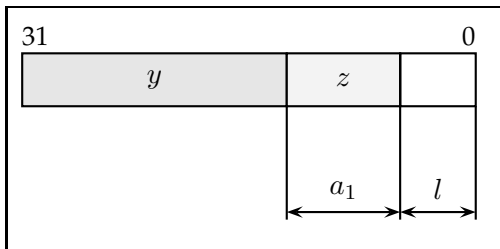


Figure 10: address used to index the BTB
only the z part of the address is used to index the buffer

If 2^l consecutive addresses map to the same set, then the least significant l bits of an address are not used to index the entry.

Starting with an example to a BTB that's parameters are all known:

It is a buffer with 512 entries, 4 ways (that means it has 128 sets) and an "unused" part that is $l = 4$ bits wide.

Because 128 sets have to be indexed the z part has to hold $a_1 = 7$ bits.

Addresses $y_i.z_j.0$ to $y_i.z_j.15$ map to the same set, because the least significant l bits are *not* used for indexing.

When executing 512 branch instructions with varying the distance between consecutive branch instruction's addresses, several phenomens can be observed:

(512 branch addresses *should* fit in a buffer with 512 entries)

For a distance of 2 bytes, 8 consecutive branch addresses ($y_i.z_j.0, y_i.z_j.2, \dots, y_i.z_j.12, y_i.z_j.14$) map to the same set. Yet, there are just 4 ways to store them, so the last 4 branch addresses will override the previous 4. In the next iteration of the outer loop, the addresses can not be found in the BTB and dynamic prediction is unavailable, therefore every branch will be statically predicted. If the branches are coded to be conditional *forward*-jumps that are always *taken*, then static prediction will always *fail*.

For a distance of 4 bytes, 4 consecutive addresses ($y_i.z_j.0, y_i.z_j.4, y_i.z_j.8, y_i.z_j.12$) map to the same BTB entry, so that every way is filled and all 512 addresses are stored in the BTB. Therefore *no* mispredictions will occur. (\rightarrow fig. 11)

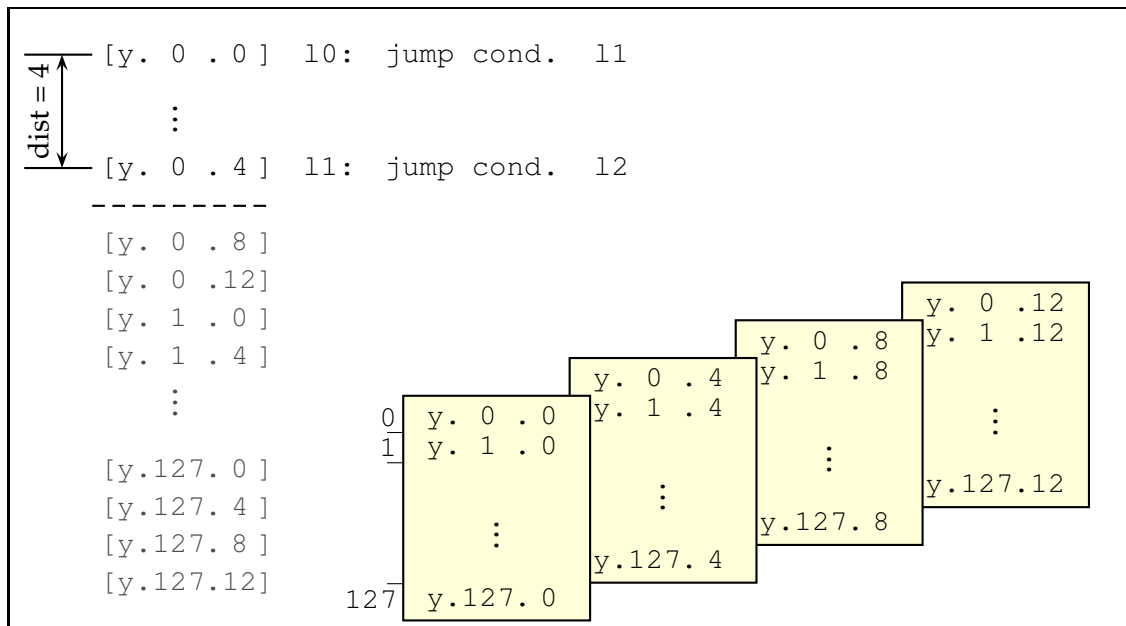


Figure 11: conditional branches at a distance of 4 bytes

A distance of 8 bytes will also fit in the buffer because a group of addresses $y_i.z_j.0, y_i, z_j.8$ needs only 2 ways and an 16 byte-distance is no problem too. (→ fig. 12)

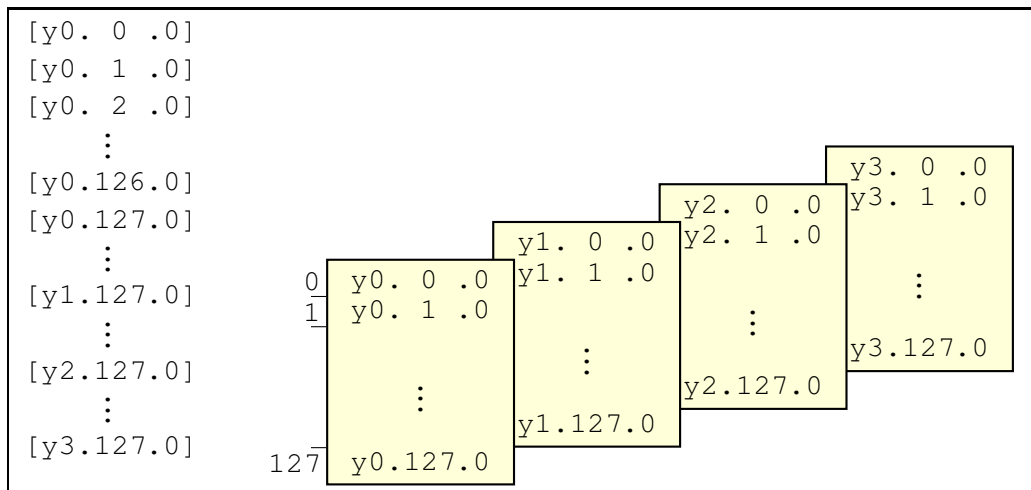


Figure 12: conditional branches at a distance of 16 bytes

However, addresses with distances greater than 16 will *not* fit in the BTB because some cache sets remain unused (marked as **free** in figure 13):

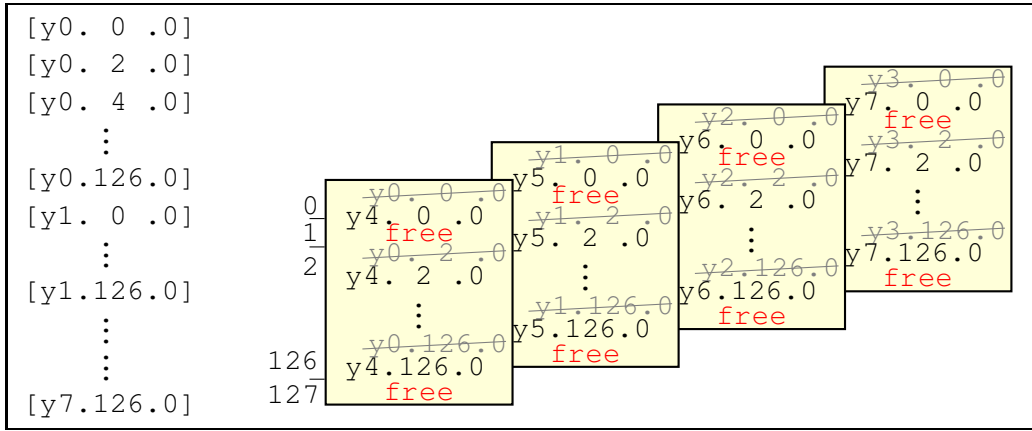


Figure 13: conditional branches at a distance of 32 bytes

If one address is $y_i.z_j.0$ the next following address for a distance of 32 bytes would be $y_i.z_{j+2}.0$ and the set with the index z_{j+1} will remain unused, so that the addresses *cannot* fit into the buffer. So for distances greater than 16 the misprediction ratio (MPR) will always be high. The conclusion is that the "unused" $l = 4$ bits of an address lead to a maximum "fitting distance" of $D = 16$ and that the sum of 3 "fitting distances" results from the 4-way associativity.

The 4 bits of l do not refer to an associativity of 4 !

If $l = 3$ then there would be 3 fitting distances, too. Yet the highest possible distance was not 16 but 8.

From this example it can be generally derived that:

1. f "fitting distances" lead to an associativity of

$$W = 2^{F-1} \Rightarrow w = F - 1$$

2. the highest "fitting distance" D leads to

$$l = \log_2(D)$$

3. N_{BTB} entries at an associativity of W form $S = \frac{N_{BTB}}{W}$ sets, so that the z part must contain S addresses, that means:

$$a_1 = \log_2(S) = \log_2\left(\frac{N_{BTB}}{W}\right)$$

The benchmark allows to vary the distance between subsequent conditional jump instructions (`jump`) and the number of these. Also, any conditional branch is directed forward, so that static prediction will mispredict them.

A problem faces the "surrounding" loop, that allows repetition of the whole jump scenario so that the influence of inaccuracy of the performance counters can be reduced.

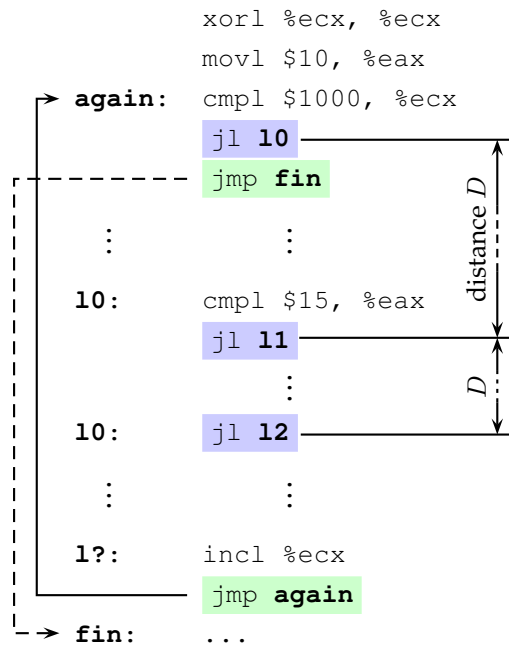


Figure 14: structure of the BTB benchmark

Figures 15 and 16 show a code example for a microbenchmark with a distance $D = 32$ and a number of $B = 3$ conditional branches.

The instructions really *executed* in both examples are the same, except that figure 15 uses *forward* and figure 16 *backward* branches.

The ellipsis symbolize any assembler instructions that are used to fill the distance D .

The size of the unconditional `jmp` instructions depends on the jump distance. For distances less than 128 B no more than 2 B are needed, instead of 5 B for greater distances. However the distance can not easily be calculated because the distance itself depends on the size of the unconditional jump.

To circumvent the problem, avoid combinations with

$$\langle \text{number_of_branches} \rangle \cdot \langle \text{distance} \rangle \approx 128$$

Or at least be aware that the first distance D might not be of correct size.

Figure 15: code snippet of the BTB microbenchmark with *forward* branches

```

__asm__ __volatile__ (
"xorl %%ecx, %%ecx\n\t"
"movl $10, %%eax\n\t"
"again: cmpl $100000, %%ecx\n\t"
"jl 10\n\t"
"jmp fin\n\t"
    "movl $10, %%eax\n\t"
    "movl $10, %%eax\n\t"
    "movl $10, %%eax\n\t"
    "movl $10, %%eax\n\t"
    "movl $10, %%eax\n\t"
"10: cmpl $15, %%eax\n\t"
"jl 11\n\t"
    "movl $10, %%eax\n\t"
    "movl $10, %%eax\n\t"
    "movl $10, %%eax\n\t"
    "movl $10, %%eax\n\t"
    "movl $10, %%eax\n\t"
    "movl $10, %%eax\n\t"
"11: jl 12\n\t"
    "movl $10, %%eax\n\t"
    "movl $10, %%eax\n\t"
    "movl $10, %%eax\n\t"
    "movl $10, %%eax\n\t"
    "movl $10, %%eax\n\t"
"12: incl %%ecx\n\t"
"jmp again\n\t"
"fin:"
::: "eax", "ecx"
);

```

Figure 16: code snippet of the BTB microbenchmark with *backward* branches

```

__asm__ __volatile__ (
"xorl %%ecx, %%ecx\n\t"
"movl $10, %%eax\n\t"
"jmp again\n\t"
"10: incl %%ecx\n\t"
"jmp again\n\t"
"11: jl 10\n\t"
    "movl $10, %%eax\n\t"
    "movl $10, %%eax\n\t"
    "movl $10, %%eax\n\t"
    "movl $10, %%eax\n\t"
    "movl $10, %%eax\n\t"
    "movl $10, %%eax\n\t"
"12: cmpl $15, %%eax\n\t"
"jl 11\n\t"
    "movl $10, %%eax\n\t"
    "movl $10, %%eax\n\t"
    "movl $10, %%eax\n\t"
    "movl $10, %%eax\n\t"
    "movl $10, %%eax\n\t"
    "movl $10, %%eax\n\t"
"again: cmpl $100000, %%ecx\n\t"
"jl 12"
::: "eax", "ecx"
);

```

Number of BTB entries

To use that benchmark it is necessary to know the number of BTB entries. Either this information can be found in some documentation or it is gained with the benchmark itself:

Start at the smallest possible distance $D = 2$ and a small number of branches, e.g. 32.

- Ⓐ If the MPR is low, increase the number of branches until the MPR is high. The highest number of branches that did not cause a high MPR is the number of BTB entries N_{BTB} .
- Ⓑ If the MPR is high, then increase the distance D until the MPR gets low. (leave the number of branches unchanged!)
When a distance with a low MPR is found, continue with step Ⓐ.

3.3 Caching

Legend of used variables

$\{a, b, \dots\}$	cache line
$\{A_1, B_1, \dots\}$	address range that fills one L1 cache way
$\{A_2, B_2, \dots\}$	address range that fills one L2 cache way
$\{A_0, A_1, \dots\}$	an address range that fills L1/L2 completely (here the number does <i>not</i> refer to L1/L2, it just distinguishes different addr. ranges)

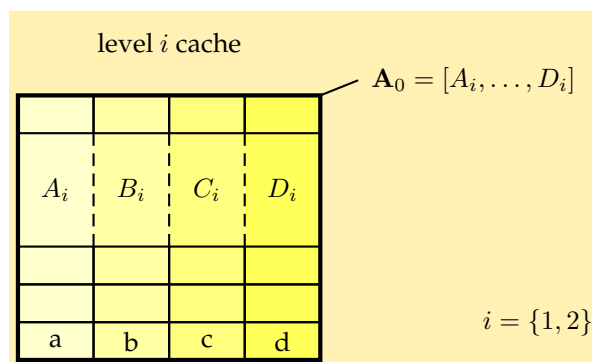


Figure 17: cache variables

3.3.1 Adjacent cache line prefetch - PIV

To test the sectored cache line fill and its disabling I wrote a short program that allocates a continuous memory block as big as the second level cache (512 KB) and accessed (read) only every second line within this block. If the adjacent cache line prefetch is enabled (default), *all* addresses of the memory area should be cached and cause *no* L2 misses on a second read, but if this feature is *disabled* only *half the addresses* will be available in L2. Because it is not known at which address the memory block starts - at the first byte of a sector or at the second, the test is executed twice. The first time the memory is accessed starting at an offset of 0 B, the second time at an offset of 1 B.

3.3.2 Caching strategy of the L1 cache

write-allocate/write-no-allocate

Simple to test is the usage of write-allocate/write-no-allocate because if L1 holds data that has been *written to*, the cache utilizes a write-allocate policy.

So the following steps have to be performed:

1. invalidate caches (at least L1), so that no data is cached

```
wbinvd
```

2. *write* to as many addresses as fit into L1

```
write(A0)
```

3. *read* these addresses and count the L1 misses ($\rightarrow n_{\text{miss}}$)

```
read(A0)  $\rightarrow n_{\text{miss}}$ 
```

\rightarrow if the misses are approximately as high as the number of read addresses, write-**no**-allocate is used

→ if there are just a few misses, the data has been stored in L1 $\hat{=}$ write-allocate

write-through/write-back

Write-through/write-back are cache *hit* policies, so the data, the test is working on, has to be *in* L1. If write-through is used then any write is performed in L1 *and* in L2 too, as opposed to write-back, where only the data in L1 is updated. If L1 is filled with data that has been *written* to and new addresses are loaded, then the modified data has to be purged out of L1. In the case of a write-through L1 cache the "old", modified data is already resident in L2 and can immediately be overwritten, whereas a write-back L1 cache has to first write back the data into L2, before the new can be loaded. The time needed for loading the "new" data should be longer in the latter case.

The structure of the benchmark is as follows:

1. fill L1 twice through *reading* the addresses \mathbf{A}_0 for the first fill and \mathbf{A}_1 for the second

```
read( $\mathbf{A}_0$ )
read( $\mathbf{A}_1$ )
```

2. *write* to the addresses \mathbf{A}_1 *already in L1* (we are examining a write *hit* policy)
 - if L1 uses write-back then L2 is *not* updated

```
write( $\mathbf{A}_1$ )
```

3. *read* addresses \mathbf{A}_0
 - (these have to be loaded from L2 and push out the modified data \mathbf{A}_1)
 - and take the time (→ t_{mod})
 - and count the L2 accesses (→ n_{mod})

```
read( $\mathbf{A}_0$ ) →  $t_{\text{mod}}, n_{\text{mod}}$ 
```

4. repeat step 1 (fill L1 twice)
 - as a result addresses \mathbf{A}_1 are cached in L1

```
read( $\mathbf{A}_0$ )
read( $\mathbf{A}_1$ )
```

5. repeat step 3 (read addresses \mathbf{A}_0)
 - time t_{unmod}
 - L2 accesses n_{unmod}
 - because the data has *not* been modified, this time, it can be purged regardless of the applied policy and $t_{\text{unmod}}, n_{\text{unmod}}$ correspond to $t_{\text{w-through}}, n_{\text{w-through}}$

```
read( $\mathbf{A}_0$ ) →  $t_{\text{unmod}}, n_{\text{unmod}}$ 
```

→ if the time is longer and the count is bigger in the case of the modified data then a write-back policy is used for L1

$$\left. \begin{array}{l} t_{\text{mod}} > t_{\text{unmod}} = t_{\text{w-through}} \\ n_{\text{mod}} > n_{\text{unmod}} = n_{\text{w-through}} \end{array} \right\} \Rightarrow \text{write-back}$$

3.3.3 Replacement strategy

When there were some documents with information to base my findings on caching strategy, there is almost none (and if, not extensive) information on replacement policies. As far as I found out, the Intel manuals only cover the Netburst microarchitecture when stating

“All caches use a pseudo-LRU (least recently used) replacement algorithm.”

[Int, p. 1-19]

However, it is never mentioned *which* pLRU strategy is applied.

The statement of [Sea00] to the Pentium II is more clear because more simple:

“L1 uses a 4-way set associative mapping which divides the 512 lines into 128 sets of 4 cache lines.

Each of these sets is really a least recently used (LRU) list.”

Because it seems that Intel® makes use of LRU based strategies the benchmarks are aimed in that direction.

As can be seen in figure 4 the pLRUm algorithm has some disadvantages compared to pLRU:

1. it needs more pLRU bits
namely one per way in each set $\longrightarrow N_{\text{bits}, m} = W \cdot S$
whereas pLRU needs 1 bit less per set $\longrightarrow N_{\text{bits}, t} = (W - 1) \cdot S$
2. the moment the last “old” entry of a set is replaced, marked as “new” and *all* other entries are marked “old”, the history of these lines is lost
 \rightarrow see figure 18

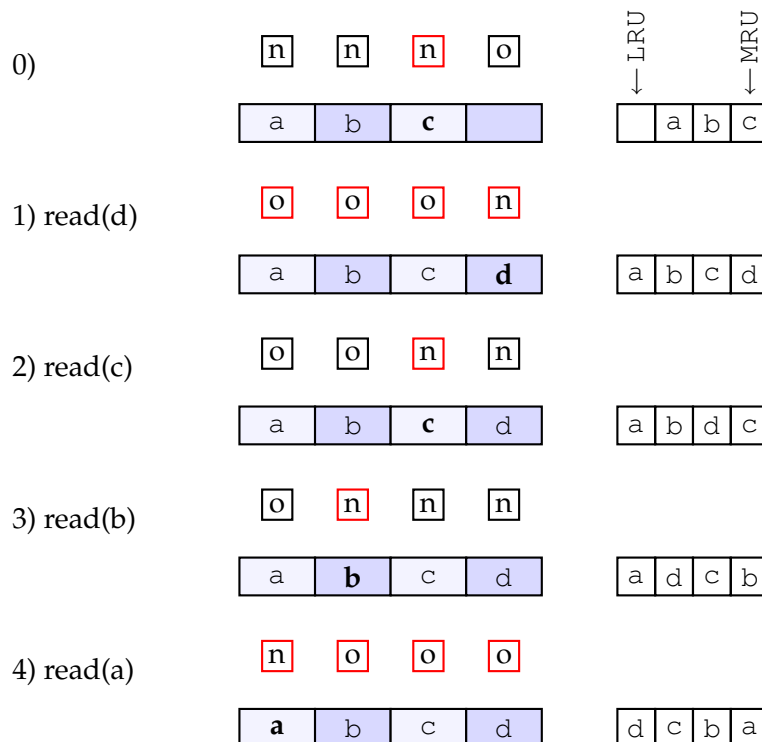


Figure 18: pLRUm history loss

Figure 18 shows quite clearly that the MRU-based pseudo LRU policy is an approximation to the strict LRU algorithm. After lines a , b , c , d have been read, to be loaded in the L1 cache, they are read in the reverse order (c , b , a) to make a the most and d the least recently used entry (step 4: LRU stack is d , c , b , a).

That means, when replacing lines, d should be the first, c , b the next and a the last. However we assumed that the cache ways are filled from left to right (step 0, 1), therefore the "old" lines will be replaced in the order b , c , d !

Following these thoughts my assumption was that if a pseudo LRU strategy was used, the tree based algorithm would be preferred.

To analyse the underlying policy one could fill the cache, reload some "ways" to make these addresses the most recent ones and afterwards load any new "way" to purge an old one - which one, shall be the indication for the used algorithm.

The detailed structure looks like:

1. read as many addresses as fit into L1
 PII/III/IV have a 4-way L1 cache, so addresses A_1, B_1, C_1, D_1 have to be loaded

```
read(A1)
⋮
read(D1)
```

2. reload (read) some "old" ways to make them the MRU ones

```
read({A1, ..., D1})
⋮
```

3. load (read) a "new" way which overrides another one

```
read(E1)
```

4. read the "old" addresses and count the L1 misses for each way

```
read(A1) → nA1
⋮
read(D1) → nD1
```

→ the way with the most misses is the one been replaced

If the second level cache is that large, that *one* of its ways can hold more lines than fit into L1, it is sure that the reloading of one L2 way really reads from and updates the lines in L2, not only in L1.

$$a_2 \geq a_1 + w_1$$

If this condition is met, the presented algorithm can be used to analyse the replacement strategy of L2, too.

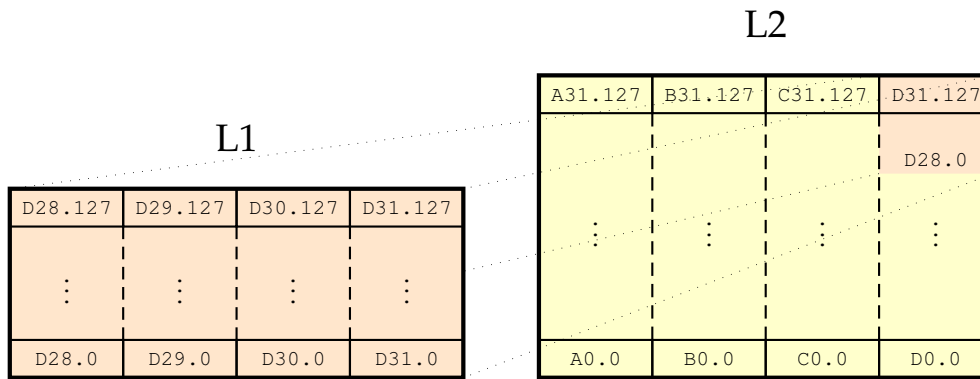


Figure 19: sizing relations L1 - L2 based on PII

If ways have been loaded in the order A_2, B_2, C_2, D_2 and afterwards way C_2 shall be reloaded, it is sure that none of the addresses $C0.0 - C31.127$ still reside in L1, because they have been purged when loading D_2 .

To obtain several hints on the applied replacement policy I decided to vary the reloading of the 3 oldest ways A_i, B_i, C_i . Resulting possibilities are:

$$\% / A_i / B_i / A_i, B_i / C_i / A_i, C_i / B_i, C_i / A_i, B_i, C_i$$

After carrying out the first experiments, I realized that there are different possibilities how a cache with a pLRU_t policy can be loaded:

Either the tree bits are used and the empty cache is filled in tree order, or more simple, the cache fills the first (e.g. from left to right) free entry and only uses the tree bits, when all entries of a set are filled. Figure 20 shows a 4-way cache set that uses a pLRU_t policy with *tree based* filling of empty entries, whereas fig. 21 presents a filling of the *first* free entry.

Once again, updated tree bits are boxed red and black arrows symbolize the path through the tree.

Every step shows the cache set *after* the given instruction has been executed.

Step 0 shows the "fresh" set after the cache has been invalidated.

Steps 1 to 4 present the filling and steps 5, 6 explain how the *pseudo*-LRU algorithm can be tricked to replace an entry other than the least recently used.

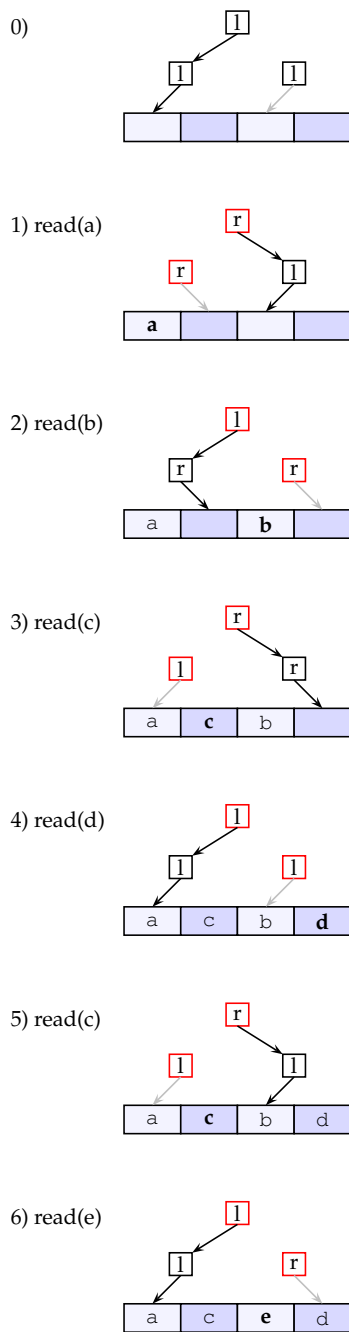


Figure 20: tree based fill

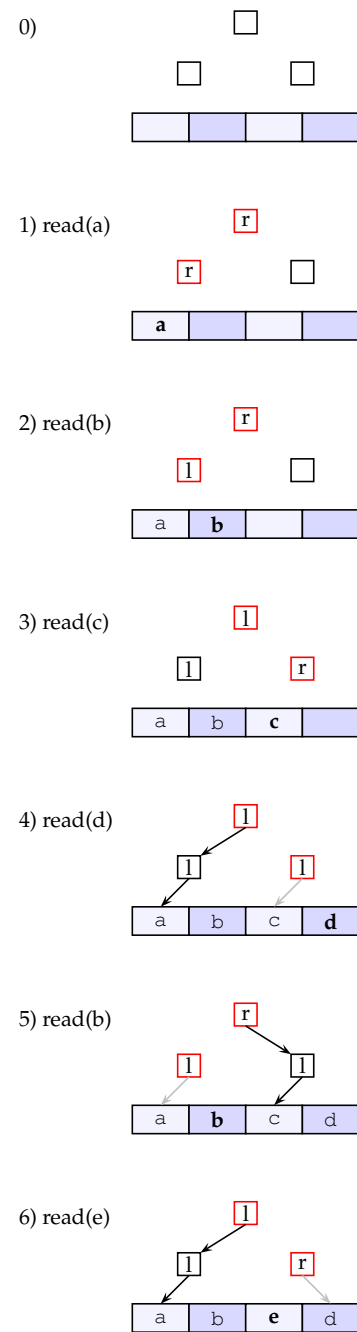


Figure 21: sequential fill

In steps 0 to 3 of drawing 21 the tree bits do not have arrows indicating the path to the LRU line to stress that this path is *not* evaluated as long as there are free entries.

Table 6 faces the results of varying the ways to be reloaded between tree based and the sequential filling.

The leftmost column gives the combinations of the 3 "oldest" ways to be reloaded. Columns 2 to 5 show the cache ways that should be replaced.

Lightblue backgrounded rows are those which have a different result depending on the filling method.

Table 6: cache ways expected to be replaced through a pLRU strategy

reload	4 ways		8 ways	
	tree based fill	sequential fill	tree based fill	sequential fill
none (%)	<i>A</i>	<i>A</i>	<i>A</i>	<i>A</i>
<i>A</i>	<i>B</i>	<i>C</i>	<i>B</i>	<i>E</i>
<i>B</i>	<i>A</i>	<i>C</i>	<i>A</i>	<i>E</i>
<i>A, B</i>	<i>C</i>	<i>C</i>	<i>C</i>	<i>E</i>
<i>C</i>	<i>B</i>	<i>A</i>	<i>B</i>	<i>E</i>
<i>A, C</i>	<i>B</i>	<i>B</i>	<i>B</i>	<i>E</i>
<i>B, C</i>	<i>D</i>	<i>A</i>	<i>D</i>	<i>E</i>
<i>A, B, C</i>	<i>D</i>	<i>A</i>	<i>D</i>	<i>E</i>

How the pseudo LRU tree based replacement strategy is applied on a 8 times associative cache is illustrated in figure 22. The drawing is to be understood as the others of this kind before.

This benchmark analysed which ways of a cache are being replaced when reading a "new" way not yet cached. To obtain several hints on the underlying policy the number and order of reloaded ways A_i, \dots, C_i could be changed to influence the pLRU history.

Yet there is still another simple method to test the replacement strategy:

First the cache is filled with addresses A_0 . Afterwards *one* "new" way is read and it is checked which "old" way it replaces. Then the cache is filled again with A_0 but this time *two* "new" ways are loaded and it is noted which two "old" ways are replaced by them. This method is repeated until as many "new" ways are loaded as fit into the cache, that means until all "old" ways have been purged.

With that incremental replacing of one to W ways of a cache it can be observed which ways the pseudo LRU algorithm selects as least recently used and that helps to identify the used algorithm.

The structure of this benchmark is as follows:

1. read as many addresses as fit into L1
 PII/III/IV have a 4-way L1 cache, so addresses A_1, B_1, C_1, D_1 have to be loaded

```
read( $A_1$ )
⋮
read( $D_1$ )
```

2. load (read) *one* "new" way which overrides another *one*

```
read( $E_1$ )
```

3. read the "old" addresses and count the L1 misses for each way

```
read( $A_1$ ) →  $n_{A_1}$ 
⋮
read( $D_1$ ) →  $n_{D_1}$ 
```

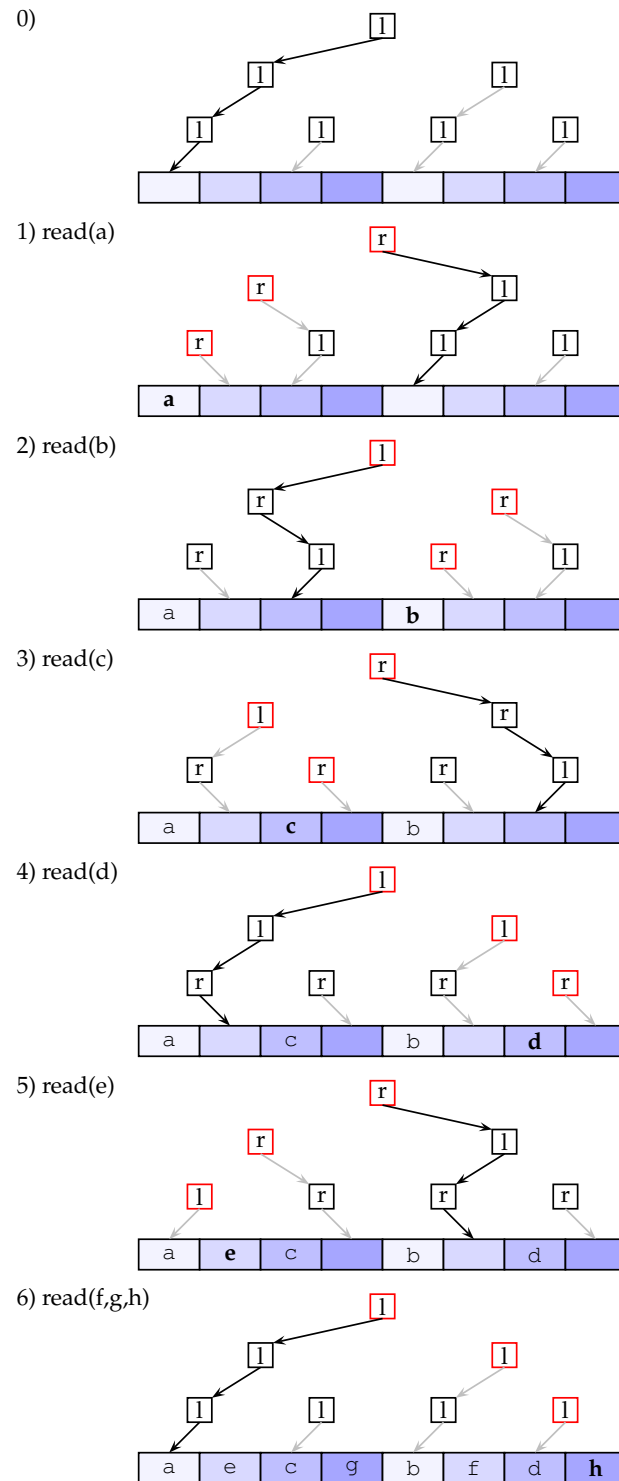


Figure 22: example of a 8-way cache with a pLRU_t policy and tree based filling

→ the way with the most misses is the one been replaced: X_1

4. repeat step 1

```
read(A1)
⋮
read(D1)
```

5. load (read) *two* "new" ways which override another *two*

```
read(E1)
read(D1)
```

6. repeat step 3

```
read(A1) → nA1
⋮
read(D1) → nD1
```

→ the ways with the most misses are the ones being replaced: X_1, X_2

One of the two replaced ways is that, been replaced in step 3, so the other one holds **new** information.

⋮

9. repeat step 3

→ the ways with the most misses are those being replaced: X_1, X_2, X_3

⋮

12. repeat step 3

→ the ways with the most misses are those being replaced: X_1, X_2, X_3, X_4

$X_k = \{A_1, B_1, C_1, D_1\}, \quad k = \{1, 2, 3, 4\}$

As explained before (→ fig. 19, p. 24), this test can be applied to L2 too. Only address ranges have to be adapted:

$A_2, \dots, D_2/A_2, \dots, H_2^3$ have to be read to fill L2 and $E_2, \dots, H_2/I_2, \dots, P_2^3$ are those "new" ways to purge the "old" ones.

³4-way/8-way L2

4 Worst case on caching

Because access to caches is much faster than access to main memory, caches may greatly improve performance. However if data is not found in cache it has to be retrieved from memory and even worse if the cache is already filled with modified data, which has to be written back to memory before new data can be loaded into it, it needs much more time than a single load from main memory.

4.1 Cache flooding I

[LH] describes the worst case when working with a 2-level memory cache architecture and how to achieve that case. Conditions to that so called "double purge" configuration are:

- * 2-level cache architecture
- * at least a 2-way L1
- * write-back strategy
- * a strict LRU cache line substitution mechanism
- * L2 is at least $a_1 \cdot a_2$ times bigger than L1
(a_1, a_2 are the address widths of L1/L2)

4.1.1 Double Purge Scenario

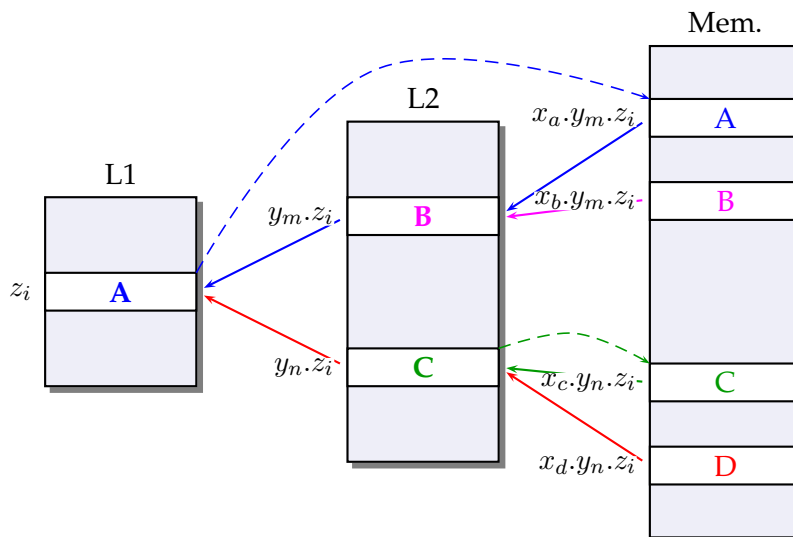


Figure 23: cache configuration,
solid arrows indicate mapping relations,
access to **D** results in "double purge case"

The uppercase alphabetic characters (A,B,C,D) in figure 23 denote a whole cache line and the solid arrows indicate the mapping relations from memory to L2 and from L2 to L1.

The cache lines **A**, **B**, **C** contain modified data.

Cache line **D** maps to the entry z_i of L1 which is already filled with **A**, so for caching **D** in L1, line **A** has to be purged to L2. However L2 cannot hold **A** because modified line **B** occupies the corresponding entry, so **A** is written-through to memory.

→ first write operation to main memory, symbolized through the **blue dashed arrow** in fig. 23

To load **D** into L2, entry $y_n.z_i$ is the only one that can be used, so the modified line **C** has to be written-back to memory, to free the necessary space.

→ second write to memory, symbolized through the **green dashed arrow** in fig. 23

All in all 2 write (hence the name *double purge*) and one read access have to be performed before the data is available to the processor.

4.1.2 Cache flooder – as described in [LH]

The algorithm presented in [LH] floods the cache in such a way that almost any memory access results in a double purge scenario.

Figure 24 shows the state of the L1, L2 caches after being flooded for an exemplary L1 cache with only 4 sets.

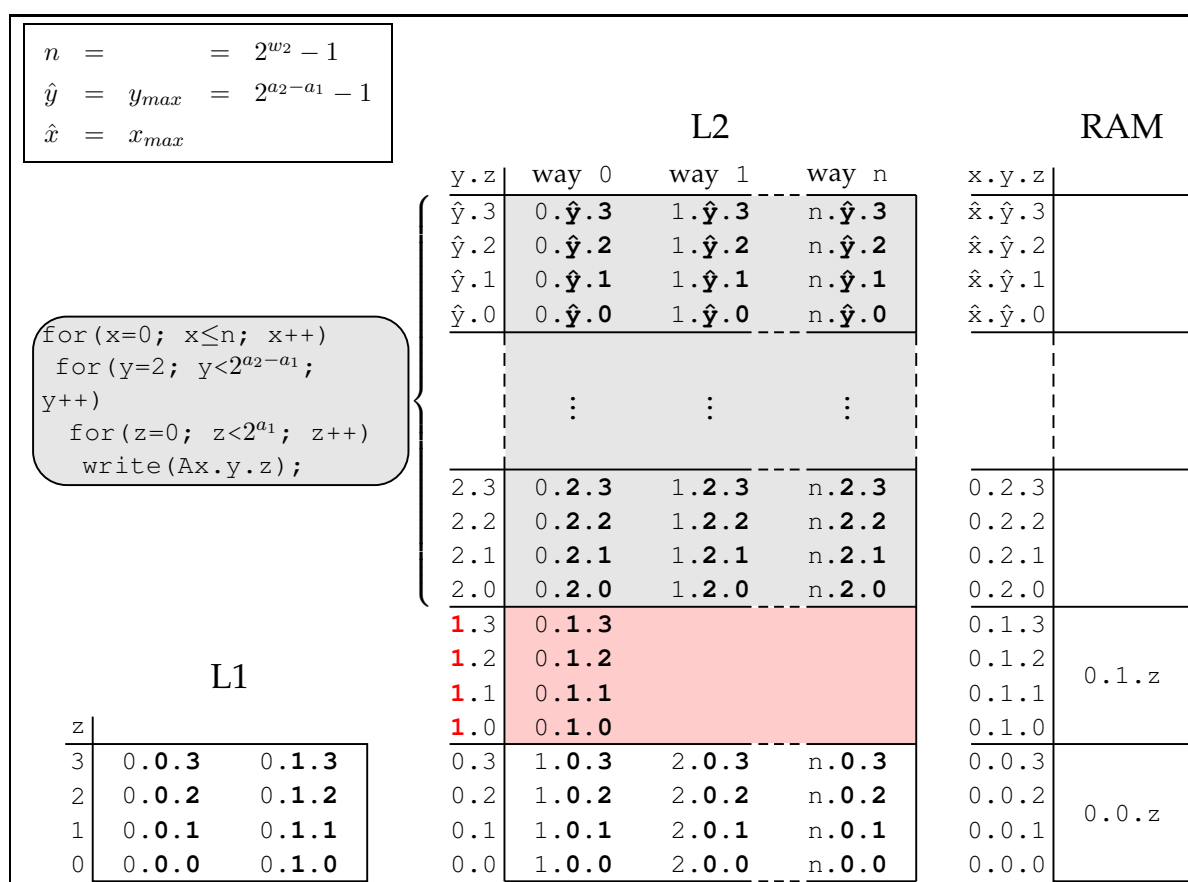


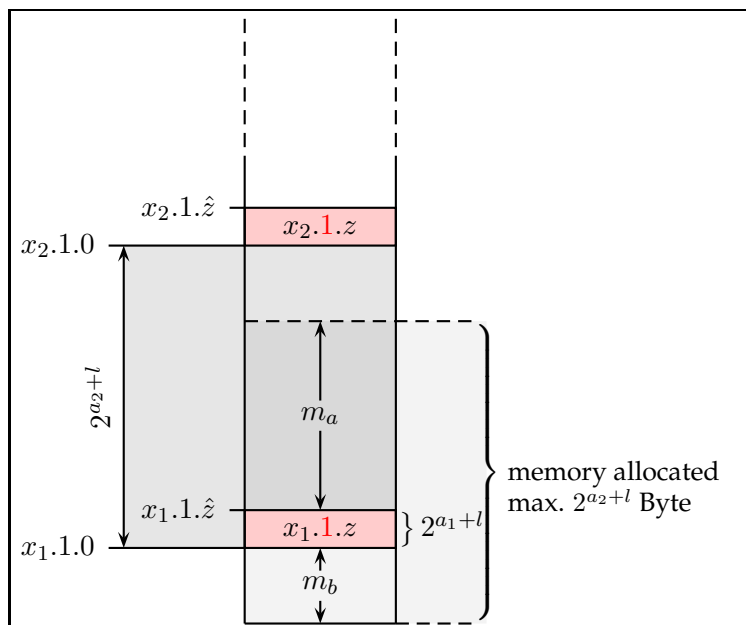
Figure 24: cache contents after being flooded

As mentioned above, not every memory access results in the worst case. All the data already buffered in the caches must not be referenced which does not face up a problem because this data belongs to the memory of the flooder and if it is not shared, it is not accessed by anyone else.

More difficult to protect is the red marked area which is mapped to addresses with $y = 1$. Access to data that is to be stored there will not result in a double purge case because the L2 lines are invalid and will simply be overwritten by the corresponding L1 lines because those are more recent.

That means that any access to addresses with $y = 1$ has to be restricted. [LH] suggests Cache Coloring which requires changes in memory management.

Another much simpler possibility is to allocate a block of memory that is twice as big as actually needed plus the number of bytes that could be taken up by addresses with $y = 1$. From that memory block only the part above (m_a in figure 25) or below (m_b) the ($y = 1$)-zone will be used, depending on its size. This solution is greedy but simple to implement - only physical addresses are needed, that means it runs in kernel space only.



In the example on the left (fig. 25) the m_a memory block would be used.

Figure 25: separation of memory in scopes with addresses having $y \neq 1$

Therefore if m bytes are needed, $b = 2 \cdot m + 2^{a_1+l}$ bytes have to be requested. However not more than a maximum of 2^{a_2+l} byte can be allocated because otherwise there might be more than one ($y = 1$)-zones in it!

$$b = 2 \cdot m + 2^{a_1+l} \stackrel{!}{<} 2^{a_2+l}$$

Implementation of the cache flooder

Table 7 helps extracting the different parts (x, y, z) of an address:

Table 7: representing special binary numbers

description	representation	example
only the n least significant bits are <i>set</i>	$(1 \ll n) - 1$	$0b000 \underbrace{111}_{n=3} = 0b001000 - 1$ $= (1 \ll 3) - 1$
only the n least significant bits are <i>unset</i>	$-(1 \ll n)$	$0b111 \underbrace{000}_{n=3} = 0b111111 - 0b0111$ $= -1 - ((1 \ll 3) - 1)$ $= -(1 \ll 3)$

```

for(x = 0; x < 2w2; x++)
  for(y = 2; y < 2a2-a1; y++)
    for(z = 0; z < 2a1; z++)
      write(Ax.y.z);
for(z = 0; z < 2a1; z++)
{
  for(x = 0; x < 2w2; x++)
  {
    for(j = 0; j < 2w1 - 1; j++)
      write(Aj.0.z);
    write(A(x + 2w1 - 1).0.z);
  }
  for(j = 0; j < 2w1 - 1; j++)
    write(Aj.0.z);
  write(A0.1.z);
}

```

The highest accessed address is $A(x + 2^{w_1} - 1).0.z$ whereas x and z are incremented as long $x < 2^{w_2}$ and $z < 2^{a_1}$ is fulfilled. As a result the highest accessed address is:

$$\begin{aligned}
 & A(2^{w_2} - 1 + 2^{w_1} - 1).0.(2^{a_1} - 1) \\
 & = A(2^{w_1} + 2^{w_2} - 2).0.(2^{a_1} - 1) \\
 & = A\hat{x}.0.\hat{z}
 \end{aligned}$$

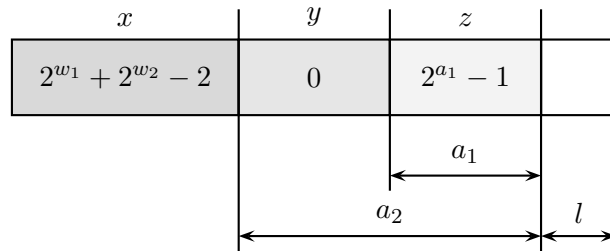


Figure 26: highest address accessed by the cache flooder algorithm

Finally

$$(\hat{x} \ll (a_2 + l)) + (\hat{z} \ll l) + 1$$

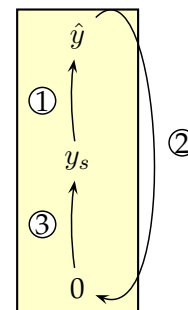
bytes have to be allocated.

The $+1$ results from the fact that even if only address 0 is accessed exactly 1 Byte is used. In general: If addresses $\{0, 1, \dots, a\}$ are accessed, at least $a + 1$ bytes must be available.

The amount of memory that can be allocated with `kmalloc` is depending on the architecture and the kernel configuration. However a common value is 128 KB.

To obtain more consecutive memory `_get_free_pages` has to be used. It allows the allocation of $2^{10}/2^{11}$ pages - architecture dependant.

The cache flooder must not work on addresses starting at $0.0.0$ but at $x_s.y_s.z_s$, because the x and z part are not needed for referencing the caches and y will run from y_s to \hat{y} , then "wrap around" to 0 and increase until $y_s - 1$. Therefore it is possible to allocate a chunk of memory and do any memory references relative to that address.



4.2 Cache flooding II

4.2.1 The algorithm

After making some test with the flooder algorithm presented in section 4.1.2 and working out the way a two level cache architecture works, I realized that the basic condition for the double purge configuration is:

1. the data in L1 is not in L2 and
2. both caches contain modified data, so the lines can not simply be overwritten

and that this can be achieved more easily and with less restrictions.

Condition 1 can not be achieved globally, but partially:

If $W_1 - 1$ ways of L1 are continuously reloaded before a new way is loaded, as often as it needs to fill L2 and to evict the $W_1 - 1$ old ways from it.

To fulfill condition 2 the first level cache must use a write-back caching strategy because otherwise modified data is always written through to L2 and can therefore simply be overwritten in L1, if free lines are needed.

As a matter of this the cache flooder can not be implemented on the Pentium IV processor because it uses a write-through L1 cache!

My algorithm is structured as follows:

1. fill L1, through reading from *and* writing to addresses $\mathbf{A}_0 = A_1, \dots, D_1$
 - read addresses to have them in L1 even if it uses a write-no-allocate policy
 - write to \mathbf{A}_0 to *modify* the data

```
read_write(A1)
  ⋮
read_write(D1)
```

2. reload $W_1 - 1$ ways A_1, B_1, C_1

```
read(A1)
read(B1)
read(C1)
```

3. load *one* new way E_1, \dots

```
read_write(E1/...)
```

4. repeat steps 2, 3 as often as needed to fill L2 and to purge the "old" addresses A_1, B_1, C_1 out of L2

as often as needed means: $R = R_s \cdot R_w - 1 = 2^r - 1$ times $r = r_s + r_w$

R_s the number of times one L1-way fits into one L2-way

$$r_s = a_2 - a_1$$

R_w the number of L2-ways

$$r_w = w_2$$

One reload of A_1, B_1, C_1 together with one loading of a new L1-way, can be left out: The addresses D_1 do not have to be evicted from L2 because they do not reside in L1 anymore (\rightarrow figure 29, the **colored** area in L2).

When implementing my flooder algorithm, I tried to use as less variables as possible, to have as few memory references as possible. Therefore I declared the cache parameters as compiler defines. Table 8 shows the connections.

Table 8: cache parameters - names and their meaning

variable	#define	meaning
l	LINELEN	order of line length
s_1	SIZE_L1	order of size of L1
s_2	SIZE_L2	order of size of L2
w_1	ASSOC_L1	order of associativity of L1
w_2	ASSOC_L2	order of associativity of L2
a_1	AWIDTH_L1	order of address width of L1
a_2	AWIDTH_L2	order of address width of L2

As explained before (\rightarrow sec. 2.4, p. 7), the address width can be calculated of the other parameters:

$$a_i = s_i - w_i - l \quad \Rightarrow \quad \text{AWIDTH_L}i = \text{SIZE_L}i - \text{ASSOC_L}i - \text{LINELEN} \quad i = \{1, 2\}$$

The number of repetitions is consequently:

$$\begin{aligned} R &= 2^r - 1 = 2^{a_2 - a_1 + w_2} - 1 \\ &= 1 \ll (\text{AWIDTH_L}2 - \text{AWIDTH_L}1 + \text{ASSOC_L}2) - 1 \end{aligned}$$

Drawings 27 to 29 illustrate the steps of flooding the CPU caches of a Pentium II.

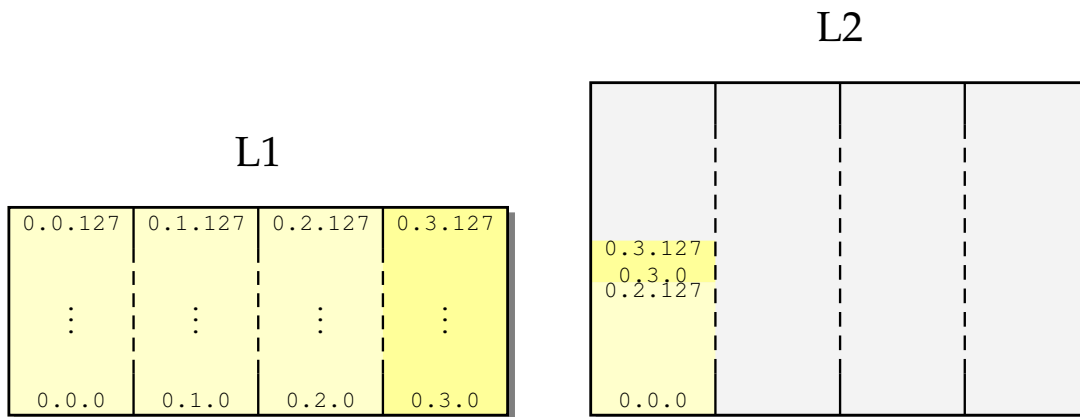


Figure 27: → step 1: fill L1

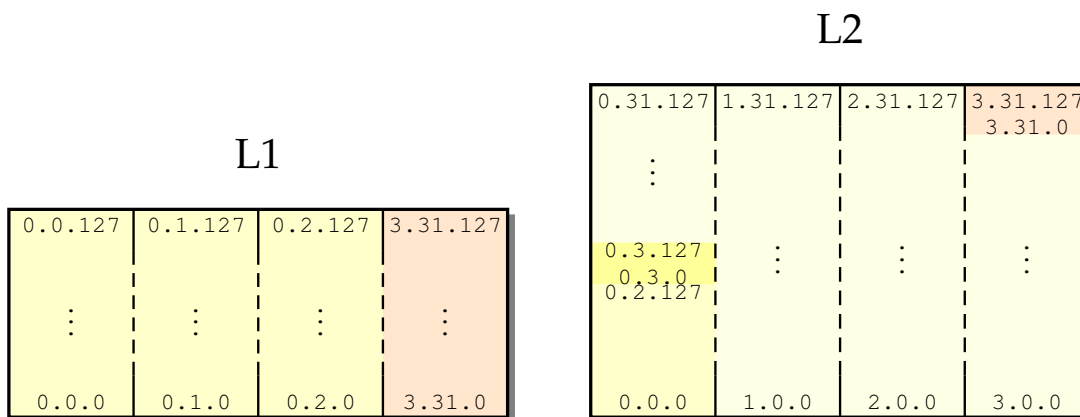
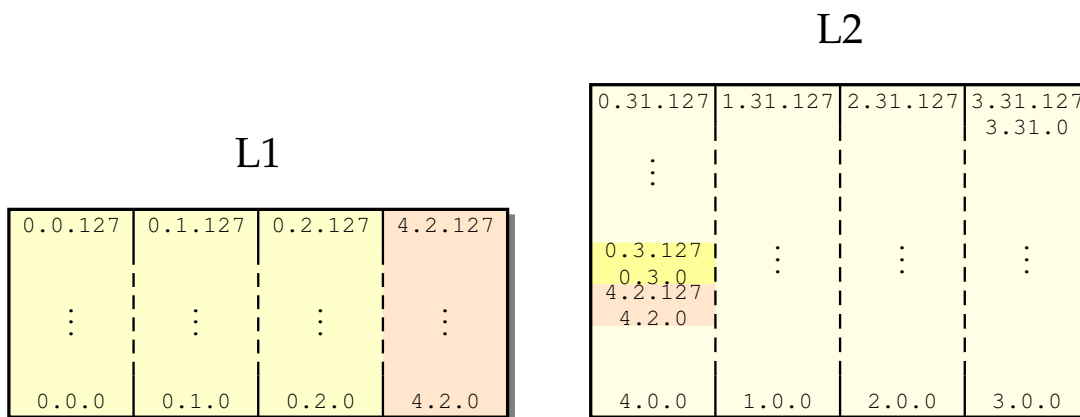
Figure 28: → steps 2, 3: using *one* L1 way to load new addresses

Figure 29: the caches in double purge configuration

Figure 27 refers to the first step of the flooding algorithm. L1 is filled with addresses A_1, \dots, D_1 . The fourth way of L1 will now be used to load new addresses.

Figure 28 shows the caches after steps 2, 3 have been executed until the whole second level cache is filled. Because of the reloading, the oldest addresses A_1, B_1, C_1 still reside in L1! The fourth (lightred colored) L1-way - which is used for loading the new addresses - holds addresses that are in L1 *and* L2.

Figure 29 presents both caches in double purge configuration:

Except the one highlighted L1-way no addresses stored in L1, reside in L2 too.

After flooding, the L1-way which was used for loading the new addresses, is the most recently used one and should therefore be the last one that is overwritten (LRU, pLRU_t). As many addresses as fit into the "first" $W_1 - 1$ cache ways can be accessed before the cache is not in the double purge configuration anymore.

The advantages of my algorithm are:

- + any addresses can be used, there are no restricted address ranges (as $y = 1$)
- + no need to do address calculations ($\rightarrow x, y, z$ parts), the caches are flooded consecutively, line by line
- + simple to understand, therefore easy to adapt
- + works with write-allocate *and* write-no-allocate policies

4.2.2 Conditions on the cache architecture

To implement the described algorithm the following conditions have to be met:

- * two level cache hierarchy
- * a first level cache that uses a write-back strategy
- * a pLRU_t with tree based filling policy applied to L1

Whereas the algorithm is easy to understand and to modify, it can be easily adapted to other replacement policies. Only the sequence of cache ways on reloading has to be changed.

4.3 Filling vs. Flooding

To achieve the worst case, caches have to be in double purge configuration, which is quite uncommon to emerge under "normal" circumstances. However filled caches is a configuration that is more probable and leads to at least 2 main memory accesses. It is sufficient that a process fills the second level cache, so that the data in L1 is stored in L2 too. If another process accesses data not yet cached (\rightarrow read from main mem.) an entry in L2 has to be freed (\rightarrow write to main mem.) and an entry has to be purged from L1 into L2.

Figure 30 shows the CPU caches of a Pentium II, filled through the following instructions:

```
__asm__ __volatile__("wbinvd");
for(i = 0; i < 1UL<<SIZE_L2; i += 1UL<<LINELEN)
    mem[i] += 1;
```

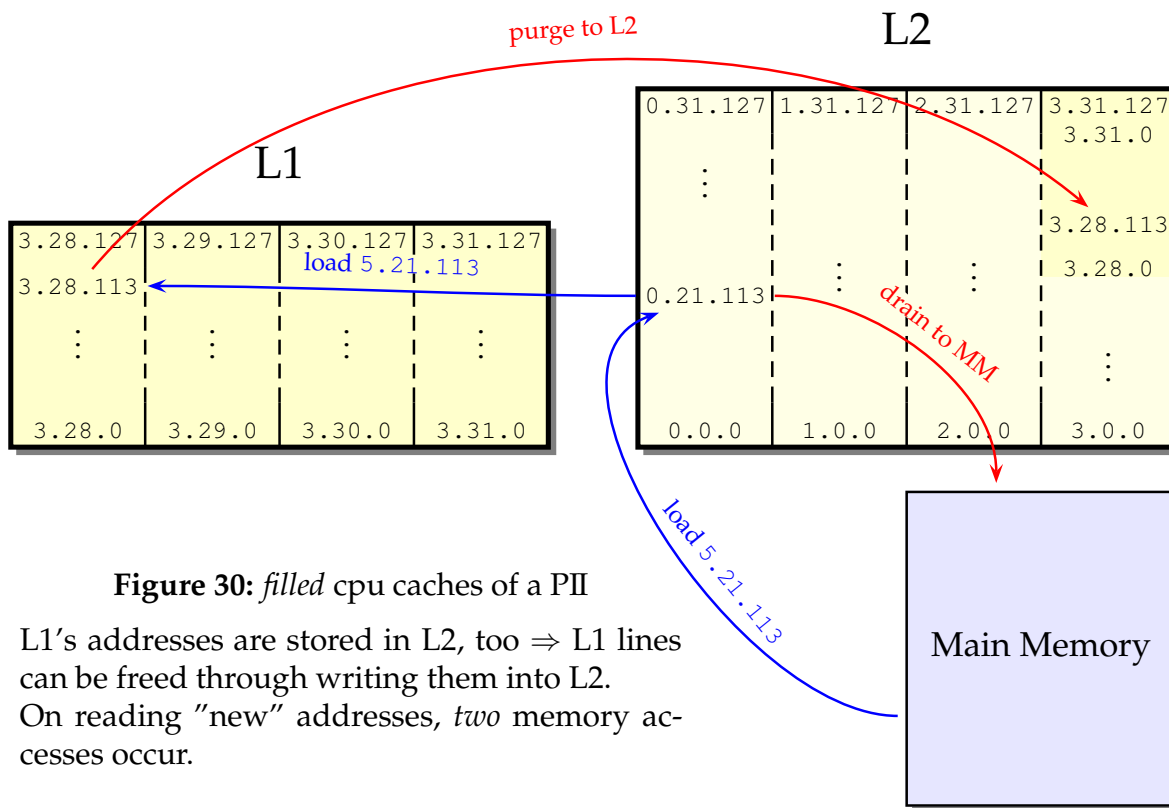
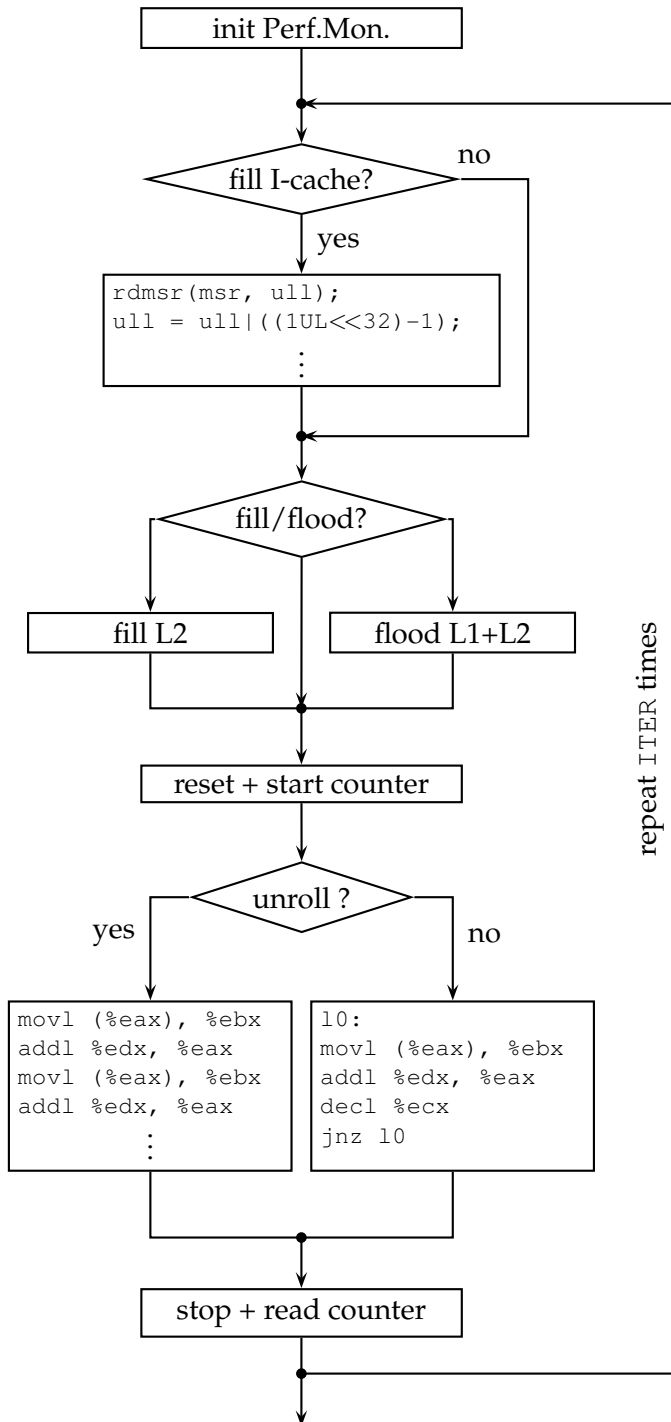


Figure 30: filled cpu caches of a PII

L1's addresses are stored in L2, too \Rightarrow L1 lines can be freed through writing them into L2. On reading "new" addresses, *two* memory accesses occur.



repeat ITER times

When running benchmarks on the flooded/filled CPU caches a memory block of 100 cache lines was read.

To fill the instruction cache a chunk of instructions - 17 548 B of machine code, more than 12 K μ ops - can be executed. As presented in figure 31 a MSR (namely the TSC) is read and the value is somehow modified but not used for any purpose.

Furthermore it is possible to choose whether the caches are flooded, filled or kept untouched and if the memory access is coded as loop or unrolled.

Figure 31: structure of the cache floader test

As pointed out above (fig. 30), filled caches require *two* main memory accesses per read of a new cache line, whereas flooded caches require *three* (\rightarrow fig. 23). When reading 100 cache lines then almost none, 200 or 300 memory accesses should be counted depending on whether the caches were "clean", filled or flooded.

5 Results

5.1 Branch prediction

5.1.1 BHR - Branch History Register

Tables 9 and 10 show the results of the BHR-benchmark, tables 11, 12 those of the local-global test.

The *mod* value is the modulo parameter that refers to the pattern length. The other columns present the misprediction ratio (MPR, in %) for 1 M iterations.

Table 9: BHR benchmark - P6 family

<i>mod</i>	MPR		
	PII	PIII	
3	0.053	0.021	
4	0.051	0.021	
5	0.050	0.021	
6	8.366	8.347	⇒ 4 bit local/ 8 bit global
7	7.176	7.157	
8	6.283	6.263	

Table 10: BHR benchmark - Netburst

<i>mod</i>	MPR	
	PIV	
7	0.005	
8	0.005	
9	0.005	
10	5.006	⇒ 8 bit local/ 16 bit global
11	4.551	
12	4.171	

Table 11: local history component
-P6 family-

<i>mod</i>	MPR		
	PII	PIII	
3	0.069	0.028	
4	0.070	0.028	
5	0.069	0.028	
6	22.27	22.24	⇒ $mod_2^* = mod_1^*$
7	19.10	19.07	
8	16.71	16.69	

Table 12: global history component
-Netburst-

<i>mod</i>	MPR	
	PIV	
5	0.006	
6	0.006	
7	5.908	⇒ $mod_2^* \neq mod_1^*$
8	4.173	
9	3.709	
10	3.340	

In both tests the MPR reaches its maximum at a parameter $mod_1^* = mod_2^* = 6$ wherefore the P6 family utilizes **local** branch history registers with a width of 4 bit.

With a single "spy" branch the MPR raises at $mod_1^* = 10$ but with two of these branches it already reaches its highest value at $mod_2^* = 7$. Therefore the conclusion must be that the Pentium IV microprocessor uses a **global** BHR with 16 history bits.

5.1.2 BTB - Branch Target Buffer

The following tables present the results of the BTB benchmark. The analysis was started at the smallest possible distance $D = 2$ and depending on the MPR (in %), the distance D and/or the number of branches B were varied.

The cycle count is an average on the number of iterations, kept constant at 100 k.

The last row of each table shows the results from a microbenchmark that uses the *same* instructions, but *backward* branches that are statically *correct* predicted.

Two cases have to be considered: the first uses “warm” caches, that means the piece of code containing the branches is executed twice and only the second time is used for measuring. In the second case, the first level instruction cache is filled through executing more than 16 KB (or more than 12 Kμops on the PIV) code and the unified L2 cache is filled as described when talking about cache flooding (p. 38).

Results on the PII/III

The “warming up” and filling of caches leads to really stable results. When repeating the experiment they only differ slightly.

But leaving out the statically predicted case (last row), there is *no* difference in “warming up” or filling!

After executing the conditional code for the first time it should be cached in the L1 I-Cache. Whereas the filling tries to achieve the opposite effect and because of the filled second level cache, instructions even have to be fetched from main memory - that means there really should be a difference.

Table 13: BTB microbenchmark results
PII, “warm” caches

#	<i>D</i>	<i>B</i>	MPR	cycles
1	2	32	72.469	240
2	4	32	0.005	66
3	4	256	0.070	584
4	4	512	0.509	1 185
5	4	1024	99.589	9 569
6	8	512	0.799	1 326
7	16	512	0.832	1 075
8	32	512	99.423	4 729
9	32	512	0.078	4 255

Table 14: BTB microbenchmark results
PII, filled caches

#	<i>D</i>	<i>B</i>	MPR	cycles
1	2	32	72.468	240
2	4	32	0.006	66
3	4	256	0.072	584
4	4	512	0.511	1 185
5	4	1024	99.592	9 570
6	8	512	0.799	1 326
7	16	512	0.851	1 075
8	32	512	99.422	4 781
9	32	512	0.061	3 295

Table 15: BTB microbenchmark results
PIII, "warm" caches

#	D	B	MPR	cycles
1	2	32	72.624	239
2	4	32	0.004	66
3	4	256	0.031	581
4	4	512	0.436	1 177
5	4	1024	99.778	9 523
6	8	512	0.667	1 316
7	16	512	2.246	1 129
8	32	512	99.658	4 731
9	32	512	0.026	4 113

Table 16: BTB microbenchmark results
PIII, filled caches

#	D	B	MPR	cycles
1	2	32	72.624	239
2	4	32	0.005	66
3	4	256	0.032	581
4	4	512	0.439	1 177
5	4	1024	99.779	9 523
6	8	512	0.668	1 316
7	16	512	3.860	1 196
8	32	512	99.657	4 775
9	32	512	0.020	3 286

As expected, tables 13/14 and 15/16 stress that both processors share the same architecture - the measured results are almost the same:

The distance of $D = 2$ is too small to map to different entries, wherefore the latter branches overwrite the previous ones and nearly none of the them can be predicted.

At a distance of $D = 4$ the MPR stays low until the number of conditional branches reaches 1024. That leads to the conclusion that the P6 family utilizes a BTB with 512 entries.

Therefore B is kept constant at $B = 512$ and only D is increased.

The low MPR of rows 4, 6, 7 shows that the 3 distances $D = \{4, 8, 16\}$ allow all of the 512 branch addresses to be saved and that $D^* = 16$ is the highest "fitting" distance.

The associativity of the BTB can be calculated as follows:

$$W = 2^{F-1} = 2^{3-1} = 4$$

and the least significant l bits, not used for indexing the buffer:

$$l = \log_2(D^*) = \log_2(16) = 4$$

To index all $S = \frac{N_{BTB}}{W}$ sets

$$a_1 = \log_2(S) = \log_2\left(\frac{512}{4}\right) = 7$$

bits are necessary.

As row 8 shows, 512 branches at a distance $D = 32$ do not fit into the buffer and are mispredicted, whereas the same number of branches but at a smaller distance can be stored (e.g. 8, 16) and correctly predicted.

When executing the test with the backward pointing branches almost no mispredictions occur, yet the time needed is almost the same as if all 512 branches were mispredicted!

Curiously the filling of caches speeds up execution: it takes 900 cycles less.

With "warm" caches the penalty per statically mispredicted branch is about 1 cycle, with filled caches about 2 cycles. But [H⁺01, p. 3] declares that the P6 family needs 10 cycles "to recover from a branch that went a different direction than the early fetch hardware predicted at the beginning of the pipeline."

If 10 cycles were needed, there should be a difference of about 5000 cycles between 512 statically predicted and mispredicted branches, therefore the measured ≈ 4700 cycles can not be

the worst case.

The reason that dynamic prediction is faster than static one is that the CPU first tries to find the branch address in the BTB before a static prediction is done. So even in the case of statically correct predicted branches (last row), the buffer lookups occur. Therefore the best case refers to *dynamically* correct predicted branches (e.g. row 4/6/7) and the worst case to *statically* mispredicted ones (e.g. row 8).

A comparison results in a $\left[\frac{4700 - 1100}{512} \right] \approx 7$ cycles per branch longer execution time for the *same* instructions!

Results on the PIV

Table 17: BTB microbenchmark results
PIV, "warm"

#	D	B	MPR	cycles
1	2	32	72.682	586
2	4	32	0.000	34
3	4	256	0.000	261
4	4	512	0.000	520
5	4	1024	0.000	1 039
6	4	2048	0.007	2 121
7	4	4096	0.046	29 224
8	4	8192	99.860	390 655
9	8	4096	0.107	32 459
10	16	4096	0.100	32 053
11	32	4096	99.827	189 100
12	32	4096	0.013	90 452

Table 18: BTB microbenchmark results
PIV, filled caches

#	D	B	MPR	cycles
1	2	32	74.952	616
2	4	32	0.003	90
3	4	256	0.003	1 008
4	4	512	0.006	1 574
5	4	1024	0.445	1 715
6	4	2048	0.012	2 131
7	4	4096	0.047	29 050
8	4	8192	99.859	390 653
9	8	4096	0.095	32 619
10	16	4096	0.099	32 035
11	32	4096	99.828	189 108
12	32	4096	0.013	90 449

Applying the same formulas on the data in table 17/18 leads to the conclusion that

- * the BTB of the Netburst architecture has 4096 entries,
- * $W = 2^{3-1} = 4$ ways,
- * $l = \log_2(16) = 4$
 $\Rightarrow 2^4 = 16$ consecutive addresses map to the same set
- * $a_1 = \log_2\left(\frac{4096}{4}\right) = 10$ bits are used for indexing the buffer

Here again, the filling of caches does not lead to the wanted effect, only branches at small distances are influenced and it takes longer to execute them.

The last two rows are both statically predicted - the MPR of row 11 shows that all 4096 branches are mispredicted (because they are pointing *forward*), whereas the *backward* pointing branches (row 12) are all correct predicted.

The cycle difference between the 4096 statically predicted (row 12) and statically mispredicted

branches (row 11) is ≈ 24 cycles per branch - a number directly linked with the 20 stage misprediction pipeline of the PIV.

When comparing the best case (4096 *dynamically* correct predicted branches) with the worst (4096 *statically* mispredicted conditional jumps) the result is amazing:

—————→ 39 cycles penalty per mispredicted branch! ←————

To verify that our assumptions on the branch architecture are correct, the benchmark can be used to measure the CPU cycles needed to execute $B = \{B_1, B_2, \dots\}$ branches at a constant distance $D = \text{const.}$

Figure 32 shows the obtained results for $B = \{32, 64, 128, 256, 512\}$ at a distance $D = 4$ bytes, for "warmed up" caches. The cycle count is the average of 100 k iterations that clearly shows the linear dependence from the number of jump instructions.

The graphs show as well, that the Netburst architecture needs not even half as many cycles as the P6 architecture!

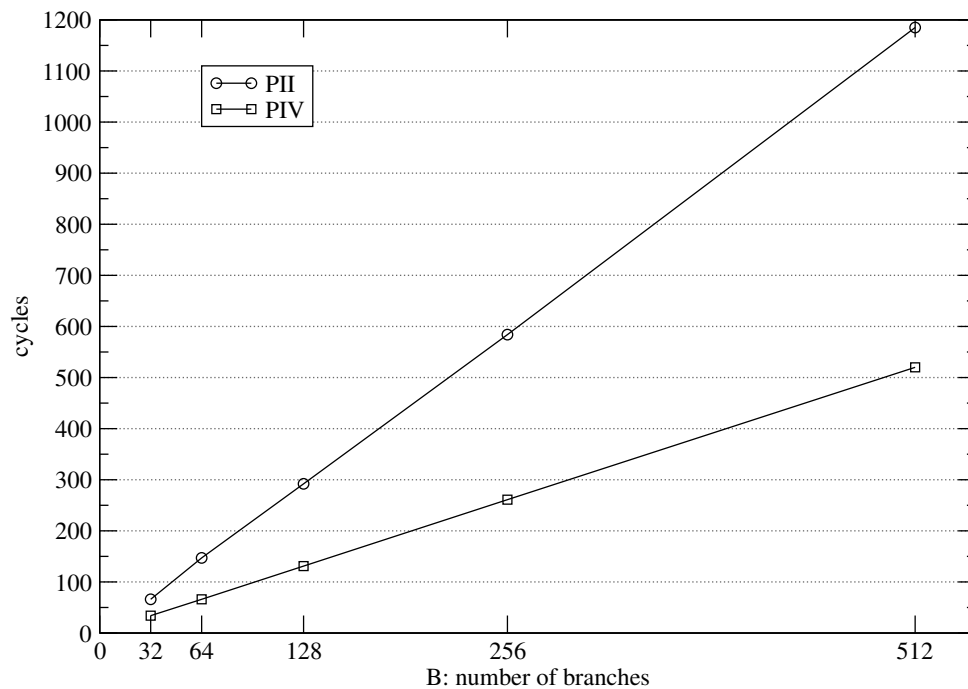


Figure 32: the number of cycles increases linear with the number of branches

5.2 Caching

5.2.1 Adjacent cache line prefetch - PIV

Figure 33 shows that it does not matter whether the first or the second line in a sector is read, the whole sector is fetched from memory, anyway – if 2-line-prefetch is enabled. Otherwise only the accessed line is loaded.

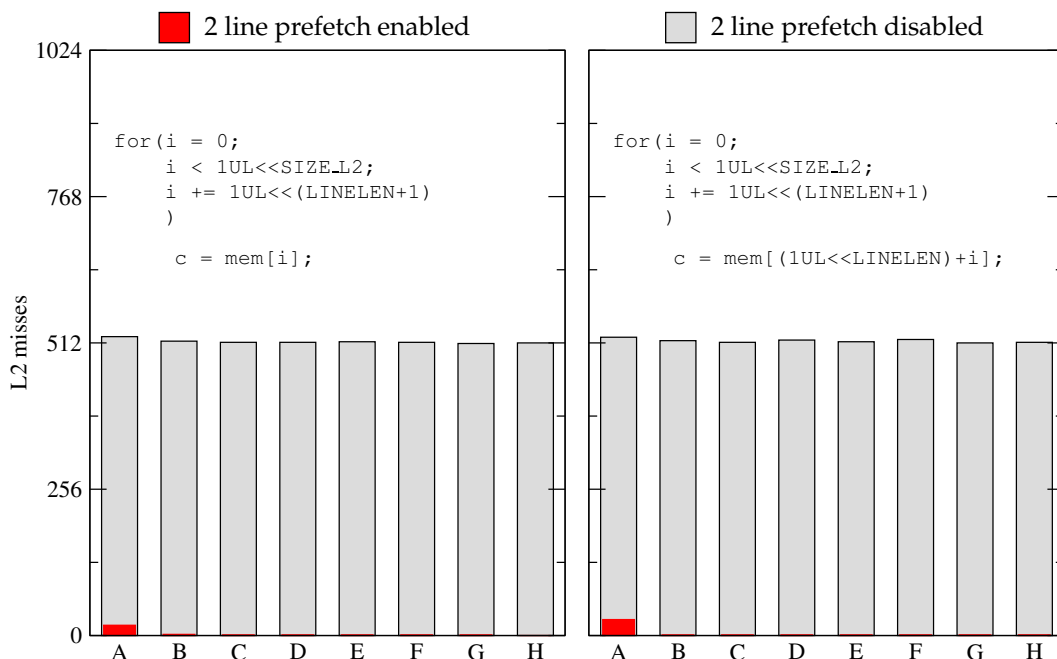


Figure 33: enable vs. disable adjacent cache line prefetch

5.2.2 L1 - caching strategy

write-allocate/write-no-allocate

Results on the PII/III

512 lines can be stored in L1 and $\bar{n}_{\text{miss}} = 16$ misses were encountered. This miss ratio of 3.125 % makes it obvious that a write-allocate strategy is used.

Results on the PIV

128 lines can be stored in L1 and $\bar{n}_{\text{miss}} = 22$ misses were encountered. This time the miss ratio of 17.188 % is not as small as on the PII/III, yet it is too small for a no-write-allocate strategy. So again, write-allocation is used.

write-through/write-back

Results on the PII/III

The performance monitoring of the P6 family allows to count the cycles the L2 data bus is busy and the number of L2 address strobes. CPU cycles were counted with the TSC (Time Stamp Counter).

The outcome of step 3 (modified cache content, p. 21) and 5 (unmodified cache content → w-through reference, p. 21) was:

Table 19: results of the caching strategy benchmark - PII

	cycles	cycles L2 d- bus busy	L2 strokes	addr
probed	20 726	12 336	2 057	
w-through	10 704	8 398	1 063	

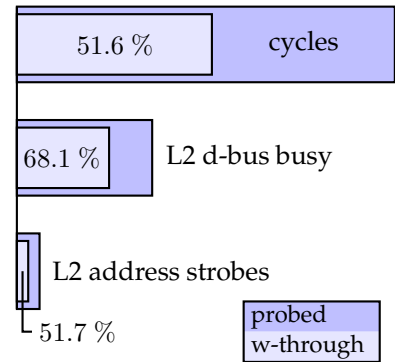


Figure 34: visualization of Tab. 19

Diagram 34 shows clearly that the modified data in the L1 cache has to be *written back* to L2 first, before it can be replaced: twice as many cycles and L2 address strokes are necessary.

It is obvious that the first level cache of the P6 family is a write-back cache.

This fact is in conclusion with Intel's article [Hay] which describes differences between the Netburst architecture and "Earlier Pentium Family Processors" (Pentium Pro, II, III).

Accordingly, [Seb01] is wrong in assigning a write-through policy to the Pentium III.

Results on the PIV

The performance monitoring of the Pentium IV does not allow to count the same events as on earlier processors, that is why I decided to take the front side bus (FSB) reads and writes as indicator for L2 cache accesses. And even if these counts were not representative the results of the PII showed that the CPU cycles are meaningful enough to differentiate between write-through and write-back.

Table 20: results of the caching strategy benchmark - PIV

	cycles	FSB reads	FSB writes
probed	16 420	128	0
write-through	16 677	129	0

It arises that there is almost no difference whether the cache content of L1 has been modified or not and the conclusion must be that the Netburst architecture uses a L1 cache with a write-through policy.

This is in line with information in [Hay], [H⁺01] and [Int].

5.2.3 Replacement strategy

Results of the cache replacement analyse on the PII/III

Figure 35 consists of two graphs. The upper one refers to the first level, the lower one to the second level cache. Each of these graphs consists of 8 subgraphs where each results out of different reloading combination. The titles of these subgraphs declare which ways have been reloaded. $A_i B_i$ means way A_i has been read first and afterwards way B_i . The abscissa shows the letter of the corresponding cache way. Although these letters are without an index it is

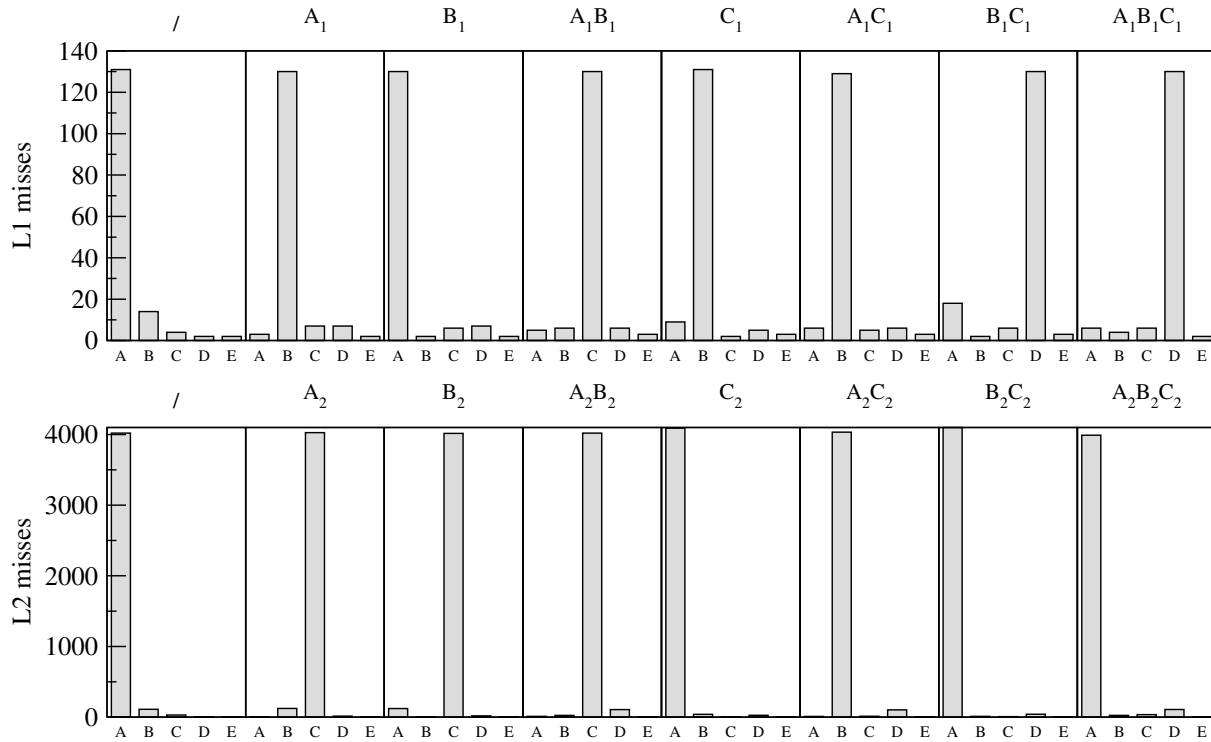


Figure 35: reload addresses to influence pLRU history- PII

clear that those of the upper graph belong to L1 and A stands for A_1 . Those of the lower graph represent the ways of L2, here A stands for A_2 .

The titles of the subgraphs of figure 35 can be used to index the first column of table 6. Columns 2 and 3 (PII has 4-way caches, L1 and L2 too) can then be compared with the graphs: the way ($\{A, \dots, D\}$) with the most misses (either 128 for L1 or 4049 for L2) should correspond to one of column 2 or 3, tree based or sequential fill.

Table 21 summarizes only the ways with the most L1 misses:

Table 21: results of the replacement policy benchmark - PII, L1 cache

reloaded way(s)	replaced way	tree based fill
/	A	A
A	B	B
B	A	A
A, B	C	C
C	B	B
A, C	B	B
B, C	D	D
A, B, C	D	D

The replaced ways correspond exactly to those replaced if a pLRU policy with tree based filling would be used.

To explore the way lines are replaced in the second level cache, the lower graph of figure 35 has to be compared to columns 2, 3 of tabular 6.

This time the free-entry based method matches the measured pattern:

Table 22: results of the replacement policy benchmark - PII, L2 cache

reloaded way(s)	replaced way	free-entry fill
/	A	A
A	C	C
B	C	C
A, B	C	C
C	A	A
A, C	B	B
B, C	A	A
A, B, C	A	A

The second level cache of the Pentium II uses a pLRU strategy with free-entry filling.

At the moment of writing this thesis, I stumbled over [Int97] which confirms my explorations:

“The caches employ a write-back mechanism and a pseudo-LRU replacement algorithm.”

The fact that there is so many different and so less detailed information (the type of pLRU policy is not mentioned!) available, shows that my microbenchmarks *are* of use.

Results on the PIV

Counting L1 misses on the PIV is not that precise as on the earlier Pentium processors because the first level cache has only 32 sets and because the bus and CPU frequencies are much higher. When counting that little events even some misscounts are quite severe.

These facts have to be considered when analysing the L1 cache, as done in the upper most graph of figure 36.

The comparison of the cache ways with the highest miss counts with those in tabular 6 gives the following result:

Table 23: results of the replacement policy benchmark - PIV, L1 cache

reloaded way(s)	replaced way	tree based fill
/	A	A
A	B	B
B	A	A
A, B	C	C
C	A	B
A, C	B	B
B, C	A	D
A, B, C	D	D

Except the red marked ones, the replaced ways correspond to those being replaced when a tree based fill method is used. Taking the explained inaccuracies into account it seems that the first level cache of the Pentium IV uses a pLRU strategy with tree based filling.

The L2 cache utilizes 1024 sets, wherefore the event-misscounts do not play any role. However there are still two cases that are really confusing because either two L2 ways are purged or none complete way is replaced, but the misses are distributed between almost all cache ways:

- * The reloading of two ways with the addresses A, B and the following reading of way I leads to the nearly complete eviction of the ways C and D .
It is important to note that not half the addresses C and half the addresses D are purged from L2 to obtain *one* free way, but *both* address ranges C, D are freed!
- * The cache behaves the other way around when reloading A, B, C because this time almost every address range A, \dots, H that has been cached in L2, experiences an approximately equal number of evictions, to store I .

The highlighted rows signal that there are differences in the address ranges that should be replaced (column 3) and those that *are* replaced (second column). Although not identical, it seems that the L2 cache of the PIV uses a pLRU policy with a tree based filling, too.

Table 24: results of the replacement policy benchmark - PIV, L2 cache

reloaded way(s)	replaced way	tree based fill
/	A	A
A	B	B
B	A	A
A, B	C, D	C
C	B	B
A, C	B	B
B, C	D	D
A, B, C	?	D

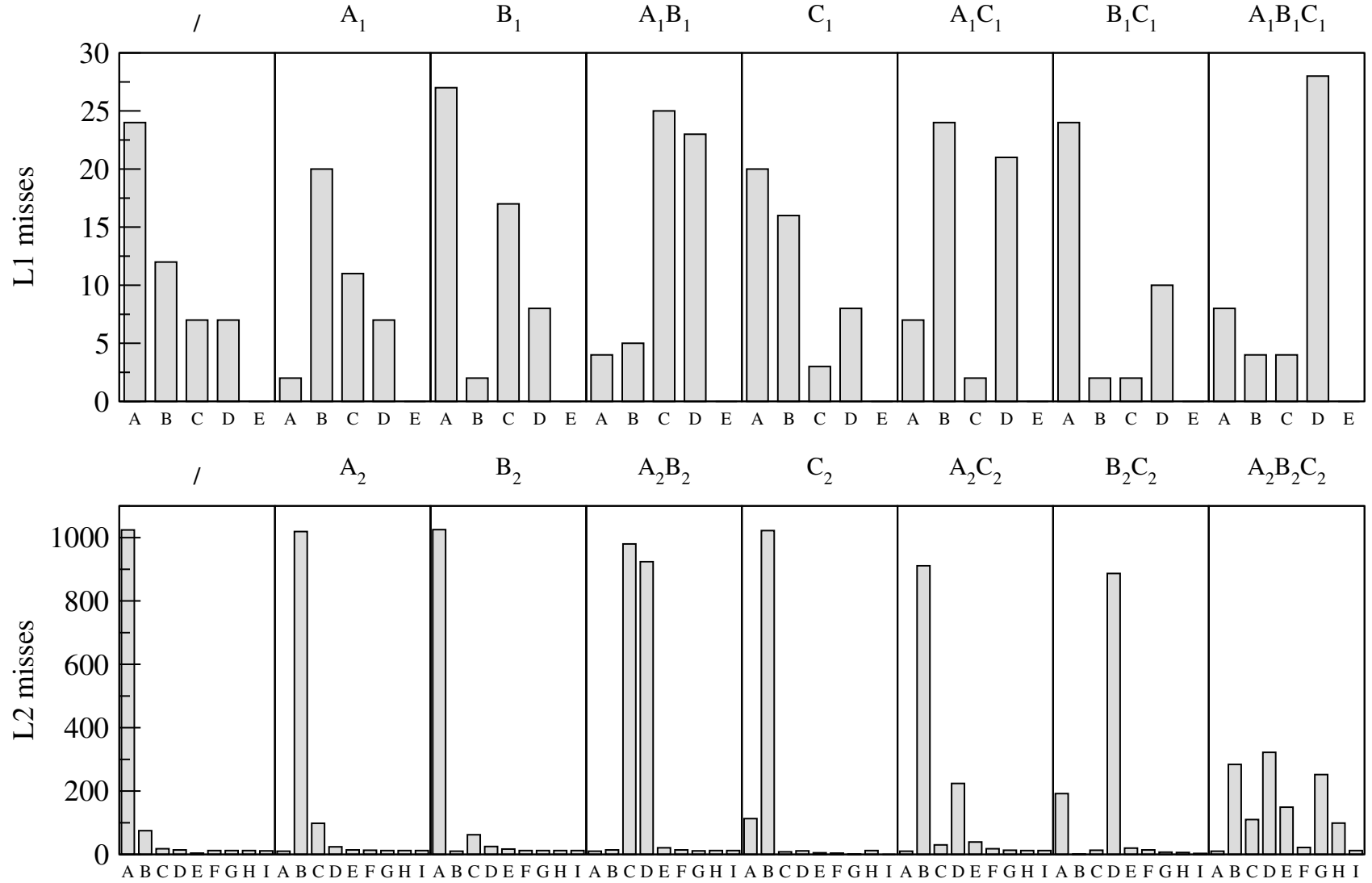


Figure 36: reload addresses to influence pLRU history - PIV

5.3 Cache flooding

Results on the PII

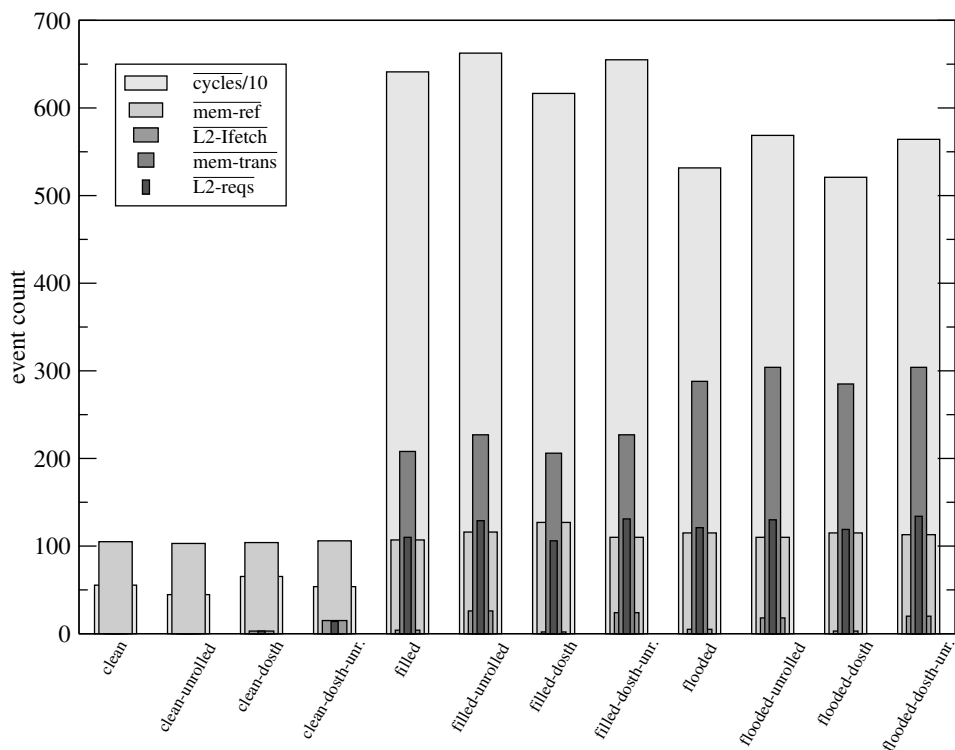


Figure 37: PII: comparison of "clean", filled and flooded caches

Diagram 37 can be horizontally divided into 3 sections:

The first contains the leftmost four bars and refers to clean caches that are neither filled nor flooded. Section number two belongs to filled caches and contains the middle 4 bars. The rightmost 4 bars form the third section which stands for flooded caches in double purge configuration.

Each section consists of 4 bars. The first bar presents the results with a clean first level I-cache and the memory accessed in a loop. The second bar of a section stands for an unrolled access. The third and fourth bar represent a filled I-cache (*dosth* → do sth.: "useless" instructions to fill the L1 I-cache → fig. 31). The third bar refers to a memory access in a loop and the fourth to the unrolled code.

Because 100 cache lines were read, the memory reference count (*mem-ref*: loads from / stores to any memory type) is about 100 in any configuration.

With clean caches no memory transactions (*mem-trans*: completed memory transactions) occur - any data can be fetched from caches, at least from second level.

When the L1 instruction cache is filled (bar 3+4 of fig.37) the instructions needed to read the data, have to be fetched from L2 (*L2-Ifetch*) and of course the unrolled access (4th bar) needs more instructions than the one coded as loop (3rd bar).

If caches are kept untouched, the unrolled memory access is the fastest. This is because any data and instruction is cached and no comparison of a counter with a final value has to be done as it has in a loop.

When looking at filled and flooded caches (bars 5 to 12) it is obvious that the access with filled caches takes more time than with flooded ones!

If the memory transaction count would not proof that memory has been accessed 300 times in the flooded case and "only" 200 times in the filled, it could be doubted that the cache flooding algorithm (presented in sec. 4.2.1 on p. 33) works, but because 300 memory accesses result out of 100 memory references, caches really have been in double purge configuration.

With "dirty" caches (filled or flooded) the unrolled reading takes longer than that done in a loop because more instructions have to be fetched from main memory.

Results on the PIII

Although belonging to the same family of microprocessors, the PIII behaves different than the PII.

This time the access of flooded caches really takes longer as accessing only filled ones - as figure 38 proofs.

As well, the whole test needs almost twice as long as on the PII (\rightarrow cycles/20).

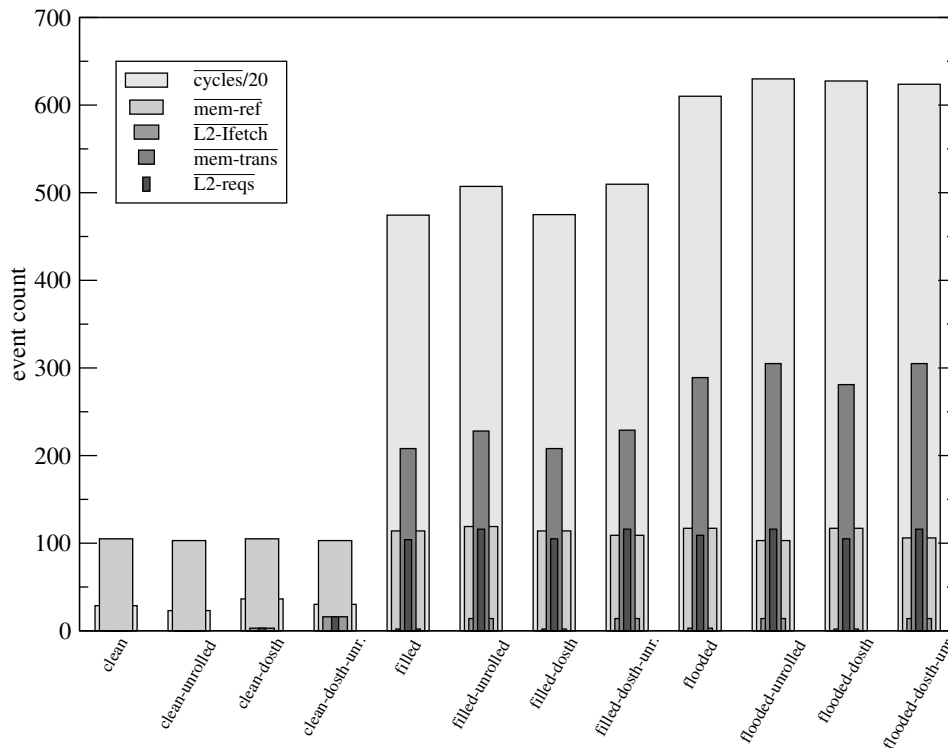


Figure 38: PIII: comparison of "clean", filled and flooded caches

On the PIII, accessing "clean" caches is about 27 times faster than accessing flooded ones!

As can be seen in figure 38, almost 450 cycles are necessary to read the data that is cached in L1, whereas $\approx 12\,500$ cycles are needed to read the data if caches have been in double purge configuration.

Results on the PIV

The following abbreviations are used in the next 2 diagrams:

loads \longrightarrow memory loads
 IOQ-alloc. \longrightarrow read + write bus transactions

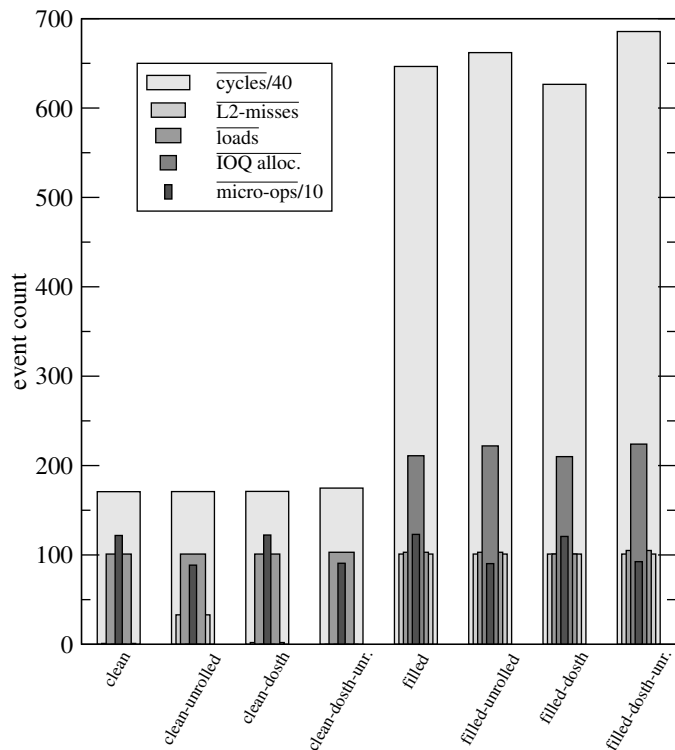


Figure 39: PIV: comparison of “clean” and filled caches – prefetching disabled

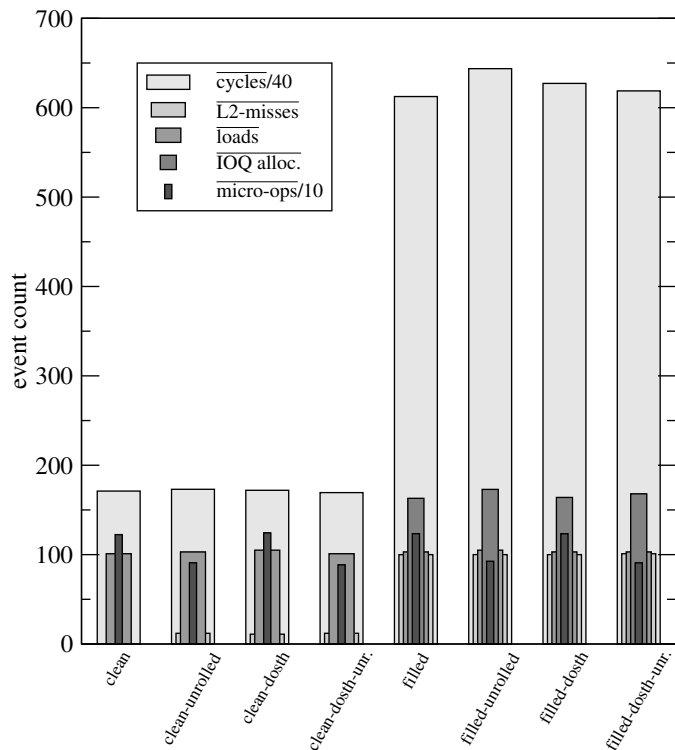


Figure 40: PIV: comparison of “clean” and filled caches – prefetching enabled

As pointed out in section 4.2.1 the Netburst microarchitecture with its write-through L1 cache does not allow to get the caches into double purge configuration. Therefore the comparison (fig. 39, 40) only covers clean and filled caches, but with prefetching enabled and disabled. Here prefetching is to be understood as the hardware prefetcher *and* the adjacent cache line filling.

Even at first sight it is obvious that reading the 100 cache lines takes much longer on the PIV than on the PII / PIII:

No matter if prefetching is enabled or not, reading the data with clean caches takes almost the same amount of time: 5500 cycles - not influenced by filling the Trace Cache and / or unrolling the loop – that is about 12 times slower than the access on the P6 architecture!

On the Netburst architecture the line size is twice as big, but the number of bits that can be read / written per clock cycle has been doubled, too (→ [Hay]).

If caches have been filled, the reading needs not many cycles less if prefetching is enabled, although the number of IOQ allocations is about 45 smaller. Even the L2 miss count is not reduced by the prefetching, although every second read should get its data from L2 because the whole sector (= 2 lines) has been fetched.

6 Conclusions

This work deals with the analysis of CPU cache and branch prediction architecture. The most common techniques and strategies have been explained and based on this understanding, algorithms have been derived that allow to explore a processors cache and branch prediction architecture.

With these tests it is possible to obtain information on the applied caching strategy, including the replacement policy. Among other things, the organization of the branch target buffer (BTB) can be examined without any advance information, so that the search of this data can be omitted.

Based on this knowledge, best and worst case examples can be constructed.

In that context I found out that the cache flooder, as presented in [LH], can be improved and I described an alternative cache flooding method that is easy to understand and to implement and with the knowledge gained through the results of the benchmarks it should be no big effort to adapt it to get CPU caches on different platforms into "double purge" configuration.

All the tests / microbenchmarks have been executed on Pentium II, III and IV processors.

These are the results that have been gathered so far, with the help of the presented methods and algorithms:

The P6 family utilizes local branch history registers of 4 bit width and a branch target buffer with 128 sets.

The Netburst architecture has one global BHR that is able to store the last 16 outcomes of conditional jump instructions. The BTB has 4096 entries, that means 1024 sets. On both architectures the least significant 4 bits of a branch instructions address are not used to index the buffer and its associativity is 4. Therefore only distances of $2 < D \leq 32$ allow all entries to be used.

The tests showed that dynamic prediction is a lot faster than static one and that it is wise to "warm up" the caches (to which the BTB belongs, too), to stabilize and shorten execution time.

CPU caches of the examined microprocessors were identified as follows:

The first level cache of the PII/III uses a write-back, write-allocate mechanism with a pLRU_t replacement strategy with tree based filling. A pLRU_t algorithm with free entry filling is applied to the second level cache.

The PIV utilizes a write-through, write-allocate policy for its L1 cache. PLRU_t with tree based filling is the replacement strategy used in both cache levels.

This document described how to use the performance monitoring capabilities available on many processors, directly, without any additional libraries or kernel modifications. Together with the explained algorithms, that opens the possibility to modify the benchmarks to run on different architectures, so that more information can be gathered and compared.

Of course, there are still several aspects left unexplained:

For example why the filling of caches needs longer than flooding on the PII or why the filling of the I-Cache has that little effect in conjunction with the BTB benchmark.

It should be explained why the cache flooder only works as expected on the PIII and why cache accesses on this processor are half as fast as on the examined PII.

A Conceptual formulation

Technische Universität Chemnitz
Fakultät für Elektrotechnik und Informationstechnik
Professur Prozessautomatisierung

Aufgabenstellung für Diplomarbeit

Name, Vorname: John, Tobias geb. am 27.01.1981

Studiengang: Elektrotechnik

Studienrichtung: Automatisierungstechnik

Thema: **Instruction Timing Analysis for Linux/x86-based
Embedded and Desktop Systems**

(Ausführliche Aufgabenstellung siehe Rückseite)

Name des Betreuers: Dr. Baumgartl (Fakultät Informatik)

Betreuer außerhalb der TUC:

Tag der Ausgabe: 05.04.2005

Abgabetermin: 04.10.2005

Tag der Abgabe:


Prof. Dr.-Ing. habil. A. Farschtschi
Vorsitzender des Prüfungsausschusses


Prof. Dr.-Ing. P. Protzel
Verantwortlicher Hochschullehrer

Figure 41: copy of the 1st page

Zielstellung:

Echtzeitaspekte sind in wachsendem Maße auch in Standard-PC-Umgebungen relevant, gleichzeitig werden x86-basierte Prozessoren zunehmend in eingebetteten Systemen eingesetzt.

Für die Konstruktion von Echtzeitsystemen stehen Standardtechniken zur Verfügung, die jedoch die Effizienz der Systeme reduzieren. Häufig wird daher auf die Garantie bestimmter Systemparameter verzichtet und überdimensionierte Hardware genutzt.

Zielstellung der Arbeit ist es, quantitative Aussagen über das Zeitverhalten aktueller x86-basierter Prozessorarchitekturen unter dem Betriebssystem Linux zu erhalten.

Im Einzelnen sind dabei folgende Aspekte relevant:

- Vergleich der Typen Pentium 4 und AMD Elan SC 410
- Entwurf und Vermessung von Microbenchmarks, die einzelne Einheiten (ALU, FPU, MMX-Einheit, ISSE) der Prozessoren gezielt ansprechen
- Vermessung typischer Betriebssystemdienste (häufig genutzte Systemrufe) auf beiden Architekturen
- Vermessung vorgegebener Echtzeitapplikationen (z.B. aktuelle Multimedia-Codecs)

Die Arbeit wird gemeinsam durch die Professur Prozessautomatisierung der Fakultät für Elektrotechnik und Informationstechnik (Prof. Protzel) und die Juniorprofessur Echtzeitsysteme der Fakultät für Informatik (Dr. Baumgartl) betreut.

Figure 42: copy of the 2nd page

References

- [And] Jeff Andrews. Branch and loop reorganization to prevent mispredicts. www.intel.com.
- [H⁺01] Glenn Hinton et al. The microarchitecture of the pentium 4 processor. *Intel[®] Technology Journal*, 2001. Q1.
- [Hay] Bryan Hayes. Differences in optimizing for the pentium[®] processor vs. the pentium[®]III processor. Whitepaper 44010.
- [Int] Intel[®]. *IA-32 Intel[®] Architecture Optimization Reference Manual*.
- [Int97] Intel[®]. *Pentium II Processor Developer's Manual*, 1997. 243502-001/24333503.
- [Int04] Intel[®]. *IA-32 Intel[®] Architecture Software Developer's Manual*, volume 3, chapter 10. Intel[®] Corporation, 2004.
- [LH] Jork Löser and Hermann Härtig. Cache influence on worst case execution time of network stacks. Technical report, Dresden University of Technology.
- [Mil04] Milena Milenkovic. Performance evaluation of cache replacement policies. Technical report, University of Alabama in Huntsville, 2004.
- [MMK04] Milena Milenkovic, Aleksandar Milenkovic, and Jeffrey Kulick. Microbenchmarks for determining branch predictor organization. Technical report, University of Alabama in Huntsville, 2004.
- [san] w³.sandpile.org.
- [Sea00] Chris B. Sears. The elements of cache programming style. Technical report, Google Inc., 2000. Proceedings of the 4th Annual Linux Showcase & Conference Atlanta.
- [Seb01] Filip Sebek. Cache memories and real-time systems. Technical report, Mälardalen University, 2001. p. 24,25.
- [sou] w³.sourceforge.net.
- [Sto01] Jon Stokes. The Pentium 4 and the G4e: an Architectural Comparison. Technical report, 2001.