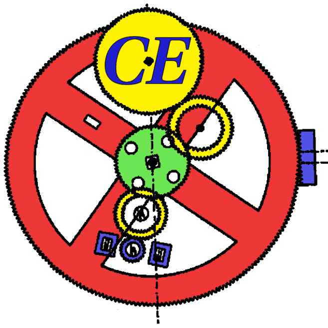


Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl

2003

MSc THESIS

Adapting the Cache Line Size to Reduce Off-Chip Memory Traffic

P.J. de Langen

CE-MS-2003-12

Abstract

Energy consumption is an increasingly important aspect of microprocessor design, especially for battery powered embedded systems. As processor speed increases and also the disparity in processor and memory speed, latency reduction techniques are often employed to reduce the time the processor is stalled, waiting for a memory access to complete. These techniques, as well as the increase in instruction issue rate itself, increase the amount of data transferred between processor and memory. Since off-chip communication requires a significant amount of power, power reduction can be achieved by reducing the number of transferred bytes. In this thesis, we try to determine how much data traffic is actually 'wasted' by conventional direct-mapped caches. Furthermore, we will try to derive which parameters of the direct-mapped cache are most influential on the amount of produced traffic. An effort is made to reduce the amount of unnecessary transferred data traffic between the processor and the off-chip memory system. This is done by using techniques that attempt to predict the amount of temporal/spatial locality exhibited by instructions, based on their previous behavior. Finally, a simple technique is proposed to reduce recurring conflict misses in direct-mapped caches.

TU Delft

Faculty of Electrical Engineering, Mathematics and Computer Science

Adapting the Cache Line Size to Reduce Off-Chip Memory Traffic

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

P.J. de Langen
born in Groningen, the Netherlands

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Adapting Cache Line Size to Reduce Off-Chip Memory Traffic

by P.J. de Langen

Abstract

Energy consumption is an increasingly important aspect of microprocessor design, especially for battery powered embedded systems. As processor speed increases and also the disparity in processor and memory speed, latency reduction techniques are often employed to reduce the time the processor is stalled, waiting for a memory access to complete. These techniques, as well as the increase in instruction issue rate itself, increase the amount of data transferred between processor and memory. Since off-chip communication requires a significant amount of power, power reduction can be achieved by reducing the number of transferred bytes. In this thesis, we try to determine how much data traffic is actually 'wasted' by conventional direct-mapped caches. Furthermore, we will try to derive which parameters of the direct-mapped cache are most influential on the amount of produced traffic. An effort is made to reduce the amount of unnecessary transferred data traffic between the processor and the off-chip memory system. This is done by using techniques that attempt to predict the amount of temporal/spatial locality exhibited by instructions, based on their previous behavior. Finally, a simple technique is proposed to reduce recurring conflict misses in direct-mapped caches.

Laboratory	:	Computer Engineering
Codenumber	:	CE-MS-2003-12

Advisor: B.H.H. Juurlink

Member: J.S.S.M. Wong

Member: G.N. Gaydadjiev

To my parents for their endless love and support

Contents

List of Figures vii

List of Tables ix

Acknowledgements xi

1 Introduction 1

2 Traffic Measurements 3

2.1 Background . . . 3
2.2 Benchmarks . . . 5
2.3 Experimental Results . . . 7
2.3.1 Traffic Inefficiency for Direct-Mapped Caches . . . 7
2.3.2 Isolating the Factors That Contribute to the Inefficiency . . . 10
2.3.3 Traffic × Delay . . . 14
2.4 Conclusions . . . 14

3 Caches with Adaptive Block Sizes 17

3.1 The Dual Data Cache . . . 18
3.2 The Unified Dual Data Cache . . . 20
3.3 Experimental Results . . . 21
3.4 Implicit Associativity in the DDC . . . 25
3.5 Conclusions . . . 25

4 Improving the (U)DDC 27

4.1 Simplification of the Finite State Machine . . . 27
4.2 Changing the Initial Prediction . . . 30
4.3 Alternative Prediction Algorithm . . . 31

5 Reducing Conflicts 37

5.1 Bypass/Temporal in Case of Conflict . . . 38

6 Conclusions and Directions for Future Work 41

Bibliography 44

List of Figures

2.1 Amounts of traffic produced by direct-mapped caches. 9
2.2 Traffic×Delay for direct-mapped caches. 15

3.1 Code Example: Summation Loop 17
3.2 Code Example: Unrolled Summation 18
3.3 State transitions of the prediction algorithm used by the DDC. 20
3.4 Amounts of traffic produced by the DDC and UDDCs. 22
3.5 Part of memset.c 23
3.6 Amounts of traffic produced by (U)DDCs with random prediction. 26

4.1 Reducing the 3-state FSM to a 2-state FSM. 31
4.2 Amount of traffic produced by (U)DDCs with different initial prediction. . 32
4.3 The proposed 4-state FSM. 33
4.4 Amount of traffic produced by (U)DDCs using the 4-state FSM. 35

5.1 Diagram of the conflict detection principle. 38
5.2 Amount of traffic for different benchmarks. 39

List of Tables

2.1 The used benchmarks from the Mediabench suite. 6
2.2 Traffic inefficiencies for 32-byte direct-mapped caches. 8
2.3 Caches compared to isolate the factors that contribute to the traffic inefficiency 11
2.4 Inefficiencies Ratios of 2kB caches. 11
2.5 Inefficiencies Ratios of 4kB caches. 12
2.6 Inefficiencies Ratios of 8kB byte caches. 12

Acknowledgements

I would like to express my sincere gratitude to the entire CE lab, but especially to Ben Juurlink. His patience, enthusiasm, and keen editorial eye have made this both a valuable and memorable experience.

Delft, The Netherlands
June, 2003

Pepijn de Langen

1 Introduction

A cache is a relatively small and fast memory located between the processor and main memory. Usually, a cache is placed on the same chip as the processor, whereas the main memory is located off-chip. The primary use of caches is to reduce the amount of time a processor is stalled while waiting for data from the memory system. However, by using a cache on the processor chip, the amount of traffic between the processor and the off-chip memory is also reduced. Since the amount of power used for transferring data between the processor and the memory system is significant in modern systems [4], effective caching can be very important if power is an issue.

The effectiveness of the cache is based on the principle of locality. There are two types of locality, *temporal* and *spatial*. When data is requested multiple times within a certain amount of time, it can be useful to hold a copy of it in the cache. This principle is called temporal locality. Temporal locality is exploited by just holding the data whenever it is requested. Counters, for example, have high temporal locality since they are often repeatedly used. Spatial locality is found when multiple data locations near to each other are accessed within a certain amount of time. By fetching multiple consecutive words instead of only the requested word on a cache miss, this locality can be exploited.

Spatial locality can be found in most applications, but especially in applications that operate on a lot of data. Scientific applications that use vectors and matrices used to be the main part of these. Nowadays, a lot of spatial locality may also be found in applications from the field of multimedia, that often use a lot of data in the form of video frames or audio streams.

Most caches arrange data in *lines* of multiple consecutive words, primarily to exploit spatial locality. Since multiple different memory locations may be mapped to the same cache entry, part of the address of the currently cached item needs to be saved in order to remember the actual memory location of this item. This information is commonly called the *tag*. By arranging data in lines, only one tag has to be remembered for multiple locations. This greatly reduces the amount of space needed for the tags. In this work, the term *cache line* and *cache block* are used interchangeably.

Fetching whole cache lines when only one word is needed may of course increase the amount of time needed to transfer the requested data. This, in fact, can lead to an increase in the average memory access time. To resolve this, various techniques have been proposed [8]. These include:

early restart As soon as the requested word arrives, this word is send directly to the processor, which is then allowed to continue execution.

critical word first The memory first sends the requested word, which is then send directly to the processor on arrival. The rest of the cache block is then filled, while the processor continues execution.

The influence of large blocks on the average memory access time may also be decreased by eliminating events, in which the amount of time stalled depends on the block size. Techniques that eliminate such events include:

write buffers Words that are to be sent to memory are moved to a separate buffer, allowing the processor to continue while the transfers to memory are done concurrently.

An other effect from using longer lines is the increase in produced traffic. Since communication between processor and memory system may significantly contribute to the amount of dissipated power, reduction of this power may be achieved by reducing the amount of traffic. Also, lack of bandwidth between the processor and the memory system may eventually become the bottleneck to higher performance [3]. In this thesis, an effort is made to examine by how much traffic may be reduced and how this can be done.

This thesis is organized as follows. In Chapter 2, it is shown how and how much unnecessary traffic is generated by conventional direct-mapped caches. Chapter 3 describes and analyzes two caches that attempt to distinguish between data that exhibits *temporal* and data that exhibits *spatial* locality. This is done by predicting the locality exhibited by load and store instructions, based on their earlier behavior. Although these caches are primarily build to lower the average memory access time, we investigate if they can also be used to reduce the amount of produced traffic. First, increasing the hit-ratio of the cache will also be beneficial to the amount of traffic. Secondly, the possibility to fetch smaller blocks on accesses that exhibit no spatial locality can also decrease the amount unnecessarily fetched data. Improvements to these caches are described and experimentally validated in Chapter 4. In Chapter 5, some methods to avoid conflict misses (i.e., misses resulting from lack of associativity) are discussed. Conclusions and suggestions for future work are given in Chapter 6.

Traffic Measurements 2

The main focus when designing caches if often put on reducing the average memory access time. However, most techniques used for this purpose, such as increasing the line size or prefetching, increase the amount of traffic between the cache and the off-chip memory. Two problems may arise from this. First, bandwidth stalls may occur. As the gap between the speed of processors and memory increases, these stalls will become more frequent. This will eventually become a performance bottleneck [3]. The second problem is the waste of power, caused by unnecessary traffic, as has also been found by Catthoor et al. [4]. Especially for battery powered embedded devices, power usage is an important design aspect. This work is primarily focussed on reducing the amount of traffic produced by data caches, in order to reduce power usage. In other words, an effort is made to reduce the total amount of traffic, instead of bandwidth stalls. In this thesis, only data caches will be considered. Instruction caches have very different usage and requirements compared to data caches. Therefore, the results found for data caches in this thesis need not be applicable to instruction caches.

In this chapter, a quantitative analysis is made of the traffic usage of direct-mapped data caches. This is done by measuring the amount of traffic produced by a cache. A metric called *traffic inefficiency* is then used to show how much traffic is 'wasted' by a typical cache. Similar experiments have been performed by Burger et al. [3] for SPEC92 and SPEC95 benchmarks, but our work differs in the following ways. First, our benchmarks are taken from the MediaBench suite [13], which is more representative of embedded applications. Second, *request traffic*, i.e., the addresses sent from the processor to the off-chip memory, is also incorporated in our calculations. Burger et al. did not consider request traffic and their results are, therefore, biased in favor of smaller blocks. Third, we also consider *traffic* $\times$ *delay* since many traffic reduction techniques (e.g., reducing the block size) can have a negative impact on performance. Given the real-time nature of many embedded applications, performance should not be neglected. A fourth difference is the way how contributions to the traffic inefficiency are calculated. This will be explained in more detail in Section 2.1.

A preliminary version of this chapter has appeared in [5].

2.1 Background

To determine how and by how much traffic reduction may be achieved, we have simulated direct-mapped and fully-associative caches of various sizes. When a store instruction is issued, the words corresponding to the associated cache block are fetched into the cache and the called word is updated. This is called *write-allocate*. At this point, the actual memory location has a stale copy of the written word. A cache designer may choose to have this location updated immediately after the store instruction. This is called *write-*

through. On the other hand, the update operation may be delayed until the cache block is replaced, which is called *write-back*. For the latter method, additional bits are required to keep track of the parts of the cache that are more up-to-date than the main memory. These bits are often called *dirty bits*, and may be assigned per word, per (sub)block, or per multiple blocks. If a word is written to a location that has already been marked 'dirty' in the cache, traffic is saved compared to when using write-through, since the intermediate value is never written back to the main memory. All caches in this thesis use write-back, and have one dirty bit assigned per cache block. In other words, a dirty bit is assigned to the same part of the cache as is covered by one tag. This implies that all transfers, to or from the main memory, are of equal size. If dirty bits are assigned to blocks of the same size as the transfer size, using write-back will always produce less traffic than using write-through. In this chapter this is the case for caches with blocks of 4 bytes. If dirty bits are assigned to larger blocks, more words that are actually 'clean' will be marked 'dirty'. One single write will mark a whole cache line as 'dirty', while it is uncertain if any locality can be exploited. As a result, using write-back can result in more traffic than using write-through. In general, however, using write-back produces significantly less traffic than using write-through. This has also been the case in our experiments. Our benchmarks are assumed to be memory bound, hence our delay calculations are only based on the delay caused by memory accesses.

To measure how much more off-chip traffic than necessary is generated by a cache, we use *traffic inefficiency*. The traffic inefficiency of a cache C is defined as:

$$\textit{traffic inefficiency}_C = \frac{\textit{traffic generated by cache } C}{\textit{traffic generated by the MTC}},$$

where MTC stands for *minimal traffic cache*. In this work, the minimal traffic cache is take to be a fully-associative cache with Belady's OPT replacement strategy [2] and a block size equal to the transfer size, which in this work is taken to be 4 bytes. As will be explained later, this cache is not really the minimal traffic cache, but an estimate.

Our calculations include *request traffic*, i.e., the addresses sent from the processor to the off-chip memory. As mentioned before, Burger et al. only considered data traffic, and their results are therefore more biased in favor caches with a small block size.

The total amount of traffic is calculated as follows.

- Cache hits do not produce any traffic.
- A cache miss produces CLS bytes of data traffic, where CLS is the cache line size in bytes.
- This increase is doubled, if one or more items in a displaced line were dirty. Note that at this point there is a choice how write-back is implemented. A dirty bit could be used for the whole line or one could use a dirty bit per word on the cache line. In the latter case, traffic can be saved by not writing back the words that are clean. To profit from this, however, the cache would need to be able to read and write an arbitrary number of words to and from the cache. For simplicity, we do not allow such variable transfer sizes and assume that caches write whole cache lines back to the memory system.

- Request traffic is calculated as 4 bytes for every cache miss. The total amount of traffic can thus be determined as:

$$\textit{total traffic} = \textit{data traffic} \times (1 + \frac{4}{\#\ \textit{bytes per cache line}}).$$

  Since all data brought to and from the cache needs to be addressed with 4 bytes, the request traffic can be derived as a fraction of the data traffic.

Due to the fact that we also consider request traffic, caches with larger lines may actually produce less traffic than the MTC. Temam [15] proposed an *Extended Belady's* algorithm, which minimized the cache miss ratio by optimally exploiting temporal and spatial locality. This algorithm, however, is designed to minimize miss ratio instead of the amount of traffic, as is the case with the original algorithm by Belady. Nevertheless, both algorithms provide a good estimate for the achievable traffic reduction and for sake of simplicity we will employ the original version. Also, the more simple *Belady's* can be easily compared with the same cache with a *least-recently-used* strategy, in order to show the influence of the replacement strategy. When using the *Extended Belady's*, this would much harder to find.

Another reason why OPT need not be optimal is because we use write-back instead of write-through. Temam [15] notes that Horowitz et al. [9] proposed an extension to Belady's algorithm that considers both fetch and write-back traffic. He also notes, however, that Belady [2] indicated that this would only slightly increase the amount of fetch traffic. Optimizing for write-back traffic besides fetch traffic is therefore not expected to significantly affect the result.

Contributions to the traffic inefficiency can be derived by comparing caches that differ in only one parameter. By comparing two caches that only differ in block size, for example, the influence of block size on the total amount of traffic and thus the traffic inefficiency can be found. Burger et al. subtracted two inefficiencies to calculate the contribution of a single cache parameter, such as the block size, to the inefficiency. In this work, the ratio between the two is used, since this provides more information on the influence of the various parameters. Assume, for example, that a cache C_1 is compared to a cache C_2, that for one benchmark the traffic inefficiency of C_1 is 2 and that of C_2 1, and for another benchmark the traffic inefficiencies are 100 and 50, respectively. Then Burger et al. define the inefficiency gap for the first benchmark as 1 and for the second as 50. Our metric, on the other hand, will be 2 in both cases, since in both cases cache C_1 produces twice as much traffic as C_2.

2.2 Benchmarks

The benchmarks we employed are taken from the *MediaBench* [13] suite. These benchmarks represent a large portion of the routines used in common multimedia applications, and are therefore quite representative for today's embedded applications. We have not used the *MiBench* [7] embedded benchmark suite, because it was not available to us at that time. Furthermore, most benchmarks from the *Mediabench* suite are also contained in the *MiBench* suite.

Benchmark	Input	# of mem. refs.	% loads	% stores
adpcm-dec	clinton.adpcm	$4.59 \cdot 10^5$	71.4	28.6
adpcm-enc	clinton.pcm	$4.59 \cdot 10^5$	85.6	14.4
jpeg-dec	testimg.jpg	$1.18 \cdot 10^6$	67.1	32.9
jpeg-enc	testimg.ppm	$6.55 \cdot 10^4$	68.2	31.8
mpeg2-dec	mei16v2.m2v	$3.28 \cdot 10^7$	80.7	19.3
g721-dec	clinton.g721	$4.90 \cdot 10^7$	75.8	24.2
g721-enc	clinton.pcm	$4.78 \cdot 10^7$	76.3	23.7
pegwit-dec	my.sec & pegwit.enc	$5.31 \cdot 10^6$	78.0	22.0
pegwit-enc	my.pub & pegwit.plain	$8.52 \cdot 10^6$	76.8	23.2
gsm-dec	clinton.pcm.gsm	$8.39 \cdot 10^6$	64.2	35.8
gsm-enc	clinton.pcm	$5.19 \cdot 10^7$	77.8	22.2
epic	test_image.pgm	$7.47 \cdot 10^6$	89.9	10.1
unepic	test_image.pgm.E	$1.64 \cdot 10^6$	54.0	46.0
mesa-mipmap	-	$1.65 \cdot 10^7$	64.2	35.8
mesa-texgen	-	$2.56 \cdot 10^7$	63.8	36.2
mesa-osdemo	-	$2.88 \cdot 10^6$	66.8	33.2
ghostscript	tiger.ps	$3.60 \cdot 10^8$	64.3	35.7

Table 2.1: The used benchmarks from the Mediabench suite.

The benchmarked applications are listed in Table 2.1, along with the number of memory references and the used inputs. In the following, we will give short descriptions of these applications.

JPEG (*author: Independent JPEG Group*)
Lossy image compression. Both compression (encoding) as decompression (decoding) benchmarks are used.

MPEG (*author: MPEG Software Simulation Group*)
Decoding of a MPEG-2 video bitstream.

GSM (*author: The Communications and OS Research Group at the Technische Universität Berlin*)
GSM 06.10 full-rate speech transcoding at 13 kbit/s, using residual pulse excitation/long term prediction coding (RPE/LTP). Both encoding and decoding routines are used.

ADPCM (*author: Jack Jansen*)
Adaptive Differential Pulse Code Modulation. Speech compression and decompression, using the Intel/DVI ADPCM code. Note that this code differs from the CCITT ADPCM standard.

G.721 (*author: Sun Microsystems, Inc.*)
CCITT G.721 voice compression and decompression.

PEGWIT (*author: George Barwood*)
Public key encryption and decryption, using an elliptic curve over GF(2^{255}), SHA1 for hashing, and the symmetric block cipher square.

Ghostscript (*author: Aladdin Software*)
A Postscript™ interpreter.

Mesa (*author: Brian Paul*)
A 3-D graphics library which uses an API similar to the one used by OpenGL™. Stripped versions of the `mipmap`, `osdemo`, and the `texgen` demos are used.

EPIC (*author: Eero Simoncelli*)
Efficient Pyramid Image Code, with compression algorithms based on a biorthogonal critically-sampled dyadic wavelet decomposition and a combined run-length/Huffman entropy coder. Both encoding and decoding are done.

2.3 Experimental Results

With direct-mapped caches, a requested address always maps to the same cache line. Hence, the direct-mapped cache requires only one tag-comparison, in contrast to caches that have associativity. Due to this and the fact that replacement is also trivial for direct-mapped caches, adding associativity to a cache will generally decrease its speed and increase its power usage per access. We are, therefore, mainly interested in direct-mapped caches. For several reasons, including lack of associativity, these direct-mapped caches can be quite inefficient in terms of traffic, however. In this section, experimental results are presented in order to show how much traffic is wasted by direct-mapped caches. Also, an effort is made to determine how this is caused and how traffic reduction might thus be achieved.

This section is organized as follows. First, the traffic inefficiencies of direct-mapped caches of various sizes are shown. After that, the factors that contribute to the traffic inefficiency are isolated. Finally, *traffic*×*delay* is considered.

2.3.1 Traffic Inefficiency for Direct-Mapped Caches

In order to calculate traffic inefficiencies, we simulated direct-mapped caches with blocks of 32 bytes as well as the minimal traffic cache (MTC). Besides requiring significantly more space on the die, larger caches also increase the capacity of the bus, and are thus slower or more power consuming. Hence, for a battery powered embedded device, a small cache would be preferable. Therefore, we will simulate caches with sizes in the range from 256 bytes to 32 kilobytes.

The resulting traffic inefficiencies of direct-mapped caches are presented in Table 2.2. From this table, it can be seen that 75% of these inefficiencies are between 0 and 9, but sometimes they are an order of magnitude larger. Overall, the results indicate that large savings can be achieved. Furthermore, it can be observed that for some benchmarks the traffic inefficiency is not monotonically decreasing in the cache size. Sometimes, the traffic inefficiency increases significantly, after which it decreases again.

Benchmark	256B	512B	1kB	2kB	4kB	8kB	16kB	32kB
`adpcm-dec`	6.51	3.33	2.72	2.36	56.21	2.45	0.62	0.62
`adpcm-enc`	7.92	4.93	3.97	3.44	47.05	2.57	0.65	0.65
`jpeg-dec`	6.22	6.18	5.85	6.54	4.36	3.07	5.20	3.64
`jpeg-enc`	6.40	6.49	6.28	4.97	3.91	1.37	0.84	0.74
`mpeg2-dec`	14.66	16.81	26.53	40.81	18.72	13.49	3.61	1.33
`g721-dec`	26.74	131.15	45.53	25.41	56.24	15.97	5.26	5.23
`g721-enc`	15.19	222.20	39.08	21.02	26.95	17.68	13.06	13.04
`pegwit-dec`	7.30	6.90	6.76	6.74	6.00	6.24	6.63	7.45
`pegwit-enc`	7.54	7.04	6.80	6.45	5.97	6.07	7.20	8.16
`gsm-dec`	3.31	3.17	4.01	2.77	3.34	9.82	0.78	0.76
`gsm-enc`	19.97	18.73	2.07	5.55	3.72	8.95	1.76	1.74
`epic`	5.13	4.04	3.75	3.73	3.63	1.70	1.31	1.11
`unepic`	2.80	2.43	2.21	1.76	1.33	1.22	1.22	1.26
`mesa-mipmap`	8.50	9.02	5.80	5.03	3.37	1.80	1.12	0.77
`mesa-texgen`	4.25	5.57	7.38	7.58	6.15	4.78	3.15	2.15
`ghostscript`	14.55	13.05	11.35	13.69	18.63	7.05	5.28	3.14
Average	9.81	28.81	11.25	9.86	16.59	6.51	3.60	3.23

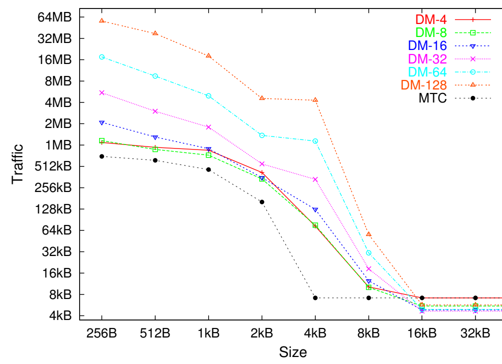
Table 2.2: Traffic inefficiencies for 32-byte direct-mapped caches.

This can be explained more clearly by looking at Figure 2.1, which depicts the amount of traffic as a function of the cache size for various benchmarks. The results for direct-mapped caches with different block sizes are shown as well as the minimal traffic cache. In these graphs, the curve labelled `DM-n` denotes a direct-mapped cache with a block size of `n` bytes and `MTC` denotes the minimal traffic cache.

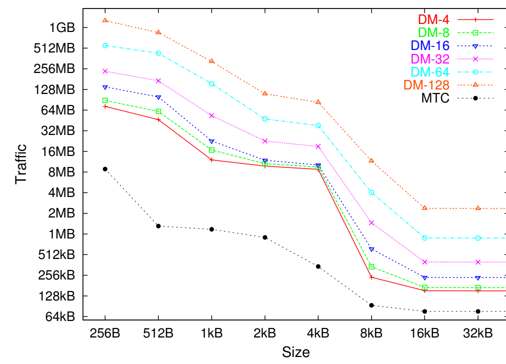
In general, if the cache size is increased, the amount of traffic decreases. This happens because *capacity misses*, e.g. misses due to the limited size of the cache, can be eliminated by enlarging the cache. Furthermore, if the cache is large enough to hold the entire working set and, consequently, there are no more capacity misses, the amount of traffic does not decrease further for larger caches. An example of this can be seen in Figure 2.1(a). When the direct-mapped caches are larger than 16kB, the amount of traffic no longer decreases. For the MTC this already occurs around 4 kilobytes. As a result, traffic inefficiency will also become constant when the cache size exceeds a certain limit.

For some benchmarks, like `pegwit-enc` whose results are depicted in Figure 2.1(e), the different configurations benefit almost equally from an increased cache size. None of the caches is able to hold the entire working set. For these benchmarks the traffic inefficiency hardly depends on the cache size, as can be seen in Table 2.2.

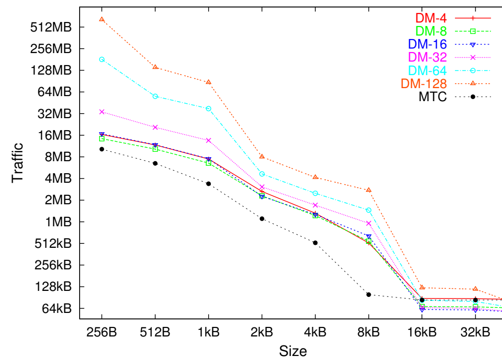
Sometimes, when the cache size is doubled, the amount of traffic decreases by more than a factor of 2. This can be explained as follows. Assume two small and simple direct-mapped cache, with a block size of 1 word. The first cache, C_1, has a capacity of 4 words, while the second, C_2, has a capacity of 8 words. Assume an application



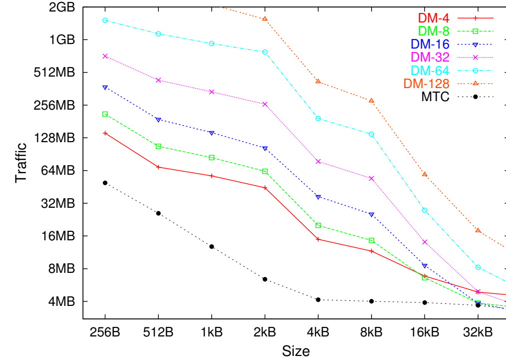
(a) adpcm-enc



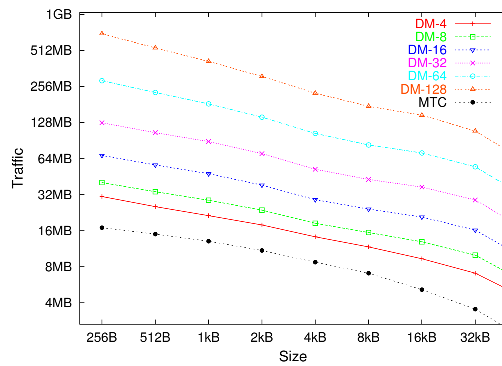
(b) g721-dec



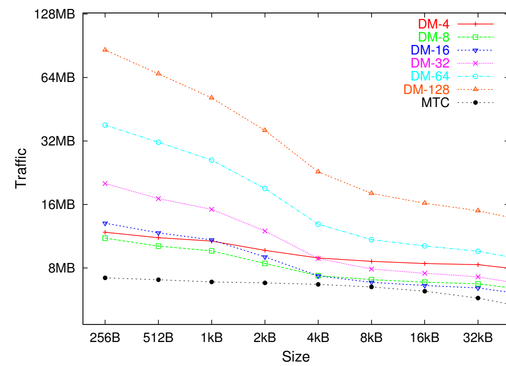
(c) gsm-enc



(d) mpeg2-dec



(e) pegwit-enc



(f) unepic

Figure 2.1: Amounts of traffic produced by direct-mapped caches.

that operates intensively on an arrays of words: $A[0..7]$. Because of the limited size and associativity of the cache, $A[i]$ and $A[i+4]$ $(0 \leq i \leq 3)$ will be mapped to the same place in the cache C_1. With this cache, repeated access to the elements of $A[]$ will discard other elements of $A[]$ from the cache. With cache C_2, however, this problem will not occur, since all elements of the array $A[]$ can be fit in the cache. In this case, in fact only 8 compulsory misses occur, independently of the amount of times the array is accessed. If such accesses are frequent in an application, the disparity between the amounts of traffic produced will become quite large.

The MTC will always benefit from an increased cache size, unless it is already able to hold the entire working set. Reason for this is that the MTC is a fully-associative cache, and thus no *conflict misses* occur with this cache. With direct-mapped caches, however, conflict misses do occur. Therefore, direct-mapped caches may not benefit at all from the increase in size, since a significant number of conflict misses may still remain. This can be seen, for example, in Figure 2.1(b) where in some caches the amount of traffic hardly decreases if the size of the direct-mapped cache is doubled from 2kB to 4kB. Increasing the cache size will eventually resolve all conflict misses. Since the direct-mapped cache may not be able to resolve these conflict misses in a range of cache sizes, and because the MTC always benefits from this increase, the traffic inefficiency can become very high in such cases.

So, because the direct-mapped caches and the MTC may not benefit equally from increased size, the traffic inefficiency may increase if cache size increases. Examples of this can also be seen in Figure 2.1(a) and 2.1(b), where the distance between the lines corresponding to the direct-mapped cache with 32-byte blocks and the MTC shows sudden increases for certain sizes of the cache. However, if a cache is made large enough, the amount of traffic and thus also the inefficiency will eventually reach a constant value. This is because the entire working set can then be kept in the cache, and all cache misses are *compulsory* or *cold-start* misses incurred the first time a data item if referenced. The inefficiency will then be at most 8, the relative difference in block size.

Not considering request traffic, a smaller block size always results in less traffic. Since in our experiments the request traffic is not ignored, a cache with smaller blocks can produce more traffic than a cache with larger blocks. As noted before, the MTC is not really a lower bound, since it may produce more traffic than a cache with blocks larger than 4 bytes. This can be seen, for example, in Figures 2.1(a) and 2.1(f), where the amount of request traffic is significant for caches with small blocks. For `adpcm-enc`, even the direct-mapped cache with blocks of 32 bytes produces less traffic than the MTC with 4 bytes per block. This results in traffic inefficiencies smaller than 1, as can be seen in Table 2.2.

2.3.2 Isolating the Factors That Contribute to the Inefficiency

There are several factors that contribute to the traffic inefficiency, such as the block size, the associativity, and the replacement strategy. Although the real contributions to the inefficiency are not independent, in this section we attempt to isolate their contribution by comparing caches that differ only in one parameter. The caches compared in these experiments are listed in Table 2.3. As explained in Section 2.1, in the work of Burger et

	Associativity, block size, replacement policy	
Factor	**Experiment 1**	**Experiment 2**
I. Associativity	direct-mapped, 32B	fully-associative, 32B, LRU
II. Replacement	fully-associative, 32B, LRU	fully-associative, 32B, MIN
III. Blk. size (DM)	direct-mapped, 32B	direct-mapped, 4B
IV. Blk. size (MTC)	fully-associative, 32B, MIN	fully-associative, 4B, MIN

Table 2.3: Caches compared to isolate the factors that contribute to the traffic inefficiency

Benchmark	**Associativity**	**Replacement**	**Block Size (DM)**	**Block Size (MTC)**
`adpcm-dec`	1.16	2.89	1.48	0.71
`adpcm-enc`	1.21	4.20	1.32	0.68
`jpeg-dec`	3.00	1.59	2.59	1.37
`jpeg-enc`	3.83	1.37	2.87	0.95
`mpeg2-dec`	9.34	2.26	5.87	1.93
`g721-dec`	29.54	1.38	2.31	0.62
`g721-enc`	27.40	1.27	2.65	0.61
`pegwit-dec`	1.21	1.50	3.91	3.73
`pegwit-enc`	1.23	1.37	3.95	3.84
`gsm-dec`	1.02	4.24	1.16	0.64
`gsm-enc`	1.04	3.12	1.23	1.72
`epic`	4.64	1.25	2.90	0.64
`unepic`	2.00	1.45	1.24	0.61
`mesa-mipmap`	6.07	1.21	2.59	0.68
`mesa-texgen`	1.50	2.70	2.93	1.87
Average	6.27	2.12	2.60	1.37

Table 2.4: Inefficiencies Ratios of 2kB caches.

al. [3] the traffic inefficiency of the first cache is subtracted from the inefficiency of the other cache to compute the *inefficiency gap*. Since this metric does not allow different benchmarks and different cache sizes to be compared, we compute the relative difference, by dividing the inefficiency of the first by the latter. To avoid confusion, we will refer to our metric as the *inefficiency ratio*.

The inefficiency ratios are presented in Tables 2.4, 2.5, and 2.6 for caches of 2kB, 4kB, and 8kB, respectively. The most interesting changes in efficiency seem to be in this range, as can be seen in Table 2.2. From these tables the traffic inefficiency can again be found:

$$\textit{traffic inefficiency} = F_{assoc} \times F_{repl} \times F_{blk_size_MTC},$$

where F_{accoc}, F_{repl}, and $F_{blk_size_MTC}$ denote the factors listed in the columns labelled Associativity, Replacement and Block Size (MTC), respectively.

Benchmark	Associativity	Replacement	Block Size (DM)	Block Size (MTC)
adpcm-dec	90.03	1.00	3.06	0.62
adpcm-enc	72.39	1.00	4.57	0.65
jpeg-dec	1.96	2.06	1.84	1.08
jpeg-enc	2.96	1.47	2.47	0.90
mpeg2-dec	6.07	2.64	5.21	1.17
g721-dec	24.68	3.14	2.16	0.72
g721-enc	21.02	2.03	2.32	0.63
pegwit-dec	1.23	1.40	3.56	3.48
pegwit-enc	1.19	1.35	3.67	3.72
gsm-dec	2.41	1.92	1.29	0.72
gsm-enc	4.15	1.35	1.22	0.66
epic	3.90	1.47	2.67	0.64
unepic	1.65	1.35	0.99	0.60
mesa-mipmap	3.53	1.39	1.85	0.69
mesa-texgen	4.81	1.46	2.35	0.87
Average	16.13	1.66	2.61	1.14

Table 2.5: Inefficiencies Ratios of 4kB caches.

Benchmark	Associativity	Replacement	Block Size (DM)	Block Size (MTC)
adpcm-dec	3.92	1.00	1.74	0.62
adpcm-enc	3.95	1.00	1.80	0.65
jpeg-dec	2.72	1.41	1.24	0.80
jpeg-enc	1.36	1.57	0.96	0.64
mpeg2-dec	14.18	1.58	4.66	0.60
g721-dec	11.84	2.07	6.22	0.65
g721-enc	9.94	2.60	5.93	0.68
pegwit-dec	1.24	1.45	3.51	3.48
pegwit-enc	1.10	1.46	3.66	3.81
gsm-dec	11.43	1.29	1.87	0.66
gsm-enc	10.79	1.23	1.28	0.67
epic	1.57	1.75	1.36	0.62
unepic	1.52	1.35	0.92	0.59
mesa-mipmap	1.68	1.61	1.09	0.67
mesa-texgen	4.91	1.41	1.99	0.69
Average	5.47	1.51	2.54	1.05

Table 2.6: Inefficiencies Ratios of 8kB byte caches.

The first factor we consider is associativity. To compute the contribution of associativity, a direct-mapped cache is compared to a fully-associative one that employs LRU replacement. The first column of Tables 2.4, 2.5, and 2.6 show that the influence of associativity on the amount of traffic depends on the benchmark as well as on the size of the cache. Note that this factor is 1 when both caches are large enough to hold all referenced data. Although in general this factor decreases as the cache size is increased, we again observe that sometimes the inefficiency ratio increases if the cache is enlarged. This is because with the direct-mapped caches, a lot of conflict misses may still occur, in contrast to the fully associative ones.

The second factor is the replacement strategy. To determine the influence of this factor, a fully associative cache with 32-byte blocks is compared to the MTC with the same block size. Again, since there are no conflict nor capacity misses in a cache that is sufficiently large, this ratio will eventually become one. Because both caches need not benefit equally from increased capacity, the ratio can sometimes increase if the cache size is increased. This occurs, for example, in both `pegwit` benchmarks for cache sizes of 4 and 8 kilobytes. Overall, the replacement strategy contributes little to the amount of traffic. When the cache is small, however, effective replacement can reduce the amount of traffic up to 4 times.

The third factor is the block size. For both the direct-mapped cache and the MTC, we calculate the ratios in produced traffic between 32-byte and 4-byte variants. Since request traffic is included in our calculations, this ratio may be smaller than 1. For this comparison, the inefficiency ratios are between 1 and 8 for most direct-mapped caches. Since using different block sizes also influences the cache hit and miss statistics, this factor need not be bounded by 8. For the MTC, the block size is less important. Actually, for most of the benchmarks and cache sizes, the MTC with a block size of 4 bytes produces more traffic than the MTC with a block size of 32 bytes. In these cases, the applications exhibit enough spatial locality to make using larger blocks useful. By enlarging the block size, the usable capacity is decreased unless all requests are exactly aligned after another. The number of *capacity* misses can thus be increased. In these cases, however, this does not contribute a lot and the reduction of *compulsory* misses makes the cache with larger blocks the more efficient one.

Interesting behavior can be seen for both `pegwit` benchmarks. For these benchmarks, the inefficiency ratios resulting from employing larger blocks are about the same for both the direct-mapped cache and the MTC, independent of the cache size. The reason for this is that these benchmarks exhibit little spatial locality, so caches with larger blocks have about the same number of hits and misses as the caches with smaller ones.

Overall, associativity is the most important factor, especially when caches are small. We remark that this is different from the work of Burger et al., where the most important factor was found to be the block size. A plausible explanation is that we consider request traffic whereas Burger et al. do not. This shows that, if the total amount of traffic in and out of the cache is to be considered, request traffic should not be neglected.

For caches large enough to hold the entire working set, the amount of spatial locality determines whether or not large blocks are beneficial. In Figure 2.1, for example, `adpcm-enc` generates about the same amount of data traffic for all direct-mapped caches larger than 16kB, indicating that most data could be cached concurrently. For `g721-dec`,

however, the caches with different block sizes do not produce an equal amount of traffic. Experiments with larger direct-mapped caches with 32-byte blocks, the results of which are depicted in Figure 3.4 of Chapter 3, show that the amount of produced traffic further decreases for sizes over 64 kilobytes. For `g721-dec`, although the working set does not fit in the cache, quadrupling the size of the cache from 16 to 64 kilobytes seems not to be beneficial at all to the amount of traffic.

2.3.3 Traffic × Delay

Many traffic reduction techniques (e.g., reducing the block size) can have a negative impact on performance. Given the real-time nature of many embedded applications, this should not be neglected. We consider *traffic×delay* as an important figure of merit.

As mentioned before, we only consider loads and stores. For calculation of the total delay, the *hit time* is taken to be 1. The *miss penalty* is derived as:

$$\textit{miss penalty} = 39 + \frac{\textit{block size}}{\textit{transfer size}},$$

where the *transfer size* is assumed to be 4 bytes in our experiments.

In Figure 2.2, traffic×delay is depicted as a function of the cache size for a number of benchmarks. In these figures, the MTC is excluded, since it uses an off-line replacement algorithm and thus has no reasonable delay.

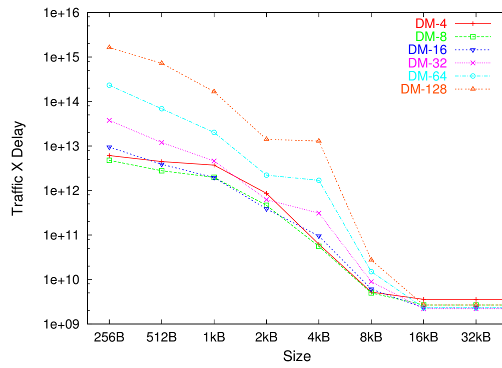
Like traffic, delay will also reach a certain constant value when the cache is large enough to hold the whole data-set. The general trend displayed by the graphs in Figure 2.2 does not differ much from the corresponding graphs in Figure 2.1. However, some differences can be noted. For some benchmarks, such as `adpcm-enc` or `unepic`, the cache configuration that is optimal with respect to traffic×delay is different from the cache configuration that is optimal with respect to traffic. Others benchmarks like `pegwit-enc` show no relative change at all.

It can be concluded, that adding delay to the traffic figures does not significantly change the general trend. In other words, a cache that is optimal with respect to traffic is in most cases also optimal with respect to traffic×delay. The main reason for this is that, through the inclusion of request traffic, the influence on access time is already partially taken into account. The task of caching the right items is in fact quite similar for reducing both traffic and access time.

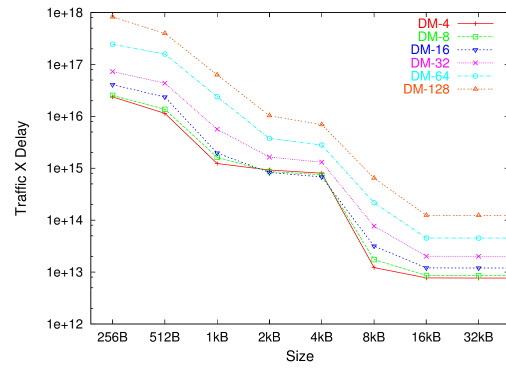
2.4 Conclusions

It is concluded that large amounts of traffic can be wasted by direct-mapped caches. In many cases, we observed that the amount of traffic was significantly reduced when the size of the direct-mapped cache was increased from 4kB to 8kB. For larger caches, however, increasing the size may not reduce the amount of traffic produced at all. Furthermore, lack of associativity can be disastrous for the amount of produced traffic, especially when the cache size is small.

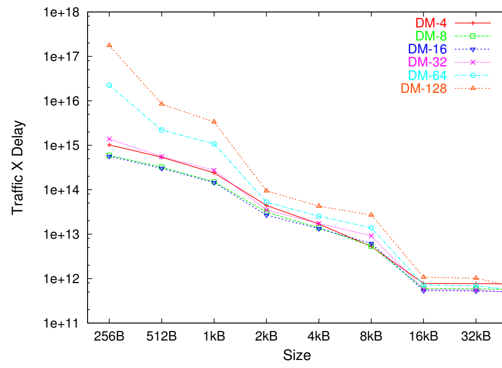
For caches with 4-byte blocks, the amount of request traffic is equal to the amount of data traffic. Still, this cache produces the least traffic for several configurations. For



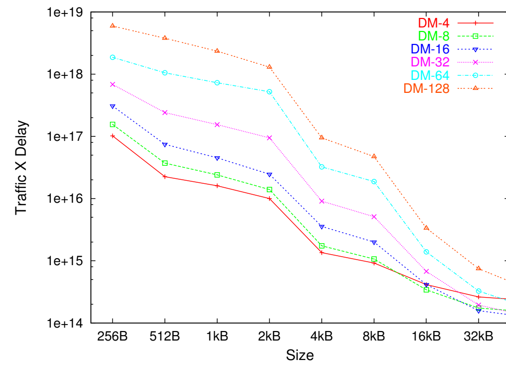
(a) adpcm-enc



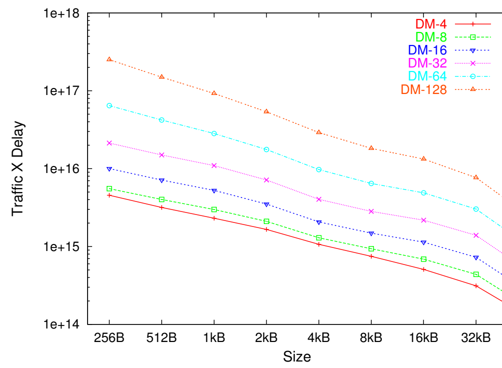
(b) g721-dec



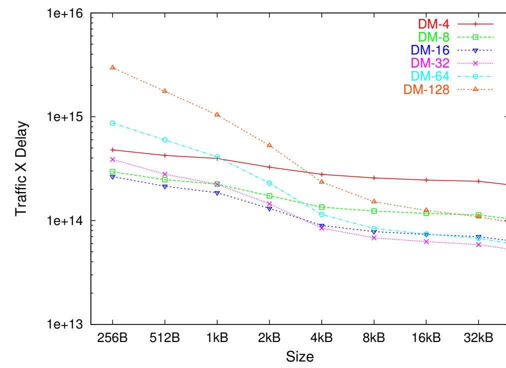
(c) gsm-enc



(d) mpeg2-dec



(e) pegwit-enc



(f) unepic

Figure 2.2: Traffic×Delay for direct-mapped caches.

small caches, this is caused by the smaller number of conflict-misses in comparison to caches with larger blocks. Since, per miss, more bytes need to be transferred for larger blocks, these conflicts may also become very expensive for caches with a large block size. Furthermore, caches with larger blocks write more words back to memory that are not necessarily dirty.

Larger blocks seem to outperform smaller ones only if the cache is large enough and the application exhibits a sufficient amount of spatial locality. If neither is the case, smaller blocks are preferable.

When looking at traffic×delay, some caches with small blocks show to be less preferable. However, overall the addition of delay does not change the general trend of these figures much. Most caches that performed well with only traffic as metric, also perform well with traffic×delay.

In Chapter 3, an effort will be made to decrease the amount of produced traffic by using caches that employ multiple block sizes.

3 Caches with Adaptive Block Sizes

Chapter 2 has shown that both the line size of the cache as well as the (lack of) associativity may have a significant impact on the amount of produced traffic. Lack of associativity combined with large lines may lead to huge traffic inefficiencies, because conflict misses become more frequent and more expensive in terms of traffic.

Fetching lines of several words may be beneficial if there is a sufficient amount of spatial locality. If spatial locality is not present, a cache should fetch small blocks. Therefore, it may be beneficial to fetch a minimum of words on requests that exhibit only temporal locality. Furthermore, by using larger blocks, more data items in the cache are replaced. Since the discarded data might have been of higher priority than the newly acquired items, this can decrease speed and increase traffic. This problem, which is commonly called *cache pollution*, is less present with caches that employ smaller blocks. To avoid pollution, it may be beneficial to keep the current data in the cache, instead of replacing it with the new data. Hence, if locality does not seem to be available at all, there is no need to allocate a line in the cache and thus *bypassing* the cache may be most efficient.

Different caches that exploit knowledge of the amount and the type of locality have been proposed. Some of the implementations require information about locality to be added by the compiler [1, 17] or determine a static optimum for a single application [14]. Others track re-usage of elements to determine available locality [10]. In general, one can track memory accesses by the address of the load/store instruction or by the address of the referenced data. A disadvantage of tracking by instruction address is that locality between different instructions is hard to detect. On the other hand, behavior of individual instructions can easily be determined, which is much harder when data address are being used. This can be explained more clearly by looking at the code examples in Figure 3.1 and 3.2.

The code in Figure 3.1 shows a loop with 1 memory access inside the loop. Since the index `i` is incremented with 1 every iteration, the load instruction inside the loop clearly exhibits a lot of spatial locality. The unrolled code depicted in Figure 3.2, on the other hand, has 4 separate load instructions inside the loop. The index `i`, in this

```
for ( i = 0; i < 1000 ; i++ )
{
  sum += A[i];
}
```

Figure 3.1: Code Example: Summation Loop

```
for ( i = 0; i < 1000 ; i += 4 )
{
  sum += A[i];
  sum += A[i+1];
  sum += A[i+2];
  sum += A[i+3];
}
```

Figure 3.2: Code Example: Unrolled Summation

code, is incremented with 4. A load instructions, in this loop, will exhibit poor spatial locality with itself, if the line size of the cache is less or equal to 32 bytes. The spatial locality between the 4 loads, however, is high. The locality in the first listing can easily be predicted by using the instruction address, because the instruction exhibits spatial locality with itself. For the second loop, on the other hand, locality cannot so easily be predicted if instruction addresses are used. If, for example, a cache with 32 bytes per line is used and the element size is 8 bytes, then the load instructions inside this loop exhibit no locality on their own. In this case, the spatial locality is only exhibited between different instructions and it can therefore be detected more easily by using the addresses of the data itself. With the latter method, locality is predicted for parts of the memory, instead of for instructions. Although this allows better detection of locality between different instructions, predicting locality for a single instructions, like the one in Figure 3.1, cannot be done as well with data addresses as with instruction addresses. The *Spatial Locality Detection Table* (SDLT) and the *Memory Address Table* (MAT) by Johnson et al. [10], for example, would be able to detect this type of locality between different instructions. These kinds of prediction are, however, beyond the scope of this thesis.

This work focuses on caches that try to predict available locality per instruction, such as the *Dual Data Cache* (DDC) proposed by González et al. [6] and the *Unified Dual Data Cache* (UDDC) proposed by Juurlink [12], both of which will be examined in this chapter. Although these caches were originally designed to reduce memory stalls, we will investigate if they can also be used to reduce the amount of traffic generated by a cache. Besides the fact that stalls and traffic can both benefit from a high cache hit rate, also the ability to fetch blocks of different sizes and reduction of cache pollution can be beneficial to the amount of traffic produced by a cache.

3.1 The Dual Data Cache

The dual data cache, as defined by González et al. [6], consists of two separate caches and a structure called the *prediction table*. The two caches may differ in capacity, line size, associativity, and optionally replacement strategy. Typically, one of these caches has a line size equal to the transfer size to memory and is dedicated to host temporal data, while the other attempts to exploit spatial locality by having a large line size. These caches will thus be referred to as 'temporal cache' and 'spatial cache'.

On every cache access, both caches are probed concurrently. Elements residing in the spatial cache will never be allocated in the temporal cache. Elements that reside in the temporal cache, however, may afterwards also be allocated in the spatial cache. This occurs when a word from the same spatial cache line is not found in either cache and is then allocated in the spatial cache. Because of this, data may reside in both caches. Hence, if a store instruction hit both caches, then the data should also be written to both caches. If load instruction hits both caches, the data from the temporal cache is used, since this cache always contains the most up to date information. Whenever the requested item is not found in either cache, the prediction table is consulted on how the item should be cached.

The prediction table is a small cache itself, indexed by the program counter (PC). Each entry consists of the following fields:

PC The program counter (instruction address) of the load or store instruction is used as the *tag*.

Last address The last data address requested by this instruction.

Stride The difference between the last and the second data address that was referenced by this instruction.

State Holds information on past behavior regarding the stride.

Prediction Holds information on past behavior regarding the prediction.

The locality prediction table is accessed at the same time as the temporal and spatial cache. If no entry corresponding to the current instruction address is found, an entry is allocated. The locality prediction table is updated on every load/store and, consequently, a prediction is always available if the entry is not found in either the temporal or the spatial cache. There are three different predictions:

Temporal A (small) temporal cache line is fetched.

Spatial A (large) spatial cache line is fetched.

Bypass The requested word is fetched, but it is not allocated in either cache.

Entries in the prediction table follow the state transitions depicted in Figure 3.3. It is further noted that, according to González et al., the prediction field is only used in the *Steady* state. In both the *Initial* and the *Transient* state, the default prediction is used, which is *temporal*. The last address field is always updated. The fields that hold the last stride, the length of the stride, the state, and the current prediction are updated according to the rules in Figure 3.3. A stride is considered small (`small(S)` is true) if it is smaller than the size of a spatial cache line. If, in the *Steady* state, a stride is detected that differs from the previous value, the state becomes *Initial*. On this event, the prediction field is updated according to a function, labelled $f(S, L)$ in Figure 3.3. This function is defined as follows. If, on this transition, the value in the stride field (i.e. the previous stride) is too large to exploit spatial locality, the prediction will become *temporal* unless the instruction is expected to interfere with itself. This test, which is

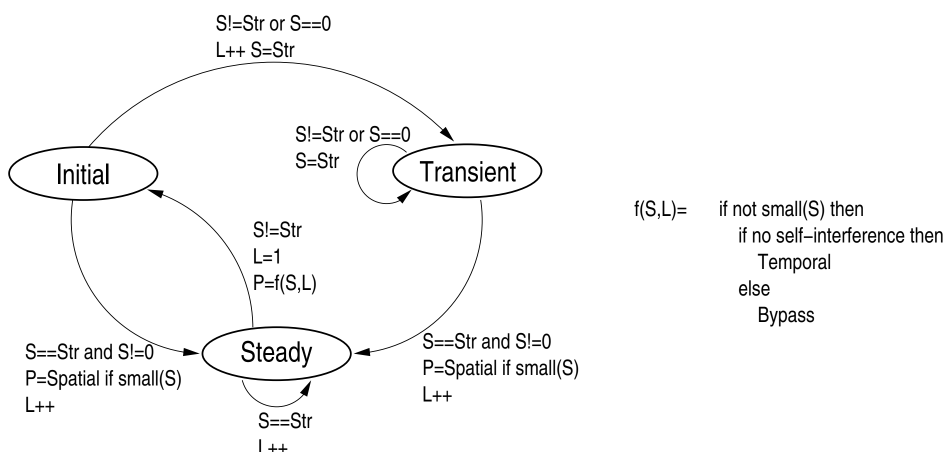


Figure 3.3: State transitions of the prediction algorithm used by the DDC.

labelled `self-interference` in Figure 3.3, is done in the following way. If an array is accessed with a stride larger than the line size, the number of concurrently cacheable elements is less or equal to the number of sets. If the distance between two accesses, measured in cache lines, is not coprime with the number of sets, not all cache lines will be used and the maximum number of concurrently cacheable array elements is thus much less. More precisely, the maximum number of elements L_{max} can be found by:

$$L_{max} = \frac{\textit{cache size}}{GCD(\#sets, \textit{set-stride})} \qquad ,$$

where GCD is the *greatest-common-divisor* and *set-stride* is the stride in cache lines. Since, in general, the number of sets in a cache is a power of 2, the GCD will also be a power of 2. In fact, one may count the number of right-hand side zeroes in the binary representation of a stride, to derive the GCD. If, for some stride, the number of right-hand side zeroes is N, then the GCD of the number of sets and this stride will be 2^N. Now, if this GCD multiplied by the length of the stride is larger than the size of the cache, this instruction will sweep itself out of the cache. If this is the case, the prediction becomes *bypass*.

3.2 The Unified Dual Data Cache

The unified dual data cache (UDDC) uses the same prediction table and state transitions as the DDC. The main difference is that, in contrast to the original dual data cache, the unified version has only one cache structure to hold data. This way, the cache capacity need not be statically assigned to either temporal or spatial data, as is the case with the DDC.

When spatial locality is predicted, the UDDC will act as a conventional direct-mapped cache and a whole cache line will be allocated. If temporal locality is predicted, only a smaller part of the cache line will be used. Bypasses are treated the same as with the DDC.

As described by Juurlink [12], one can choose between two different types of the UDDC. In the first variant, which will be referred to as `UDDC-A`, the tag is shared among a number of smaller blocks. Each small block has a *valid-bit* and optionally a *dirty-bit* to allow sub-block caching. The second variant, which will be called `UDDC-B`, has separate tags for each 'temporal' sub-block. This allows the cache to exploit much more of its capacity when caching data that is classified as temporal. This implementation, however, requires a large amount of space on the die to be assigned for tags. If a temporal block is defined to be 4 bytes, for example, the tags will take up about half the size of the total cache structure.

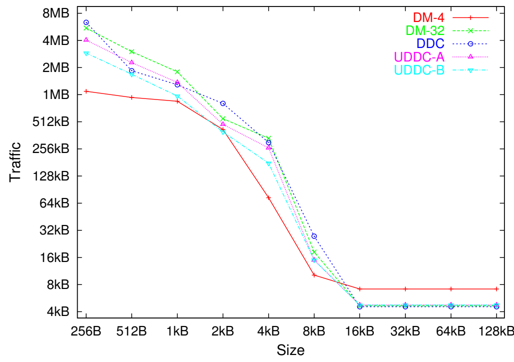
3.3 Experimental Results

We have simulated the DDC and the UDDC with the following parameters. A DDC with a capacity of N bytes is split equally in a $N/2$ temporal part and $N/2$ spatial part. For both the DDC and the UDDC a temporal block is equal to the transfer size, which is taken to be 4 bytes. A spatial block consists of 32 bytes for both caches. The prediction table is a direct-mapped cache with 128 sets. The latter is chosen to be rather large since our main interest is the performance the prediction algorithm instead of the table itself.

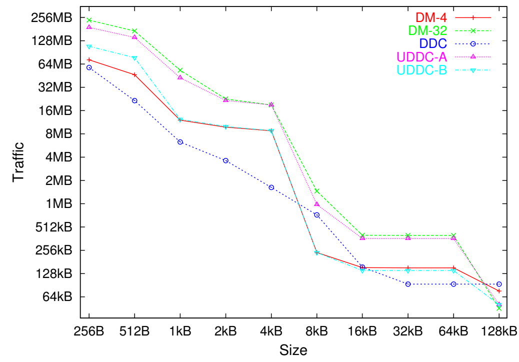
In Figure 3.4, the results for various benchmarks are depicted for the DDC, the UDDCs, and two direct-mapped caches. The direct-mapped caches are labelled `DM-n`, where `n` denotes the line size in bytes.

From Figure 3.4, it can be seen that both the DDC and the UDDCs produce less traffic than direct-mapped caches for a number of benchmarks. Especially for `g721-dec`, and `pegwit-enc` they can significantly reduce the amount of traffic, compared to direct-mapped caches with 32 byte cache lines. Some other benchmarks, like `mpeg2-dec`, only show improvement for smaller caches. As cache capacity is increased, the amount of conflicting cache accesses will decrease and as a result, larger blocks become increasingly preferable. This can also be seen from Figure 3.4, where for caches of 128 kilobytes, the direct-mapped cache with blocks of 32 bytes (DM-32) results in the least amount of traffic for most benchmarks. For some benchmarks the DDC and the UDDC produce more traffic than the DM-32 cache, especially for larger caches. For `gsm-dec`, both the DDC and the UDDC can produce up to 8 times as much traffic. Although this exceptionally bad performance is only found in one benchmark, it still shows that, under certain circumstances, the used prediction algorithm can perform terribly bad. This will be discussed in more detail.

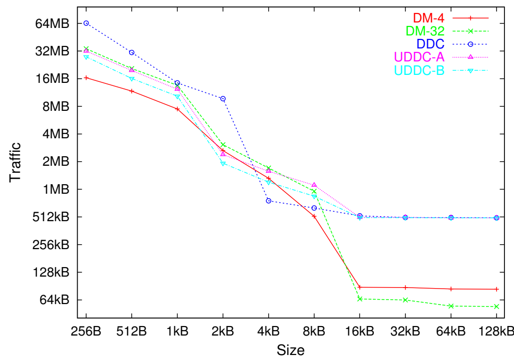
From the simulation output, we have found that for `gsm-dec` a significant amount of cache accesses is predicted *bypass*. From Figure 3.3 can be seen that there are two ways for the saved prediction to become *bypass*. First, the initial prediction (i.e., the prediction that is assigned to newly allocated entries in the locality prediction table) is *bypass*. Secondly, the prediction may be set to *bypass* when self-interference is detected on transition from the *Steady* to the *Initial* state. Since in both the *Initial* and the *Transient* state the prediction is always *Temporal*, the state must be *Steady* for the predictor to output *bypass*. This state is only reached if the stride is constant. So, *bypass* can be predicted if the stride does not vary for two consecutive executions of the



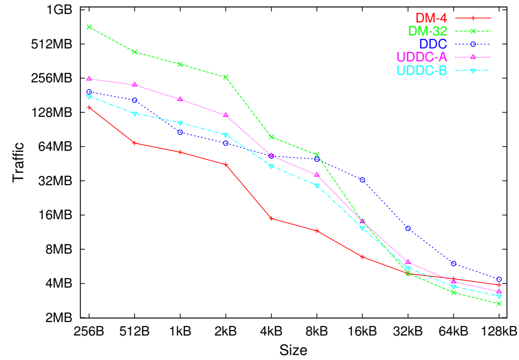
(a) adpcm-enc



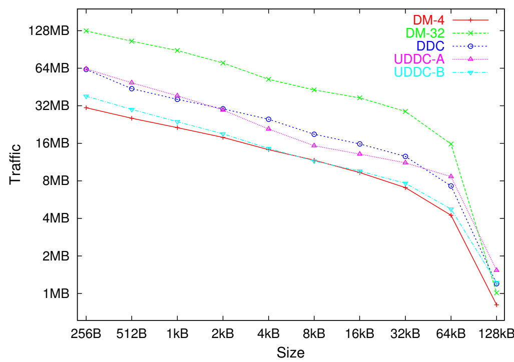
(b) g721-dec



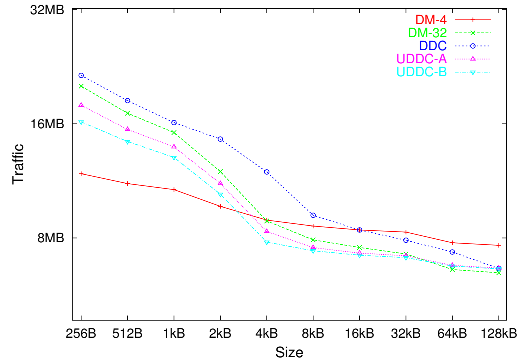
(c) gsm-dec



(d) mpeg2-dec



(e) pegwit-enc



(f) unepic

Figure 3.4: Amounts of traffic produced by the DDC and UDDCs.

```
/* Write 8 'op_t' per iteration until less than 8 'op_t' remain. */
xlen = len / (OPSIZ * 8);
while (xlen > 0)
{
  ((op_t *) dstp)[0] = cccc;
  ((op_t *) dstp)[1] = cccc;
  ((op_t *) dstp)[2] = cccc;
  ((op_t *) dstp)[3] = cccc;
  ((op_t *) dstp)[4] = cccc;
  ((op_t *) dstp)[5] = cccc;
  ((op_t *) dstp)[6] = cccc;
  ((op_t *) dstp)[7] = cccc;
  dstp += 8 * OPSIZ;
  xlen -= 1;
}
```

Figure 3.5: Part of memset.c

same load/store instruction and the stride is larger than the line size of the spatial cache, or when self-interference is detected. Simulation output shows that, with the `gsm-dec` benchmark, self-interference is never detected, however. Therefore, for all occurrences where *bypass* is predicted, this is still the initial prediction. In other words, prediction has not been set to either *spatial* or *temporal.* This can only occur if a stride is constant and large enough. Instructions can reach this *bypass* predicting state in 2 accesses and will remain in this state as long as the stride does not change.

By disassembling the code, we found that the bypassed memory access in this case do not originate from code that is actually part of the gsm routines. The responsible code was found to derive from `memset()`, a C-function that is part of the standard GNU C library. The function `memset()` is one of a collection of functions that, in general, use a hardware-specific implementation In our case, a generic implementation was used which contains the code listed in Figure 3.5. Where the variable `op_t`, defined as an unsigned long int, has a size of 4 bytes.

The source code in Figure 3.5 shows an unrolled loop. Unrolling loops is a commonly used technique to improve performance of pipelined superscalar processors. For the DDC and the UDDC, however, this code performs poorly since there is a high degree of spatial locality but the prediction algorithm fails to detect it. An iteration of the loop depicted in Figure 3.5 causes a direct-mapped cache to produce 32 bytes data traffic for fetching the line on the first access. Since *write-back* is used, the data is not yet written back to memory. Instead the cache line is marked 'dirty'. The total amount of traffic produced by this direct-mapped cache, including 4 bytes for request traffic, is thus 36 bytes. Note that, because of the now 'dirty' cache line, another 36 bytes will eventually be spent for writing the data back to memory. For the DDC and the UDDC, assuming that all accesses in this loop bypassed the cache, the amount of traffic is computed as follows. Every bypass costs 4 bytes data traffic and 4 bytes request traffic. Since this data is written directly to memory, no cache line is allocated and no dirty bits are being set.

The total amount of traffic, produced by the DDC or UDDC in a single iteration of the before mentioned loop, is thus 64 bytes. If only this loop were executed, the the direct-mapped cache would produce more traffic than the DDC or UDDC. The main advantage of the direct-mapped cache over the DDC and UDDC, is that the direct-mapped cache has the previously written bytes still in the cache. If, after this loop, one of these elements is referenced by a load or store instruction, the element may still reside in the direct-mapped cache, and no traffic will be produced. For the DDC and the UDDC, however, accesses to these locations will result in a cache misses. If the cache is thus sufficiently large, the direct-mapped cache will benefit from caching the data from the store instructions in Figure 3.5. The DDC and the UDDC will bypass these accesses, independent of the size of the cache. In general, this would lead to a disparity of a factor 2 at most. The reason for the much larger factor with the `gsm-dec` benchmark, is that in this application, a significant number of written memory locations are never accessed by load instructions. The main part of these memory locations, that are only written to, make up the output buffers. In normal operation, these output buffers are of course read by some other application, by the operating system or are handled in some other way. However, since we only simulated a single applications, using standard input and output, these memory locations seem to be only written to. For the direct-mapped cache, these store instructions do not add to the amount traffic. If *bypass* is predicted for these accesses with the DDC and the UDDCs, these caches will keep adding to the amount of traffic. Hence the disparity in traffic between the caching and bypassing cache can ever increase. In this case, the disparity of 8 is somewhat misleading, since we do not know how this buffer is read. If the instructions that read these memory location are also bypassed, the disparity in traffic could increase even more. If these loads are cached, the amounts of traffic could still differ by a factor of 2.

The UDDC-B produces less traffic than the UDDC-A, and for some cases the difference is significant. This disparity derives from the higher number of conflicts that occur in the UDDC-A because it has 8 times as few tags as the UDDC-B. For memory accesses that are predicted to exhibit spatial locality, the effective capacity of the DDC is half the size of the capacity of the UDDCs. Therefore, if the UDDCs benefit from an increase in cache size, for example with `g721-dec` from 8 to 16 kilobytes, then for the DDC this improvement can be seen from 16 to 32 kilobytes. In about half the cases, the UDDCs perform better than the DDC. With `unepic`, for example, the UDDCs produce less traffic than the direct-mapped caches, whereas the DDC increases the amount of traffic. In cases where the DDC produces less traffic than the UDDCs, however, the disparity can be far more significant. With `g721-dec` in Figure 3.4(b), for example, the UDDC-A can produce more than 8 times as much traffic as the DDC.

For this, we give two possible explanations. First, it appears that a side-effect of the DDC is that it is, in fact, two-way set-associative. With the DDC, a temporal predicted item cannot interfere with a spatial predicted one. The UDDC does not have this property and may thus produce more traffic than the DDC. This effect can be tested by comparing the results of the original DDC with a one that has a prediction table which randomly predicts *spatial* or *temporal*. This is covered in Section 3.4. Another potential drawback of the UDDC-A is that dirty blocks, that need to be written back to memory, always require transfer of a whole cache line, although only one temporal block might

actually be dirty. When dirty blocks from the temporal cache of the DDC are written back, only one word of data traffic is produced.

3.4 Implicit Associativity in the DDC

Because of the split cache design of the dual data cache, *temporal* and *spatial* data do not interfere with each other, in contrast to the unified dual data cache, where interference between the two can occur. To show the effect of this implicit associativity of the dual data cache, the original DDC is compared to a cache with the same design, except for the prediction algorithm. The prediction in this new DDC will be chosen randomly from either *temporal* or *spatial*. This cache will be referred to as *Random DDC*. Note that, besides effect of the split cache design, also the quality of the original prediction algorithm can thus be determined.

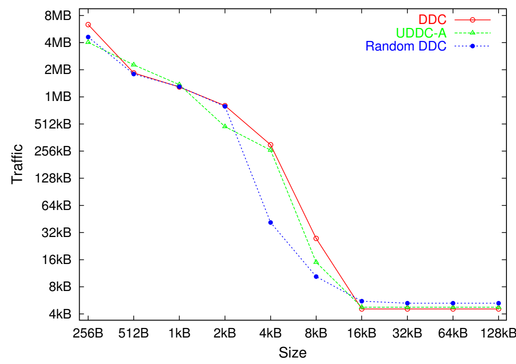
In Figure 3.4, the results are depicted. In this figure, `DDC` denotes the original DDC, `UDDC` denotes the Unified DDC type A, and `R-DDC` the DDC with random prediction.

Not surprisingly, when the `gsm-enc` benchmark is being used, the R-DDC produces far less traffic than the DDC and UDDC for larger cache sizes. Here, the ineffective prediction of *bypass* by the original prediction algorithm is clearly the cause of the enormous amount of traffic, which is not generated by the cache with random prediction. For some benchmarks, like `pegwit-enc`, the original prediction algorithm seems to work quite well. Overall, however, the prediction algorithm proposed by González et al. does not seem to perform substantially better than the random prediction. For several configurations, for example `adpcm-enc` with caches of 4kB or 8kB, the random prediction is actually more effective. In more than 40% of all tested cases, caches with random prediction produced less traffic than the caches with the original prediction algorithm.

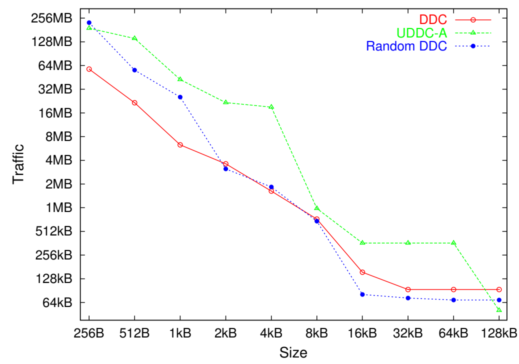
We conclude that one of the main reasons why the dual data cache sometimes performs better than a direct-mapped cache is because of its split cache design. The implicit associativity is, in fact, of more importance than the prediction algorithm, which is effective only in few cases.

3.5 Conclusions

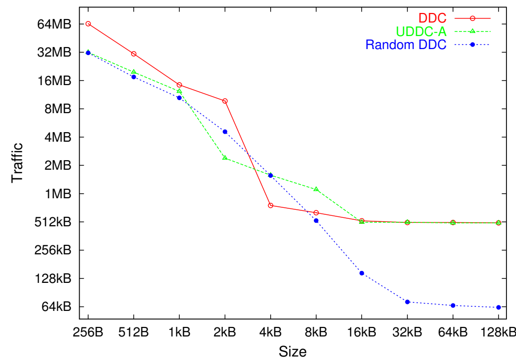
It has been shown that dual data caches can, in principle, reduce the amount of traffic compared to direct-mapped caches. The prediction algorithm as proposed by González et al., however, does not perform as well as was expected. Furthermore, we have observed that loop-unrolling, which is a quite commonly used performance improvement for superscalar processors, causes the prediction algorithm to predict ‘bypass’. This has been shown to be very inefficient, since the data in unrolled loops might exhibit high spatial locality. By simulating the DDC with a random prediction algorithm, instead of the original algorithm, it has been shown that most significant savings, made with the DDC, are in fact accomplished by the implicit associativity in this cache.



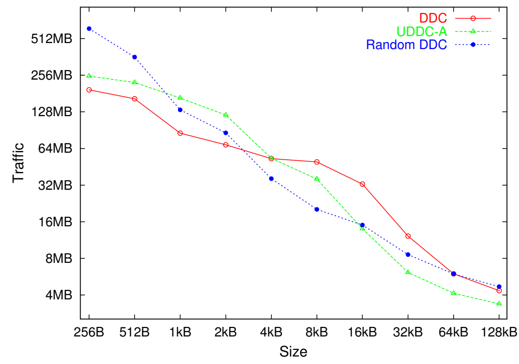
(a) adpcm-enc



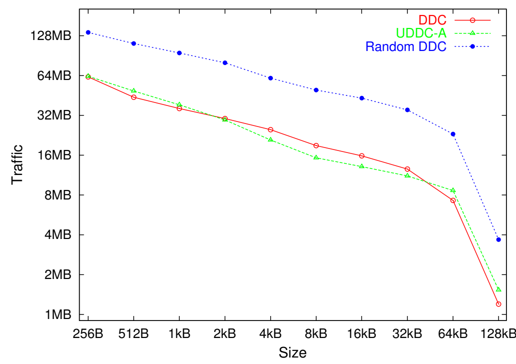
(b) g721-dec



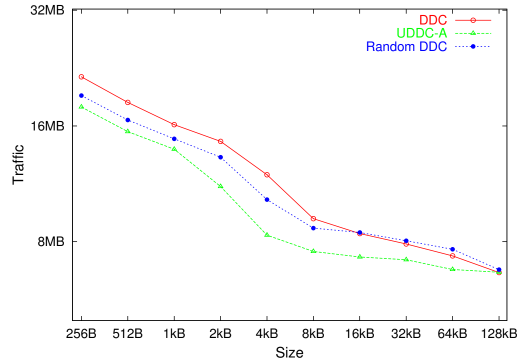
(c) gsm-enc



(d) mpeg2-dec



(e) pegwit-enc



(f) unepic

Figure 3.6: Amounts of traffic produced by (U)DDCs with random prediction.

Improving the (U)DDC

4

The results found in Chapter 3 show that both the DDC and the UDDC can reduce the amount of produced traffic to some extend. For some benchmarks, however, these results are disappointing. Especially the prediction to bypass the cache shows to be disastrous for at least one application. This chapter is organized as follows. First, we make a side-step and proof than the the original finite state machine can be reduced to an equivalent machine with only two states. Hence, the state can be represented with one bit less. Second, we will investigate if less traffic is produced, if the initial predictions are set to *temporal* rather than *bypass* for both the DDC and the UDDC. Third, both the DDC and the UDDC are tested using a different prediction algorithm with 4 states.

4.1 Simplification of the Finite State Machine

The dual data cache employs a prediction algorithm that is described by the finite state machine (FSM) depicted in Figure 4.1(a). In this section, we will proof that this FSM is equivalent to the FSM with only 2 states, shown in Figure 4.1(b).

Theorem 1. *The 2-state finite state machine $\mathcal{M}_2$ will produce the same output as the original finite state machine $\mathcal{M}_1$, for all possible input strings.*

Proof. The original FSM will be referred to as $\mathcal{M}_1$, and the new FSM will be called $\mathcal{M}_2$. Both finite state machines use a set of *input symbols* $\mathcal{I} = \{0, 1\}$, where 1 denotes that the current stride is equal to the previous one, and unequal to 0. A 0 input symbol denotes that either the current stride is different from the previous one or they are both zero. Furthermore, the original FSM uses a set of 3 states $\mathcal{Q} = \{\sigma_{Initial}, \sigma_{Transient}, \sigma_{Steady}\}$. Both $\mathcal{M}_1$ and $\mathcal{M}_2$ use 4 functions to compute the new state, and the values of S_{prev}, L, and P. Here, S_{prev} denotes the previous stride, L denotes the length of the stride, and P denotes the value of the prediction field. Furthermore, we will use S_{curr} for the current stride. For $\mathcal{M}_1$, the transition function $g : \mathcal{Q} \times \mathcal{I} \rightarrow \mathcal{Q}$ is defined as:

$$
\begin{aligned}
g_n(\sigma_{Initial}, 0) &= \sigma_{Transient},\\
g_n(\sigma_{Initial}, 1) &= \sigma_{Steady},\\
g_n(\sigma_{Transient}, 0) &= \sigma_{Transient},\\
g_n(\sigma_{Transient}, 1) &= \sigma_{Steady},\\
g_n(\sigma_{Steady}, 0) &= \sigma_{Initial},\\
g_n(\sigma_{Steady}, 1) &= \sigma_{Steady}.
\end{aligned}
$$

To update the values of the last stride (S_{prev}), the length of the stride (L), and the prediction field (P), the functions $h : \mathcal{Q} \times \mathcal{I} \rightarrow S$, $i : \mathcal{Q} \times \mathcal{I} \rightarrow L$, and $j : \mathcal{Q} \times \mathcal{I} \rightarrow P$ are

used. The function h is defined as:

$$h_n(\sigma_{n-1}, e) = \begin{cases} S_{prev} & \textit{if } \sigma_{n-1} = \sigma_{Steady} \textit{ and } e = 0, \\ S_{curr} & \textit{otherwise.} \end{cases}$$

The function i, to update the length of the stride, is defined as:

$$\begin{aligned} i_n(\sigma_{Initial}, 0) &= L_{n-1} + 1, \\ i_n(\sigma_{Initial}, 1) &= L_{n-1} + 1, \\ i_n(\sigma_{Transient}, 0) &= L_{n-1}, \\ i_n(\sigma_{Transient}, 1) &= L_{n-1} + 1, \\ i_n(\sigma_{Steady}, 0) &= 1, \\ i_n(\sigma_{Steady}, 1) &= L_{n-1} + 1. \end{aligned}$$

Finally, the function j is defined as:

$$\begin{aligned} j_n(\sigma_{Initial}, 1) &= \textit{spatial if small}(S_{prev}), \\ j_n(\sigma_{Transient}, 1) &= \textit{spatial if small}(S_{prev}), \\ j_n(\sigma_{Steady}, 0) &= f(S_{prev}, L_{n-1}), \\ j_n(\sigma_{n-1}, e) &= P_{n-1} \textit{ otherwise}, \end{aligned}$$

where $f(S_{prev}, L_{n-1})$ is the function used to update the prediction field, as described in Section 3.1. This FSM, as proposed by González et al., is shown in Figure 4.1(a).

The proposed FSM with only two states, $\mathcal{M}_2$, is presented in Figure 4.1(b). In this FSM, the *Initial* and the *Transient* state are combined to one state, which we again will call *Initial*. The new *Steady* state in Figure 4.1(b) is the same state as in the original FSM. Hence, the reduced set of states of this new FSM is $\mathcal{Q}' = \{\sigma_{Initial}, \sigma_{Steady}\}$. Like $\mathcal{M}_1$, this new FSM also predicts the saved prediction field P if the state is *Steady*, and predicts the default prediction otherwise. For this two-state FSM, the transition function $g' : \mathcal{Q} \times \mathcal{I} \rightarrow \mathcal{Q}$ is given by:

$$\begin{aligned} g'_n(\sigma, 0) &= \sigma_{Initial}, \\ g'_n(\sigma, 1) &= \sigma_{Steady}, \qquad \forall \sigma \in \mathcal{Q}'. \end{aligned}$$

The function $h' : \mathcal{Q}' \times \mathcal{I} \rightarrow S$, which updates the stride field for this FSM, is equal to the function h, which was used with the original FSM. To update the *length* field (L) in this FSM, a function $i' : \mathcal{Q}' \times \mathcal{I} \rightarrow L$ is used, defined as:

$$\begin{aligned} i'_n(\sigma_{Initial}, 0) &= 2, \\ i'_n(\sigma_{Initial}, 1) &= L_{n-1} + 1, \\ i'_n(\sigma_{Steady}, 0) &= 1, \\ i'_n(\sigma_{Steady}, 1) &= L_{n-1} + 1. \end{aligned}$$

The function $j' : \mathcal{Q}' \times \mathcal{I} \to P$ is defined as:

$$\begin{aligned} j'_n(\sigma_{Initial}, 1) &= \textit{spatial if small}(S_{prev}), \\ j'_n(\sigma_{Steady}, 0) &= f(S_{prev}, L_{n-1}), \\ j'_n(\sigma_{n-1}, e) &= P_{n-1} \ \textit{otherwise.} \end{aligned}$$

To prove that these two finite state machines are equivalent, we have to show that for every possible input, $\mathcal{M}_1$ produces the same outputs as $\mathcal{M}_2$. In other words, we have to show that for some function $t : \mathcal{Q} \to \mathcal{Q}$ it holds that

$$\begin{aligned} t(g(\sigma, e)) &= g'(t(\sigma), e), \\ h(\sigma, e) &= h'(t(\sigma), e), \\ i(\sigma, e) &= i'(t(\sigma), e), \\ j(\sigma, e) &= j'(t(\sigma), e), \qquad \forall \sigma \in \mathcal{Q}, e \in \mathcal{I}. \end{aligned}$$

We define t as:

$$t(\sigma_n) = \begin{cases} \sigma_{Initial} & \textit{if } \sigma_n = \sigma_{Transient}, \\ \sigma_n & \textit{otherwise.} \end{cases}$$

This may also be written as:

$$t(\sigma_n) = \begin{cases} \sigma_{Initial} & \textit{if } \sigma_n \neq \sigma_{Steady}, \\ \sigma_{Steady} & \textit{if } \sigma = \sigma_{Steady}. \end{cases}$$

Since

$$t(\sigma_n) = \sigma_n, \qquad \forall \sigma \in \mathcal{Q}',$$

This leads to

$$t(g(\sigma_{n-1}, e)) = \begin{cases} \sigma_{Initial} & \textit{if } g(\sigma_{n-1}, e) \neq \sigma_{Steady}, \\ \sigma_{Steady} & \textit{if } g(\sigma_{n-1}, e) = \sigma_{Steady}. \end{cases}$$

Because

$$g(\sigma_{n-1}, e) = \sigma_{Steady} \qquad \textit{iff } e = 1,$$

it follows that

$$t(g(\sigma, e)) = g'(t(\sigma), e), \qquad \forall \sigma \in \mathcal{Q}, e \in \mathcal{I}.$$

Now, it remains to prove that:

$$\begin{aligned} h(\sigma, e) &= h'(\sigma, e), \\ i(\sigma, e) &= i'(\sigma, e), \\ j(\sigma, e) &= j'(\sigma, e), \qquad \forall \sigma \in \mathcal{Q}', e \in \mathcal{I}, \end{aligned}$$

and

$$\begin{aligned} h(\sigma_{Transient}, e) &= h(\sigma_{Initial}, e), \\ i(\sigma_{Transient}, e) &= i(\sigma_{Initial}, e), \\ j(\sigma_{Transient}, e) &= j(\sigma_{Initial}, e), \qquad \forall e \in \mathcal{I}. \end{aligned}$$

By definition,

$$h(\sigma, e) = h'(\sigma, e), \qquad \forall \sigma \in \mathcal{Q}',$$

and

$$h(\sigma_{Transient}, e) = h(\sigma_{Initial}, e).$$

For the first FSM, the length field L is always equal to 1 if the state is $\sigma_{Initial}$. From this, it follows that:

$$i_n(\sigma_{Initial}, 0) = L_{n-1} + 1 = 2 = i'_n(t(\sigma_{Initial}), 0).$$

Since

$$g(\sigma_{Initial}, 0) = \sigma_{Transient},$$

it also holds that

$$i_n(\sigma_{Transient}, 0) = L_{n-1} = 2 = i(\sigma_{Initial}, 0),$$

and we can therefore conclude that

$$i(\sigma, e) = i'(t(\sigma), e), \qquad \forall \sigma \in \mathcal{Q}, e \in \mathcal{I}.$$

Furthermore, j' is defined the same as j for $\sigma \in \mathcal{Q}'$. Since also

$$j(\sigma_{Inital}, e) = j(\sigma_{Transient}, e), \qquad \forall e \in \mathcal{I},$$

we can conclude that

$$j(\sigma, e) = j'(t(\sigma), e), \qquad \forall \sigma \in \mathcal{Q}, e \in \mathcal{I}.$$

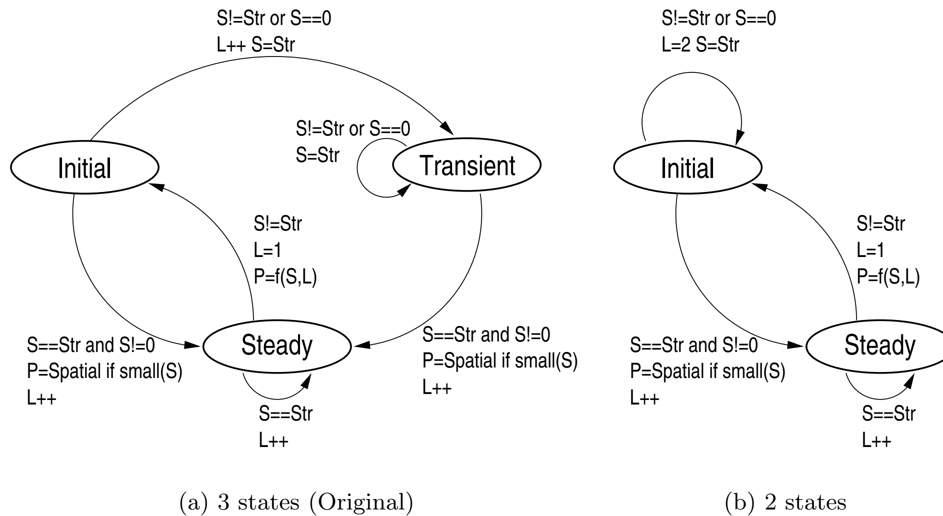
Hence, the new 2-state FSM is equivalent to the original FSM proposed by González et al. □

4.2 Changing the Initial Prediction

As shown in Figure 3.4, predicting bypasses can cause the DDC or UDDC to produce significantly more traffic than direct-mapped caches. Since these bypasses originate from the 'initial prediction', changing this initial prediction to *temporal* might prevent inefficient bypassing from occurring. To show whether or not this reduces the amount of traffic, a DDC and a UDDC of type A are simulated with the initial prediction set to *temporal* instead of *bypass*.

Figure 4.2 shows the results of changing the initial prediction to temporal, labelled as `Temp-DDC` and `Temp-UDDC` for the DDC and the UDDC-A respectively. The original DDC and UDDC-A are also depicted in this figure, denoted as `Orig-DDC` and `Orig-UDDC` respectively.

For almost all benchmarks and cache sizes, the initial prediction does not seem to be of much influence on the amount of traffic. Only small differences are detected between the versions that use *bypass* and the ones that use *temporal* as the initial prediction. Because the prediction algorithm always predicts *temporal* if not in the *Steady* state, and



(a) 3 states (Original)

(b) 2 states

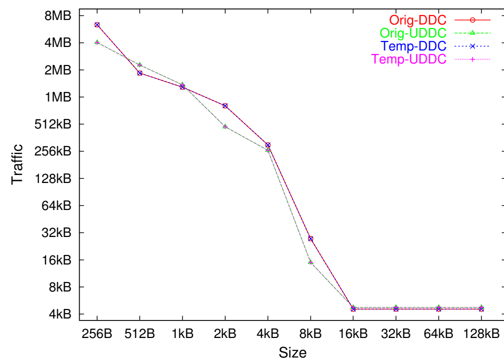
Figure 4.1: Reducing the 3-state FSM to a 2-state FSM.

because in the *Steady* state the prediction becomes *spatial* for small strides, differences only occur when the state becomes *Steady* and the stride is large. For most benchmarks, the amount of produced traffic is about equal for the two different initial predictions, and it can therefore be concluded that large strides do not occur very often in these applications.

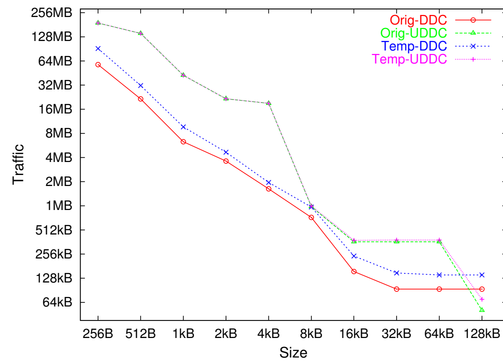
Since the little differences are almost always in favor of the cache with the initial prediction set to *bypass*, we can conclude that, in principle, using this initial prediction can be efficient. With `gsm-dec`, however, bypassing is disastrous to the amount of traffic. With this benchmark, the bypasses produced by the DDC and the UDDC derive from the initial prediction, rather than from the 'self-interference' detection algorithm. Therefore, for applications like `gsm-dec` in Figure 4.2(c), an initial prediction of *temporal* is highly preferable. Furthermore, the savings made for `gsm-dec` by having a *temporal* prediction, are far more significant than the small increase in traffic for some other benchmarks. In order to efficiently use *bypass* as the initial prediction, one has to make certain that behavior like in `gsm-dec` will not occur. Otherwise, setting the initial prediction to *bypass* is highly inadvisable.

4.3 Alternative Prediction Algorithm

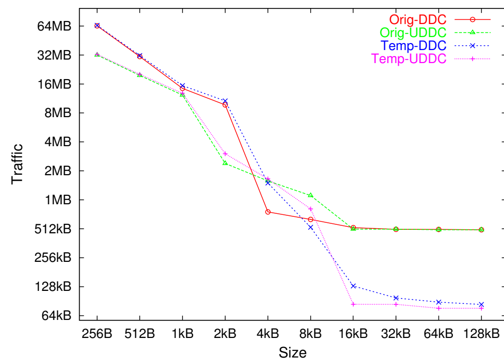
In the previous chapter, the performance of the original prediction algorithm, as proposed by González et al., was shown to be disappointing for a number of cases. Although the amount of traffic could be reduced for some combinations of benchmarks and cache sizes, no guarantee exists that the DDC or the UDDC would produce less traffic than a conventional direct-mapped cache with 32 bytes per cache line. In at least one case, a tremendous increase in traffic was shown for the DDC and the UDDC. Furthermore, in Section 3.4, we have shown that a random algorithm could, in some cases, reduce



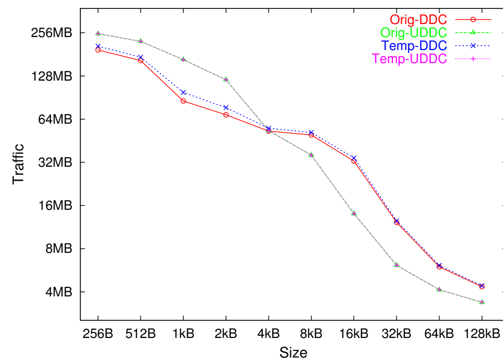
(a) adpcm-enc



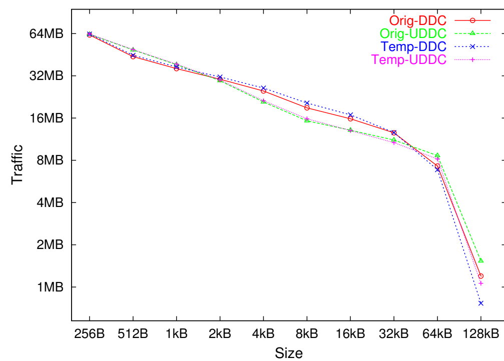
(b) g721-dec



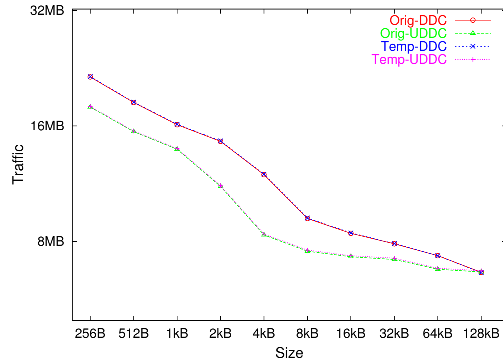
(c) gsm-dec



(d) mpeg2-dec



(e) pegwit-enc



(f) unepic

Figure 4.2: Amount of traffic produced by (U)DDCs with different initial prediction.

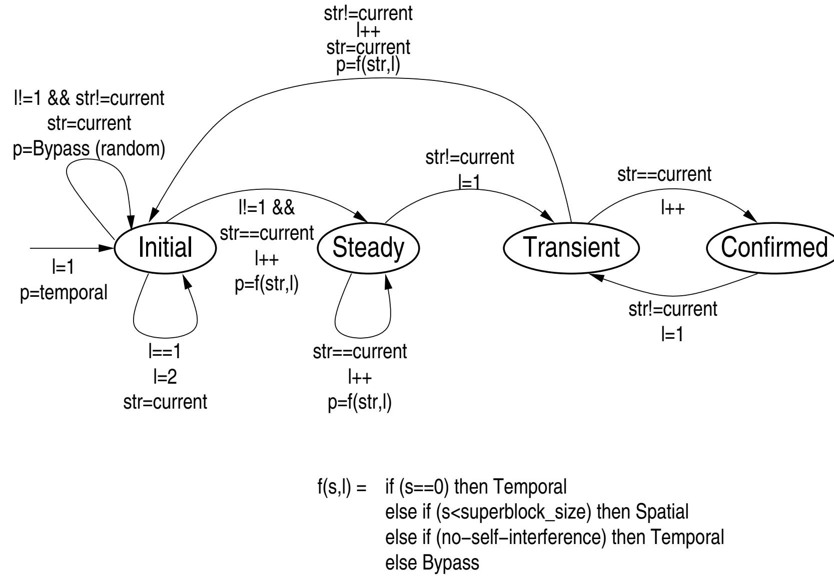


Figure 4.3: The proposed 4-state FSM.

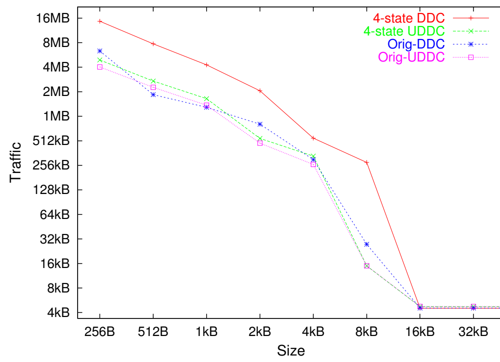
the amount of traffic as well as the algorithm by González et al. Several aspects of the original prediction algorithm have been shown to be ineffective. This has led us to redesign the prediction algorithm, which will be examined in this section. The 4-state FSM of this new algorithm is depicted in Figure 4.3.

Figure 4.3 shows the state transitions of the proposed algorithm. This finite state machine is largely based on one proposed by Juurlink [11]. This design can be explained as follows. First, we choose to always use the prediction field, in contrast to the original FSM where this field is only used in *Steady* state. Since we found that an initial prediction of *bypass* could be the source to an enormous increase in traffic, and since overall caches with a line size of 4 bytes save more traffic than ones with 32 bytes per line, the initial prediction will be *temporal*. The prediction will remain in *Initial* state, until two consecutive strides are equal. Note that if the length of the stride L is 1, there is no previous stride yet, and the prediction will remain in *Initial* state with the original prediction. If the stride keeps changing, the prediction will become *bypass*. Since the only possibility to leave this *Initial* state is when the same stride is detected twice in a row, the following state will always be *Steady*. This *Steady* state will be kept as long as the same stride is detected. On transition to and while remaining in the *Steady* state, the prediction will be determined by the $f(S, L)$ function. This function is different from the function with the same name in the original FSM, but it has some similarities. Most noticeable, it includes the *self-interference* detection scheme. Note that González et al. only used this function on the end of a stride, whereas we will use it every time the prediction field is updated. This allows us to detect self-interference earlier and start bypassing the cache. If the stride is unequal to the previous one, and the current state is *Steady*, the new state will be *Transient*. On this transition, only the length field L is updated. The stride and prediction fields are left unchanged. For functions that operate on multiple rows, arrays

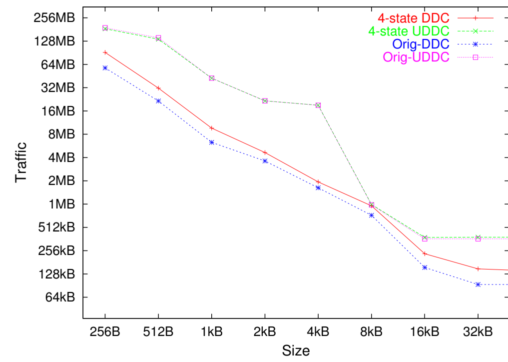
or other indexed data structures, the constant stride is interrupted when the end of one row is reached and operation is switched to the following one. On such an event, the prediction for the first row may well satisfy for the latter. This is implemented in our FSM by the *Confirmed* state. If, in *Transient* state, we detect a stride that is equal to the stride in *Steady* state, the new state will be *Confirmed*. In this state, nothing is done except for checking whether or not the stride remains the same value. If this is not the case, the FSM will go back to the *Transient* state. In the *Transient* state, the stride may also show to be different from the earlier one in *Steady* state. In this case, the new state will become *Initial* again, the stride is updated, and the prediction is updated according to $f(S, L)$.

Figure 4.4 shows the results, obtained with the new prediction algorithm. In this figure, `4-state DDC` and `4-state UDDC` denote respectively the DDC and the UDDC (type A), using the newly proposed prediction algorithm. For comparison, the same caches using the original prediction are also drawn in this figure. Clearly, this new prediction algorithm does not perform as well as expected. In fact, the only major improvement to the original algorithm is with the `gsm-dec` benchmark. As was already shown in Section 4.2, the initial prediction of the original algorithm is the main reason for the disparity in traffic in this case. Furthermore, it can be noted that for both the DDC and the UDDC, the two different prediction algorithms result in an about equal amount of traffic for almost all benchmarks and sizes. When these new caches are compared with the original ones where the initial prediction set to *temporal*, depicted in Figure 4.2, hardly any difference in traffic can be detected. This shows that the small differences between corresponding caches in Figure 4.4 are mainly caused by the different initial prediction.

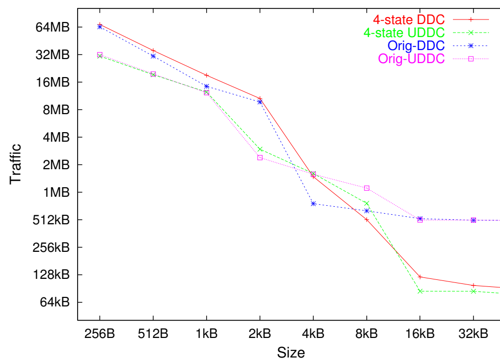
From these results, we conclude that the effect of using one of the above prediction algorithms is mainly based on the decision to predict *spatial* for small strides and *temporal* for larger ones. Overall, smaller blocks produce less traffic than larger ones, as was shown in Chapter 2. Therefore, any prediction algorithm that predicts *temporal* in a substantial number of cases can be employed to reduce the amount of traffic. Using separate caches to store differently predicted items, as is done for the DDC, can eliminate conflict misses and, therefore, reduce the amount of produced traffic.



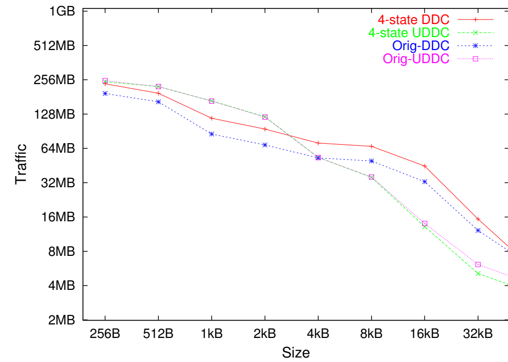
(a) adpcm-enc



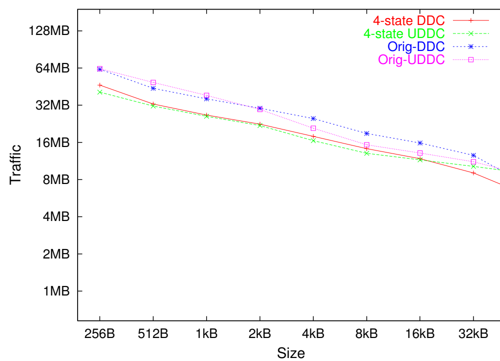
(b) g721-dec



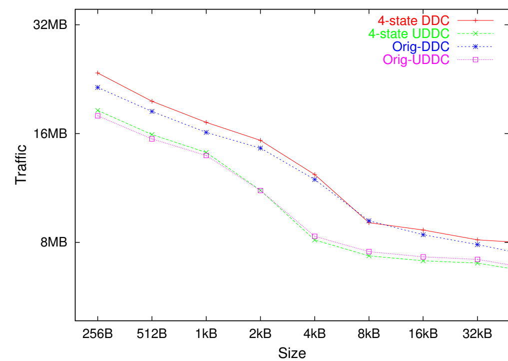
(c) gsm-dec



(d) mpeg2-dec



(e) pegwit-enc



(f) unepic

Figure 4.4: Amount of traffic produced by (U)DDCs using the 4-state FSM.

5 Reducing Conflicts

As explained in Chapter 2, some multimedia routines may access memory in such a way that the capacity of direct-mapped caches is poorly used. The distance between the data addresses of two consecutive executions of a load/store instruction can be large, reducing the possibility to exploit spatial locality. The most common example of this is the column-wise traversal of a large matrix. Since the used data structures can be quite large, a lot of conflict misses are very likely to occur if a cache lacks associativity. Besides causing *cache pollution*, instructions that access large vectors can also eventually interfere with themselves. The conventional direct-mapped cache may therefore refresh caches lines again and again without ever reusing the data. This process is very likely to happen in applications that operate on large blocks of data, such as in the field of multimedia.

González et al. [6] included a mechanism in their design of the dual data cache, which functions as follows. If the stride changes while in steady-state, the prediction is updated if the previous stride was to big to catch any spatial locality. From the length and stride fields it can be determined if this structure will fit in the cache or not. If it will not fit, it is bypassed on future references, as it otherwise may interfere with itself. Otherwise, the prediction will become temporal.

While this method may resolve conflicts, it may also wrongly predict to bypass the cache. The fact that a cache will not fit all references made by a certain instruction does not imply that caching would be detrimental. In Section 4.2, it was shown how predicting *bypass* can turn out to be very inefficient the amount of traffic, although in this case it was not caused by this method.

It is clear that data that is accessed with a small stride should always be cached in a way to exploit spatial locality. If the stride is too large to capture multiple references in one line, however, caching may still be beneficial, as it is still possible that this or even other instructions use the cache line again. This can, however, not be determined without information being shared among different instructions. In other words, it would require the information to be accessible though data addresses instead of instruction addresses, like in the *Spatial Locality Detection Table* proposed by Johnson et al. [10]. This is, however, beyond the scope of this thesis.

If an instruction makes multiple references to the same line in the cache, interference with another instruction can be detected by the simple fact that we expect a cache-hit on the second and further accesses to the same line by this instruction. If the line is not in the cache, while it has been cached during the previous execution of this instruction, it is clearly conflicting with another instruction. In the following section, we will examine two caches that employ this method to detect conflicts.

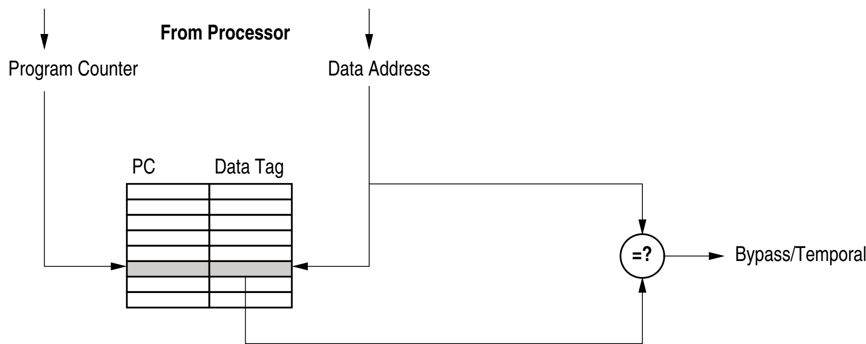


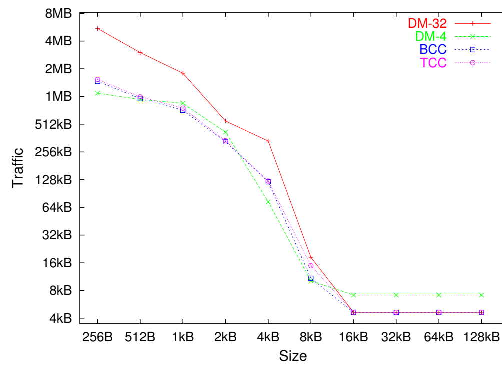
Figure 5.1: Diagram of the conflict detection principle.

5.1 Bypass/Temporal in Case of Conflict

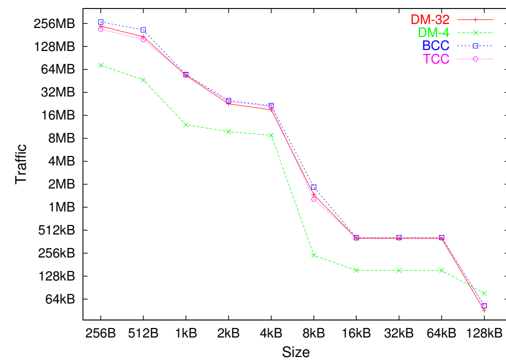
As an example of how conflicts may be detected and resolved using the method described above, we propose two caches. The first cache will be called the *Bypass in Case of Conflict* cache (BCC). This cache operates like a normal direct-mapped cache with the additional possibility to bypass the cache. When an access is to bypass the cache, only an amount of bytes equal to the transfer size to memory are fetched. It has a prediction table which only records instruction and data addresses. The prediction table is accessed at the same time as the normal cache unit and an entry is allocated to the instruction if it is not found in this table. If the instruction address is found in this table, the data address in the corresponding entry of this table is updated and compared to the data address that is currently requested, to detect if these are contained in the same cache line. If this is the case, we expect the currently referenced data to be already available in the cache, since it has been fetched last time this instruction was executed. From this, it follows that if a cache miss is encountered, the requested data item has been replaced by another instruction. To resolve this conflict, the current reference will not be cached. As long as the instruction accesses the same cache block, all accesses by this instruction will bypass the cache. When eventually another block is being accessed, again the whole cache block is fetched and allocated in the cache.

The second cache organization we propose, is based on the same principle. This cache, however, employs *sub-block* caching, like the unified dual data cache (type A). It employs the same prediction table as the BCC cache. The only difference is that when a conflict is detected, this cache will fetch a small sub-block and allocate it in the cache. Since, in this way, temporal locality can still be used, this cache will be called the *Temporal in Case of Conflict* cache (TCC).

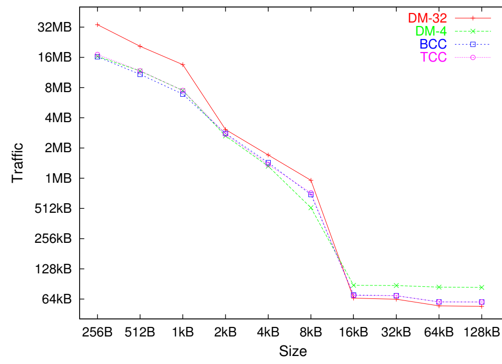
Both proposed caches employ a structure as depicted in Figure 5.1. On every access, this structure send 1 bit of information to the cache. On a cache hit, this information is simply ignored, since no data will be retrieved from memory. Whenever a cache miss is detected, the cache consults this bit on if or how this item should be cached. In related work by Tyson et al. [16], a dynamic cache model is proposed that uses a 2-bit counter to record the hits and misses for different instructions. Since the prediction table can be



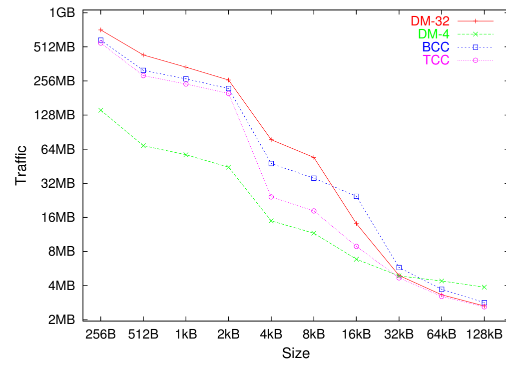
(a) adpcm-enc



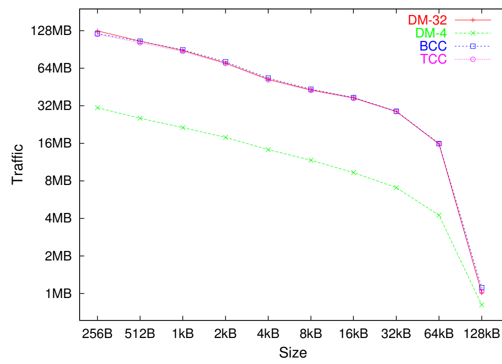
(b) g721-dec



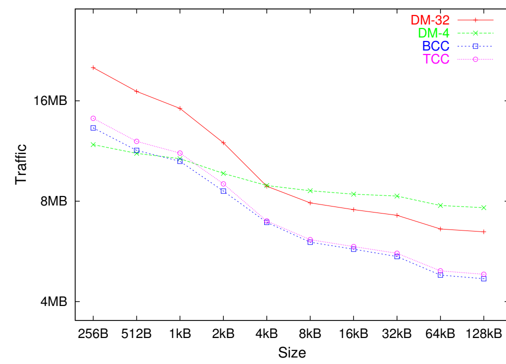
(c) gsm-dec



(d) mpeg2-dec



(e) pegwit-enc



(f) unepic

Figure 5.2: Amount of traffic for different benchmarks.

updated without information from the cache and since the produced information is only used on a cache miss, using this proposed method does not increase the cycle time.

In Figure 5.2, the results of the *Bypass in Case of Conflict* cache and the *Temporal in Case of Conflict* cache are depicted, labelled `BCC` and `TCC` respectively. For reference, a direct-mapped cache with a block size of 32 bytes (`DM-32`) and one with a block size of 4 bytes (`DM-4`) have been included in these figures. In these experiments, a direct-mapped prediction table of 128 entries was used.

From Figure 5.2 it can be seen that, overall, the amount of generated traffic can be reduced by using the *Bypass in Case of Conflict* or the *Temporal in Case of Conflict* cache. Sometimes, however, these caches produce more traffic than the direct-mapped cache with a line size of 32 bytes. Except for the `mpeg2-dec` benchmark in Figure 5.2(d), these increases are not significant. In fact, only in one case, with the `mpeg2-dec` benchmark and the BCC cache of 16kB, a significant increase in traffic is measured. Using the TCC cache, where a sub-block is cached if a conflict is detected, never increases the amount of traffic significantly.

For large capacity caches in general, and with some benchmarks like `pegwit-enc` also for smaller ones, the amounts of traffic produced by these new caches and the direct-mapped cache are about equal. Especially for smaller caches and in case of `unepic` also for larger ones, the BCC and the TCC cache can save a significant amount of traffic, compared to the direct-mapped cache with 32 bytes per line.

Overall, the *Temporal in Case of Conflict* cache is the cache that produces that least amount of traffic, in these experiments. It can be concluded that usage of this technique to avoid repeating conflict misses, can significantly reduce the amount of produced traffic.

6 Conclusions and Directions for Future Work

In this thesis we have shown that conventional direct-mapped cached can waste a significant amount of traffic. The inefficiency of these cache varies among different benchmarks and sizes and is mainly caused by the lack of associativity. Furthermore, it has been shown that inefficiency of a direct-mapped cache can depend very much on its capacity. Most benchmarks showed at least one point in the range of tested sizes, where an increase in cache size by factor 2 resulted in a decrease in traffic by a significantly larger factor. Also, most benchmarks also showed to produce an equal amount of traffic for some caches of different sizes.

Adaptively varying the block size, such as with the *Dual Data Cache* and the *Unified Dual Data Cache*, can in principle reduce the amount produced traffic, compared to direct-mapped caches. The prediction algorithm, as proposed by González et al., does not always produce good predictions, however. If loop-unrolling is employed, the amount of traffic produced by these caches can increase significantly. Furthermore, if the prediction field is initially set to *bypass*, this can cause a significant increase in traffic. An initial prediction of *temporal* has also showed to hardly produce more traffic, while costly incorrect predictions are avoided. Simulations of the same cache structures with a random prediction algorithm have shown that a large part of the saving made by the *Dual Data Cache* is, in fact, due to the implicit associativity of this cache.

While we did not succeed in producing a more efficient prediction algorithm, we have shown that the prediction algorithm by González et al. can be improved in several ways. Furthermore, it must be noted that, in general, smaller blocks have shown to produce less traffic. Caches with small blocks, however, demand a significant amount of space on the chip to store the tags. This makes smaller blocks less preferable. Consequently, the *Dual Data Cache* requires significantly more space than a direct-mapped cache or the *Unified Dual Data Cache* (type A) of the same capacity.

The main advantage of the *Dual Data Cache* over the *Unified Dual Data Cache* is the implicit associativity of the first. In the *Unified Dual Data Cache*, stored spatial data is easily polluted with temporal data. Furthermore, the *Unified Dual Data Cache* was not allowed to write an arbitrary amount of dirty words back to the memory, hence more bytes were marked ‘dirty’. Savings may be made for this cache, by adding dirty bits and allowing the cache to write only those words back to memory that are in fact dirty.

A simple method to detect conflicts in a direct-mapped cache has been proposed. Two caches that employ this detection method were proposed: the *bypass in case of conflict* cache and the *temporal in case of conflict* cache. We have shown that number of conflict misses can be reduced by using these mechanisms.

Finally, we like to note that, when simulating applications, input and output buffers should be treated with special care. If either input or output is of significant size, reading from standard input or writing to standard output may yield incorrect results, since in

practice, this is possibly implemented differently.

In future work, a comparison could be made with methods that predict using data addresses instead of instruction addresses, like the *MAT/SDLT* by Johnson et al. We also believe that a more efficient prediction algorithm can be constructed for the *Dual Data Cache* and *Unified Dual Data Caches*.

Furthermore, the result obtained in this thesis could be extended to include detailed delay calculations. Also, many applications from the *MiBench* benchmark suite have not yet been tested.

The introduced conflict detection mechanism could be implemented in any cache that has more than one option on a cache miss, for example *bypassing* or *sub-block caching*. For all such cache designs, experiments can be performed to show the impact of employing this conflict detection mechanism.

Bibliography

[1] S. Anantharaman and S. Pande, *An Efficient Data Partitioning Method for Limited Memory Embedded Systems*, Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems, 1998, pp. 205–218.

[2] L.A. Belady, *A Study of Replacement Algorithms for a Virtual Storage Computer*, IBM Systems Journal **5** (1967), no. 2, 5:78–101.

[3] D. Burger, J.R. Goodman, and A. Kägi, *Memory Bandwidth Limitations of Future Microprocessors*, Proc. Int. Symp. on Computer Architecture, 1996, pp. 78–89.

[4] F. Catthoor, F. Franssen, S. Wuytack, L. Nachtergaele, and H. De Man, *Global Communication and Memory Optimizing Transformations for Low-Power Signal Processing Systems*, VLSI Signal Processing Workshop, 1994.

[5] P.J. de Langen and B.H.H. Juurlink, *Off-Chip Memory Traffic Measurements of Low-Power Embedded Systems*, Proc. ProRISC Workshop on Circuits, Systems and Signal Processing, 2002.

[6] A. González, C. Aliagas, and M. Valero, *A Data Cache with Multiple Caching Strategies Tuned to Different Types of Locality*, Proc. Int. Conf. on Supercomputing, 1995, pp. 338–347.

[7] M.R. Guthaus, J.S. Ringenberg, D.Ernst, T.M. Austin, T.Mudge, and R.B. Brown, *MiBench: A free, commercially representative embedded benchmark suite*, IEEE 4th Annual Workshop on Workload Characterization, 2001.

[8] J.L. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach*, 2nd ed., Morgan Kaufmann Publishers Inc., 1996.

[9] L. P. Horwitz, R. M. Karp, R. E. Miller, and S. Winograd, *Index Register Allocation*, Journal of the ACM (JACM) **13** (1966), no. 1, 43–61.

[10] T.L. Johnson, M.C. Merten, and W.W. Hwu, *Run-Time Spatial Locality Detection and Optimization*, Proc. Int. Symp. on Microarchitecture, 1997, pp. 57–64.

[11] B.H.H. Juurlink, private communication, 2002.

[12] ———, *Unified Dual Data Caches*, Proc. EUROMICRO Symp. on Digital Systems Design, Sept. 2003, To appear.

[13] Ch. Lee, M. Potkonjak, and W.H. Mangione-Smith, *MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communicatons Systems*, Int. Symp. on Microarchitecture, 1997, pp. 330–335.

[14] P. Petrov and A. Orailoglu, *Performance and Power Effectiveness in Embedded Processors - Customizable Partitioned Caches*, IEEE Trans. on CAD **20** (2001), no. 11, 1309–1318.

[15] Olivier Temam, *Investigating Optimal Local Memory Performance*, Architectural Support for Programming Languages and Operating Systems, 1998, pp. 218–227.

[16] G. Tyson, M. Farrens, J. Matthews, and A. Pleszkun, *A New Approach to Cache Management*, Proc. Int. Symp. on Microarchitecture, Nov. 1995.

[17] Y. Yamada, T. Johnson, G. Haab, J. Gyllenhaal, W. Hwu, and J. Torrellas, *Reducing Cache Misses in Numerical Applications Using Data Relocation and Prefetching*, Technical report crhc-95-04, Center for Reliable and High Performance Computing, Apr. 1995.

Curriculum Vitae

Pepijn de Langen was born in Groningen, the Netherlands in 1976. He studied Electrical Engineering at the Delft University of Technology, where he plans to graduate with a Master of Science degree in Computer Engineering in August 2003. He qualified 3 times for the Northwestern European Regional Contest of ACM's International Collegiate Programming Contest. His research interests include, but are not limited to, computer architecture, algorithms, computer graphics, and parallel computing. Pepijn is a member of the ACM.