Instruction cache for the C-processor

Bormans, J.E.H.M.

Award date:
1989

August 1989.

Master thesis report:

Instruction cache for the C-processor.

By: *J.E.H.M. Bormans*

Supervisor: *Prof. ir. M.P.J. Stevens*

Coaches: *ir. F.P.M. Budzelaar*

ir. W.J. Withagen

Eindhoven University of Technology,

Department of Electrical Engineering,

Digital Systems Group.

Abstract

This master thesis is the result of my graduation project at the Digital Systems group of the Department of Electrical Engineering of the Eindhoven University of Technology. A study is made on instruction caches. Caches are used to improve the processor performance at the cost of a small quantity of fast memory and the associated control logic. This study is targeted towards an instruction cache needed for the "C-processor", which is in development at the Digital Systems group, but has a very general character. Different instruction cache organizations, replacement algorithms, prefetch algorithms, are studied, as well as the effects of changing the cache, the block or the transfer block size. A read and a fetch buffer are introduced to reduce the average access time to the cache. The buffers also reduce the number of accesses to the cache RAM, which decreases the necessity of dual-ported RAM.
Trace driven simulations are necessary to design caches in a well-considered manner. A simulator has been written, which takes timing parameters into account. This is especially important when prefetching and/or a fetch buffer is used. Traces needed for these simulations, however, were not available due to the lack of a compiler for the C-processor. Therefore, a trace generator has been written to produce artificial traces. Simulation results based on these traces can be used to indicate tendencies. To draw more well-founded conclusions, however, real traces are needed.
This work will be continued by Yun Chao Hu. He is going to develop an instruction cache generator, which includes many of the features discussed in this report.

Acknowledgements

In the first place I would like to thank my coaches ir. Frank Budzelaar and ir. Willem Jan Withagen, and professor Stevens. They supported and encouraged me, but gave me also a large freedom in this graduation project. I appreciated this very much. I also would like to thank everyone else who contributed to this work. Especially thanks to ir. Jean-Paul Smeets, with whom I had numerous valuable discussions.

Table of contents.

1. Introduction.

Improvements in very large scale integration technology have made it possible to produce extremely high-performance microprocessors. However, the rate at which data can be supplied to a processor has become a limiting factor, due to pin count constraints and a relative less progress in the cost versus speed ratio of memory. Memory hierarchies can be used to obtain a higher bandwidth at the cost of just a small quantity of fast memory and its associated control logic. Cache memories can be used to level out the speed mismatch between the processor and the main memory. The effectiveness of a cache is based on the property of locality, both temporal and spatial, in the reference pattern. Temporal locality refers to the fact that requested information is likely to be requested again in the near future. Spatial locality refers to the fact the probability of reference is not uniformly divided over the address space, but is higher for information located near the current request. Caches are nearly transparent in the architecture of the processor. No compile-time effort is thus necessary to exploit them in an effective way, in contrast to single or multiple register sets. The advances in the integration circuit density have made it possible to implement a cache on the processor chip. This way the on-chip bandwidth can be as high as necessary. Even when the on-chip cache is carefully designed to limit the off-chip references, it may be necessary to add an extra off-chip memory hierarchy level in order to increase the off-chip bandwidth on a cost-effective base.

At the Digital Systems group, of the Department of Electrical Engineering of the Eindhoven University of Technology, a processor is being developed which is intended for effective execution of programs written in the language C, and related high level languages. Separate instruction and data caches are integrated in this "C-processor". As part of my graduation project at the mentioned department, a study is made in order to implement an instruction cache in an effective way.

The backgrounds of the C-processor project and the modules of the C-processor which are of interest for the instruction cache will be treated in the next chapter. The third chapter will be used to explain the general theory of instruction caches. In the fourth chapter a number of aspects will be studied more closely and based on this some new features will be introduced. A simulator is necessary to investigate these features, cache organizations, cache algorithms, etc. This simulator is described in the fifth chapter. The results of the simulations are presented in chapter six. The final chapter contains conclusions and recommendations for future research.

2. Backgrounds and specifications.

2.1. The C-processor project.

As part of a project to get a better and more systematic grip on the design aspects of digital systems, a number of "test" design projects have been started at the Digital Systems group of the Department of Electrical Engineering of the Eindhoven University of Technology. The aim is to retrieve common design steps and methods in order to automate, or at least partly automate, these steps.

One of these test designs is the C-processor. The C-processor project was started by F.P.M Budzelaar [Budzelaar] in 1987, coached by Prof. ir. M.P.J. Stevens. Besides the design philosophy in general, he investigated the requirements to execute compiled C-programs effectively. Based on this, he specified a software definition of the processor and the global processor architecture. He decomposed the first level of the design in functional blocks and worked out some of them in more detail. The instruction set and the high level decomposition are further elaborated by W.J. Withagen [Withagen]. He wrote two high level models, each for a specific task: A model to test the instruction set and its functionality and a model which describes the decomposed modules at a high hardware level. This high level hardware description can be used as a testing vehicle for future decompositions and gate level implementations.
A number of people are currently active on different fronts of the C-processor, both on the implementation of the functional blocks as on software topics, like compilers and debuggers.

Although this thesis is primarily intended for the instruction cache of the C-processor, it has a rather general character. Therefore, it can also be used as guideline for example a cache generator, comparable to already existing RAM generators, PLA generators, etc. The link to the main project will be clear.

2.2. The C-processor.

The parts of the C-processor which are of interest for this thesis are depicted in Figure 1. At the main memory side the instruction cache is connected to the bus unit and the memory management unit, at the side of the processor to the instruction unit. These units are discussed in the next sections.

2.2.1. The bus unit.

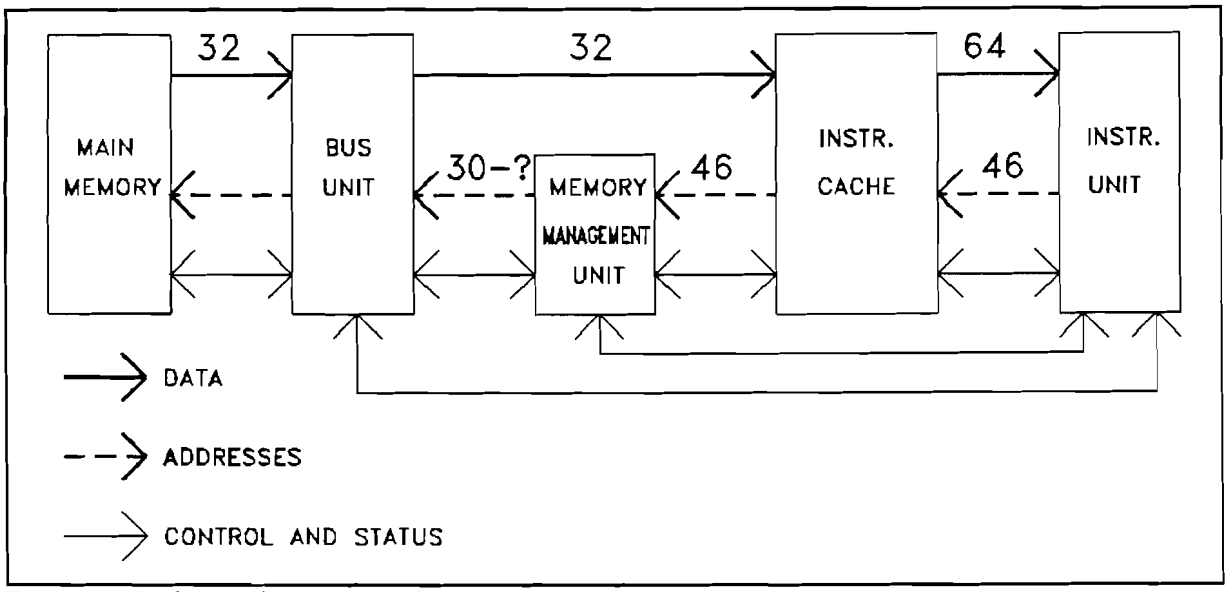

Figure 1: The instruction cache environment.

The bus unit is the interface to the outside world. Besides requests from the instruction cache, it receives requests from the data cache. The bus unit is able to serve these requests simultaneously. There are separate buses to the outside world for instructions and data. (*Harvard organization.*)
The instruction bus is 32 bits wide. The quantity of 32 bits will be called *quads* from now on.

All signals coming from or going to the outside world go through the bus unit for adaption of the signal levels. The bus unit takes also care of the external bus control.

2.2.2. The memory management unit.

The C-processor is able to address 4 gigabyte. Since quads instead of single bytes are addressed, 30 bits are required for an address. The available amount of RAM, however, will be less than needed for the complete address space. The major part of the information will be stored on magnetic, or in future on optical, storage media.

This requires a *memory management unit*, which, among other things, has to translate the *virtual addresses* into *real* (or *physical*) *addresses*. The memory management unit receives addresses of 46 bits. The 16 most significant bits are *process identification bits*. The purpose of these bits will be explained in paragraph 2.4.

The memory management unit is placed at the main memory side of the cache. The cache thus uses virtual addresses and is therefore called a *virtual address cache.* Most caches are *real address caches*. These caches have the disadvantage that for each cache reference the virtual address has to be translated into the real address. Although some parallelism is possible for this translation and the location of data in the cache, the cache access time is increased by this translation. To reduce the cache access time, a virtual address cache is used. An address translation is only required after a miss. If the instruction and data cache have separate memory management units, or when the memory management unit is able to serve both simultaneously, this translation can be speeded up by translating the virtual address as soon as the cache receives a request. In that case the physical address is already available when a miss occurs. A more sophisticated approach is introduced in [Wood], where parts of the memory management unit are integrated in the cache to achieve in-cache address translation.

If the memory management unit receives a request to a page which is not in the physical address space, it runs an operating routine on the processor, e.g. to swap in a page from harddisk. Since this routine will use the cache itself, a status signal at the main memory site of the cache is necessary which indicates that the requested information is not available. Otherwise the cache will wait until a data valid signal is given and the operating routine never can be run.

2.2.3. The instruction unit.

A 64 bits bus is used to transport the instructions between the cache and the instruction unit. The available bandwidth between the cache and the instruction unit is thus twice as large as the bandwidth between the main memory and the cache. The necessity of the 64 bits bus towards the instruction unit will be clear from considering that many instructions will be longer than 4 bytes. The 32 bits connection with the outside world is imposed by pin constraints.

Instead of specifying the address of the 64 bits, the instruction unit specifies the address of the first of the two quads. The cache has to deliver the quad at the specified address and the quad at the next address. By allowing the 64 bits to be quad aligned instead of 64 bits aligned, a more efficient data transport is possible between the cache and the instruction unit.

The C-processor is highly pipelined in order to be able to execute almost all instructions in one clock cycle. The double quad is shifted in a buffer before the instructions are decoded. The bytes ahead of the current instruction will be discarded, the bytes behind it will be used, in case no jump occurs, for the next instruction(s). It is, however, also

possible that the instruction is longer than 8 bytes and two requests are necessary before it is completely in the buffer of the instruction unit.

The first of the pipeline stages is intended for the delay from the instruction cache. As long as the cache is able to service requests from the instruction unit within one clock cycle, the processor will not be delayed.

The number of clock cycles between the current and the next request for the cache thus depends on:
- the time required by the cache to deliver the two quads.
- the number of the instructions contained in the previous and current double quad.
- the execution time of these instructions.

2.3. Separate versus combined caches.

A separate instruction and a separate data cache are used instead of one single cache. This is because of:
- The total cache bandwidth increases, since the two caches can work parallel.
- Both caches can be tailored to the specific instruction or data reference patterns.
- In contrast to a data cache, a main memory update algorithm does not have to be implemented for an instruction cache. (No self-modifying code is assumed.)

To prevent that some information is stored in both caches, a compiler is required which ensures that data and instructions are stored in separate blocks. (As we will see, large blocks instead of single items are stored in caches.) Immediate operands are to be considered as parts of instructions.

A disadvantage of two separate caches is that the storage size for instructions and data is fixed. One homogeneous cache would use the cache memory more efficient, since the amount of instruction and data storage could be adapted to the optimum partitioning for a certain program.

2.4. Task switching.

The C-processor is likely to be used in a multiprogramming environment, so the process running on it can alternate every few milliseconds. The contents of a cache can differ from main memory after a task switch, since a new program may have been loaded in main memory.

After a task switch most caches are *flushed*, by invalidating the data. This has the disadvantage that when a process is restarted, all information belonging to it is lost, although the process by which it was interrupted may have used just a small part of the cache.

A possible solution to this problem is *working set restoration*. Before a task switch is executed, a record is made of the contents of the cache. The contents, or the most recently used part of it, is restored when the process is restarted. Although it will decrease the penalty of task switching, it is unlikely to be used in caches, due to the complexity of implementing it.

Another possibility is splitting the cache in two or more parts. One could for example use an *user* and *supervisor* cache, since many task switches occur due to switching between user and supervisor state. Obvious improvement is however doubtful, because the effective cache size for one process is smaller.
Besides this, cache coherence problems occur, since the information shared by the user and supervisor process may not be entirely distinct.

The strategy chosen for the C-processor is quite different. A *process identification number* is assigned to each process. Each new process gets a process identification number which is one higher than that of the last new process. The process identification numbers of ended processes are not reused. Therefore, a relative large number of 16 bits is used for it. The process identification bits are added to the addresses and form the most significant part. The address of a quad thus consists of 46 bits (16 + 32 - 2).

This way the necessity of a cache flush after each task switch is removed. Theoretically the cache should be flushed before process identification number 0 is assigned again after 2^{16} new processes. If this is really necessary is doubtful, but, on the other hand, it will not lower the processor performance noticeably and the logic to invalidate the data is necessary anyway, since the cache has to be flushed after a processor reset.

The improvement obtained by using process identification numbers is highly influenced by the mean number of references between task switches and the cache size.

3. Theory of cache memories.

This chapter contains an overview of the theory of instruction caches. Appendix A contains a list of the used literature. Especially the survey about cache memories by A.J. Smith has often been used [Smith_A.82]. Aspects which only apply to data caches, like main memory update algorithms, will not be treated. Aspects related to shared-memory multiprocessor systems, like cache coherence, will also not be discussed.
This chapter will concentrate on the main issues of instruction caches, some less important aspects will be treated in the next chapter.

3.1. Optimizing cache design.

Optimizing a cache design has several aspects:

1. Maximizing the *hit ratio*, which is defined as the number of references to the cache that result in a hit divided by the total number of references.
2. Minimizing the cache access time. The cache access time is considered as the critical path of the processor.
3. Minimizing the *miss penalty*, which is the (extra) delay due to a miss.
4. Minimizing the hardware complexity of the cache algorithms.

The following facets are also important, especially for on-chip caches:

5. Minimizing the area.
6. Minimizing the traffic between main memory and cache, since the limiting factor for microprocessors is often the bus bandwidth.

Some of these aspects support each other, others are in conflict. An obvious example is the trade-off between the hit ratio and the cache size. An important but often not recognized trade-off is that between the hit ratio and the cache access time. The reason these aspects are contradictory, will be clear from considering the fact that to increase the hit ratio, a larger cache or a better, and thus more complex, algorithm is required. In both cases the access time is likely to increase.

Besides hit ratio, *miss ratio* is often used. Its definition will be clear.
The *traffic ratio* is often used to indicate the traffic between the main memory and the cache. It is defined as the ratio between the number of quads transferred from main memory to the cache and the number of quads requested by the processor.

3.2. Cache organization.

The organization of a cache determines the way a main memory address is mapped into a cache location. There are three common ways to organize a cache:

1. *fully associative mapped*
2. *direct mapped*
3. *set associative mapped*

To explain these organizations an example will be used. A cache is assumed which is able to store 512 quads. These quads are grouped in *blocks*[1] of 8 quads, for a reason which will be explained further on. As stated above, the memory address consists of 46 bits, from which the 16 most significant are process identification bits.

3.2.1. Fully associative mapped cache.

In a *fully associative mapped* cache the memory address is divided into a *tag field* and a *word field* [2]. Independent from the organization method, the tag field of an address has to be compared against the tag belonging to a block, which determines the original main memory location of the quads in the block. The word field is used to select a quad within a block.

Processing a request in a fully associative mapped cache involves the following steps (see Figure 2):

1. scan the block tags until the tags of the request and the block are identical.
2. check the *status information* of that block. At this point we assume that the status information consists only of a *data_valid* bit.
3. if this data_valid bit is set, the quads of the block are valid and the requested quad can be selected using the word field.

The question whether the remaining blocks should be checked or a miss should be reported in case a block is found with the right tag field, but the wrong status information, depends on details of the placement algorithm.

A block has to be read from main memory and has to be stored in the cache when a miss occurs. In a fully associative mapped cache there is a complete placement freedom and the main memory block can be stored in any cache block. A *replacement algorithm*,

[1] *Line* instead of *block* is sometimes used in literature to indicate such group of data elements.

[2] *Quad field* would be a more suited name for this field, since quads instead of words are selected by it. *Word field*, however, is the usual name for it and so it will be used here.

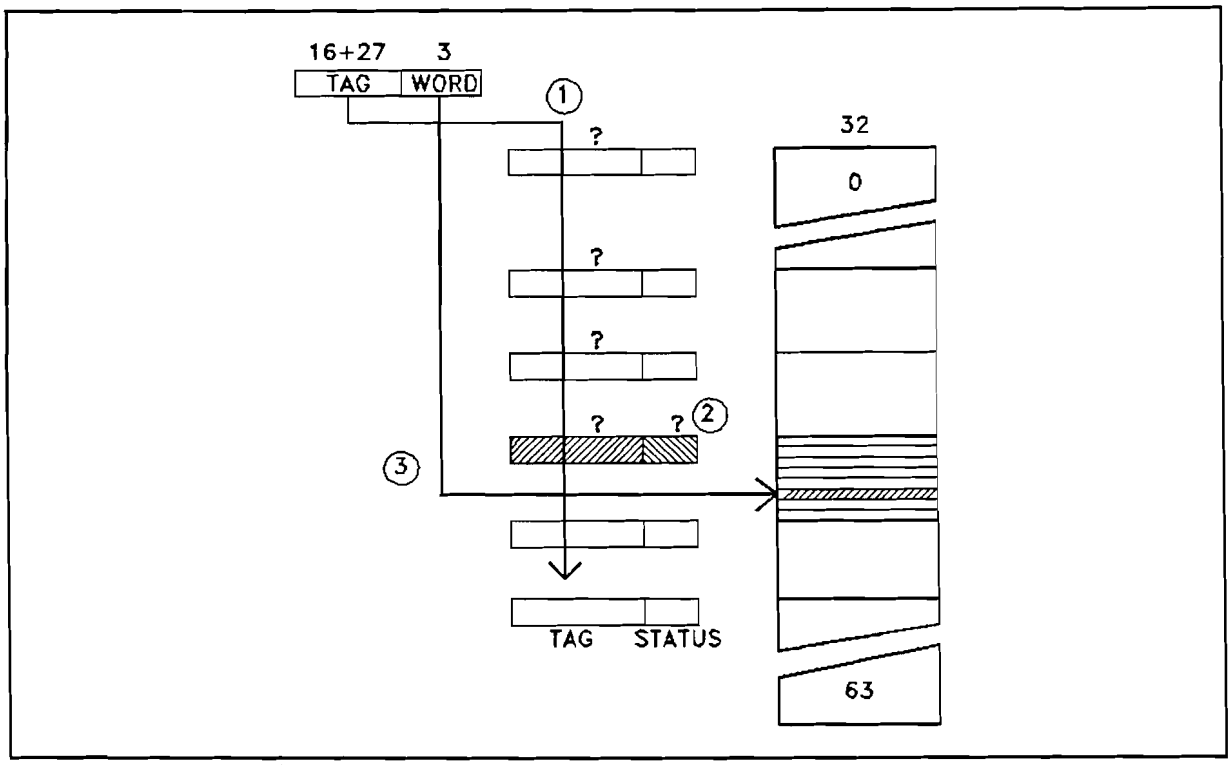

Figure 2: Fully associative mapped cache.

see paragraph 3.5, is necessary to decide which cache block has to be replaced.

If the conventional addressing method would have to be used with regular memory, the following steps would be involved for each block until the right block is found:

- generate the address which will select the tag field of the block.
- read the tag and the status information.
- compare the block tag with the tag of the specified address in a central comparator and check the status information.

Since the conventional addressing method is far too cumbersome, the usage of *contents addressable memory* (*CAM*) is unavoidable in a fully associative mapped cache.

3.2.2. Direct mapped cache.

In a *direct mapped cache* the memory address is divided into three parts, see Figure 3. Besides the tag and word field, there is a *block field*. The following steps can be distinguished in processing a request:

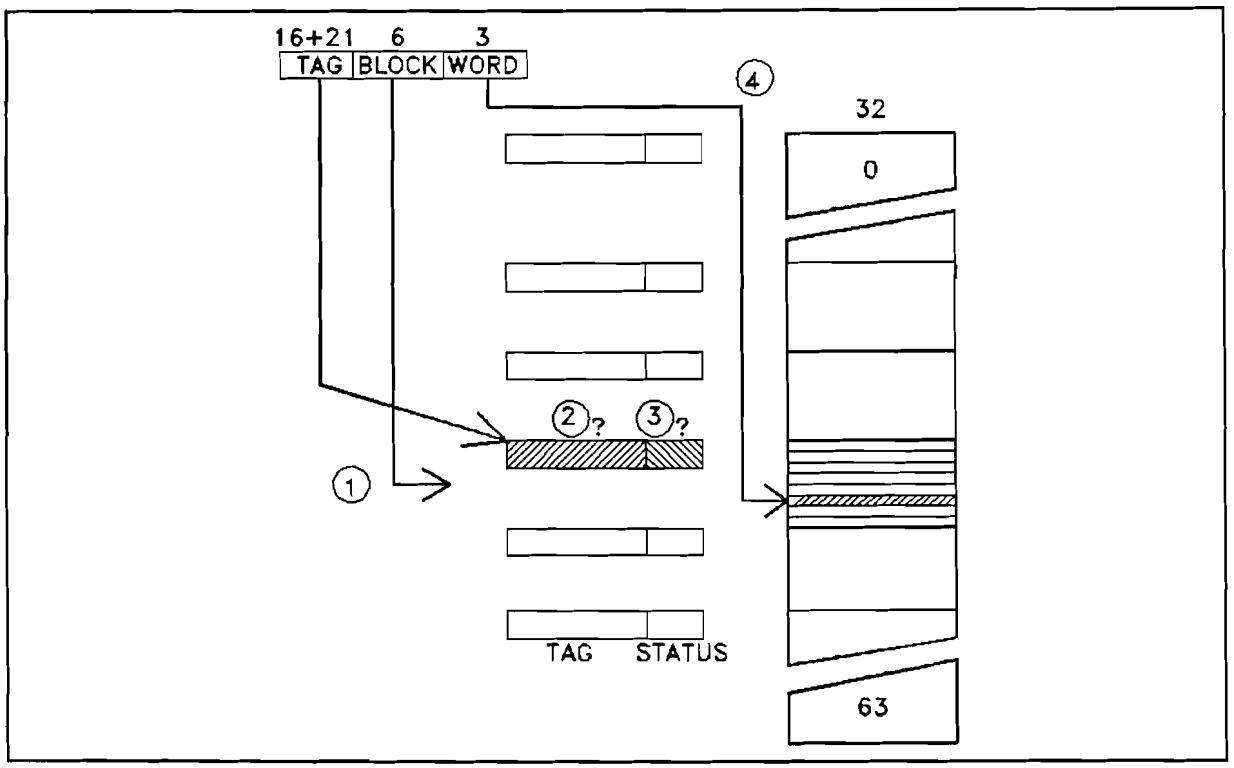

Figure 3: Direct mapped cache.

1. the block field is used as an address[1] and is decoded in the conventional way. It points directly to a block in the cache.
2. the tag of the block is compared against the tag of the request.
3. the status information is checked.
4. if there is a hit, the word field is used to select the quad within the block.

No associative search has to be done, since a main memory block can potentially be found in only one of the cache blocks. This has the advantage of locating a block much faster.

Since a block can only be stored at one place in a direct mapped cache, the block which has to be replaced by a main memory block after a miss, is fixed.
No replacement algorithm is thus necessary as for a fully associative mapped cache. However, although such an algorithm takes some time, a better overall cache performance can be obtained as a result of a better replacement of blocks. Consider, for example, a partly filled cache. In a direct mapped cache a block may have to be removed from the cache after a miss, although there are still empty blocks. A fully

[1] The number of blocks has to be a power of 2.

associative mapped cache does not have this disadvantage (when an appropriate replacement algorithm is chosen).

3.2.3. Set associative mapped cache.

From the preceding sections it is clear that a totally free or a totally fixed placement of blocks both have advantages and disadvantages. An intermediate form of the described methods, *set associative mapping*, combines the advantages and limits the disadvantages.

A memory address is divided in three fields: Tag field, *set field* and word field. See Figure 4:

1. the set field points to the set of cache blocks that may contain the block with the requested quad.
2. the tag field is compared against the tags of the blocks in the set.
3. if one of the tags match, the status information of that block is checked.
4. if there is a hit, the word field is used to select the quad within the block.

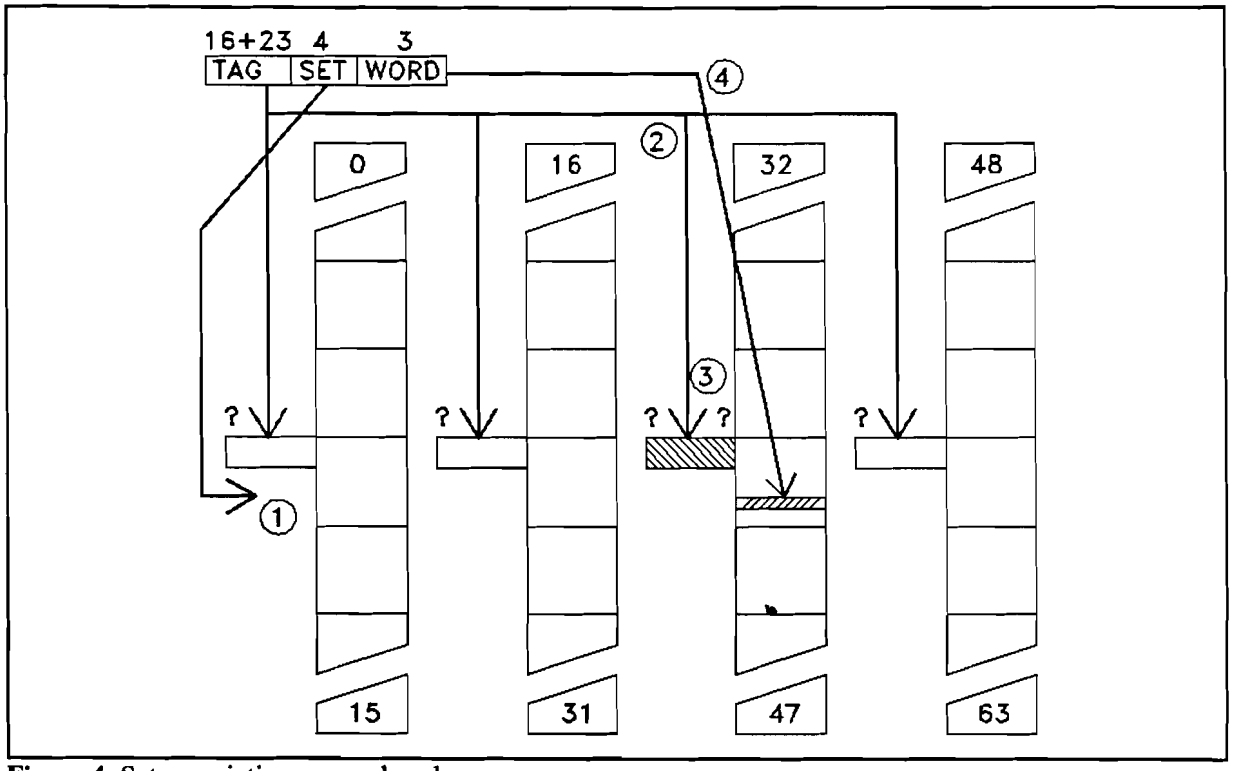

Figure 4: Set associative mapped cache.

In Figure 4 a 4-way set associative cache is drawn, having a set size of 4[1].

Actually, the fully associative and direct mapping methods are special cases of set associative mapping. If the set size is 1, the mapping is direct. If the set size is equal to the total number of blocks, the mapping is fully associative.

As long as the set size is limited, no CAM is required for a set associative cache. In our example the tag (and status) bits of the blocks can be read and checked simultaneously in four separate comparators. These comparators can also be used for the blocks in the other sets, whereas a separate comparator is included for each block in real CAM.

The most simple method to map a memory address into a set number is *bit selection*. This means that the bits of the set fields are used as an address, which is decoded in the conventional way, like a block is selected in direct mapped caches. Some other algorithms have been proposed, which decrease the miss ratio. They have, however, never been used since there is only a slight improvement, whereas the implementation is slower and more expensive. Therefore they will not be discussed here.

3.3. Cache parameters.

Besides the mapping method, the sizes of a number of cache parameters have to be chosen. In this paragraph the cache size, set size and block size will be discussed. *Transfer blocks* will be introduced in the section about the block size. The size of these transfer blocks will be discussed in section 3.3.4.

3.3.1. Cache size.

The area required for a cache can be divided into three parts:
- RAM for data
- RAM for overhead information (tag bits, status information, etc.)
- area required for multiplexers, buses, control logic, etc.

The area required by the RAMs can be estimated easily, an estimation of the last part is more difficult.

From now on "cache size" will be used to indicate the number of quads which can be

[1] The number of "parallel caches" is equal to the number of blocks in a set and not, as sometimes wrongly stated in literature, equal to the number of sets. The number of sets is sixteen in our example.

stored, "cache area size" to indicate the area required for the total cache. In most cases, however, it will be clear from the context.

The cache size is known as the most important cache parameter, since it highly affects both the miss ratio and the cache area size. The larger the cache, the more old information can be saved. This decreases the miss ratio. Since the usefulness of old information declines in time, only a slight improvement of cache performance is found beyond a certain cache size.

3.3.2. Set size.

To study the set size, the cache size and the block size have to be fixed. Hence the set size and number of sets are to be regarded as inversely related.

Set size is just a different term for the scope of associative search. The larger the set size, the lower the miss ratio will be. This is because a larger placement freedom improves the performance of the replacement algorithm. Beyond a certain set size, however, this improvement is likely to increase very little if at all. Therefore the miss ratio will not decrease significant beyond a certain set size.

A smaller set size, on the other hand, results in a faster and less expensive implementation.

If a real address cache had to be developed instead of a virtual address cache, an other consideration would also had to be taken into account.
An improvement in the speed of such a cache could be obtained by overlapping the virtual address translation and the set selection (This parallelism was already mentioned in paragraph 2.2.2.) This is possible because the only bits which have to be translated are those which specify the page address. The remaining bits, which specify the quads within a page, do not have to be translated and are immediately available.
Therefore the bits in the set and word field should be less than or equal to those immediately available bits. This way an upper bound of the number of sets can be obtained, since the block size, and thus the size of the word field, is supposed to be fixed.

3.3.3. Block size.

The reason why quads are grouped in blocks will be made clear in this section.

The main advantage of grouping them is that the amount of the address tags, status information, replacement information, buffer circuitry, etc, decreases. This *block overhead* decreases when the block size is enlarged.

A simple example, in which only the tag and data_valid bits are considered, is used to show this. A direct mapped cache of 512 quads is assumed.

In Table 1 the percentage overhead bits is calculated for different block sizes and the total number of bits is compared to the total number of bits in case the quads would not have been grouped together.

Table 1: Overhead due to tag and data valid bits as function of block size.

block size in quads	no overhead bits	total no bits	overhead in %	% of original size
1	19456	35840	54	100
2	9728	26112	37	73
4	4864	21248	23	59
8	2432	18816	13	53
16	1216	17600	7	49
32	608	16992	4	47
64	304	16688	2	47

If we assume equal sizes of data and overhead bits, the last column of Table 1 can be seen as the saving in RAM area compared to a cache with a block size of one. These area savings can be used to increase the number of data bits of the RAM. Large blocks can thus be used to increase the effective cache size.

The effect of enlarging the block size will sometimes even be larger than shown above, because more overhead bits than tag and data_valid bits are required in some cases, as it will turn out in the following paragraphs. The tag bits are the main part of the overhead bits, but the extra bits required for the LRU replacement algorithm (paragraph 3.5.) are also significant, especially in a fully associative mapped cache. The influence of the other overhead bits is very small.

Besides the decrease in block overhead, there are some other positive effects of enlarging the block size:

- More adjacent data is stored at once, which tends to reduce the miss ratio. *Prefetching* could be, from this point of view, an alternative for large blocks. Prefetching is discussed in paragraph 3.4.
- A block can be read in *burst mode*, which reduces the average main memory access time. If information is fetched in burst mode, it is worthwhile to define the *scaled traffic ratio* besides the traffic ratio. It is equal to the traffic ratio divided by the ratio of the average main memory access time with working in burst mode and without it. The traffic ratio is a criterion for the amount of transported data, the scaled traffic ratio for the required time for it.
- A smaller penalty of cache flushes, which occur in many caches during task switching, is obtained by enlarging the block size. This way the cache will be filled faster.

There are two important disadvantages of enlarging the block size. Firstly, the miss penalty will increase, because a complete block instead of one quad has to be fetched from main memory. Secondly, the usefulness of extra loaded quads becomes less than the usefulness of the replaced quads. Quads probably will be loaded into the cache which never will be used. This effect is called *memory pollution*. Both miss and traffic ratio increase because of memory pollution. Large block should be used from this point of view if most misses result from flushes due to task switching, small blocks if most occur in a completely filled cache (*steady state*). The optimum block size will thus also depend on the rate of task switching.
Another negative effect of enlarging the block size is that the placement freedom will be reduced, since all quads in a block must have the same tag.

There is however a way to limit the increase in traffic ratio and miss penalty, without sacrificing the savings in overhead bits. This is possible by dividing the blocks in smaller *transfer blocks*. This will be discussed in the next section.

It will be clear that the block size has a great impact on the performance of the cache. The size, the access time, the miss ratio and the traffic ratio are influenced by it, some of these because of different effects.

3.3.4. Transfer block size.

In the remainder of the text "blocks" will be used to indicate groups of quads which have the same address tag, "transfer blocks" for groups of quads which are transferred as one

unit between main memory and cache[1].

Each transfer block requires a data_valid bit, since not all transfer blocks of a block have to be stored in the cache. Other status information, like replacement information (paragraph 3.5), is required for a complete block only. Dividing blocks in transfer blocks increases the number of overhead bits thus only very slightly.

Although dividing blocks in transfer blocks is positive for the miss penalty and the traffic ratio, it is not for the miss ratio. The miss ratio will generally be higher, because only a part of a block contains valid data. We have, however, to compare designs with equal areas, rather than designs with equal quantities of data. Since the usage of small transfer blocks permits larger blocks, it could also be considered as a way to reduce overhead bits and thus to obtain a cache which is able to store more quads. From this point of view it will be clear that the miss ratio not necessarily have to be higher. This is studied extensively in [Alpert].
Another disadvantage is that the placement freedom will be reduced. This can be seen by considering that instead of dividing a large block in smaller transfer blocks, a number of small (transfer) blocks are enforced to have the same tag.

If distinction is made between blocks and transfer blocks, an address can be separated in a tag field, a set field, a *transfer block field* and a word field.

3.4. Prefetch algorithm.

There are two ways to fetch a block from main memory:
- *demand fetch*: the block is fetched when it is needed.
- *prefetch*: the block is fetched before it is needed.

By means of prefetching the miss ratio can be reduced considerably.

The following questions should be asked:
- which information should be prefetched?
- how much information should be prefetched?
- when should a prefetch be initiated?

3.4.1. Information to be prefetched.

[1] Besides this combination of names, *blocks* and *sub-blocks*, *address blocks* and *transfer blocks* or *sectors* and *blocks* are often used in literature.

To avoid demand fetches as much as possible, a good prediction is necessary of the information which will be used in the near future. Obviously the subsequent information of the information currently being used should be prefetched.

A simple method to store sequential information is the usage of large blocks. This has, however, many disadvantages as we have seen in paragraph 3.3.3.
A slightly better method is *load-forward* [Hill]: only the part of the block forward of the target of the fetch is fetched instead of the entire block[1]. This limits the traffic ratio, since not needed information is loaded less likely.

Both methods fail if a request is made for a quad outside the current block, and are therefore not to be regarded as real prefetch methods. The simplest real prefetch method is known as *one-block-lookahead*: the subsequent block of the currently referenced one is considered for prefetching. The size of these blocks will be discussed in the next section.

A more advanced prefetch algorithm could try to predict jumps. Different approaches can be used:
- The instructions could be decoded within the cache. The more complex the instruction format is, the more difficult this will be.
- Modifications of base registers could be used to initiate prefetches. This is only worthwhile if there is a certain degree of pipelining in the processor.
- The compiler could generate special instructions for the cache to simplify the prediction of jumps. In that case a cache is not totally transparent anymore.

It must be noted that an optimum prefetch algorithm (100% jump prediction) does not allow a small cache, since the limited bus bandwidth requires a low traffic ratio. This is obtained by reusing the stored information and not by creating such an optimum pipeline. On the contrary, because of the effect of memory pollution, prefetching is increasingly useful with increasing cache size.

Jump prediction, however, will not be used in the prefetch algorithm of the instruction cache, since it is already included in a more effective way in the C-processor. After all, the C-processor is highly pipelined and special techniques are used to minimize the probability of false jump prediction.

[1] Each (transfer) unit of the block requires a separate data_valid bit.

3.4.2. Prefetch block size.

If too much information is prefetched, memory pollution appears as it did for large block sizes. The smaller the cache, the larger this effect will be. Prefetching will always increase the traffic ratio. Another problem is that the storage of prefetched quads will delay the servicing of requests from the processor, unless multiported RAM is used.

These effects can be limited by dividing the blocks into smaller prefetch blocks. The link to the previously introduced transfer blocks will be clear.
It is also obvious that, from of a view of simplicity, no distinction should be made between the sizes of demand fetched and prefetched transfer units. Therefore the calculation of the optimum transfer block size should also be based on the optimum prefetch block size.

If, however, the optimum prefetch transfer block size differs too much from the optimum demand fetch transfer block size, the one-(transfer-)block- lookahead principle may have to be abandoned. In the next chapter it will be investigated if it is rewarding to prefetch multiple transfer blocks instead of the immediate next one only. In that case a smart prefetcher could be used, which fetches near transfer blocks with a high priority and more distant transfer block with a low priority. For the time being, however, we assume that the one-transfer-block-lookahead method is used.

Another suggestion is based on the fact that misses often occur in bursts. When the miss ratio is currently high, the amount of prefetched information should be reduced, since the usefulness of prefetched quads is very low at that moment [Hoevel_83]. This approach could of course also be used for demand fetched transfer blocks [Hoevel_82].

3.4.3. Initiation of prefetches.

The one-transfer-block-lookahead prefetch method can be used in different ways. A simple method is *prefetch_always*: a request to a transfer block implies the prefetch of the next transfer block, regardless whether or not this transfer block is already in the cache. *Prefetch_on_misses* is also very simple: if the reference to a transfer block results in a miss, the next transfer block is prefetched.

A more complicated method is *tagged_prefetching*: if the reference to a transfer block results in a miss, or if it is the first time the transfer block is referenced by the

processor, the next transfer block is prefetched. To accomplish this a *used_before* bit[1] is required for each transfer block. This bit is set to zero when the transfer block is prefetched and set to one as soon as that transfer block is referenced by the processor. If a transfer block is demand fetched, the used_before bit is set to one immediately. A transfer block is prefetched after a demand fetch has been executed, or when a transfer block is referenced which used_before bit was zero.

Besides these, there is a method which uses a different approach. Instead of basing the decision of prefetching on the current request, the cache could be checked for the transfer block to be prefetched. This method is called *prefetch_lookup*. This method is the most effective one, but has the disadvantage of an extra cache access. Such extra accesses are likely to increase the average cache access time.

This method can be refined by also considering the availability of the current transfer block. For example, if it was not in the cache, the cache lookup for the next transfer block could be skipped. To distinguish the refined method from the basic method, *prefetch_lookup_on_hits* and *prefetch_lookup_always* will be used.

If jump prediction would be used by the prefetch algorithm, prefetch_lookup(_always) could be used. In a loop, for example, the transfer block to which is jumped, is likely to be in the cache already.

If small transfer blocks are used as prefetch units, a prefetch for the next transfer block has to be considered as soon as the current transfer block is entered and has to be executed as fast as possible. If large blocks are used on the other hand, circumstances are conceivable, in which the initiation of a prefetch would be better after passing, for example, the middle of the current block.

3.5. Replacement algorithm.

As indicated before, a replacement algorithm is needed in a cache which is not direct mapped. The replacement algorithm has to decide which block in the set will be removed to store a main memory block. This happens when:

- a miss occurs. However, if the block is divided into transfer blocks, replacement is not always necessary. The block to which the missing transfer block belongs, can already be in the cache.

[1] *Tag* bit is the original name, but is not used to avoid confusion with the bits in the tag field of an address.

- a block is prefetched which is not in cache. Or, if transfer blocks are used, when the block to which the prefetched transfer block belongs is not in the cache.

Replacement algorithms can be divided into *usage-based* versus *non-usage-based*, and *fixed-space* versus *variable-space*. Usage-based algorithms take the record of usage of a block into account. Variable-space algorithms vary the amount of space allocated to a specific process. Since the cache size is fixed and a cache is usually too small to hold the working set of more than one process, the replacement algorithm does not lend itself for sophisticated variable-spaced algorithms. Moreover, for the C-processor this is not really under discussion, since process identification numbers are used. Therefore, only fixed-space algorithms will be considered.

The following fixed-space algorithms are known:
- *random*
- *FIFO (first-in, first-out)*
- *LRU (least-recently-used)*

Only the LRU algorithm is usage-based. FIFO is not to be considered as such, because the reuse of a block does not change the replacement status.

LRU is based on the theory that if a block has been often and recently referenced, it is likely to be referenced in the near future. FIFO has the disadvantage that even when a block is frequently and continually used, it will be removed when it becomes the oldest in the set. If the random "algorithm" is used, the blocks are replaced (pseudo-)random. A detailed study about these replacement policies can be found in [Smith_J].

The LRU strategy yields the lowest miss ratio in general. It is, however, not much better than FIFO or random. The fact that random is not as bad as it may seem, will be clear from the next example.

Assume a small associative cache, which is able to store 4 blocks. Assume the program being executed is currently in a (large) loop, which could be divided in 5 such blocks (say A, B, C, D and E).
First the program blocks A, B, C and D are referenced and stored in cache. A reference to block E will result in a miss. According to both FIFO and LRU algorithms, block A will be removed to be replaced by block E. Then block A will be referenced, which again will result in a miss. Block B will be removed to store block A, etc.

Although this example generally does not correspond to reality, it will be clear that random is not the worst solution under all circumstances.

The example also shows that the optimum replacement is the replacement of the block which will be needed the farthest in the future. This cannot be implemented since it requires knowledge of the future[1], but provides an upper bound against which other replacement algorithms can be measured.

The replacement algorithm must be implemented in hardware and must execute quickly. FIFO or random are much easier to implement than LRU. Let no_blocks be the number of blocks in a set. For FIFO a counter modulo no_blocks is required for each set. This counter is incremented with each replacement and points to the next block for replacement. A (pseudo-)random replacement can be implemented in different ways, for example a counter modulo no_blocks which is incremented each clock cycle. Only one such a counter is needed for all sets.

For LRU a table is required for each set, containing the replacement order of the blocks in the set. If a new block has to be stored in the set, the block referred to by the top of the table is used. That block reference will be placed at the bottom of the table and the other references will be shifted one place up. If a block is used which is in the cache, its reference will be put at the bottom of the table and the references which were below it will be moved one place up. However, if the set size is 2, just one bit is required to decide which block has to be replaced. Updating the replacement information is just a matter of setting or resetting it, depending on which of the 2 blocks is used.
For the replacement status distinctions could be made theoretically between demand-fetched versus prefetched (but not yet referenced) blocks and normal requests versus prefetch lookups. Since differences between the replacement algorithms themselves are already small, it will not be worthwhile to take these distinctions into account considering the extra complexity.

[1] This introduces the idea of marking blocks, which will not be used again by the program, by means of a special compiler instruction. Actually, this is more suited for data caches and could be compared with the effort necessary to allocate registers effectively. [Goodman_86, McNiven]

4. Architectural considerations.

The previous chapter contained a overview of the theory of instruction caches. A number of aspects will be studied more closely in this chapter:

- How prefetching can be accomplished.
- The increase in the cache access time due to prefetching.
- How the miss penalty can be reduced, in general and when prefetching is used.
- If it is worthwhile to prefetch multiple transfer blocks instead of only the immediately next one.
- How the cache RAM can be organized effectively.

4.1. Requirements for prefetching.

Prefetching basically means "fetching information before it is requested". To make prefetching successful, however, the cache also has to be able to service quad requests[1] from the processor while it is prefetching. The best way to achieve this parallelism is to use two separate working modules. One which controls requests from the processor, one

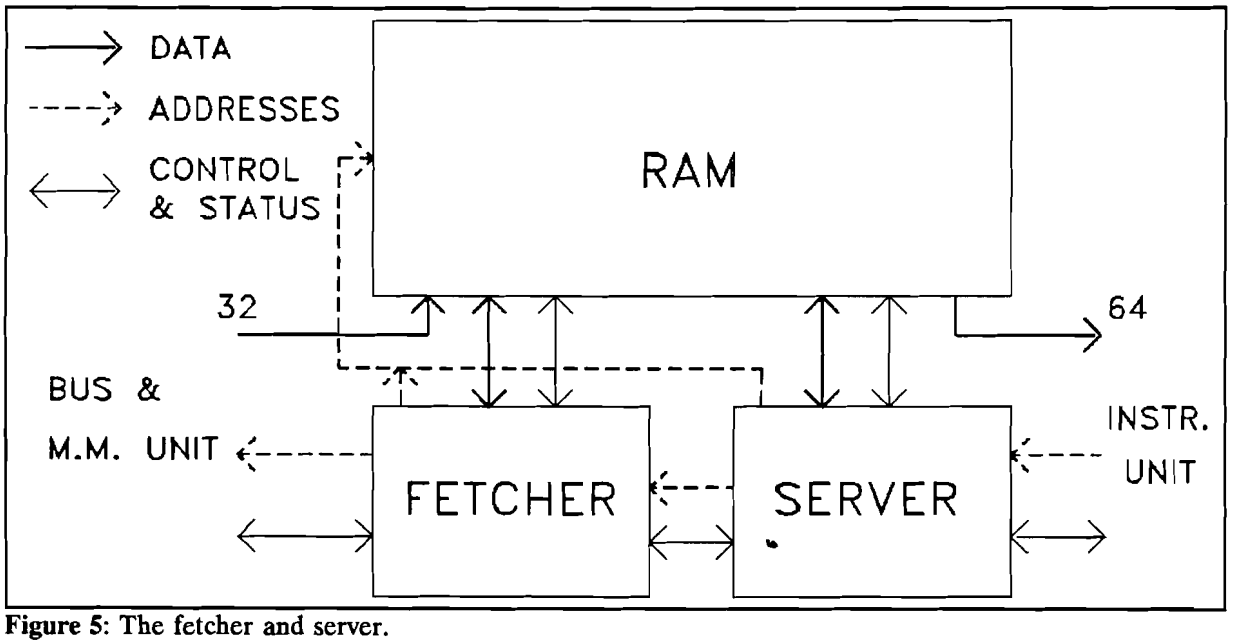

Figure 5: The fetcher and server.

[1] "Quad request" will be used to indicate a (double) quad request from the instruction unit to the instruction cache.

which controls the prefetching. The former will be called *server*, the latter *fetcher*. This is drawn in Figure 5.

The question arises which module becomes responsible when a miss occurs. Although the server is responsible for requests from the processor, it is better that the fetcher handles misses. After all, not only the requested information has to be fetched, but the complete transfer block. Since this rather complex operation is also required for prefetching, it will be obvious that prefetches and demand fetches should be controlled by the same module. (It will be clear now why "fetcher" was used above instead of "prefetcher"). Another argument is that no information can be demand fetched while the fetcher is prefetching. After all, there is only one bus to transfer instructions from main memory to the processor.

If the server receives requests for quads from a new transfer block, it is checked whether or not the next transfer block should be prefetched, possibly preceded by a demand fetch.
Due to the parallelism, however, it is possible that the fetcher is busy at that moment. This problem can be solved when the fetcher itself checks if a new prefetch has to be started according to the prefetch algorithm, after it has finished a transfer block fetch. A prefetch can thus be started by the server when a new transfer block is entered and the fetcher is not busy or by the fetcher itself.

4.2. Increase in cache access time due to prefetching.

An typical example of the trade-off between the hit ratio and the cache access time, see section 3.1, is the usage of prefetching. Prefetching will increase the hit ratio, but will also slow down the servicing of quad requests.

A quad request can be delayed in the following ways:
- The fetcher occupies the RAM for a prefetch lookup.
- A prefetched block is being stored in the RAM.
- In case of a miss, the fetcher can occupy the bus. This increases the miss penalty.

The duration of the first mentioned event is small compared to the duration of the two remaining events. Therefore, the probability of this event and the delay it causes are smaller than the probability of the other events and their accompanying delays. Furthermore, the number of lookups can be decreased by checking the cache only after a hit (prefetch_lookup_on_hits). It is of course possible to avoid lookups totally by using a different prefetch method. (See paragraph 3.4.3.) But since prefetch_lookup is a very effective method to avoid prefetching of a transfer blocks which are already in cache,

the increase in accesses to the RAM for prefetch lookups pales into insignificance besides the decrease in traffic ratio and number of unnecessary accesses to the RAM.

Collision of a quad request and the storage of a prefetched block will be studied in this paragraph, the increase of the miss penalty due to prefetching in section 4.3.3.

There are several ways to decrease the problem of collisions between quad requests and the storage of prefetched transfer blocks:

1. Service quad requests and storage of prefetched block simultaneously.
2. Decrease the duration of storing a prefetched block.
3. Decrease the number of accesses to the cache RAM.

4.2.1. Simultaneous storage of prefetched blocks and reading of quads.

The cache RAM is much faster than the main memory, even when operating in burst mode. Maybe it is possible for the server to access the cache RAM between the storage of two prefetched quads. (That is, if there is a hit. The increase in miss penalty is studied in paragraph 4.3.)

Two different approaches can be used for a single ported RAM:

- The server and the fetcher access the RAM whenever it is free. This requires a mechanism to prohibit the fetcher to use the RAM while it is used by the server, and the other way round. Both server and fetcher can thus be delayed.
- Adjust the internal timing of the cache in a way simultaneous accesses of the server and the fetcher can never occur. This could for instance be accomplished by allowing the server to access the RAM in the first half of each clock cycle, the fetcher in the second half. This is of course only possible when the RAM access time is smaller than half the cycle time, which is very improbable at high clock speeds.

Therefore, the former method seems to be the best from these two. The best solution for this problem, however, is two ported RAM. In that case the fetcher and server do not interfere at all. (Except of course when they want to access the same data). The disadvantage of two ported RAM is that it will occupy more area.

4.2.2. Duration of storing prefetched block.

The delay due to the storage of block which is being prefetched is smaller as the duration of the storage decreases. (In that case the probability of simultaneous

occurrence will also decrease.) The most obvious remedy is too decrease the prefetch transfer block size. However, this results not only in a less optimal usage of the burst mode, it also increases the requirement of prefetching multiple transfer blocks (see paragraph 4.4).

A better method to accelerate the storage of prefetched quads is a buffer between main memory and the cache:

- The quads are stored in the buffer at main memory read speed and stored in the RAM at cache RAM write speed. This is only worthwhile if the difference between these read and write times is significant.
- As above, but the prefetched transfer block is stored at once in the cache RAM. This requires a "wide" RAM. If a transfer blocks is too large to be placed on one row, the buffer could be copied to the cache in several phases or a number of RAM banks could be used parallel.

By means of the last method it is possible to store a prefetched block extremely fast in the cache RAM. The above mentioned buffer will be indicated by *fetch buffer* from now on.

4.2.3. Number of accesses to cache RAM.

Apart from the prefetch lookup accesses, a decrease in the number of accesses to the cache RAM can be obtained in the following way:

- Decreasing the number of accesses to the RAM by the fetcher. This is possible by limiting the amount of prefetched information. One way to do this is to avoid unnecessary prefetches by using the tagged_prefetching or prefetch_lookup method. Another approach is already described above: Use a fetch buffer and copy it to the cache RAM at once.
- Decreasing the number of accesses to the RAM by the server. This is possible by using a buffer between the cache and the instruction unit. If a memory request results in a hit, the total transfer block is copied to this buffer. The following quads do not require access to the cache RAM, as long as they belong to the same transfer block. The transfer blocks address, which is the total address minus the bits used to select a quad within a transfer block, has to be stored together with this buffer. A valid bit will also be required for this buffer.

The last mentioned idea, which will be indicated by a *read buffer* in the remainder of the text, has another and even larger advantage: The small buffer can be read much faster than the cache RAM itself, for it is much smaller and no associative search has to be done.

A read buffer is an intermediate form of a *remote program counter* and a *two-level cache*. The remote program counter is introduced in [Patterson]. It attempts to guess the next instruction address, so the cache can begin to fetch it out of its RAM before it is really demanded by the instruction fetcher. This way the cache access time (on hits) can be reduced. The read buffer is based on the spatial locality in reference patterns.
Two-level caches, or *multi-level caches*, are used to improve the performance of large mainframe caches by adding one or more memory hierarchies between the (main) cache and the processor. More information about multi-level caches can be found in [Bennett, Pomerene, Short].

As for the fetch buffer, the read buffer requires one "wide" RAM or several parallel ones to make copying at once possible. If it is not be possible to copy it at once, it can be partly copied or in phases. If, for example, a transfer block would be stored on two rows, only a half transfer block could be copied into the read buffer. If a request is made to the upper halve, it would be unnecessary to store also the lower halve in the read buffer.

An even more advanced scheme would be obtained, when, in that case, the lower halve of the next transfer block would be stored in the read buffer. When that part is requested, the upper part of that transfer block should be stored, etc. Although this "*in-cache-prefetching*" is rather simple as long as it is restricted to transfer blocks belonging to one block, the benefits of it should be carefully weighed against the increase in overhead. The overall performance will only increase when the read time of the read buffer is significantly lower than the cache RAM read time, a block is divided in many transfer blocks and the memory requests are highly successive.

From now on, however, it will be assumed that the read buffer contains one transfer block completely, which is copied from the cache RAM as soon as a cache RAM hit occurs. A read buffer hit will then always indicate that the same transfer block is referenced as in the previous request. This information is useful since the initialization of a prefetch has to be considered when a new transfer block is entered. As long as requests are made to the same transfer block, this is not necessary. The same approach can be used for the updating of LRU status information, although this is actually only necessary when a new block is entered. No damage is done, however, when it is updated each time a new transfer block is entered.
If no read buffer is used, the address of the previously used transfer block has to be remembered. After all, due to jumps it is not sufficient to check if the first quad of a transfer block is requested to know whether or not a new transfer block is entered.

4.3. Miss penalty.

A miss is likely to be caused by a jump in the reference pattern, which itself can be caused by a conditional expression, a function call, a function return, a task switch, an interrupt, etc. A miss can also occur when quads of a transfer block are requested which is still being prefetched. After all, a transfer block has to be stored in the cache RAM completely, before the data_valid bit can be set. It is of course also possible that another transfer block is being prefetched when a miss occurs.

Three different cases can thus be distinguished when a miss occurs:

1. The fetcher is not busy. This represents the miss penalty in its basic form.
2. The fetcher is still prefetching the transfer block which caused the miss. The actual causes of this will be under discussion in paragraph 4.4.
3. The fetcher is busy prefetching a transfer block different from the needed one. This represents the increase in miss penalty due to prefetching.

The above mentioned situations will be studied in the in the following sections.

4.3.1. The miss penalty in its basic form.

Two aspects are involved when a miss occurs and transfer blocks are fetched instead of single quads pairs. Firstly, the requested quad pair is not fetched immediately, but as part of the transfer block. Secondly, the total transfer block has to be stored in the cache RAM before quads can be read from it.

To accelerate the fetching of the required quad pair, *wrap around* could be used: The requested quad pair is fetched first, followed by the quads in the transfer block which have a higher address and finally those which have a lower address. But if wrap around is used, another fetch technique could be used. Until now we have assumed *load-through*: the quad pair is stored before it is passed to the processor. A faster method is *fetch bypass*: the requested quad pair is passed directly from the main memory to the processor and is stored in the RAM at the same time or later.

Besides that fetch bypass is basically faster, it also evades partially the problem of not being able to read from a transfer block before it is stored completely. A quad pair demand fetched according to the bypass method, is passed directly to the processor and can thus be used before the complete transfer block is stored. However, it depends on the time between the first request and the next request whether or not the third and fourth quads can be passed directly to the instruction unit or are already stored in the cache RAM. Therefore, fetch bypass is probably only applicable at the beginning of a

demand fetch, so the next quad pairs still cannot be read before the complete transfer block is stored.

At first glance two ported RAM seems to be a good solution for the problem of not being able to read from a partially available transfer block. It is not, for the data_valid bit of the transfer block can only be set when it is stored totally. Although this difficulty could be overcome somehow, it is a better idea to make the fetch buffer, which was introduced in section 4.2.2, "two ported" and to equip it with data_valid bits for each quad. As for the read buffer, a tag is required to indicate the contents of the fetch buffer.
As soon as a (double) quad is stored in such a buffer, it can be used by the processor. When the total transfer block has been fetched, it should be copied to the cache RAM.

If the processor is able to continue before a transfer block is stored completely, the question arises what should be done in case the processor requests information which does not belong to the currently demand fetched transfer block. This situation will occur more often when wrap around is used and when the main memory is slow compared to the processor. There are two options:

- finish the current demand fetch before starting the next one. This will increase the service time of the currently requested quad pair.
- stop the current demand fetch immediately. This has the disadvantage that the already fetched quads are lost. After all, the data_valid bit of the transfer block cannot be set.

If the transfer block is never referenced again, the last solution will clearly be the best. In case it is referenced again in the near future, the former will be better. The last solution will especially be bad when the transfer block is repeatedly referenced, without ever being fetched totally. This could happen, for instance, when the beginning of a loop is located near the end of a transfer block. Quads from the next transfer block will then be requested each time, before the first transfer block is fetched totally.

4.3.2. Miss while required info is being prefetched.

If a quad pair is requested which belongs to the transfer block being prefetched, the same problems occurs as described above: The requested quad pair is not fetched immediately and the complete transfer block has to be stored before the quad can be read.

A solution could be to stop the prefetch actions of the fetcher and to demand fetch the transfer block again, but now using the wrap around method. But since the prefetched

transfer block is the successor of the last transfer block used by the processor, the first quad pair of it is likely to be the required one by the processor. Therefore, it would not be wise to discard the already prefetched quads. Only a mechanism has to be provided to read a quad before the total transfer block is prefetched.

It will be clear that the usage of a fetch buffer, with separate data_valid bits for each quad, is the best solution. The fetch buffer can thus be used to decrease the access time to quads belonging to both demand fetched and prefetched transfer blocks.

Not all the quads transferred to the instruction unit will have been in the read buffer if the server can read from the fetch buffer directly or when fetch bypass is used. In order to make sure that in case of a read buffer hit the same transfer block is requested as in the previous request, the read buffer valid bit has to be invalidated when quads are read from the fetch buffer or when a miss occurs. Otherwise, it would theoretically be possible that the contents of the read buffer is used again in the future and the initialization of the next transfer block and the update of the LRU information are skipped.

4.3.3. Increase in miss penalty due to prefetching.

In the section above the miss occurred because the fetcher did not finish the prefetch actions on time. It is of course also possible that the fetcher is prefetching an other transfer block when a miss occurs. This increases the miss penalty, since the the fetcher will not be direct available.

Three approaches can be used:

- Wait until the prefetched transfer block is stored totally. The relative increase in the miss penalty can be large, especially when attempts are made to decrease the "normal" miss penalty by using a fetch buffer, wrap around or bypass.
- Interrupt the prefetch. After the demand fetched transfer block has been stored, the remaining quads are prefetched and the data_valid bit is set.
- Stop the prefetch totally. The already prefetched quads are lost.

The reason the miss occurred is a jump. Therefore, the usefulness of the already prefetched blocks will be low. Furthermore, the transfer blocks following the transfer block which caused the miss, are unlikely to be in the cache. This information should be prefetched as soon as possible. An other disadvantage of the second possibility is the increase in overhead.
According to these considerations, the last solution, stopping the fetcher totally, probably is the best.

If the server has to be able to stop the fetcher immediately on misses, the address of the transfer block being fetched has to be visible for the server. After all, the server has to know whether or not the requested information is being fetched already. If a fetch buffer is used, the tag required for it can be used for this purpose. If distinction has to be made by the server in stopping demand fetch and prefetch actions, it has to be visible to the server if the fetcher is demand fetching or if it is prefetching.

When demand fetch or prefetch actions are stopped immediately, the data_valid bit of the transfer block being fetched cannot be set. Not only the already fetched information will be lost, it is also possible that block replacement preceded it. After all, the block to which the transfer block belongs may not have been in the cache. It is thus theoretically possible that a completely filled block is replaced by a totally empty block. This disadvantage can simply be avoided in case a fetch buffer is used. After all, if a fetch buffer is used, it is not necessary to select a block in the cache and set its tag at the beginning of a transfer block fetch. This can be postponed until the transfer block is completely stored in the fetch buffer. Unnecessary block replacements can thus be avoided by changing this order.
In that case the fetch buffer has to be equipped with a used_before bit if tagged_prefetching is used. This bit should be set when the server uses the fetch buffer. When the complete transfer block has been fetched and, if necessary, a block has been replaced, this bit should be copied together with the contents of the fetch buffer to the cache.

4.4. Multiple prefetch blocks.

We have seen in the previous paragraph that the processor can request information which is still being prefetched. Causes of this will be investigated here to determine whether or not it is worthwhile to abandon the one-transfer-block-ahead principle and fetch multiple transfer blocks.

Three different causes can be distinguished:
1. The initialization of the prefetch was too short ago.
2. The prefetch actions were delayed too much.
3. The number of requests is larger than the fetcher is able to manage.

4.4.1. Initialization too short ago.

Assuming the one-transfer-block-ahead prefetch method is used, the time between

initialization of the prefetch and its requirement by the instruction unit can be too short because of:

- The previous transfer block was not entered at the beginning. So a jump must have occurred.
- The previous transfer block was processed faster by the instruction unit than usual due to the occurrence of many simple instructions. These simple instructions, however, use generally fewer bytes than more complicated instructions, so the increase in number of quad requests will be somewhat smoothed. This is an incidental case of the third point listed above and will be investigated below.
- A small forward jump occurred within the previous transfer block.

If small forward jumps occur, not necessarily within a transfer block, prefetching of multiple transfer blocks could be beneficial.

4.4.2. Prefetch operations delayed.

The prefetch of a transfer block can also not be finished on time when it is delayed in some way. This could, for example, happen when the instructions are stored in global memory of a multi-processor system. The prefetch actions can also have been delayed during the storage of the quads in the cache RAM. This can be avoided by using two ported RAM or a fetch and/or read buffer.

It will be clear that the number of transfer blocks to be prefetched is not under discussion here.

4.4.3. Processor speed too high.

If the processor generally executes faster than quads can be fetched from main memory, problems arise when the information stored in the cache is not reused often enough. In that case the transfer block size could be increased to use the benefits of the burst mode better. This is of course only effective to a certain extent. It is more effective, and more expensive, to increase the main memory speed. Prefetching of more than one transfer block will probably be not possible, since it already will be difficult to prefetch the next transfer block fast enough.

If, on the other hand, old information is reused often, other approaches could be used. Firstly, the amount of old information in the cache could be increased by increasing the cache size. Secondly, during the time the processor is reusing old information, the fetcher could prefetch as much as possible information. It is, however, not clear which

transfer blocks should be prefetched. More over, this prefetching will result in replacement of old information at a moment old information is being reused.

Concluding, prefetching of multiple transfer blocks instead of one might be desirable in the following cases:

- When small forward jumps often occur. Attempting to catch these forward jumps by increasing the number of transfer blocks to be prefetched, is a form of jump prediction. Jump prediction, however, is already included in a much smarter way in the C-processor.
- When the processor is much faster than the main memory and information should be transferred from main memory to the cache whenever possible. It will only be possible to prefetch transfer blocks different from the immediate next one, if the processor is using old information. It is, however, very difficult to decide what should be prefetched in such a situation, especially considering the fact that old information will be replaced by it.

From this it will be clear that it is probably not worthwhile to prefetch more than one transfer block. It is better to use the one-transfer-block-ahead principle and to adapt the transfer block size to both demand fetches and prefetches.

4.5. Effective cache RAM organization.

The last paragraph of this chapter will be used to investigate how the cache RAM can be organized effectively.

4.5.1. Separate data and status RAM.

Until now one RAM is assumed for data, tag and status information. Only one tag is required for all quads in a block. Some status information, like LRU bits, is also required per block. Other status information, like data_valid bits, is required per transfer block. If CAM is used, it will be evident that the complete block must be stored on one RAM row. The quads have to be selected by means of the output decoders.
Traditional RAM could also be organized this way. A RAM row is selected by the address decoder, using the set field of the address. (Assume a set size of 1, set associativity is discussed in the next section.) The tag, the status or one of the quads is selected by the output decoders. This is depicted in Figure 6a. It is of course not necessary to locate tag and status at the same location as the quads. One possibility is drawn in Figure 6b. This approach has many disadvantages:

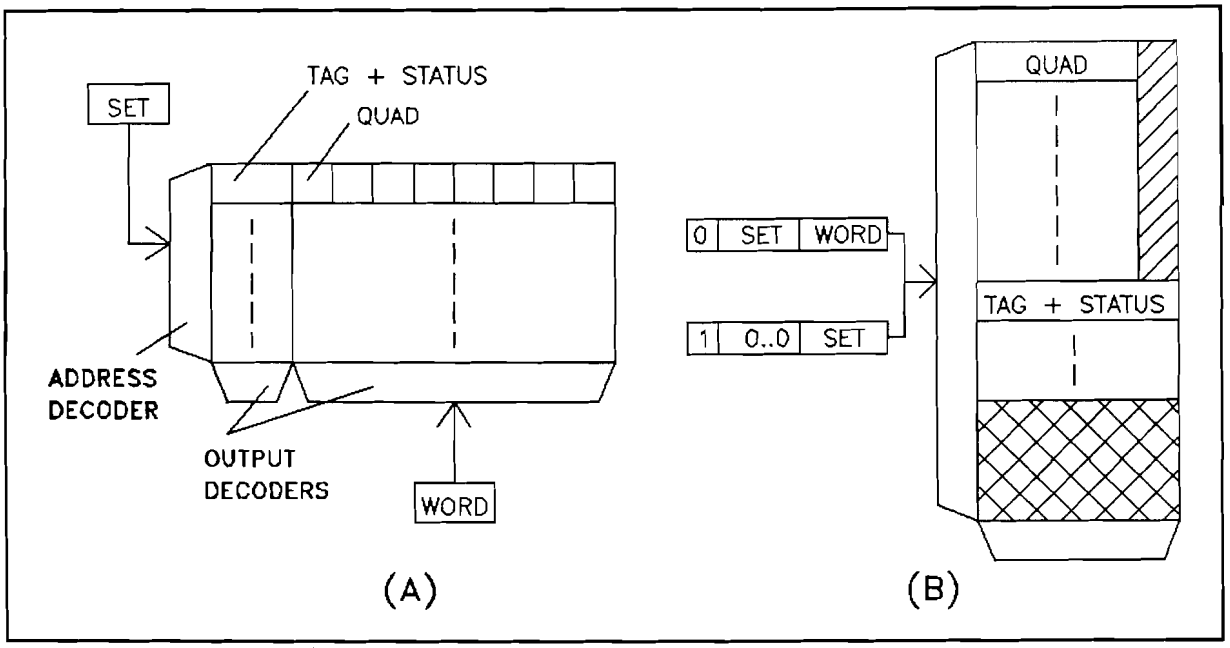

Figure 6: Data and status information combined in one RAM.

- The address necessary to select tag/status information has a different format than the address of a quad. (Or the tags should be stored in an even more inefficient way.)
- Not all bits of the rows containing quads are used. (The length of tag plus status is assumed to be longer than 32 bits). This is represented by the single shaded area.
- The address space is not efficiently used, since it is no power of 2. This is represented by the double shaded area.

Therefore, it is a better solution to use two different RAMs. One for the quads themselves, one for tag and status information. This has also certain advantages over the first approach:

- No wide RAM is necessary.
- The RAMs are smaller and thus slightly faster.
- Simultaneous access to data (quads) and tag / status information is possible.

Data RAM will be used from now on for the RAM containing the quads, *status RAM* for the RAM containing tags and status information.

In Figure 7 the global architecture of the cache is drawn, including buffers and separate RAMs.

4.5.2. Requirements for associativity.

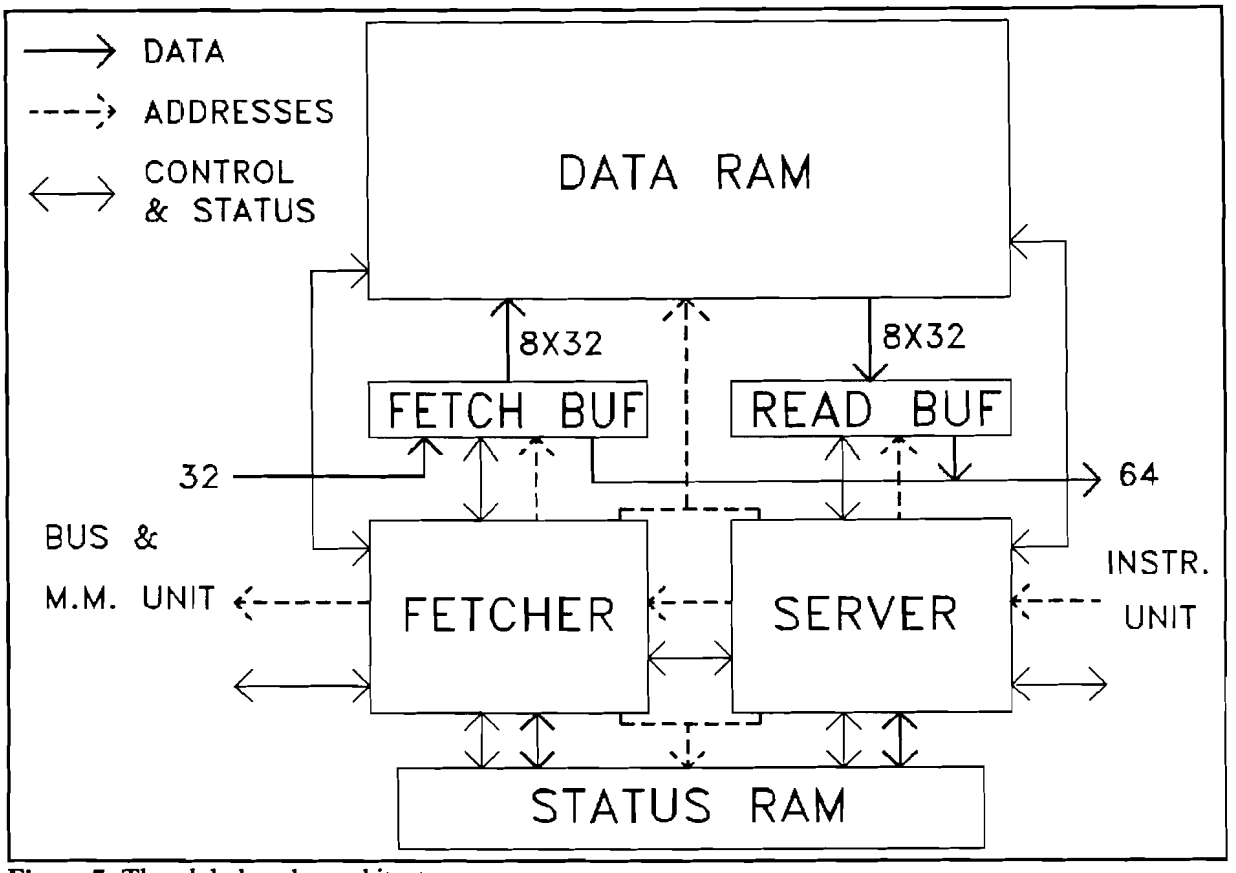

Figure 7: The global cache architecture.

Four separate modules were drawn in Figure 4 to illustrate a four way associative set. The tag and status information do indeed have to be accessible simultaneously. Different status RAMs could be used. But since these will be very small, it is better to use one status RAM, containing tag and status information of all blocks of one set in one row. This is depicted in Figure 8, assuming a set size of 2, LRU replacement, tagged_prefetching and 4 transfer blocks per block.

Quads of different blocks within a set do not have to be accessible at the same time. They can thus be placed on different rows of one data RAM. The order in which quads of different sets and blocks are placed in the RAM, depends on the format of the addresses supplied to the address decoder. The physical order of the quads is of no real importance. The address format, nevertheless, could influence the access time of the RAM. Considering for example a request from the processor. The set and word fields are direct available from the address, whereas the block field has to be generated by the server. Depending on the internal structure of the address decoder, it might for example be better to use the block field as the least instead of the most significant part of the

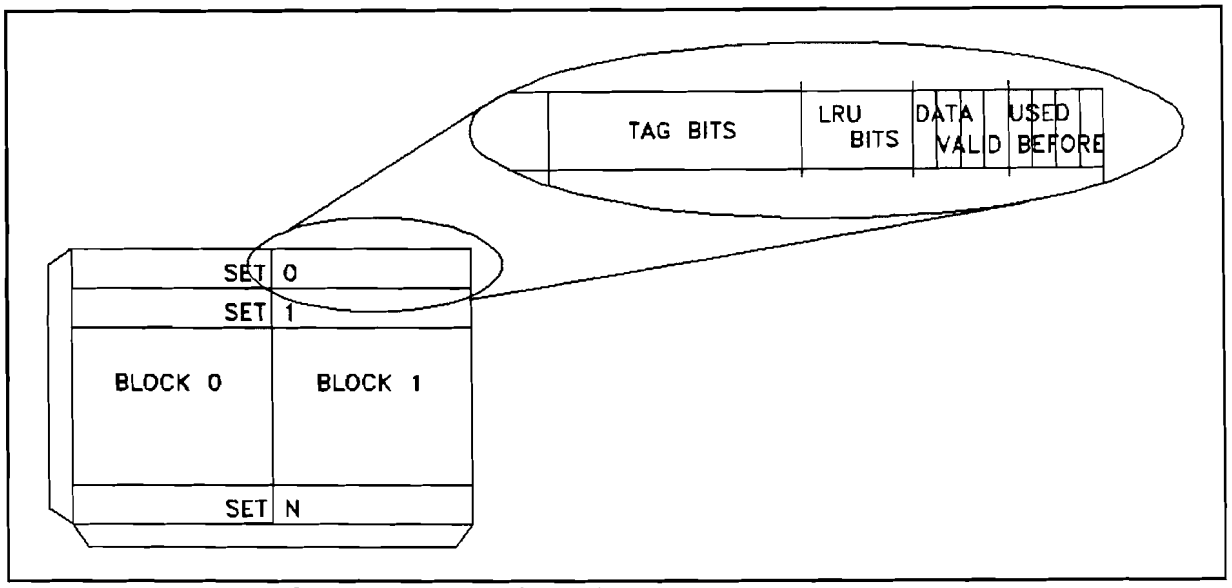

Figure 8: Status RAM of a two way associative cache.

address.

This is a good method to improve the *aspect ratio* (the ratio between the width and the length) of data RAMs of caches with a high degree of associativity. Folding is a more general solution to improve the aspect ratio of a RAM. The approach described here, however, requires less additional area, since the area needed for buses does not increase. This is inevitable when folding is used.

4.5.3. Requirements for read and fetch buffer.

The read and fetch buffer, earlier introduced in this chapter, have a very big influence on the cache layout. After all, their benefits are exploited best when complete transfer blocks can be copied to or from them. This requires a wide data RAM. (However, not as wide as represented by Figure 6a, where tag, status and a complete block was stored on one row.) The status RAM is not affected by the requirement that complete transfer blocks have to be stored on one row.

If the cache has a set size larger than 1, the method described in the previous section could be used when the data RAM becomes too wide. An example of the data RAM of such a cache is depicted in Figure 9. Blocks of 32 quads, divided over 4 transfer blocks, are assumed. The address which has to be supplied to the address decoder, consists successively of the set field, the block field and the transfer block field.

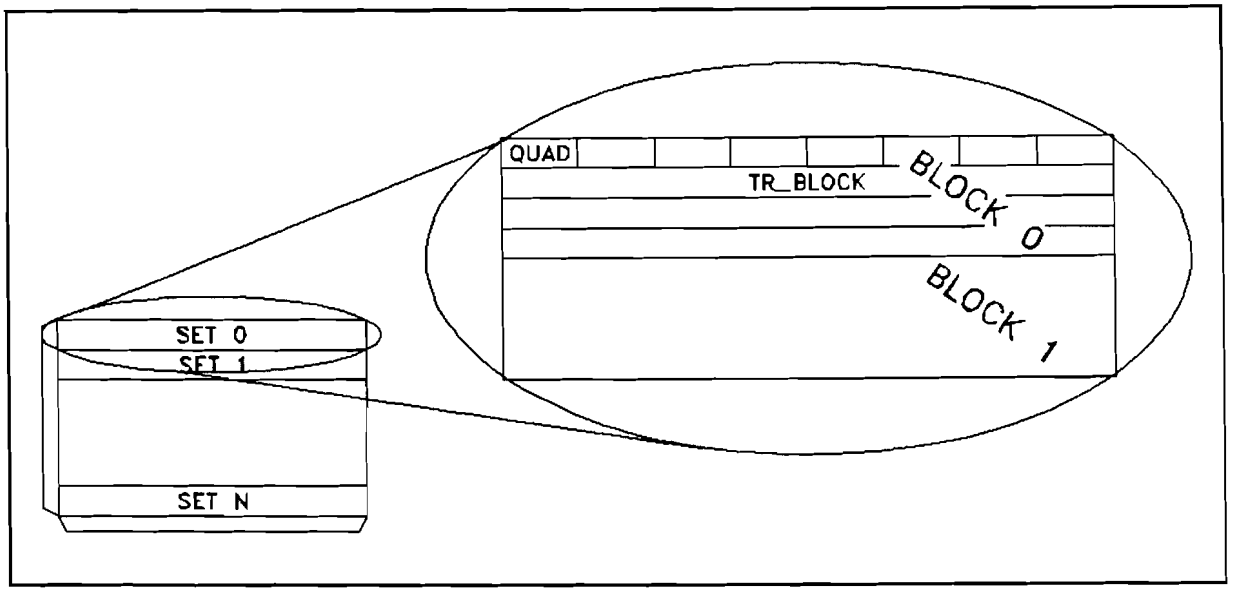

Figure 9: Data RAM of a two way associative cache.

If the data RAM is still too wide, a transfer block could be stored on two rows. Then a transfer block has to be copied from or to a buffer in two phases. Another solution is to use two data RAMs. (If the RAM is too wide because of the aspect ratio, these RAMs should be placed under each other. If it is too wide because of restrictions imposed by the available RAM generators, they can of course be placed side by side.)

The read and fetch buffer are placed side by side for clearness in the global architecture depicted in Figure 7. In reality they have to be placed below each other, in line with the data RAM. This way complex placement of the buses between the RAM and the buffers, which are as wide as transfer blocks, can be avoided.

4.5.4. Requirements for quad aligned requests.

The 64 bits requests of the processor are quad aligned. If complete transfer blocks are stored on RAM rows, two quads can be read simultaneously as long as they are in the same transfer block. Simple adjustments of the output decoders are necessary to accomplish that if quad i is selected, quad i+1 is also selected. To deliver quads which are not in one transfer block fast, the data RAM could be organized as in Figure 10. The transfer blocks are divided over two RAMs and an adder is used. If this adder is enabled, it increases the address for the upper half of the data RAM by one. For the first 7 output decoders the above applies. If output decoder 7 is selected, output decoder 0 and the adder are enabled, in order to read the first quad of the next transfer block. The adder will decrease the access time, but will always be necessary when information

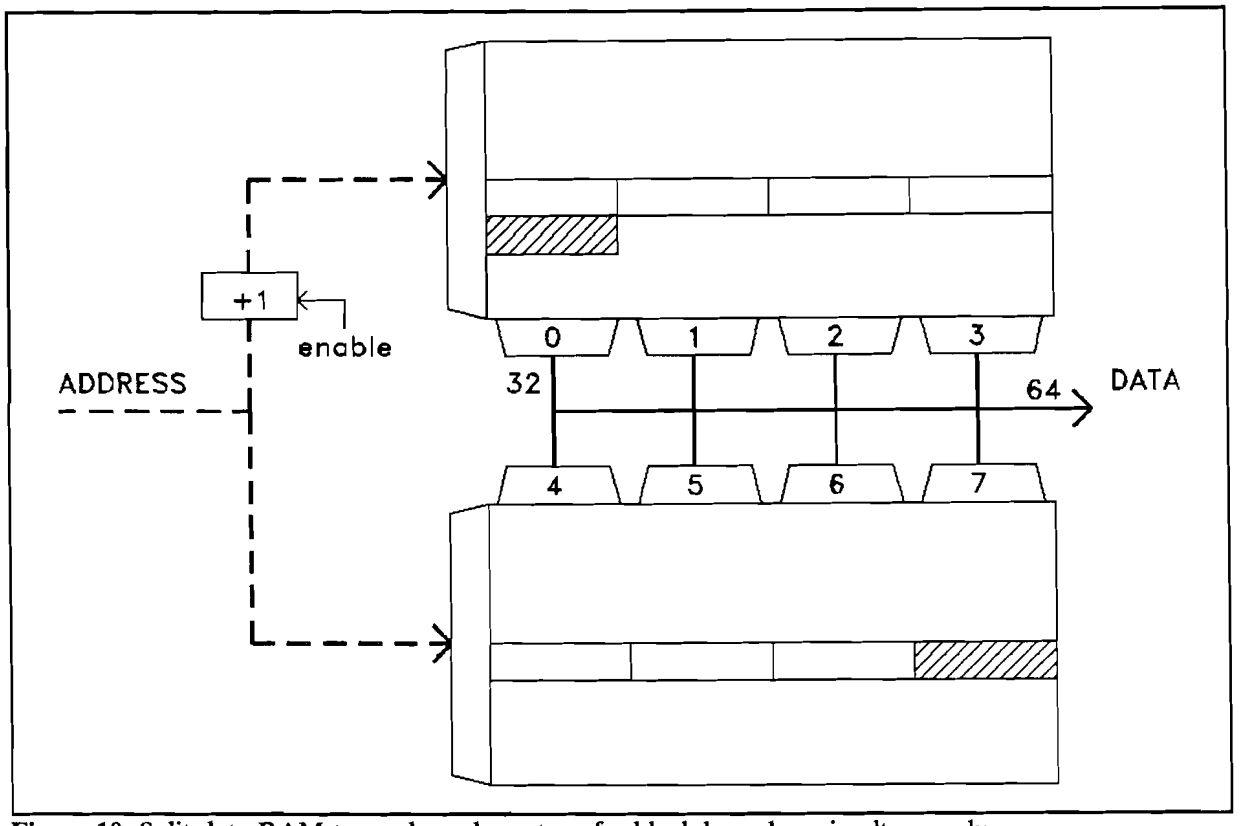

Figure 10: Split data RAM to read quads on transfer block boundary simultaneously.

is requested which is not stored on one RAM row. It is thus possible to read the two quads simultaneously from the RAM, even when they are not from the same transfer block. This approach, however, is only useful in case the next transfer block belongs to the same block and is indeed available.

If a read buffer is used, it could be split in two parts to contain two half transfer blocks or it could be doubled to store two complete transfer blocks.
However, regardless of the usage of a read and/or fetch buffer, the most simple solution is to store the first quad in a buffer when the requested quads are not in the same transfer block. As soon as the second is available, the ready signal to the instruction unit can be given. The instruction unit, however, places the quads in a buffer before the instructions will be decoded. The extra buffer stage can thus be eliminated by using a separate ready signal for each quad. This also has the advantage that the instruction unit does not have to wait for the second quad when it only needs bytes from the first.

The last sketched approach affects the interconnection between the instruction unit and the cache. This possibility was taken into account because the interconnection between both modules was not defined exactly yet and thus had to be reconsidered. It must thus be seen as a higher level consideration, rather than a change in an already defined

interconnection.

If fetch bypass is to be used on misses, a separate buffer in the cache or two ready signals towards the instruction unit will also be necessary. After all, there is only a 32 bits bus available to transport instructions from the main memory to the cache.

5. The simulator.

Many different cache parameters have been introduced in chapter 3 and 4. These parameters are related to sizes, algorithms or options. The effects on the optimization criteria are easy to predict for most of them. It is, however, difficult to decide which choice is best, since the criteria will be affected in different ways. For other parameters, however, it is even hard to predict the effects on single criteria. Increasing the block size, for example, has both positive and negative effects on the hit ratio.

Since it is not possible to implement different configurations and experiment with them, simulations are needed to study cache performance. Paragraph 5.1 describes the approach used to built the cache simulator.

5.1. Methodology.

Two aspects can be distinguished in the methodology used for the simulator. The method itself of course and the preconditions to implement it in a way it can be used effectively.

5.1.1. Trace driven simulations.

The most used method to study cache performance is *trace driven simulation*. The input of such a simulation, a trace, is a list of the requests from the processor to the cache, expressed in addresses. A trace has to be generated by executing compiled programs. If the processor itself is not available, like in this project, a software model of the processor is required. A software model of the C-processor was indeed available during this graduation project. The compiler, however, was not.

A trace generator was written to solve this problem. It generates traces containing basic elements of reference patterns. The characteristics of these artificial traces are discussed in paragraph 5.4. It will be clear that conclusions based on these traces are temporary.

Since programs can have various reference patterns, it is important to select a set of them which is in accordance with the real workload. Attention has to be paid to the way the results of different traces are combined. One approach is to weigh them according to the lengths of the traces. This has the disadvantage of emphasizing the larger traces over the smaller ones, although their relative lengths may not be representative for the part the programs have in the real workload.

Task switching can be simulated by inserting cache flushes in the traces. Therefore, the simulator has to be capable to recognize and to execute this instruction and possible other cache instructions. If appropriate intervals are chosen, this is a good way to simulate task switching for caches which are indeed flushed after a task switch. The C-processor, however, is not, since it avoids rigorous cache flushes by using process identification numbers. More accurate results can be obtained when a *system trace* is used instead of multiple traces, which contain one program each. Besides alternating user programs, tasks of the operating system should be included. After all, on many machines the operating system itself has a large impact on the workload. It will be clear that it is difficult to obtain such a realistic system trace, especially when the system itself is not implemented yet.

A good discussion about the importance of the workload choice can be found in [Smith_A.85b]. The impact of task switches on cache performance is studied in [Barsamian, Haikala, Kaplan].

5.1.2. Goals.

The following considerations were taken into account writing the simulator:

1. Several aspects are involved in optimizing cache design. To be able to make well-founded decisions, all these aspects have to be reflected in the simulation results.
2. All, or almost all, options for instruction caches found in literature have to be included in the simulator.
3. Besides this, it has to be possible to investigate the benefits of a fetch buffer, a read buffer and the possibility to stop a prefetch and/or a demand fetch immediately when a miss occurs.
4. The simulator is intended for comparing different implementations, rather than predicting the performance of single implementations exactly. Much attention thus has to be paid to the ease of running many simulations and comparing the results.
5. All simulations have to be rerun in the future, since the used traces are artificial and the conclusions based on them are temporary. This has to be possible without much extra effort.

Due to the possible usage of prefetching and a number of other options, it is necessary to include timing information in the simulation model. This will be explained in paragraph 5.3, which treats the timing model. By means of this timing model, it is possible to calculate metrics like the average access time of the cache.

Besides the average access time to the cache, the required area is also an important

optimizing criterium. However, no attempt is made to include a sophisticated area model. In the first place because the simulator is intended for comparisons and not for exact figures. Further, the data and status RAMs require the largest amount of area. Exact sizes could be computed for the RAMs, but the technology and cache layout have to be known. The number of bits, however, is also a good criterion for their sizes. The amount of area required for the server, the fetcher, buses, etc, on the other hand, makes up a much smaller impact on the total area and is hard to estimate. Therefore, only the total number of storage bits and the percentage of overhead bits are counted.

A method to avoid difficult comparisons between cache performance and required area, is to use *constant area cache simulations*. This method was used in [Alpert] to investigate the advantages of dividing blocks in transfer blocks, without losing sight of the fact that fewer overhead bits are necessary when larger blocks are used. A total number of storage bits is assumed and dependent on the cache configuration an effective data storage capacity is calculated. The main drawbacks are that it is rather complex and that cache sizes are obtained which are not necessarily powers of 2. Besides this, it is actually only useful to study techniques which reduce the amount of overhead bits considerably. The area model is too limited to include other effects. Therefore, this approach has not been used.

From the options mentioned in chapter 3 en 4, the following can not be simulated:

- decrease of the transfer block size when a burst of misses occurs. A constant transfer block size is assumed during one simulation. It is, however, possible to investigate to which extend misses occur in bursts.
- smart prefetch algorithms, for example using jump prediction. Jump prediction, however, is achieved in an other way in the C-processor, as described before.
- working set restoration. The C-processor uses a different approach to limit the penalties of task switching: process identification numbers. The benefits of this approach can easily be investigated by using traces in which the process identification numbers really are used and traces in which a cache flush is issued after each task switch.
- no "in-cache-prefetching" with respect to the read buffer is used. It is also not possible to prefetch multiple transfer blocks.

The features and algorithms which can be simulated, together with the calculated quantities, are described in paragraph 5.2.

5.1.3. Simulation modes.

Two kinds of simulations are basically necessary to study cache performance. Results

averaged over a complete simulation are of course the most important information and can be used to compare different implementations. Overall ratios, average access times, etc., however, hide a lot of information. Sometimes it may be necessary to study the cache operations more closely. For example to investigate unexpected overall ratios. Therefore, in addition to a simulation mode intended for calculation of overall results, a simulation mode is provided which generates information over small intervals.

Besides these modes, a third mode is provided to investigate to what extent misses occur in bursts. This simulation feature will not be used in this report, since there is no use in examining bursts of misses based on artificial generated traces. After all, bursts of jumps are the cause of it. (Misses due to too slow prefetching are therefore not regarded as misses in this simulation mode.) If future simulations indeed indicate that misses mainly occur in bursts, it might be worthwhile to alter the simulator in order to examine the usefulness of a dynamically adapted transfer block size. An alternative for a dynamic transfer block size, could be stopping fetch operations immediately. This is much easier to implement, but has the disadvantage of losing the already fetched information.
This simulation mode can also be used to investigate *cold* and *warm hit ratios*. The former is the hit ratio after a cache flush, the latter of a completely filled cache (steady state).

Since the simulator is intended for comparison of many simulations, it is written in a way it is able to combine multiple simulations, each differing in one or more cache parameter. Such a combined simulation run will be called a *simulation frame* from now on. Since the cache parameters change in one simulation frame, it is thus not possible to consider them as constants and to merge them into the simulator by macro expansion. Such a simulator is not only easier to write, it probably will also be faster when simulating. The chosen approach, however, removes the overhead of macro expansion and compilation, and has the advantage of being able to run multiple simulations easily.

More information on these simulation modes can be found in appendix B.1.

5.1.4. Interactive and batch input modes.

Two different input modes are provided to obtain an easy to use simulator, an interactive and a batch input mode. In the interactive input mode the user is prompted for all required information: file names, simulation modes, number of simulations or the interval size (dependent on simulation mode) and those parameters which configure the cache. Except for the file names, the user only has to choose from lists containing the available options. If multiple simulations have to be run, the user only has to specify the changes relative to the previous simulation. This input mode is user-friendly with respect

to the self-explaining character, but has the disadvantage that all user actions have to be repeated when the simulations are rerun.

Therefore, all simulation and cache parameters are stored in a parameter file. A simulation with a new trace can be started easily by invoking the simulator in batch input mode, specifying the names of this parameter file, the trace file and output file as command line arguments. Besides this, the parameter file performs the task of storing the cache parameters in an ordered manner. The cache parameters are stored in a table, one row for each simulation. It is of course also possible to edit parameter files, which is very convenient when simulations have to be run which differ only slightly from simulations of which the parameter files are already available.

How the simulator can be invoked is explained in appendix B.1.

5.1.5. Representation of simulation results.

The simulation results are stored in an output table, similar to the manner the cache parameters are stored. In addition to the simulation results, other information, like the parameter file name, the trace file name and the trace length, is stored in the output file.

Since one is likely to loose track when a number of output tables have to be compared, graphs have to be used for convenience of comparison. The graphs shown in this report are generated by PLOTUTIL of Buts Electronic Systems, a program to generate graphics. A postprocessor for the simulator is written to convert results of a number of simulation frames easily to input files of PLOTUTIL.

As for the simulator itself, both interactive and batch input modes are provided for the postprocessor. How this postprocessor should be used and what its exact possibilities are, can be found in appendix B.3.

5.2. Input and outputs of simulator.

The input and output parameters of the simulator will be explained in this paragraph. The parameters directly related to the timing model, however, will be treated in section 5.3.

The input parameters of the simulator are:

- cache, block and transfer block size. These sizes are in quads.

- set size. A set associative cache is thus assumed. Direct mapped and fully associative caches, however, are just special cases of associative mapped caches.
- the replacement algorithm: LRU, FIFO or random.
- the prefetch algorithm: Besides the algorithms already introduced (prefetch_always, prefetch_on_misses, tagged_prefetching, prefetch_lookup_on_hits and prefetch_lookup_always), *prefetch_never* is possible. This "algorithm" will be clear.
- the usage of the fetch and/or read buffer. A two ported cache without buffers can also be simulated.
- the usage of improved fetch methods to decrease the miss penalty: Wrap around and/or fetch bypass.
- the possibility to stop fetch operations immediately when a miss occurs: *prefetch_stop* and/or *demand_fetch_stop*.

Table 2: Example of a parameter file of the simulator.

```
file: block16_d5.par


Simulation mode       = 3: Overall ratios.
Number of simulations = 9
Interval size         = 0 quad requests
```

	SIZE				ALGORITHM		TIME (ns)				BUFFER		FETCH	
	cache	set	block	tr-block	re-place	pre-fetch	access	la-tency	ram	cy-cle	usage	time	method	stop
1	64	2	16	16	1	5	150	200	50	100	1	0	1	1
2	128	2	16	16	1	5	150	200	50	100	1	0	1	1
3	256	2	16	16	1	5	150	200	50	100	1	0	1	1
4	512	2	16	16	1	5	150	200	50	100	1	0	1	1
5	1024	2	16	16	1	5	150	200	50	100	1	0	1	1
6	2048	2	16	16	1	5	150	200	50	100	1	0	1	1
7	4096	2	16	16	1	5	150	200	50	100	1	0	1	1
8	8192	2	16	16	1	5	150	200	50	100	1	0	1	1
9	16384	2	16	16	1	5	150	200	50	100	1	0	1	1

```
Replacement Prefetch          Buffer usage:    Fetch method:  Fetch stop:
algorithm:  algorithm:
1: LRU      1: NEVER          1: No buffers    1: Normal      1: Not
2: FIFO     2: ALWAYS         2: Fetch buffer  2: Wrap around 2: Prefetch
3: RANDOM   3: ON_MISSES      3: Read buffer   3: Bypass      3: Demand
            4: TAGGED         4: Both buffers  4: Both        4: Both
            5: LOOKUP_HIT     5: Two ported
            6: LOOKUP_ALWAYS
```

An example of an parameter file is contained in Table 2. The cache size is ranged from 64 to 16384 quads, for a block size of 16 quads. The transfer block size is equal to the block size. The LRU replacement algorithm and the prefetch_lookup_on_hits algorithm are used. No buffers are used and no wrap around, bypass, prefetch_stop and demand_fetch_stop.

Exclusive of those belonging to the timing model, the following quantities are calculated by the simulator:

- miss / hit ratios. The hit ratio is separated in hit ratios for the read buffer, the cache RAM and the fetch buffer. A reference to a transfer block which is being fetched, is regarded as a miss if no fetch buffer is used, otherwise as a fetch buffer hit, even though the quads themselves may not be in the buffer.
- the traffic ratio. The scaled traffic ratio is also calculated to take into account that the main memory is used relative shorter when information is fetched in burst mode. Actually, the traffic ratio is a criterion for the amount of data transferred from main memory to the cache, whereas the scaled traffic ratio is a criterion for the time the main memory is used for reading instructions.
- the total number of storage bits and which percentage of this is used for overhead bits. Besides tag, data_valid, used_before and LRU or FIFO bits, all bits associated with the buffers are regarded as overhead bits. After all, the buffers do not increase the storage capacity of the cache. These parameters are independent of the simulations themselves, since they only depend on the cache configuration.
- the memory pollution. The memory pollution will be defined as the number of quads not requested by the processor divided by the total number of quads read from the main memory.

No exact definition of the memory pollution has been given in chapter 3, since no exact definition is found in the literature. The definition, which is given above, indicates the amount of quads transferred from main memory to the cache for nothing. The memory pollution could, however, also have been defined as the number of quads not requested by the processor, divided by the total number of quads used by the processor.

The following approach is used to calculate the memory pollution. Instead of storing instructions in the quad entries of the cache, memory pollution information is stored in them. When a quad is read from main memory, a 1 is put in the quad entry. When it is read by the instruction unit, the contents of the quad entry is set to 0. The real contents of the quads, the instructions, are after all unnecessary during the simulations. Therefore, no instructions are really read from the main memory, stored in the cache and transported to the instruction unit. The timing model makes only an estimation of the duration of these actions. When a block is replaced or when the cache is flushed,

Table 3: Example of an output file of the simulator.

```
file: block16_d5.out                Fri Jul 28 8:22:19 1989
```

	AREA		TIMING		RATIOS						
	total # bits	%of data bits	service time	cycles	read-buf-hit %	cache ram hit %	fetch buf-hit %	miss %	mem. poll. %	traffic	scal. traffic
1	2220	7.7	281	3.25	0.0	90.2	0.0	9.8	26.8	1.23	0.57
2	4432	7.6	232	2.76	0.0	92.2	0.0	7.8	25.3	0.99	0.46
3	8848	7.4	150	1.95	0.0	95.9	0.0	4.1	29.6	0.53	0.24
4	17664	7.2	144	1.90	0.0	96.1	0.0	3.9	27.8	0.50	0.23
5	35264	7.1	141	1.87	0.0	96.2	0.0	3.8	26.5	0.49	0.23
6	70400	6.9	139	1.84	0.0	96.3	0.0	3.7	25.5	0.48	0.22
7	140544	6.7	138	1.83	0.0	96.3	0.0	3.7	25.1	0.48	0.22
8	280576	6.6	137	1.83	0.0	96.3	0.0	3.7	24.9	0.48	0.22
9	560128	6.4	137	1.83	0.0	96.3	0.0	3.7	24.9	0.48	0.22

```
Number of (double) quad requests = 100814
Trace file                       = ../t100000a.trc
Parameter file                   = block16_d5.par
```

the sum of all quad entries in the block, respectively in the cache, is determined. To take the unused quads at the end of a simulation into account, the cache is flushed after each simulation.

The output file of the above mentioned parameter file is stored in Table 3.

5.3. Timing model.

Many cache analyses concentrate on miss ratio and traffic ratio. In some cases it is possible to deduce from the miss ratio and the accompanying penalty which implementation has the lowest average access time.

Due to a number of reasons such a model is not sufficient for our analysis. Firstly, the miss penalty is no longer a function of just the transfer block size, because fetch bypass, wrap around and a fetch buffer can be used. Secondly, if prefetching is used, the miss ratio becomes also dependent on the main memory speed and the request rate of the processor. After all, a transfer block will not be direct available after initiation of a prefetch. Finally, if prefetches or demand fetches can be stopped immediately, the traffic ratio becomes also time dependent.

A timing model is thus not only needed to calculate access times directly, but also to obtain accurate measurements for miss and traffic ratios, which normally are considered as time independent quantities.

An exception of the above mentioned analyses is [Przybylski]. The influence of the main memory access time is studied extensively. However, prefetching and options like a fetch buffer are not used. This simplifies the timing model enormously. After all, the miss (and traffic ratio) are then time independent.

5.3.1. Timing parameters.

The following input parameters are used in the timing model:

- processor cycle time (*cycle_time*): This parameter is equal to 100 for all shown simulations. The timing parameters are thus not expressed in exact time units like nanoseconds, but are relative quantities.
- main memory access time (*access_time*): The time necessary to read a quad from main memory working in burst mode.
- latency time (*latency_time*): The extra delay at the beginning of a burst. This parameter should not only include the latency time inherent of the main memory, but also the delay caused by the memory management unit and the bus unit. The first quad of a transfer block is thus available after the latency_time + access_time.
- cache RAM access time (*ram_time*): Time required to read or write one or more quads from or to one row of the RAM in the cache.
- buffer access time (*buffer_time*): Time required to read or write quads from or to the fetch buffer or the read buffer.

The delays introduced by the server or fetcher modules are supposed to be included in the cache access, buffer access and latency times. No attempt is thus made to include the delay of every single action performed in the cache. The main reason of this is that it is very difficult to estimate these delays, since the exact timing of the cache is unknown.

The rate at which the processor requests quads is also of importance. The processor will not issue a request each clock cycle, since more than one instruction can be contained in two quads and not every instruction will be executed in one clock cycle. The length of complete C-processor instructions vary from 1 to 12 bytes. Therefore, each request in the trace file does not only consist of the process identification number and the address of the first quad, but also of the number of cycles between the receipt of the requested quads and the next request. If the required information is not available when

the traces are generated, an average value could be used. It should, however, not be difficult to take the lengths of the instructions into account. It would even be more realistic when the number of cycles for each instruction is also taken into account. To accomplish this exactly in accordance with the reality for the C-processor, however, will be very difficult due to the highly-pipelined nature of this processor.

The timing model is used to calculate two output parameters:
- *service_time*: The average time before a requested quad pair is available, not taking into account the quantization due to the synchronous nature of the communication between cache and instruction unit.
- *number_of_cycles*: The average number of cycles before a requested quad pair can be used by the instruction unit.

Not taking into account that algorithms of different complexity introduce different delays, does not mean that they cannot be compared well. The difference between service_time of for example LRU replacement and random replacement can be regarded as an upper bound for the additional delay when LRU is used instead of random replacement. If the LRU replacement cannot be implemented in a way the extra delay is significant lower than this upper bound, random replacement should be used. After all, other aspects, like required area and design time, also have to be taken into account.

5.3.2. Modelling parallelism in a sequential program.

The fetcher and the server work in parallel. This introduces some problems, since the language to be used for the simulator, C [Kernighan], is not capable to execute processes virtually in parallel. The following approach is used to solve this problem.

The two most important internal timing variables of the simulator are:
- *time_count*: this variable reflects the "current time" in the simulation model.
- *ready_time*: this variable indicates when the fetcher will finish the demand or prefetching of a transfer block.

Each time when a double quad request from the instruction unit has been served by the server, it is checked if the fetcher finished the fetching of a transfer block during the servicing of the request or if it will finish it before the next request will be issued. If the fetcher is not busy or if it will not be finished before the next request is issued, time_count is updated and the next request is executed. If it did finish a transfer block fetched in meanwhile, this is simulated by the program and a new prefetch is started possibly. The ready_time of this prefetch will be ready is corrected for the fact that it will have been initialized earlier in reality. Similar actions are performed when the

fetcher will be finished before the next request is issued. In both cases it will be checked if the possibly recently initialized prefetch will be finished before the next request is issued. This is possible when the user is not really interested in dynamic aspects and models very fast main memory or when the number of cycles until the next request is currently high. According to the one-transfer-block-lookahead principle, no new prefetch can then be initialized.

Not every single access of the fetcher to the fetch buffer or the cache RAM to store a demand or prefetched quad is thus modelled, but the transfer block fetches as a whole. This simplifies the modelling of the parallelism enormously, but has also some disadvantages. If there is, for example, a fetch buffer hit, a calculation has to be performed to know if the requested quads are already stored in the fetch buffer, or if not, what the delay will be. These calculations, however, are much simpler than an extensive event queue.

If the two requested quads belong to one transfer block, the double quad request is processed at once in the simulation model. Otherwise the request is split in two parts and the both quads are processed separately. Between the servicing of these two quads, it is checked whether or not the fetcher has finished a demand or prefetch in the meantime. If this is only checked after both quads are processed, as described above, the modelling of the parallelism would not correspond to reality in case demand or prefetches can be stopped immediately on misses: Assume that a fetch buffer is used and that the server is waiting for a quad which is being fetched. The transfer block to which the second quad belongs, will most likely also not be in the cache. During the processing of the second half of the request, a miss will occur and the fetcher will be stopped immediately. It is, however, possible that the complete transfer block is in the fetch buffer and should be copied to the cache RAM. This is possible when the waiting of the processor for the last quad of the transfer block is not caused by the occurence of a jump and the usage wrap around, but is caused by slow main memory. If it is not checked whether or not the fetcher is ready between the servicing of the two quads, the parallelism is not modelled correctly and the contents of the fetch buffer is probably discarded wrongly.

The above shows that the modelling of parallel processes in a sequential language is not straightforward.

5.3.3. Penalties for simultaneous access to RAM.

If no two-ported RAM is used, the server or the fetcher will be delayed when they want to access it at the same time. Therefore, a penalty is included in the timing model for

simultaneous accesses to the data RAM. No penalties are calculated for the status RAM, since the number of accesses to it is much smaller.

The timing model provides no direct indication when such simultaneous accesses occurs and what the exact delay is, since complete transfer block demand or prefetches are modelled and not single quad fetches. Therefore, an average delay is included when there is a cache RAM hit. Asynchronous internal cache operations are assumed calculating this average delay. More accurate calculations are not possible as long as the internal timing is not exactly known.

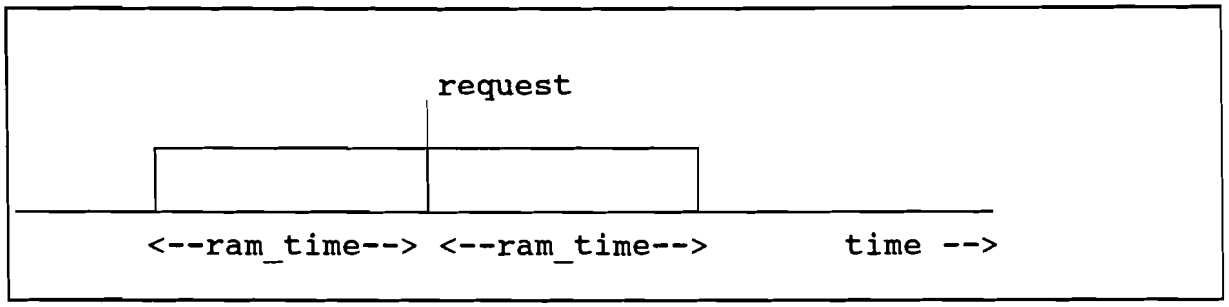

Figure 11: The time period in which processor requests and prefetch storages interfere.

If the server receives a request while the fetcher is busy, an overlap exists when a prefetched quad arrives in the time period depicted in Figure 11.
The probability that a prefetched quad arrives in this period is:

$$\text{prob\_overlap} = \frac{2 * \text{ram\_time} * \text{no\_quads}}{\text{latency} + \text{access\_time} * \text{no\_quads}}$$

No_quads is the number of quads in a transfer block. The average delay will be ram_time / 2. In half of the cases the fetcher is delayed, in the other half the server. The average delay for both fetcher and
server is thus:

$$\text{average\_delay} = \frac{0.5 * \text{ram\_time}^2 * \text{no\_quads}}{\text{latency} + \text{access\_time} * \text{no\_quads}}$$

This delay will be added to both the ready time of the fetcher and the service_time when there is a cache RAM hit. Using these average delays does of course not correspond to reality, but the overall service_time will hardly be affected by it. Circumstances are thinkable, however, in which the calculated number of cycles differs from reality due to these average values. After all, a number of small penalties may not

increase the total number of cycles, whereas one large penalty can.

No penalty will be added when the server reads the quads from the read buffer. If a fetch buffer is used, the number of accesses from the fetcher to the RAM is equal to one instead of the number of quads in a transfer block per demand or prefetch. This is modelled by dividing the average_delay defined above by the number of quads in a transfer block. The same can be told as above about using an average penalty instead of including a penalty when there is a cache RAM hit and the fetch buffer is indeed copied to the cache RAM.

5.3.4. Miss penalty.

To explain the benefits of some options more clearly and to indicate which assumptions are made in the timing model, the calculation of the miss penalty is treated in this section.

The following miss penalty is calculated for a cache equipped with a fetch buffer and the wrap around algorithm:

latency_time + no_quads_at_once * access_time

No_quads_at_once is 2 if the requested quads belong to the same transfer block, 1 if not. Depending on this, a request is processed at once or in two parts, as explained in section ?.

If a fetch buffer is used, but not the wrap around algorithm:

latency_time + (quad + no_quads_at_once) * access_time

In this case *quad* indicates the position of the processed quad in the transfer block, ranging from 0 to no_quads minus 1.

If wrap around is used, but no fetch buffer:

latency_time + ((bypass) ? no_quad_at_once * access_time :
no_quads * access_time + ((two_ported) ? 0 : ram_time) [1]

Wrap around is thus useless when no fetch buffer is used or no fetch bypass. If no two ported RAM is used, ram_time is calculated for the access of the data RAM after a transfer block has been stored in it. In this respect a fetch buffer is always two-ported.

[1]The language C expression (A) ? B : C; can be explained as: If expression A is true use expression B, if not C.

It is assumed that as soon as a quad is stored in it, it can be used directly by the server. The minimum delay in case of a fetch buffer hit, however, is buffer_time.

If nor a fetch buffer nor the wrap around algorithm is used, the following miss penalty is calculated:

```
latency_time + ((bypass) ? (quad + no_quads_at_once) * access_time :
                           no_quads * acess_time + ((two_ported) ? 0 : ram_time) )
```

A penalty is included for stopping demand or prefetches immediately when a miss occurs. Before the demand fetch is started, the time_count is increased to the minimum of time_count + access_time/2 and ready_time.

5.4. Trace generator.

It is possible to generate artificial traces by considering basic elements in reference patterns. The events which can be distinguished are listed below. A trace is generated by selecting these events according to a probability distribution, until the specified trace length is obtained.

- continuation of the current address sequence. The number of requests added to the trace when this event is selected, ranges from 0 to 20 (uniformly distributed).
- loops. Distinction is made between small and large loop lengths. The length of a small loop ranges from 0 to 20, of large loops from 0 to 100. The number of loops varies for both from 0 to 10.
- jumps. Distinction is made between near and far jumps. The displacement of near jumps ranges from 0 to 500, of far jumps from 0 to 100000. The ratio between forward and backward displacements is 9 : 1.
- function calls and returns. Distinction between near and far functions is made in exactly the same way it was made for jumps. The ratio between forward and backward displacements is 2 : 1. After each function call 0 to 50 successive addresses are assumed. After this, any of the events listed here is possible, thus not necessarily a function return. Functions can be nested to a maximum depth of 3.

It will be clear that no attempts should be made to generate a system trace this way. Therefore, all process identification numbers are equal to zero. To include the penalties of task switches nevertheless, cache flushes are inserted and relative large displacements for jumps and function are used.

Two successive addresses differ 2 instead of 1, since only the address of the first quad is supplied when a quad pair is requested. Although the process identification number is actually the most significant part of the address, each request consists of a process

identification number followed by the remaining part of the address. This is caused by the maximum of 32 bits for integers in C, whereas a complete address would require 46 (16 + 30) bits.

The traces obtained with the trace simulator are certainly inferior to real traces. The trace generator, however, has one small advantage. The characteristics of the traces can be changed easily, so it is easy to experiment with different reference patterns. Two different traces have for example been used to investigate the influence of jumps and loops. Table 4 contains the probability distribution of the events mentioned above. Both traces are of length 100000.

The trace generator certainly has some imperfections. The way loops are modelled, for example, is one of the shortcomings. They only contain sequences of addresses, although jumps and function calls are also possible within loops. No attempt is made to remove all imperfections, since the trace generator will only be used until real traces are available.

Appendix B.2 informs about the usage of the trace generator.

Table 4: Event probability distributions in % of traces used for simulations.

event	probability (%)	
	trace A	trace B
cache flush	1	1
sequence	34	34
small loop	10	7
large loop	10	3
near jump	10	15
far jump	5	10
call near function	10	10
call far function	5	5
function return	15	15

6. Simulation results.

It is hardly possible (and useful) to simulate all possible combinations of options, especially when the effects have to be studied for different cache sizes and main memory access times. The following approach has therefore been used: It is tried to improve the cache performance little by little. First the impacts of changing the set size, block size and transfer block size are studied. This is followed by choosing the best prefetch algorithm. Then the effects of using buffers or two-ported RAM are studied. Finally it is investigated if options like wrap around, bypass and prefetch_stop, result in a better cache performance. One has to be, however, very careful using this approach. After all, many cache parameters are related to each other. For example, the decision about the best block size should not be made before the effects of dividing blocks into transfer blocks is also studied.

Only the most interesting simulation results are shown in this thesis. The results of the other simulations, as well as the parameter and output tables of all simulations, are available on the Digital Systems Group. An overview of the performed simulations is contained in appendix C. This information is also important when the simulations have to be rerun with new traces in the future.

6.1. Defaults.

To avoid that all chosen sizes and options have to be listed each time, default values are specified here.

Except when simulation results are presented as function of the cache size, the cache size is equal to 1024. With exception of paragraph 6.2, the set size is always 2 and the LRU replacement algorithm is used.

Unless specified differently, no features like buffers, two-ported RAM, fetch bypass, wrap_around, prefetch_stop and demand_fetch_stop are used.

As stated before, the cycle_time is always equal to 100. The default values of the other timing parameters can be divided into two groups: Those for *static simulations* and those for *dynamic simulations*. For the latter the values are:

access_time = 150
latency_time = 200
ram_time = 50
buffer_time = 20 (If no buffers are used it is equal to 0.)

Static simulations are used to show what the simulation results are when no timing information is taken into account. The last four mentioned timing parameters are in that case equal to 1. If it is not specified if the simulations are static of dynamic, then they are dynamic.

Trace A is the default trace. (See paragraph 5.4). The number of cycles between receipt of requested quads and the next request by the instruction unit has a default value of 1.

6.2. Set size.

Figure 12 and Figure 13 contain the miss ratio and the service_time as function of the cache size for different set sizes. The prefetch_always and the LRU replacement algorithms are used. Both the block and the transfer block size are 8 quads. The simulations are dynamic. (See the previous paragraph.)

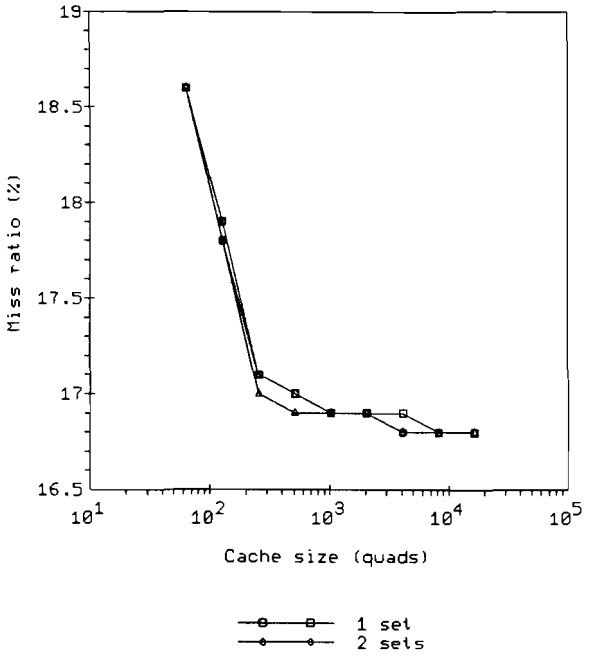

Figure 12: The miss ratio as function of the cache size for different set sizes.

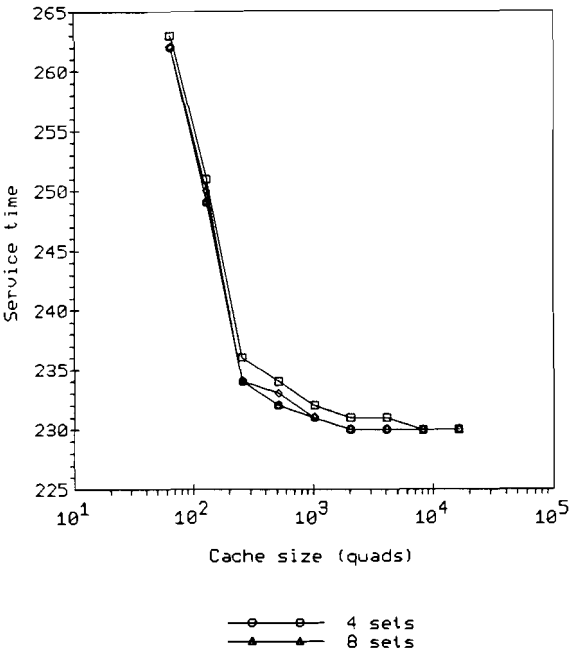

Figure 13: The service_time as function of the cache size for different set sizes.

The improvements due to larger set sizes are very small. This is probably caused by the main disadvantage of the used traces: No functions are contained in the loops. Especially for these kinds of reference patterns set sizes larger than 1 are profitable. The artificial

traces, therefore, are not suited to study changes in the set size.
According to the literature [Smith_A.82], no large improvement is to be expected when the set size is increased. Other cache parameters affect the cache performance much more. In many cases a set size of 2 is chosen, since the benefits of larger set sizes are often very small.

Since the differences are already small when the LRU replacement is used, no experiments are done with the random and the FIFO replacement algorithms. After all, these algorithms will probably not be better than LRU, they are only used in some cases because they can be implemented more easily.

From now on the set size will always be 2 and the LRU replacement will be used.

6.3. Block size.

The effects of changing the block size are studied in this paragraph. In all cases the transfer block sizes are equal to the block sizes.

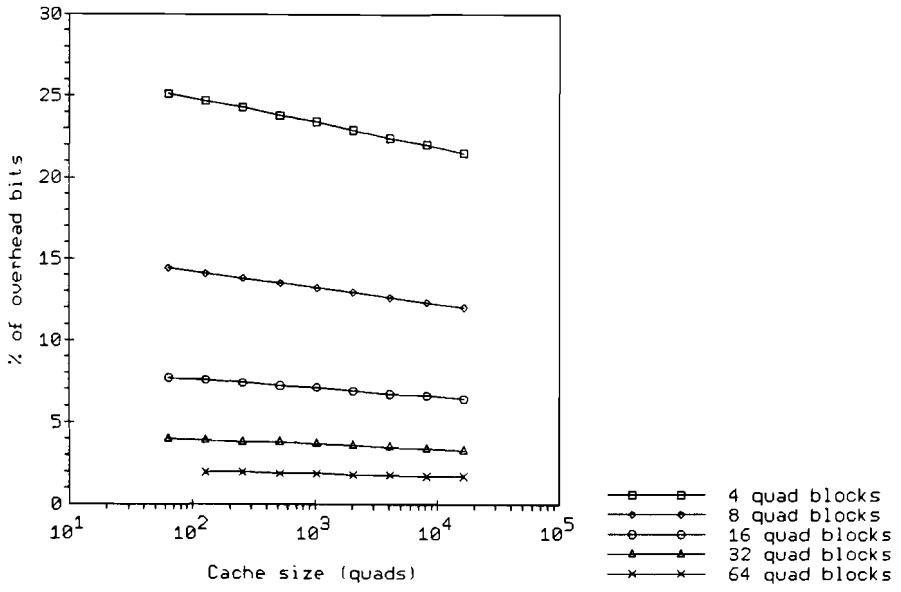

Figure 14: The percentage of overhead bits as function of the cache size for different block sizes.

The effect of increasing the block size on the number of overhead bits is shown in figure Figure 14. Besides that the percentage of overhead bits decreases for larger blocks,

another effect is visible: The percentage of overhead bits, for a certain block size, decreases when the cache size increases. This is caused by the increase of sets. Due to the larger number of set bits, the number of tag bits decreases and thus the number of overhead bits.

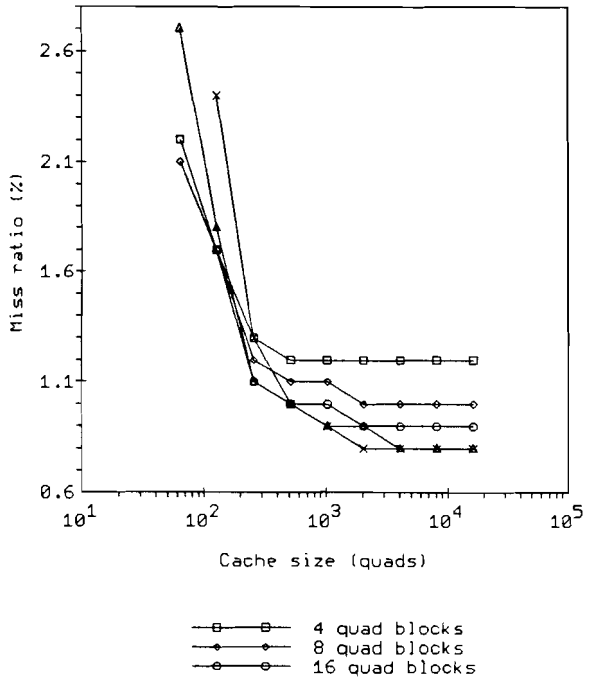

Figure 15: The miss ratio as function of the cache size for different block sizes. Static simulations, prefetch_always.

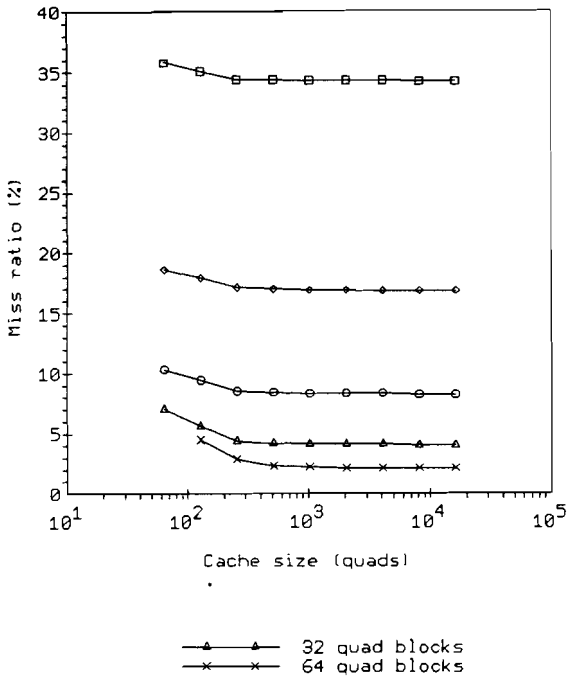

Figure 16: The miss ratio as function of the cache size for different block sizes. Dynamic simulations, prefetch_always.

The miss ratio for static simulations is shown in Figure 15. First the miss ratios decrease fast as the cache sizes becomes larger, but after a certain cache size, which depends on the block size, the miss ratio does not decrease at all.
If the cache is "large enough", the largest blocks result in the lowest miss ratio, the smallest blocks in the highest. If the cache is small, on the other hand, the usage of large blocks is adverse.

Figure 16 contains the miss ratio as function of the cache sizes for dynamic simulations. The main difference is that the miss ratio is generally much larger. This difference is caused by taking into account that information being prefetched is not immediately available, contrary to the assumptions made in the simulations of the previous figure.

Another difference is that there is hardly any decrease of the miss ratio when the cache

size becomes larger. If we look at Figure 17, where the prefetch_lookup_on_hits instead of the prefetch_always algorithms is used, we see that this is caused the usage of the inefficient prefetch_always algorithm. Figure 15 did not show the inefficiency of the prefetch_always algorithm, because the disadvantage of unnecessary prefetches is of no importance for static simulations.

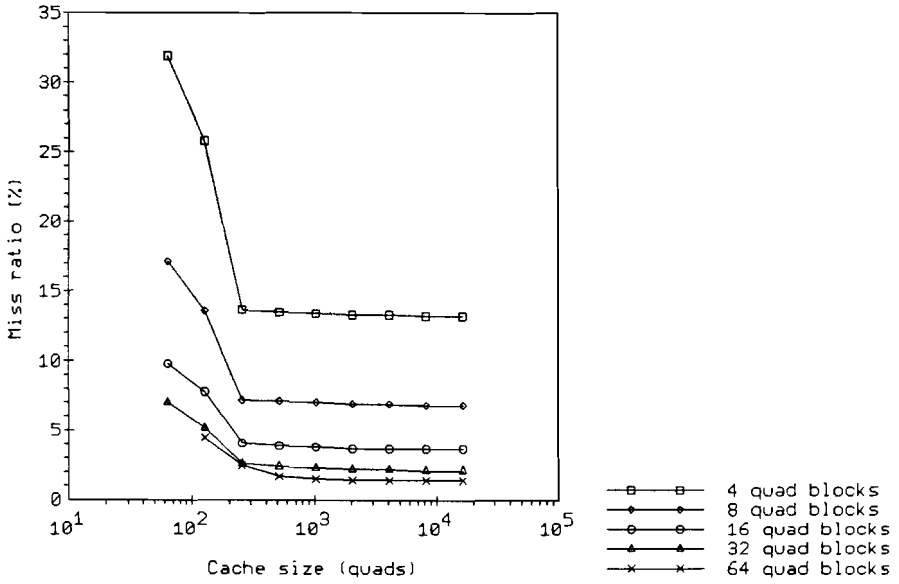

Figure 17: The miss ratio as function of the cache size for different block sizes. Dynamic simulations, prefetch_lookup.

Throughout this study no significant differences in simulations results have been found for prefetch_lookup_on_hits and prefetch_lookup_always. Therefore, no results of the latter will be shown and when prefetch_lookup is used in the remainder of the text, prefetch_lookup_on_hits is intended.

For the prefetch_lookup algorithm the miss ratio is generally lower than for the prefetch_always algorithm. The miss ratio decreases fast as the cache size increases, but improves only slightly after a certain point.

Figure 15 to Figure 17 study the influence of the block size on the miss ratio in a very limited way. After all, the blocks are not divided in transfer blocks, just two different sets of timing parameters are used, only two different prefetch algorithms are used, no options like buffers are used, etc. The number of simulations, however, is 3 * 5 * 9 - 3

= 132. (The number of figures[1] * number of graphs in each figure * number of simulations in each graph, which is equal to the number of marks. This has to be corrected for the fact that the "64 quad blocks" graphs contain only 8 simulations, since the smallest possible cache is 128 quads for this block size and a set size of 2.)
The large amount of required simulations for such a restricted study, underlines the importance of being able to run many simulations easily.

The service_times as function of the cache size for both dynamic simulations are depicted in Figure 18 and Figure 19.

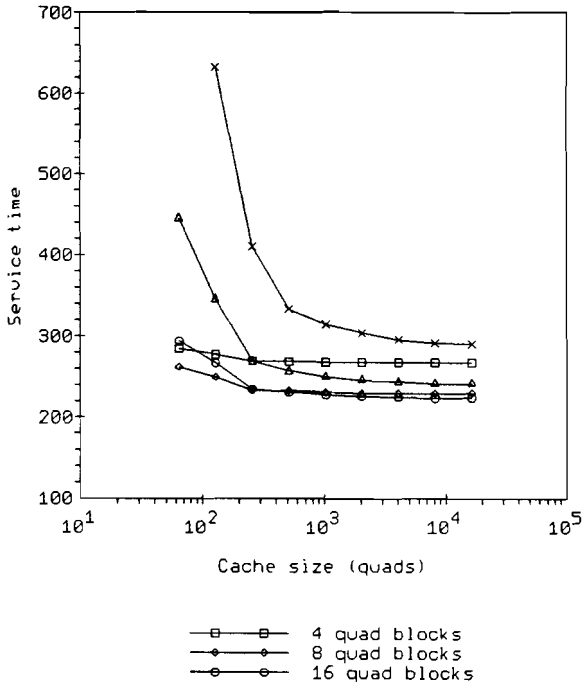

Figure 18: The service_time as function of the cache size for different block sizes. Prefetch_always.

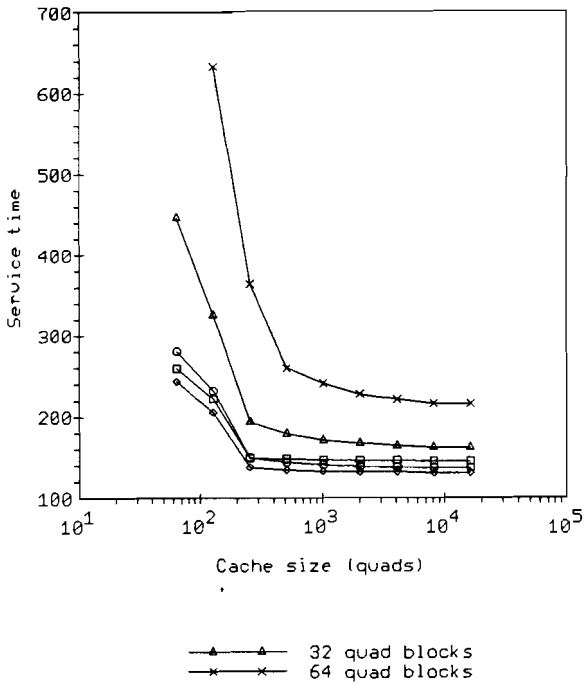

Figure 19: The service_time as function of the cache size for different block sizes. Prefetch_lookup.

These figures do not correspond with those about the miss ratios. A block size of 64 quads is according to the service_time the worst choice, whereas it was generally the best according to miss ratio. Thus, if one considers miss ratios only, even when the simulator itself takes timing into account, wrong choices can be made!
For large caches the optimum block size is 16 quads when the prefetch_always algorithm

[1] Of course did not each shown figure require additional simulations, since a number of figures can be generated from a set of simulations. E.g., the remaining figures in this paragraph, with exception of the last one, are generated from the same simulations.

is used, 8 when prefetch_lookup is used. Again, the prefetch_lookup algorithm is better than prefetch_always.

Figure 20 and Figure 21 show the traffic and the scaled traffic ratios for prefetch_always and prefetch_lookup respectively.

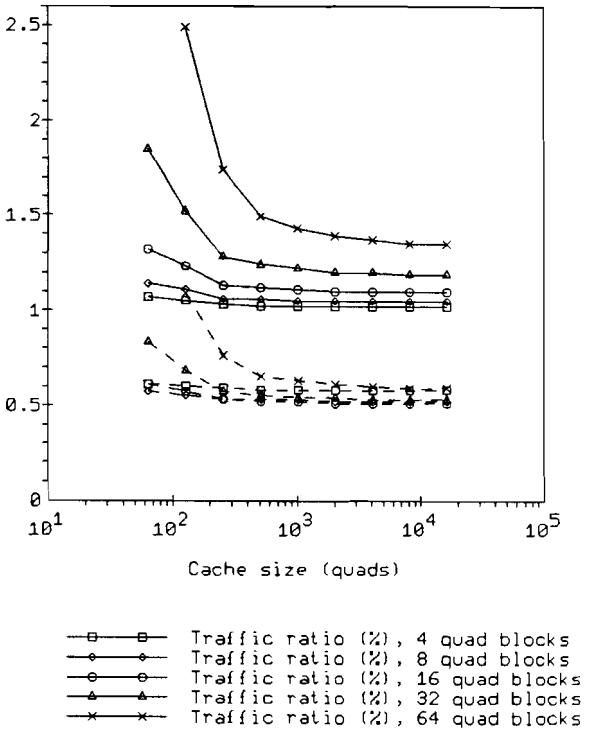

Figure 20: The traffic and scaled traffic ratios as function of the cache size for different block sizes. Prefetch_always.

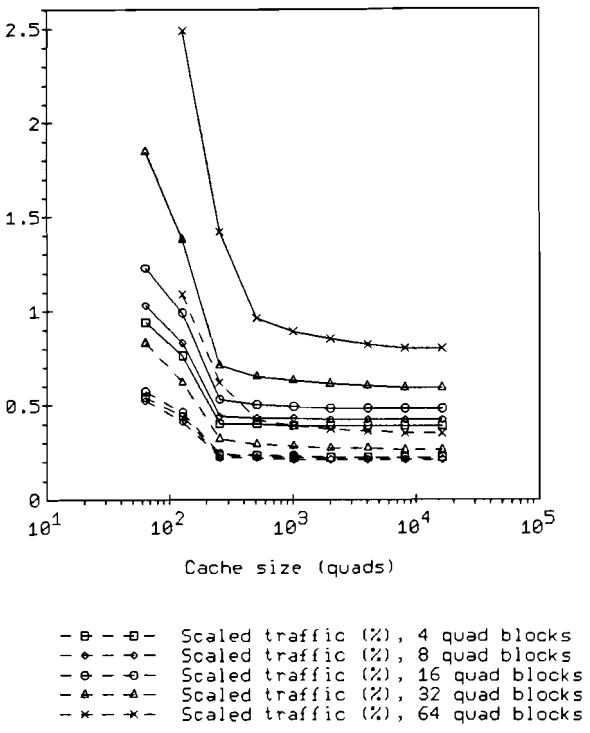

Figure 21: The traffic and scaled traffic ratios as function of the cache size for different block sizes. Prefetch_lookup.

This figures explain why the prefetch_lookup algorithm results in better performance than prefetch_always: If the latter algorithm is used, the traffic ratio is much larger. This is caused by unnecessary prefetches.
Besides this, we see that the traffic ratio becomes larger when larger blocks are used. The difference between the traffic ratio and the scaled traffic ratio is significant. Therefore, it is important to make distinction between the amount of transported data from main memory to the cache and the time required for it.

The memory pollution in case the prefetch_always algorithm is used, is depicted in Figure 22.

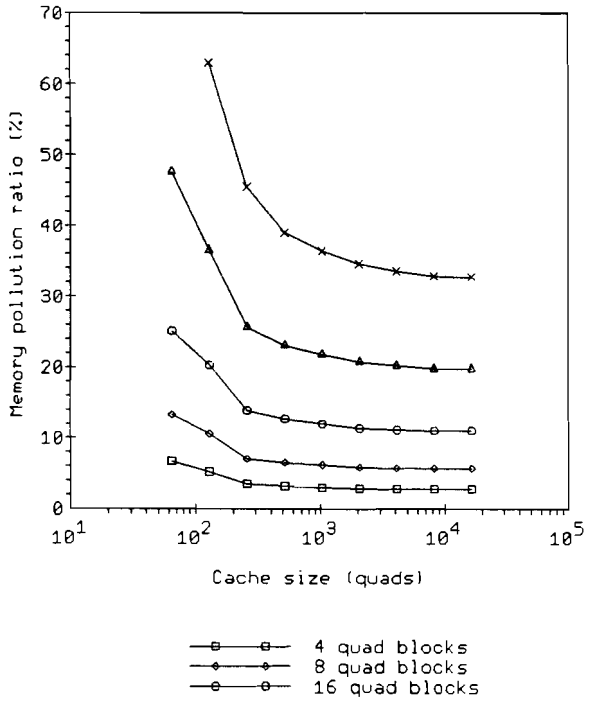

Figure 22: The memory pollution as function of the cache size for different block sizes. Prefetch_always.

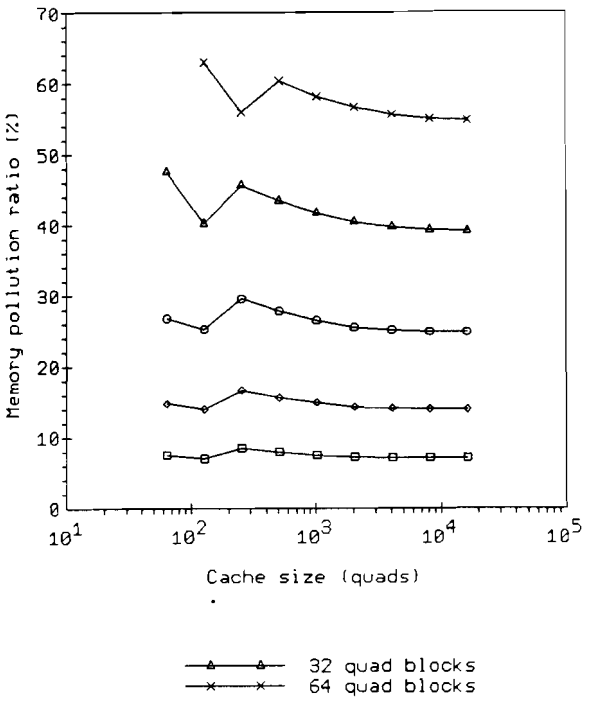

Figure 23: The memory pollution as function of the cache size for different block sizes. Prefetch_lookup.

The memory pollution is larger for larger blocks and decreases when the cache size becomes larger.
If the prefetch_lookup algorithm is used, however, the memory pollution is rather different. See Figure 23. First a small decrease can be recognized, which is followed by a small increase. When the cache size becomes larger, the memory pollution decreases only slightly and is much larger than in case prefetch_always is used. This difference is caused by the difference in the traffic ratio. The traffic ratio decreases much more for prefetch_lookup than for prefetch_always as the cache size increases. Since the memory pollution is defined as the number of unused quads divided by the total number of quads read from main memory, the memory pollution for prefetch_lookup hardly decreases.

Figure 24 shows the memory pollution for prefetch_lookup in case the alternative definition had been used for the memory pollution: the number of unused quads divided by the total number of quads read from the cache by the instruction unit.

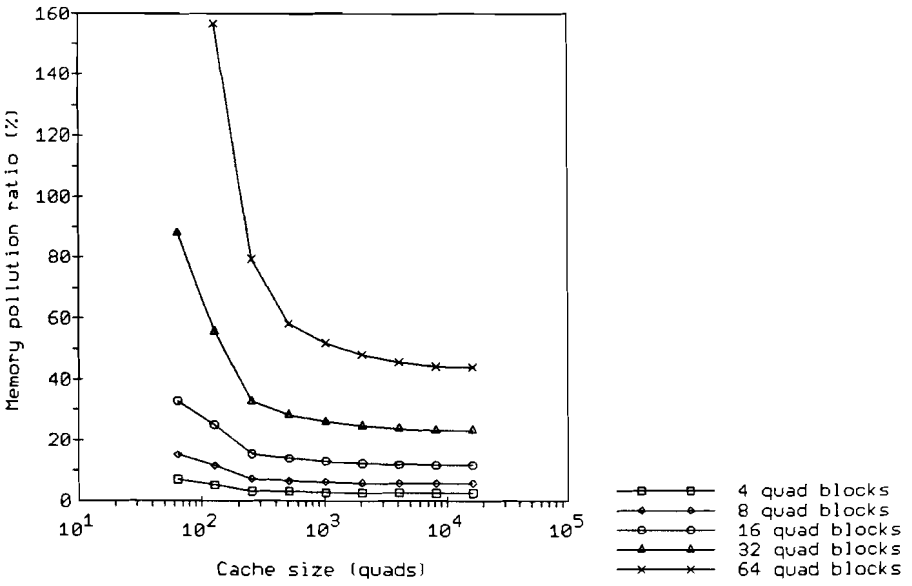

Figure 24: The memory pollution alternatively defined (see text) as function of the cache size for different block sizes. Prefetch_lookup.

6.4. Transfer block size.

The benefits of dividing blocks into transfer blocks will be studied in this paragraph. From now on, the block size will be 32 quads, unless stated differently.

The miss ratio as function of the cache size for different transfer block sizes is presented in Figure 25. Dynamic simulations are shown and the prefetch_lookup algorithm is used. The miss ratio is affected adversely when blocks are divided in transfer blocks. The smaller the transfer blocks, the worse. As for the block size, however, it is misleading to consider the miss ratio only. Figure 26, which contains the service_time, shows that the optimum transfer block size is 8 and not 32. More about the optimum transfer block size after the traffic ratio as function of the cache size has been shown.

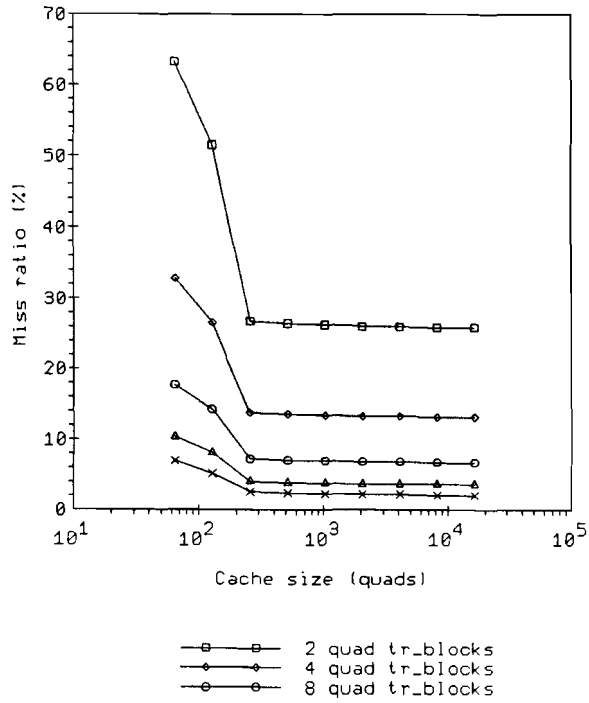

Figure 25: The miss ratio as function of the cache size for different transfer block sizes. Block size = 32.

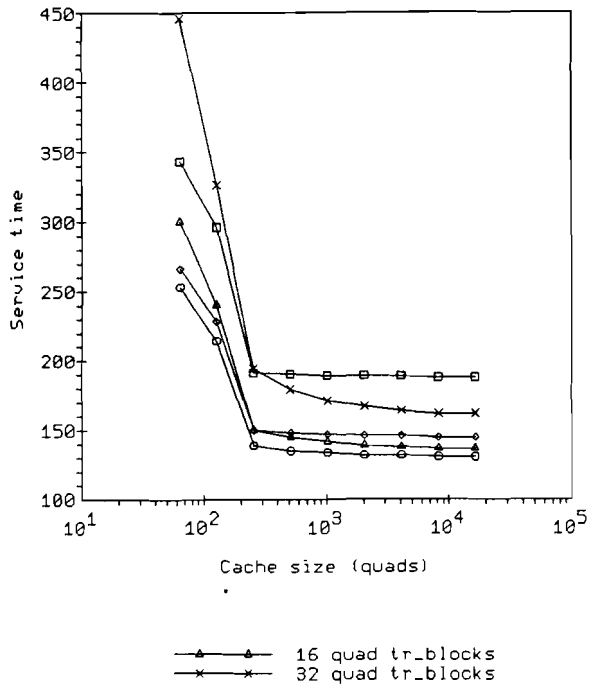

Figure 26: The service_time as function of the cache size for different transfer block sizes. Block size = 32.

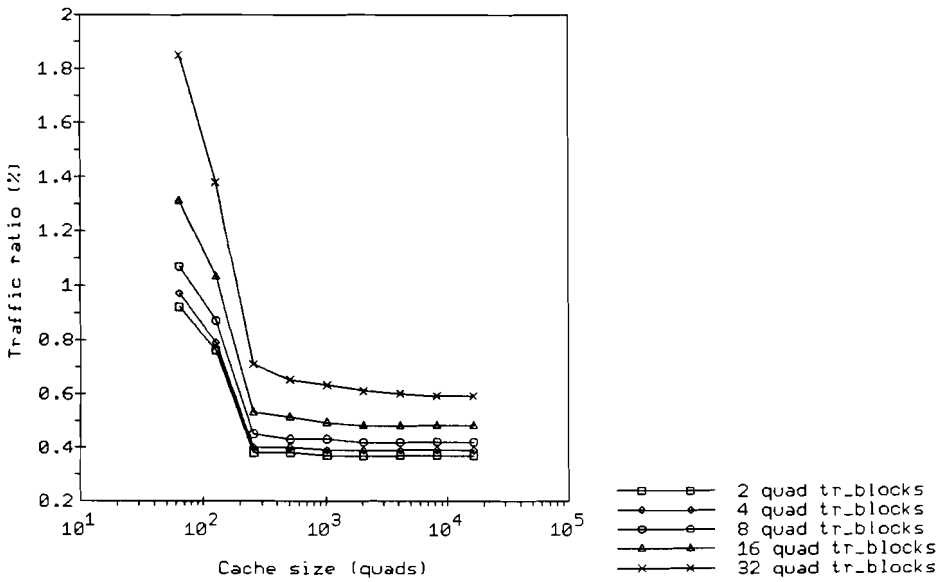

Figure 27: The traffic ratio as function of the cache size for different transfer block sizes.

Figure 27 shows the decrease of the traffic ratio when the blocks are divided in transfer blocks. (For clarity the scaled traffic ratio is not drawn.)

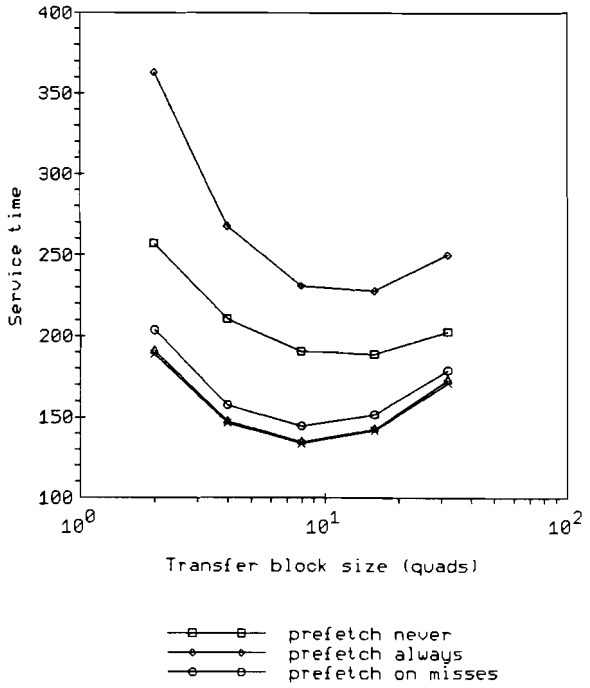

Figure 28: The service_time as function of the transfer block size for different prefetch algorithms. Block size = 32.

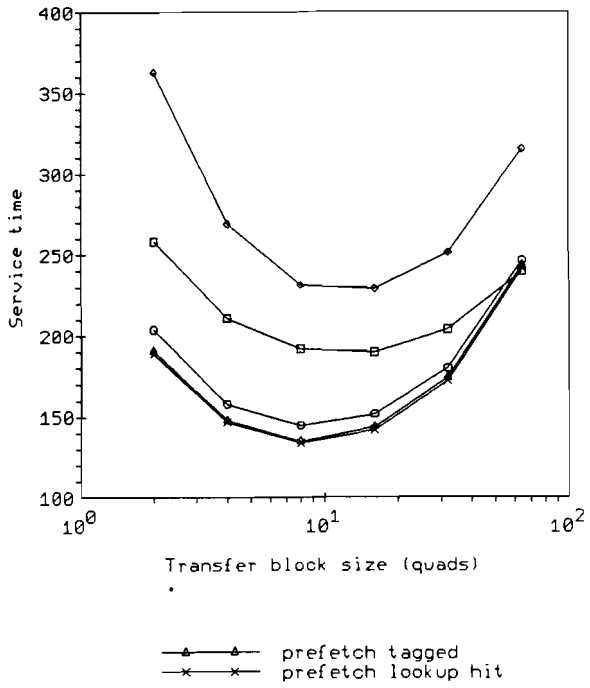

Figure 29: The service_time as function of the transfer block size for different prefetch algorithms. Block size = 64.

From Figure 26 it became clear that an optimum value exists for the transfer block size. Figure 28 and Figure 29 contain the service_times as function of the transfer block size for different prefetch algorithms for a block size of 32 and 64 respectively.

These drawings clearly show that it is profitable to divide blocks in transfer blocks. The transfer blocks, however, must not be made too small. If the prefetch_never or the prefetch_always algorithms are used, the optimum transfer block size is 16, else it is 8. The prefetch_always algorithm is the worst, even worse than prefetch_never! Prefetch_tagged is almost as good as prefetch_lookup.

Until now the latency_time has always be equal to 200. The service_time as function of the transfer block size for different latency_times is depicted in Figure 30. (Block size = 32.) The service_time is of course lower when the latency_time is smaller. For a latency_time of 0 for the optimum transfer block size is 4, and for a latency_time of 400 it is 16. The optimum transfer block size changes for different latency_times, since the benefits of fetching information in burst mode decrease when the latency_time becomes smaller.

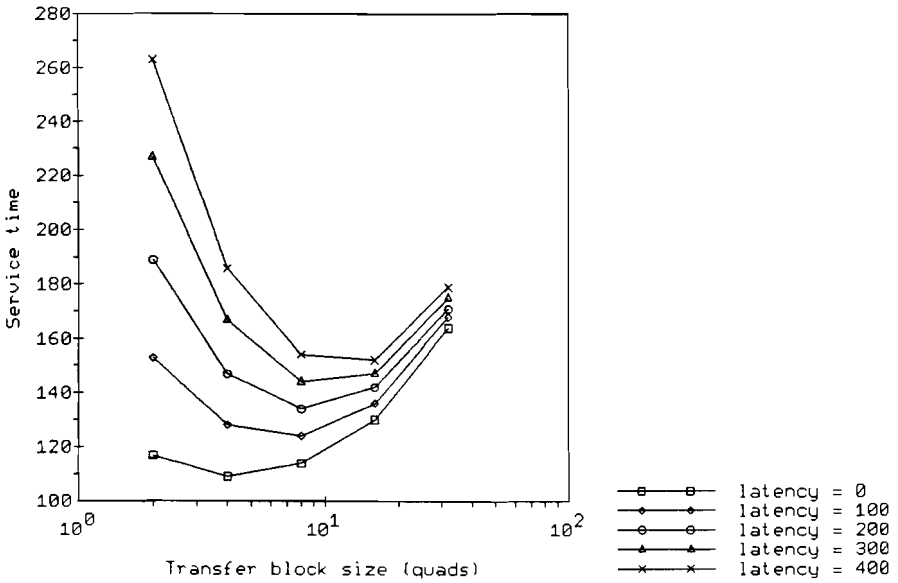

Figure 30: The service_time as function of the transfer block size for different latency times.

In the remainder of the text the latency_time will always be 200. The block size will always be 32 and the transfer block size 8.

6.5. Prefetch algorithm.

The effects of different prefetch algorithms have already been treated to a certain extent above. Therefore, only a few results are shown in this paragraph. For the difference between prefetch_lookup_on_hits and prefetch_lookup_always, see the note on page 58.

Figure 31 and Figure 32 show the miss ratio for different cache sizes of static and dynamic simulations respectively.

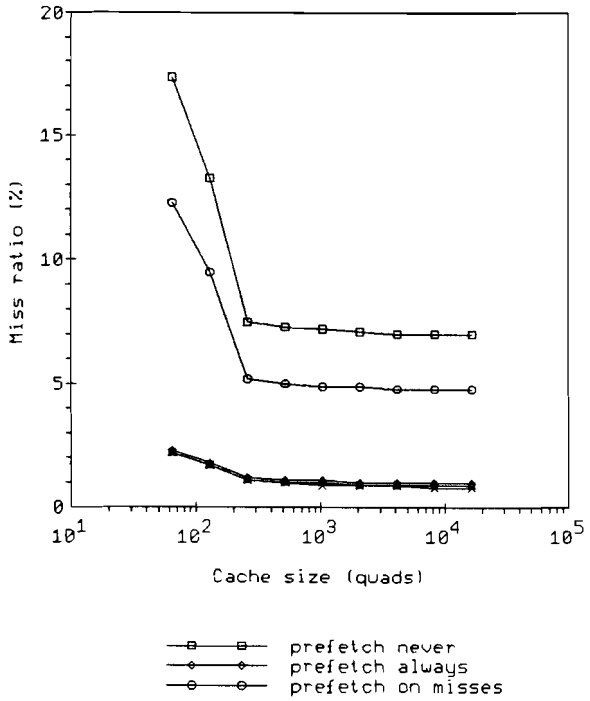

Figure 31: The miss ratio as function of the cache size for different prefetch algorithms. Static simulations.

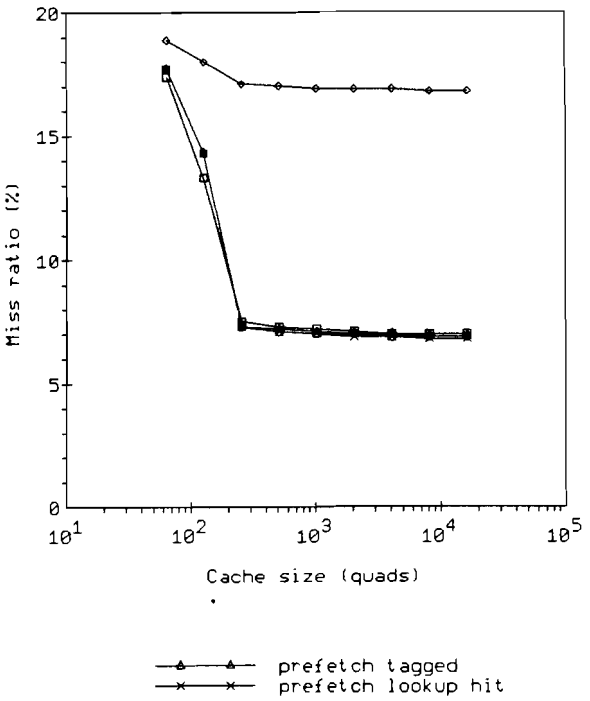

Figure 32: The miss ratio as function of the cache size for different prefetch algorithms. Dynamic simulations.

For the static simulations prefetch_never is the worst, followed by prefetch_always. For dynamic simulations, however, prefetch_always is the worst and the miss ratio is generally higher. Prefetch_lookup is the best, but not much better than the remaining algorithms. The service_time, see Figure 33, indicates larger differences.

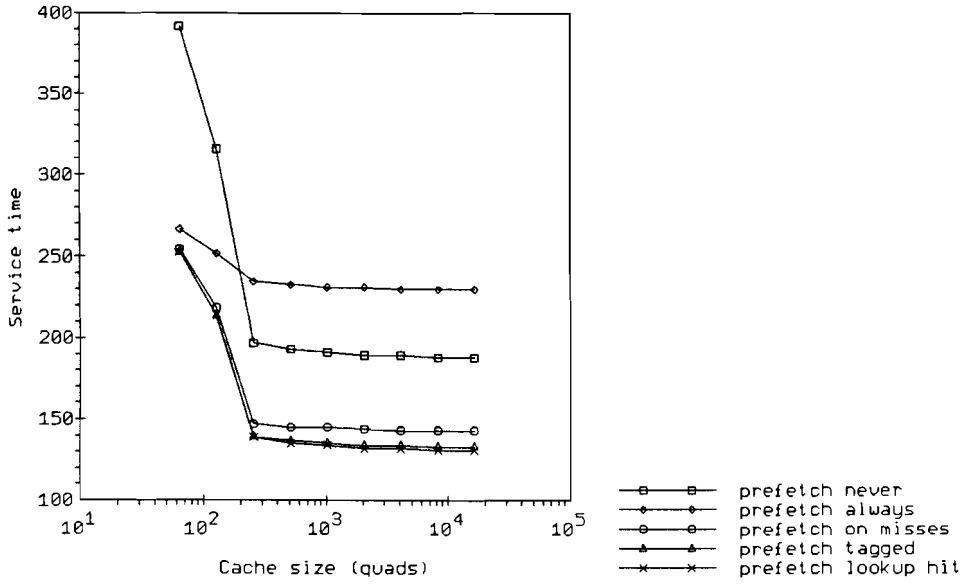

Figure 33: The service_time as function of the cache size for different prefetch algorithms. Dynamic simulations.

The usage of the prefetch_lookup algorithm results in the lowest service_time. Tagged_prefetching, however, is only slightly worse.

6.6. Buffers.

Figure 34 shows the service_time as function of the access_time. Prefetch_never is used. The usage of a read buffer is a significant improvement. The usage of a fetch buffer results in an even better performance. The best results are obtained when both buffers are used.
Two-ported RAM, on the other hand, decreases the service_time only slightly. The difference indicates the penalties calculated when the fetcher and the server try to access the cache RAM simultaneously. See paragraph 5.3.3. The penalties calculated in the simulator are thus very small. This is caused by the assumption that the internal timing of the cache is totally asynchronous. In reality these penalties will probably be larger.

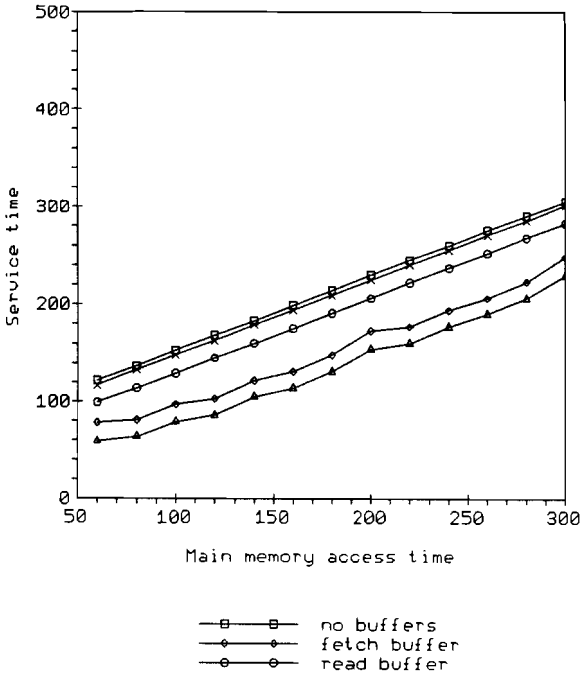

Figure 34: The service_time as function of the access_time for different buffer options. Prefetch_never, trace A.

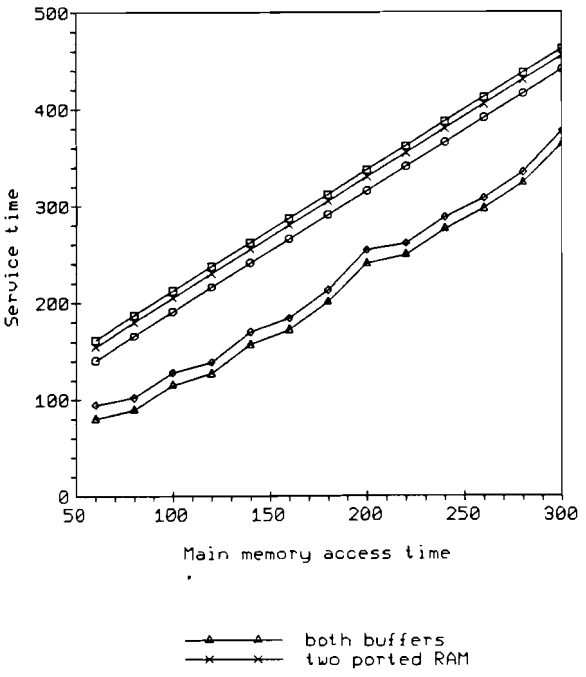

Figure 35: The service_time as function of the access_time for different buffer options. Prefetch_never, trace B.

Figure 34 showed the service_time in case trace A is used. Trace B is used for Figure 35. This trace contains relatively more jumps and less loops, see paragraph 5.4. The service_time is generally higher for this trace. The improvement due to the fetch buffer, is relatively larger when more jumps occur.

To explain the twisting graphs when a fetch buffer is used, with or without a read buffer, the access_time interval is decreased for the simulations using trace A. See Figure 36. The spikes are caused by the synchronous communication between the cache and the processor. (This became clear by using the simulation modes which generates information about each interval instead of overall ratios. See appendix B.1.). At the tops of the spikes the requested quads are available at the end of a cycle. After approximately 1 cycle the next request is made by the processor. If the access_time, however, is a bit larger, the requested quads are often available just after the beginning of a cycle. In that case the processor reads the quad approximately 1 cycle later and another cycle is waited before the next request is made. In the mean time the fetcher can already fetch the next information. Therefore, there are only spikes in the service_time, not in the number_of_cycles.

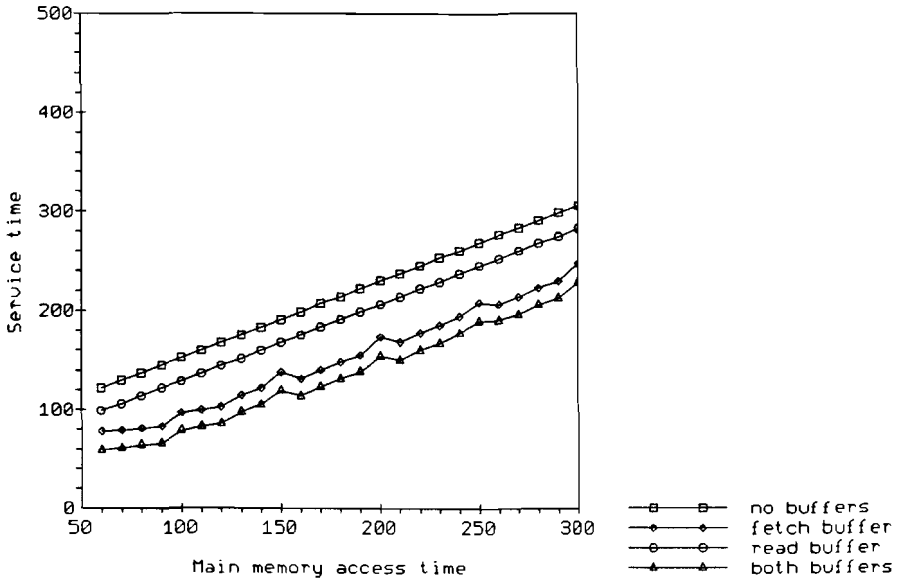

Figure 36: The service_time as function of the access_time for different buffer options. Prefetch never, trace A, smaller intervals.

Figure 37 and Figure 38 depict the service_time in case prefetch_lookup is used for trace A and trace B respectively. Due to the usage of prefetching, the service_times are lower. Again, in both cases is the usage of both buffers results in the best performance.

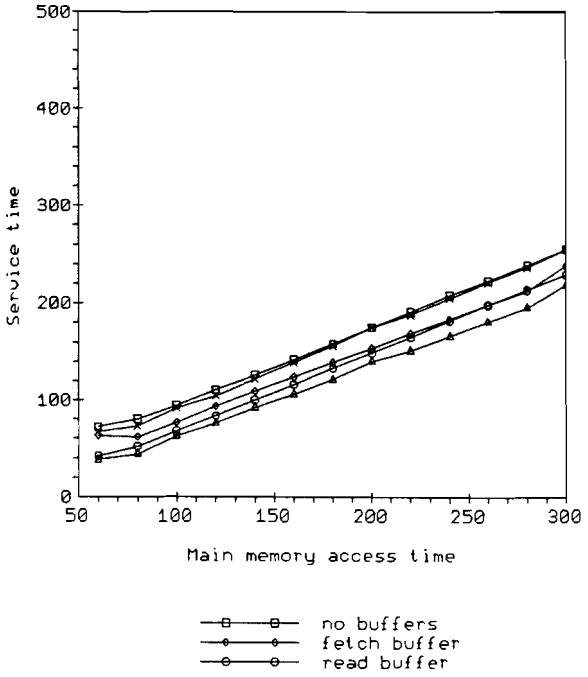

Figure 37: The service_time as function of the access_time for different buffer options. Prefetch_lookup, trace A.

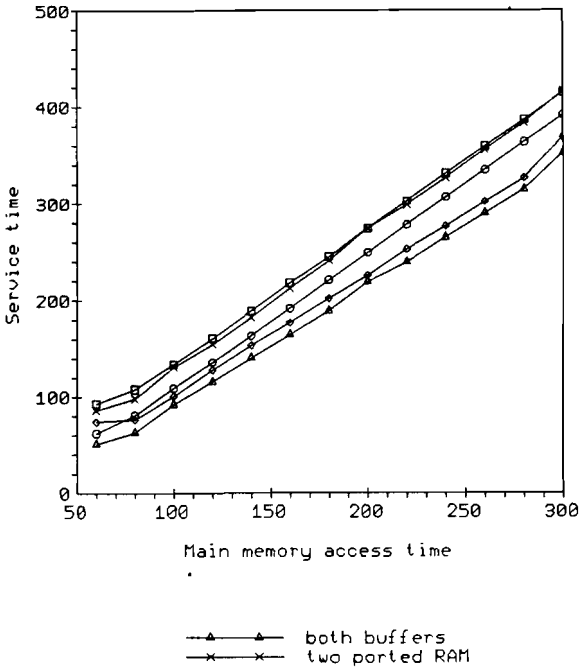

Figure 38: The service_time as function of the access_time for different buffer options. Prefetch_lookup, trace B.

If no prefetching is used, the improvement due to the fetch buffer was larger than for the read buffer. If prefetch_lookup is used, on the other hand, the simulations with trace A show the opposite. This is caused by the following effect:
If there is no fetch buffer, a demand fetch has to be finished before information can be read from it. (Assume no fetch bypass.) When the complete transfer block has been fetched, the processor starts using it and the next transfer block is prefetched. If no jumps occur, the processor will request information from this next transfer block after some time. This transfer block is then partly or completely stored in the cache.
If, however, a fetch buffer is used, the processor is able to use information which is being fetched from main memory. This decreases the initial delay, but has also one unexpected disadvantage: The fetcher is almost always busy fetching the current transfer block and seldom the next one. The next transfer block will seldom be prefetched before it is requested. Due to this effect, the benefits of a fetch buffer are restricted a bit. This effect becomes smaller when the reference pattern is less sequential. See Figure 38, where the improvement by the fetch buffer is larger than the improvement of the read buffer.

Nevertheless, it can be concluded that the usage of the fetch buffer and the read buffer

both decrease the service_time considerately.

Until now we have concentrated on the service_time. The number_of_cycles are shown in Figure 39 and Figure 40 for prefetch_never and prefetch_lookup respectively. Trace B is used in both cases.

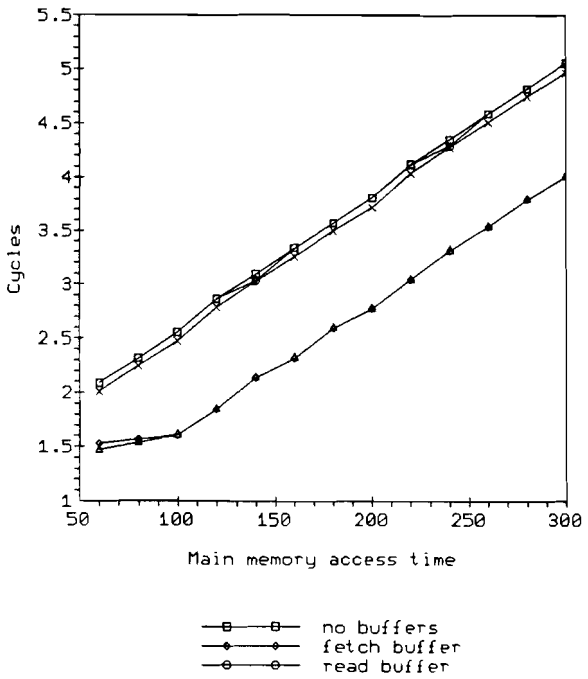

Figure 39: The number_of_cycles as function of the access_time for different buffer options. Prefetch_never, trace B.

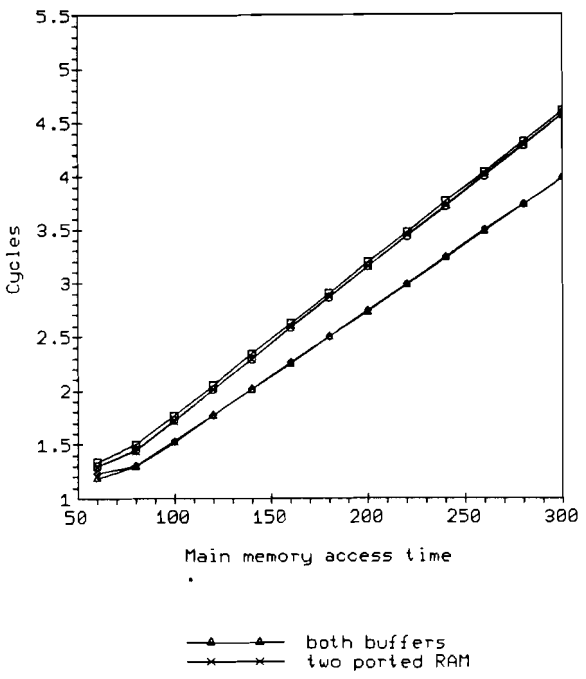

Figure 40: The number_of_cycles as function of the access_time for different buffer options. Prefetch_lookup, trace B.

The usage of a fetch buffer is in both cases a good improvement. If no prefetching is used, the usage of a read buffer does not decrease the number_of_cycles. This is logical since the buffer_time (=20) and the ram_time (=50) are both smaller than the cycle_time (=100). If prefetching is used, the usage of a read buffer results in a slightly better performance, since the number of simultaneous accesses to the RAM decreases when a read buffer is used. It is already stated that the penalties calculated by the simulator are probably to low, since they are based on total asynchronous internal timing. A read buffer, therefore, will probably decrease the number_of_cycles to a larger extent than shown above.
The benefits of the read buffer will be larger when the ram_time is larger than the cycle_time, or when the communication between the processor and the cache is asynchronous. (In that case only the service_time has to be considered.)

Figure 41 contains the miss and hit ratios when the prefetch_never is used and trace B is used for the simulations.
The miss ratio is independent from the usage of the buffers, so only one graph is drawn for it. Besides that it is independent from the usage of buffers, it is independent from the access_time. If no buffers are used, the cache RAM hit ratio is thus also independent from the access_time.
If a read buffer is used, the cache RAM hit ratio becomes much lower. This ratio and the read buffer hit ratio are both independent from the access_time.
The amount of quads read from the fetch buffer becomes larger when the access_time increases. Because of this, the cache RAM hit ratio and, if a read buffer is used, the read buffer hit ratio are dependent from the access_time.

It can thus be concluded that without prefetching, the miss and hit ratios are independent from the access_time, unless a fetch buffer is used. (Options like fetch bypass and wrap around are not taken into account here.)

Figure 42 shows the miss and hit ratios when prefetch_lookup is used. If prefetching is used, all miss and hit ratios become access_time dependent.

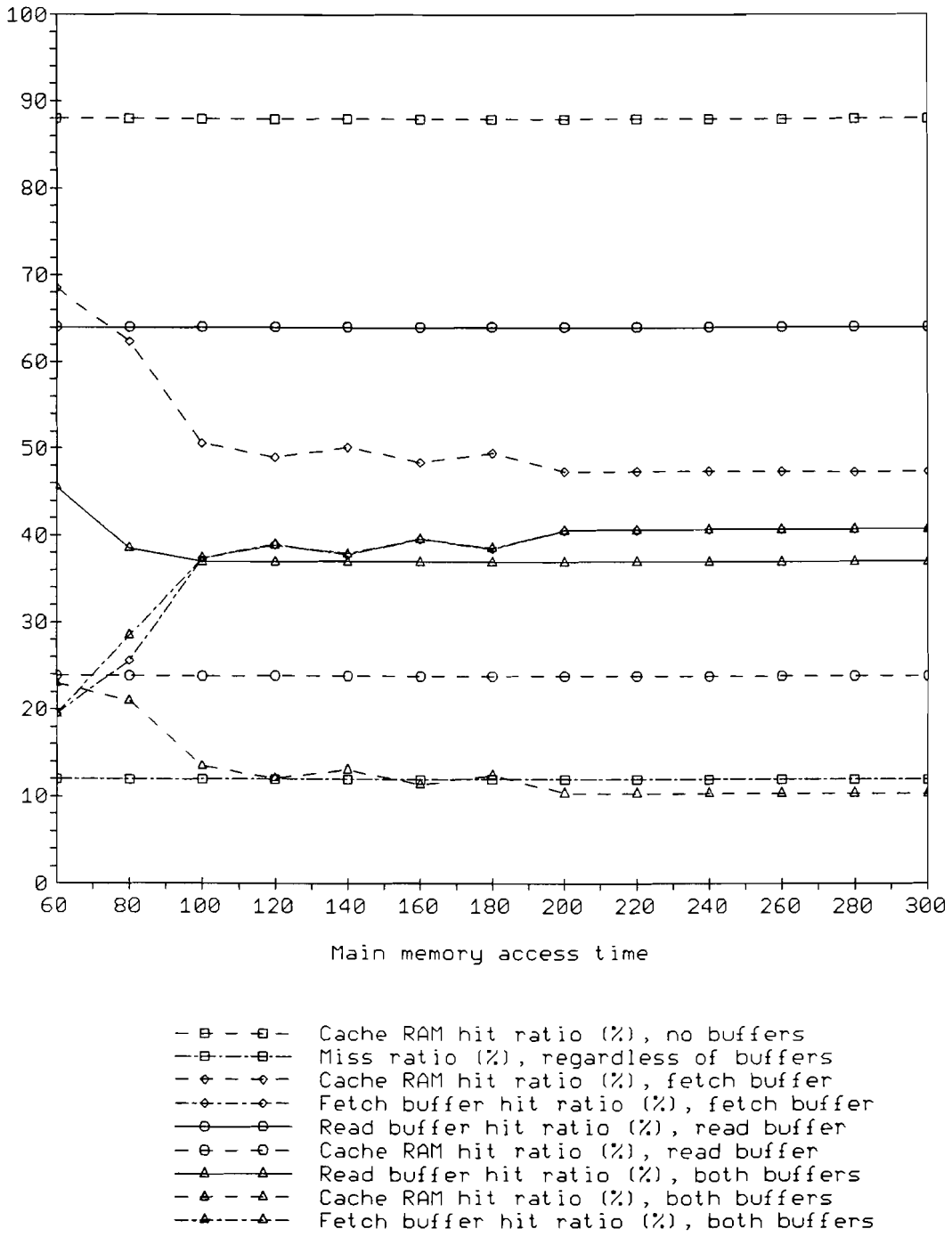

Figure 41: The miss and hit ratios as function of the access_time for different buffer options. Prefetch_never, trace B.

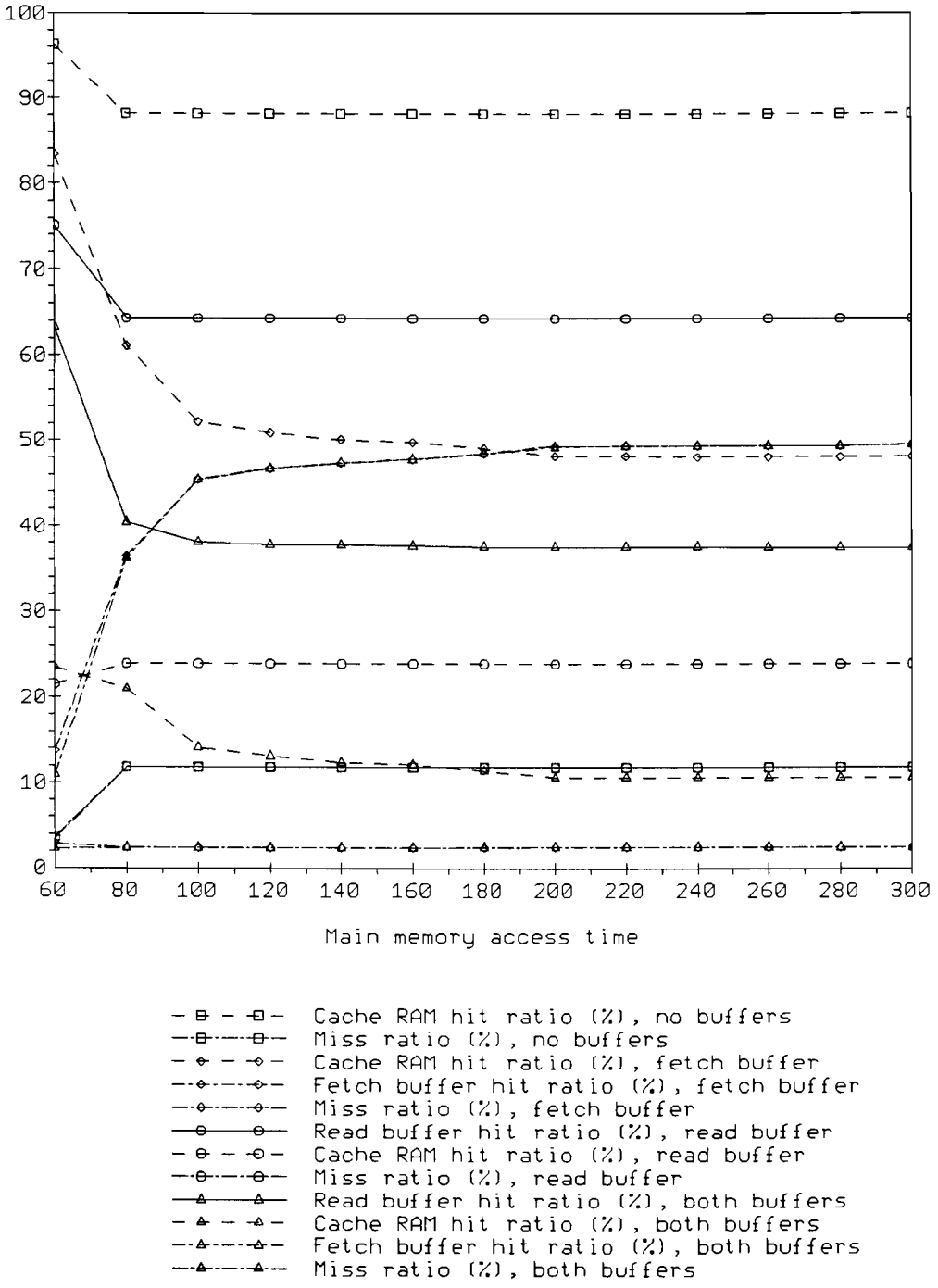

Figure 42: The miss and hit ratios as function of the access_time for different buffer options. Prefetch_lookup, trace B.

Until now the number of cycles between receipt of the requested quads and the next request by the processor has been 1. Figure 43 and Figure 44 show the service_times when this number of cycles is changed, using no buffers or both buffers respectively.

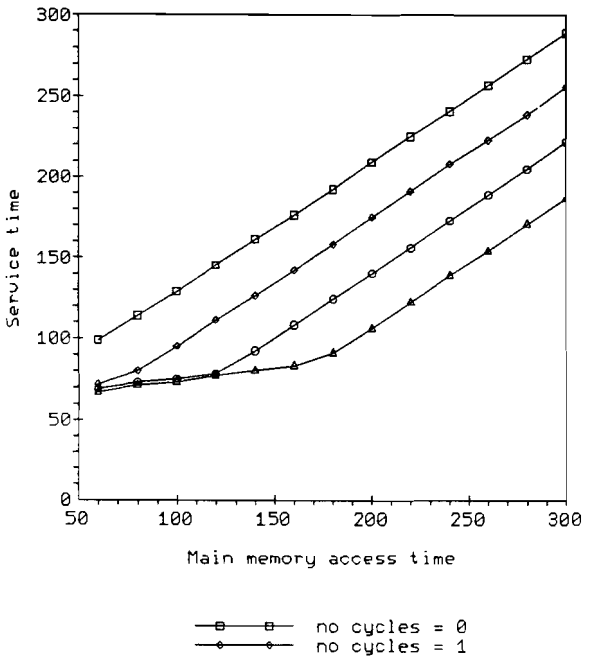

Figure 43: The service_time as function of the access_time for different number of cycles between requests. Prefetch_lookup, trace A, no buffers.

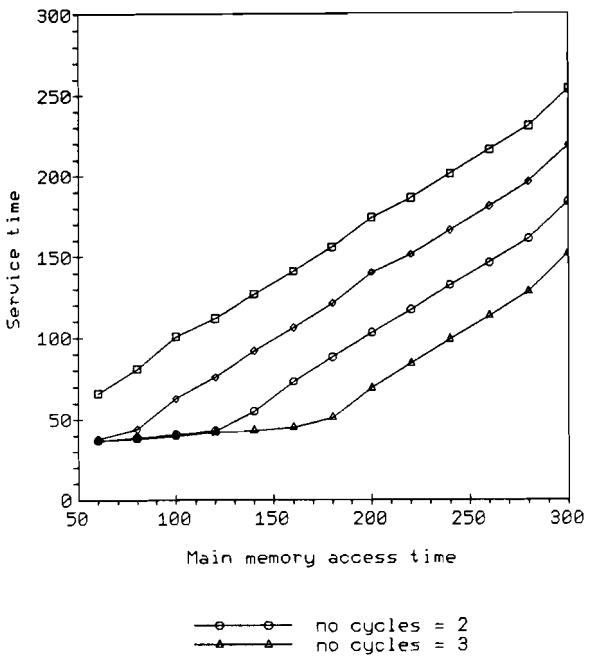

Figure 44: The service_time as function of the access_time for different number of cycles between requests. Prefetch_lookup, trace A, both buffers.

The service_time is of course lower when the processor request rate is lower. If the access_time, however, is very small, the fetcher is able to work fast enough and the differences become smaller.
After a certain access_time, the absolute differences between the graphs are almost constant. If the processor request rate increases, the same cache performance can be obtained by making the access_time smaller.
Figure 43 and Figure 44 show that the number of cycles between receipt of requested quads and the next request by the processor is very important. Therefore, it has been made possible to use variable values for this during simulations, by storing it with each request in the trace file.

6.7. Other cache options.

Finally the remaining cache options will be investigated: prefetch_stop, demand_fetch_stop, wrap around and bypass.

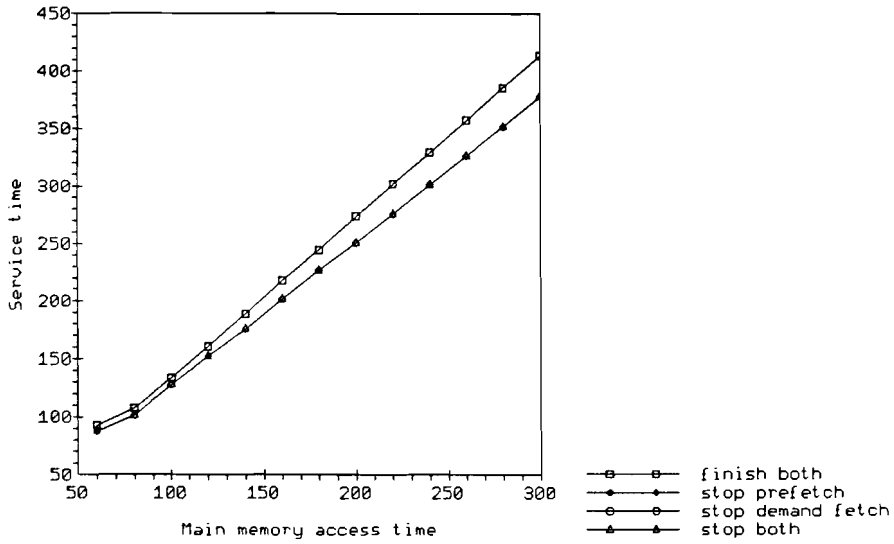

Figure 45: The service_time as function of the access_time when demand or prefetches are stopped immediately on misses. Trace B, no buffers, no wrap around.

Figure 45 shows that it useless to use demand_fetch_stop when no buffers are used. This is caused by the fact that the demand fetching of a transfer block has to be finished before the processor can use information from it when no fetch buffer is used. Therefore, it is not possible that the fetcher is demand fetching when a miss occurs. The usage of prefetch_stop, on the other hand, is an improvement.

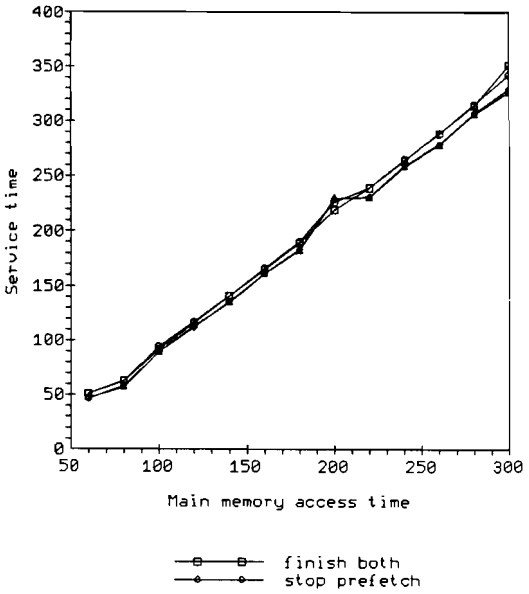

Figure 46: The service_time as function of the access_time when demand or prefetches are stopped immediately. Trace B, both buffers, no wrap around.

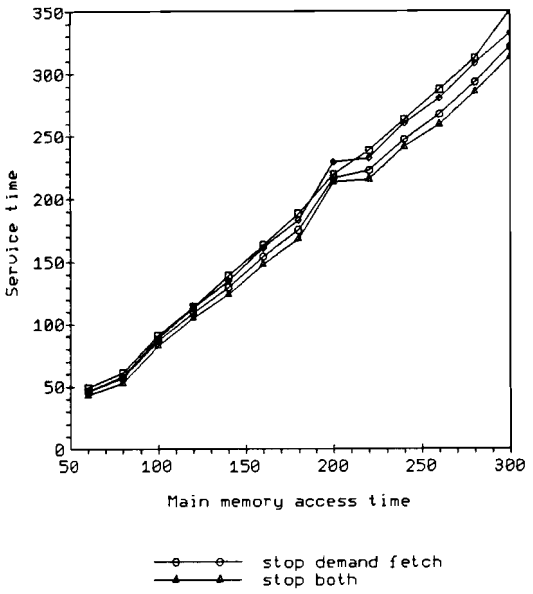

Figure 47: The service_time as function of the access_time when demand or prefetches are stopped immediately. Trace B, both buffers, wrap around.

Figure 46 and Figure 47 depict the service_times when prefetching is used without wrap around and with wrap around respectively. The advantages of the stopping of demand or prefetches immediately are not always larger than the drawbacks. Wrap around is only profitable when demand_fetch_stop is used. If wrap around is used, the usage of both demand_fetch_stop and prefetch_stop results in the best cache performance.

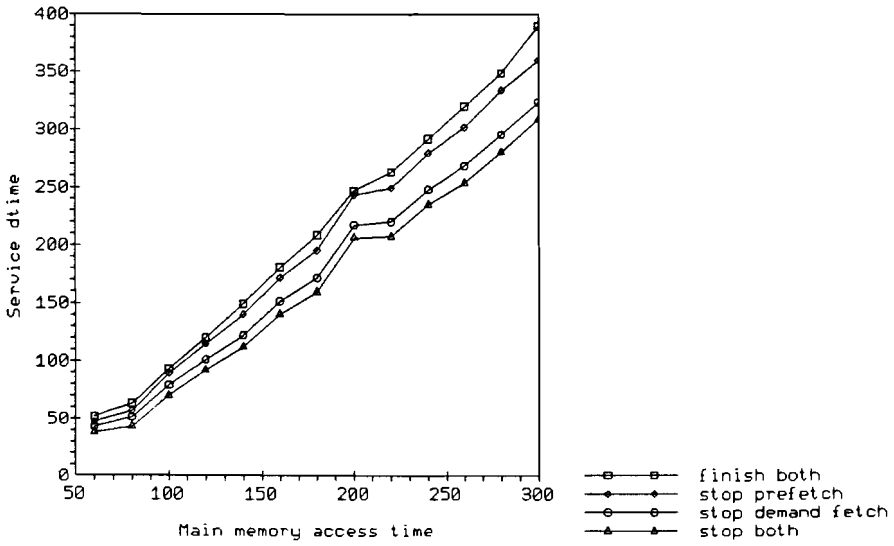

Figure 48: The service_time as function of the access_time when demand or prefetches are stopped immediately. Trace B, both buffers, wrap around. Block size = 64, transfer block size = 16.

The improvements due to demand_fetch_stop and prefetch_stop are very small. If a transfer block size of 16 instead of 8 had been chosen, the improvements would have been larger. (This transfer block size, however, is not the optimum value, it is too large.) This is shown in Figure 48.

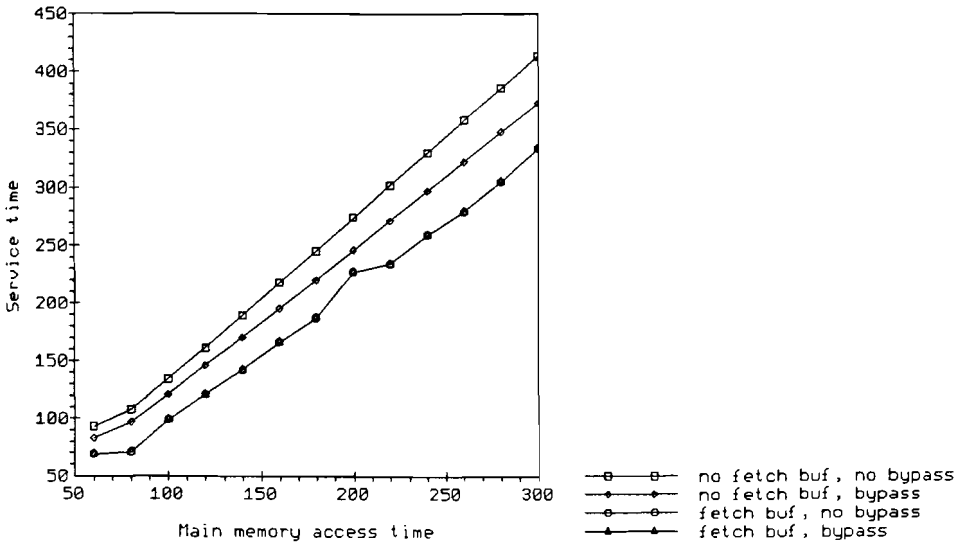

Figure 49: The service_time as function of the access_time when bypass and/or a fetch buffer is used. Trace B, wrap around, demand_stop.

Figure 49 shows the benefits of fetch bypass. Wrap around and demand_fetch_stop are used. (Otherwise fetch bypass is almost useless.) If no fetch buffer is used, fetch bypass results in a lower service_time. The usage of a fetch buffer is much more effective. If such a buffer is used, however, fetch bypass does not result in an improvement. This is in accordance with the simulation model, in which the buffers are assumed to be latches and two-ported. No additional service_time reduction is calculated in that case if fetch bypass is used.

7. Conclusions and future research.

7.1. Conclusions.

In this master thesis a study on instruction caches is presented. Cache memories can be used to increase the speed of a processor-memory system on a cost-effective base. In order to function effectively, however, caches must be designed carefully. Many cache parameters do exist. Besides the cache size, the set size, the block size and the transfer block size have to be chosen. The cache organization is reflected by the set size. A small number categories of algorithms are needed within an instruction cache. Different algorithms are possible for each category. Furthermore, a number of features, like buffers between the main memory and the cache and between the cache and the processor, are optional.

Since it is just not possible to "guess" the right sizes, algorithms and options, a simulator is indispensable. A simulator has therefore been written, which is able to perform trace driven simulations. This simulator is general in two respects: Firstly, it is able to model many cache parameters. Secondly, it is not only applicable for the instruction cache of the C-processor, but can also be used for other caches. The only connection with the C-processor is the instruction bus width from the main memory to the cache and the amount of information requested at once by the instruction unit. These interfaces, however, can be changed easily in the simulator. The usage of process identification numbers, which is characteristic for the C-processor, is not essential to the simulation model. When one wants to simulate a cache which is flushed during each task switch, one uses flush instructions in the trace file and does not change the process identification numbers, e.g. make them all zero.

It is very important to include timing information in the simulation model. Conclusions based on only metrics like the miss ratio and the traffic ratio can lead to less optimum configurations. Moreover, if prefetching, a fetch buffer or features like stopping demand or prefetches immediately after misses are used, these metrics are even not time independent anymore. Including timing information in the simulation model is thus not only necessary to calculate metrics like the average cache service time directly, but also to calculate the miss ratio and traffic ratio correctly.

Due to the large number of cache parameters, many simulations are necessary to find an optimum cache configuration. This is especially true when one wants to study this as function of cache size and main memory access time.
The ease of running many simulations and comparing the results of them is thus very

important.

The correspondence of the traces to the workload of the processor is of major concern. The simulation results based on the artificial traces can only be used to indicate tendencies. Some general conclusions based on the artificial traces are:

- after a certain cache size, the cache performance increases only slightly.
- it is worthwhile to divide blocks in transfer blocks. An optimum transfer block size can be determined for a certain cache implementation.
- one has to be careful choosing the prefetch algorithm. E.g. prefetch_always is sometimes worse than no prefetching at all. Prefetch_lookup seems to be the best choice.
- The benefits of two-ported RAM are small. The usage of a fetch buffer is in all cases a good improvement. A read buffer is able to reduce the service_time significantly, but due to the synchronous communication between the cache and the processor, the reduction in the number_of_cycles is limited. (If both the cache RAM and the read buffer access times are smaller than one clock cycle.)
- In some cases it is worthwhile to use options like prefetch_stop, demand_fetch_stop, fetch bypass or wrap around.

For more specific conclusions, like under which conditions certain cache options are beneficial, one is referred to chapter 6.

Real traces, preferably system traces, are needed to draw more well-founded conclusions. As soon as they are available, the instruction cache can be developed in a well-considered manner.

7.2. Future research.

The work on instruction caches will be continued by Yun Chao Hu as his graduation project. He is going to develop an instruction cache generator, comparable to already existing RAM and PLA generators. This generator will of course not only be used for the instruction cache of the C-processor. Therefore, the interfaces to the main memory and the instruction unit will be more flexible. Besides on-chip caches, it will also be possible to generate off-chip caches with it. This generator will be able to implement many of the features discussed in this report. Not only the cache size be will flexible, but also block, transfer block and set sizes. A flexible set size allows direct mapped, fully associative and set associative caches to be implemented. Different algorithms can be chosen and the read and the fetch buffer will be optional. It might be possible that a number of algorithms or features are eliminated based on simulations with real traces. This will not only simplify the development of the instruction cache generator, it will also prevent that the user has to choose from an unnecessary large number of options,

from which some are useless for any application.

The instruction cache simulator can be tailored to this instruction cache generator in the future. Some of the current input parameters, like the cache RAM access time and the buffer access time, must then be calculated automatically based on the specified sizes, the internal timing of the caches and the used technology. If the internal timing is known, it will also be possible to calculate more accurate penalties when the fetcher and the server try to access a single-ported RAM simultaneously. When the technology is known, it will be easy to predict the required area for the largest parts of the cache, the RAMs. When a number of test caches have been generated, it will also be possible to make good estimations of the remaining required area.
When the amount of area one is willing to sacrifice for the cache, the main memory access time and the processor cycle time is fixed or nearly fixed, the number of required simulations will be within acceptable bounds.
For the simulations traces must be used which one expects to be good representations of the workload for the processor the cache is intended for. When an acceptable or optimum configuration is found for the application, the cache can be generated automatically.

The instruction cache simulator can also serve as outline for a simulator for a data cache, which will also be necessary for the C-processor. The data cache will be more complicated than the instruction cache. Main memory update algorithms and cache coherence will also have to be taken into account. Besides this, the data cache of the C-processor must be able to serve two requests simultaneously. Due two this higher parallelism, it is possibly necessary to include a real event queue in the simulator or to use a programming language for it which is able to execute processes in parallel virtually. Papers of interest for data caches are: [Bazlen_81a, Bazlen_81b, Court, Ditzel, Eickemeyer, Goodman_83, Hasegawa, Hoichi, Kaplan, Lindsay, Stanley, Yen]. Some of these apply to both instruction and data cache, others apply to data cache caches only. I "encountered" these papers when I was gathering papers on instruction caches. Therefore, this list is certainly not complete. Especially on cache coherence much more has been written.

A. Literature.

[Alpert] Alpert, D.B., M.J. Flynn, "Performance trade-offs for microprocessor cache memories," *IEEE Micro*, Aug. 1988, pp 44-53.

[Barsamian] Barsamian, H., and A.L. DeCegama, "System design considerations of cache memories," *Digest of papers of the Six* Annual IEEE Computer Society International Conference, sept 1972, pp 107-110.

[Bazlen_81a] Bazlen, D., K.J. Getzlaff, J. Hajdu and G. Knauft, "Accelerating store-in-cache operations," *IBM Techn. Disclosure Bulletin*, vol. 23, no. 12, May 1981, pp. 5428-9.

[Bazlen_81b] Bazlen, D., J. Hajdu and G. Knauft, "Preventive cast-out operations in cache hierarchies," *IBM Techn. Disclosure Bulletin*, vol. 23, no.12, May 1981, pp. 5426-7.

[Bennett] Bennett, B.T., J.H. Pomerene, T.R. Puzak and R.N. Rechtschaffen, " Prefetching in a multilevel memory hierarchy,"

[Budzelaar] Budzelaar, F.P.M., "The structured design of a processor for the language C," Master thesis report EB 085, Technical University Eindhoven, 1988.

[Court] Court, J.F., K.L. Leiner, "Channel delays due to cpu cache back-ups," *IBM Techn. Disclosure Bulletin*, vol. 24, no. 1b, June 1981, pp 621-2.

[Ditzel] Ditzel, D.R., and H.R. McLellan, "Register allocation for free: The C machine stack cache," *Proc. ACM Symp. Architectural Support for the Prog. Lang. and Oper. Systems*, March 1982, pp. 48-56.

[Eickemeyer] Eickemeyer, R.J., and J.H. Patel, "Performance evaluation of on_chip register and cache organizations," *Proc. of the 15th Annual Int. Symp. on Computer Architecture*, May-June 1988, pp. 64-72.

[Goodman_83] Goodman, J.R., "Using cache memory to reduce processor-memory traffic," *Proc. of the 10th Annual Int. Symp. on Computer Architecture*, June 1983, pp. 124-131.

[Goodman_86] Goodman, J.R., Wei-Chung Hsu, "On the use of registers vs. cache to

minimize memory traffic," *Proc. of the 13th Annual Int. Symp. on Computer Architecture*, June 1986, pp. 375-383.

[Haikala] Haikala, I.J., "Cache hit ratios with geometric task switch intervals," *Proc. of the 11th Annual Int. Symp. on Computer Architecture*, June 1984, pp. 364-371.

[Hasegawa] Hasegawa, M., Y. Shigei, "High-speed top-of-stack scheme for VLSI processor: a management algorithm and its analysis," *Proc. of the 12th Annual Int. Symp. on Computer Architecture*, June 1985, pp. 48-54.

[Hill] Hill, M.D., A.J. Smith, "Experimental evaluation of on-chip microprocessor cache memories," *Proc. of the 11th Annual Int. Symp. on Computer Architecture*, June 1984, pp. 158-166.

[Hoevel_82] Hoevel, L.W., and J. Voldman, "Mechanism to detect bursts/gaps of cache miss activity," *IBM Techn. Disclosure Bulletin*, vol. 25, no. 7a, Dec. 1982, pp. 3397-8.

[Hoevel_83] Hoevel, L.W., and J. Voldman, "Burst-controlled prefetching to reduce finite cache penalty," *IBM Techn. Disclosure Bulletin*, vol. 26, no. 6, Nov. 1983, pp. 3056.

[Hoichi] Hoichi Cheong and A.V. Veidenbaum, "A cache coherence scheme with fast selective invalidation," *Proc. of the 15th Annual Int. Symp. on Computer Architecture*, May-June 1988, pp. 299-307.

[Kaplan] Kaplan, K.R., and R.O. Winder, "Cache-based computer systems," *Computer*, vol. 6, no. 3, March 1973, pp. 30-36.

[Kernighan] Kernighan, B.W., and D.M. Ritchie, "The C programming language," 1978, Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632.

[Lindsay] Lindsay, D.C., "Cache memory for microprocessors," *Comput. Archit. News*, vol. 9, no. 5, Aug. 1981, pp. 6-13.

[McNiven] McNiven, G.D., E.S. Davidson, "Analysis of memory referencing behavior for design of local memories," *Proc. of the 15th Annual Int. Symp. on Computer Architecture*, May-June 1988, 56-63.

[Patterson] Patterson, D.A., P. Garrison, M. Hill et al., "Architecture of a VLSI instruction cache for a RISC," *Proc. of the 10th Annual Int. Symp. on Computer Architecture*, June 1983, pp. 108-116.

[Pomerene] Pomerene, J.H., T.R. Puzak, R.N. Rechtschaffen and F.J. Sparacio, "Enhanced concurrency using a multilevel cache system,"

[Przybylski] Przybylski, S., M. Horowitz, J. Hennessy, "Performance tradeoffs in cache design," *Proc. of the 15th Annual Int. Symp. on Computer Architecture*, May-June 1988, pp. 290-298.

[Short] Short, R.T., H.M. Levy, "A simulation study of two-level caches," *Proc. of the 15th Annual Int. Symp. on Computer Architecture*, May-June 1988, pp. 81-88.

[Smith_A.82] Smith, A.J., "Cache memories," *ACM Computing Surveys*, vol. 14, no. 3, Sept. 1982, pp. 473-530.

[Smith_A.85a] Smith, A.J., "Problems, directions and issues in memory hierarchies," *Proc. of the 18th Hawaii International Conf. on System Sciences*, Jan. 1985, vol. 2, 468-476.

[Smith_A.85b] Smith, A.J., "Cache evaluation and the impact of workload choice," *Proc. of the 12th Annual Int. Symp. on Computer Architecture*, June 1985, pp. 64-73.

[Smith_J] Smith, J.E., and J.R. Goodman, "A study of instruction cache organizations and replacement policies," June 1983. pp. 132-137.

[Stanley] Stanley, T.J., R.G. Wedig, "A performance analysis of automatically managed top of stack buffers," *Proc. of the 14th Annual Int. Symp. on Computer Architecture*, June 1987, pp. 272-281.

[Withagen] Withagen, W.J., "Definition and high level description of the C-processor," Master thesis report, Technical University of Eindhoven, 1988.

[Wood] Wood, D.A., S.J. Eggers, G. Gibson, M.D. Hill et al., "An in-cache address translation mechanism," *Proc. of the 13th Annual Int. Symp. on Computer Architecture*, June 1986, pp. 358-65.

[Yen] Yen, W.C., and K.S. Fu, "Analysis of multiprocessor cache organiszations with alternative main memory update policies," *IEEE Annual International Symposium on Computer architecture*, May 1981, pp. 89-105.

B. Using the tools.

This appendix describes how the simulator, the trace generator and the postprocessor can be used and contains supplementary information about them. These programs run under both UNIX and MSDOS.

B.1. The simulator.

The simulator has two input modes, an interactive and a batch input mode. In the former mode the user is prompted for all options and the program is self-explaining. The latter, on the other hand, is very useful when simulations have to be rerun or when simulations have to be run which parameters files do not differ much from already available parameter files. Parameter files can then be generated fast by copying old ones and editing them.

The simulator can be used in interactive mode by entering:

```
ic_simul -i
```

The user is then prompted to enter the parameter, trace and output file names. The extensions .par, .trc and .out are catenated by the program, so the user should not enter them. Then the simulation mode has to be chosen (see paragraph 5.1.3). These modes can be used to obtain:

1. the number of misses in each interval. References to transfer blocks which are already being prefetched, are not regarded as misses in this mode. The misses are counted per quad and not per double quad. After all, if the quads are not in the same transfer block, it is possible that one of the two quads is available.
2. timing information and the status of the fetcher after each interval. The following information is presented:
   - the total time
   - the required time to serve the requested quads in the interval
   - whether or not the fetcher is prefetching
   - whether of not the fetcher is demand fetching
   - the time the fetcher will finish the demand or prefetch
   - the address of the transfer block being fetched
3. overall results. This is the common simulation mode.

If the last mode is chosen, the user is prompted for the number of simulations in the simulation frame. The special modes are restricted to single simulations, since large

amounts of data are produced.
If one of these special modes is chosen, the user is prompted for an interval size. This size is expressed in number of double quads requests. The overall ratios are calculated also for these modes.

Finally the user is prompted to enter the cache parameters, as described in paragraph 5.2. If there are multiple simulations, the user can choose which cache parameters have to be changed, instead of entering them all over again. The parameters which are not reentered by the user, get the values of the previous simulation.

Error checking is performed on the user inputs. An error message is displayed if an error is encountered and the user is permitted to correct it. It is always possible to change previously entered inputs, so mistakes can simply be corrected, instead of having to start all over.

The following command line syntax should be used for batch simulations:

```
ic_simul -b <parameter file> <trace file> <output file>
```

For the extensions the same applies as for the interactive mode.

Lines of the trace file which start with ";" are regarded as comment and are skipped. The "#" character is used to indicate cache instructions. The only cache instruction implemented is the flush instruction, which is issued by "#flush". Lines which do not start with ";" or "#" are regarded as (double) quad requests and are expected to have three items successively:

1. the process identification number.
2. the address of the first quad.
3. the number of cycles between deliverance of the requested quads and the next request.

B.2. The trace generator.

The trace generator, which purpose and properties are discussed in paragraph 5.4, can be invoked by entering:

```
tracegen
```

One is prompted successively to enter the name of the tracefile and the minimum length of it. The extension .trc is catenated by the program, so it should not be entered by the user. The length of the trace equals the number of double requests, the cache flushes are not counted in.

The characteristics of the traces can be changed by modifying the macros in the source file tracegen.c.

B.3. The postprocessor.

The graphs shown in this report are generated by PLOTUTIL, a program of Buts Electronic Systems. A postprocessor is written to convert parameter (.par) and output (.out) files of the simulator to input files of PLOTUTIL (.dat). As for the simulator, an interactive and batch input mode is provided. The postprocessor can be used interactively by entering:

```
postproc -i
```

The user is then prompted for the following:

- File name for future batch processing. The purpose of this file is explained below. The extension .ppb is catenated to this name, so it should not be entered by the user. The same applies for other filenames the user has to enter.
- The number of simulations frames (the number of.par/.out files) to postprocess. This number is restricted to a maximum of 5.
- The number of different output parameters. Maximum 5 different output parameters are allowed.
- The output parameters. The order in which they are entered should be in accordance with the order in the displayed list, which presents all output parameters. The numbers which reflect the different output parameters, have thus to be entered in increasing order!
- The name of the .dat file for each output parameter. Each name may equal the previous name (*the* previous, not one of the previous!), so it is possible to put the graphs of different output parameters in one drawing.
- The input parameter. Since the graphs are drawn as function of the input parameter, it is not advisable to choose one of the algorithms or the usage of buffers as input parameter. If, for example, graphs have to be generated to compare the three different replacement algorithms, it is better to use cache size, or main memory access time as input parameter and to alter the replacement algorithm over three .par files.
- The names of all .par and .out files, and a comment from the user for each pair. Since each parameter file is read, the frames do not necessarily have to contain the same number of simulations and the input parameters do not have to change uniformly over all frames.

The comment given by the user will be put in the legend of all graphs belonging to that

.par / .out pair. Graphs from different frames get different mark types. Graphs of different output parameters in one drawing get different line types. If just one output parameter is depicted in a drawing, its name is put on the y-axis. If different output parameters are stored in one drawing, their names are included in the legend. The name of the input parameter is always stored below the x-axis. For an input parameter which has to be a power of 2, like cache size, a logarithmic x-axis scale is used. Otherwise it is a linear scale.
It is possible to put defaults, like axis sizes, legend positions, dot sizes, etc, in a file called plot_defaults. The contents of this file is copied to each .dat file which is generated. The format of these commands can be found in the user manual of PLOTUTIL.

The above mentioned postprocessor batch (.ppb) file stores all relevant (no changes etc.) user inputs. New simulation results, for example due to the availability of a new trace file, can then be converted completely automatically by entering:

```
postproc -b <postprocessor batch file>
```

It is of course also possible to edit the .ppb file directly for small changes. These files are not as clear as the .par files, since all required information is just stored in it. They can, however, be understood by keeping the following in mind:

- The information is stored in the same order as entered in the interactive mode.
- The input parameter is expressed by a number which corresponds to the column order of the table of the parameter file. The cache size has number 0, the "fetch_stop" option has number 13. The output parameters are stored in the same manner. The number of total bits is thus reflected by 0, the scaled traffic ratio by 10.

C. Overview of simulations.

Only the most interesting simulation results are shown in this thesis. Since the simulations have to be rerun in the future, when real traces are available, it is important to know which parameter files and post processor batch files are already available and which graphs are generated. This appendix provides this information and tells in which directories the files can be found. The same order is used as in chapter 6.

Set sizes.

Directory:
sr_set (sr stands for simulation results)

Parameter files:
set#_s.par: static simulations, prefetch_always.
set#_d.par: dynamic simulations, prefetch_always.
set#_d5.par: dynamic simulations, prefetch_lookup.

is the set size: 1, 2, 4, 8.
The simulations are as function of: cache size.

Batch files are used to start a number of simulation frames and to postprocess the results automatically. Besides this, one parameter file and the output files are catenated. As an example, the contents of set_s.sim is shown:

```
../ic_simul -b set1_s ../t100000a set1_s
../ic_simul -b set2_s ../t100000a set2_s
../ic_simul -b set4_s ../t100000a set4_s
../ic_simul -b set8_s ../t100000a set8_s

cat set1_s.par set1_s.out set2_s.out set4_s.out set8_s.out > set_s.out

../postproc -b set
```

t100000a is trace A (length 100000). These batch files will be called *simulation batch files* from now on. Their contents will be described as below. (Unless the names of the output files are not similar to those of the parameter files, they are not mentioned.)

Simulation batch files (.par files, .trc file, .ppb file):

set_s.sim (set#_s.par, t100000a.trc, set_s.ppb)
set_d.sim (set#_d.par, t100000a.trc, set_d.ppb)
set_d5.sim (set#_d5.sim, t100000a.trc, set_d5.ppb)

Generated .dat files:
set_s_mis.dat: miss ratio, static simulations, prefetch_always.
set_s_tra.dat: traffic ratio
set_s_mem.dat: memory pollution
set_d_ser.dat: service time, dynamic simulations, prefetch_always
set_d_mis.dat, set_d_tra.dat, set_d_mem.dat,
set_d5ser.dat, set_d5mis.dat, set_d5tra.dat, set_d5mem.dat.

Each directory contains a MSDOS batch file (.bat) to generate the graphs automatically with PLOTUTIL.

Block sizes.

Directory:
sr_block

Parameter files:
block#_s.par: static simulations, prefetch_always.
block#_d.par: dynamic simulations, prefetch_always.
block#_d5.par: dynamic simualtions, prefetch_lookup.

is the block size: 4, 8, 16, 32, 64.
The simulations are as function of: cache size.

Simulation batch files (.par files, (.out files), .trc file, .ppb file):
block_s.sim (block#_s.par, t100000a.trc, block_s.ppb)
block_d.sim (block#_d.par, t100000a.trc, block_d.ppb)
block_d5.sim (block#_d5.par, t100000a.trc, block_d5.ppb)
mp_block_d5.sim (block#_d5.par, mp_block#_d5.out, t100000a, mp_block_d5.ppb)
The latter batch file controls simulations with the different memory pollution definition. This definition, however, has to be changed in the C-programs (ic_resul.c).

Generated .dat files:
bl_s_bit.dat: overhead bits.
bl_s_mis.dat, bl_s_tra.dat, bl_s_mem.dat,

bl_d_mis.dat, bl_d_ser.dat, bl_d_tra.dat, bl_d_mem.dat,
bl_d5mis.dat, bl_d5ser.dat, bl_d5tra.dat, bl_d5mem.dat,
mp_d5mem.dat: memory pollution, alternatively defined.

Transfer block sizes.

Directory:
sr_tr

Parameter files:
tr_block#_s.par: static simulations, prefetch_always.
tr_block#_d.par: dynamic simulations, prefetch_always.
tr_block#_d5.par: dynamic simualtions, prefetch_lookup.

is the transfer block size: 2, 4, 8, 16, 32.
The simulations are as function of: cache size.

Simulation batch files (.par files, .trc file, .ppb file):
tr_block_s.sim (tr_block#_s.par, t100000a.trc, tr_block_s.ppb)
tr_block_d.sim (tr_block#_d.par, t100000a.trc, tr_block_d.ppb)
tr_block_d5.sim (tr_block#_d5.par, t100000a.trc, tr_block_d5.ppb)

Generated .dat files:
tr_s_mis.dat, tr_s_tra.dat, tr_s_mem.dat,
tr_d_mis.dat, tr_d_ser.dat, tr_d_tra.dat, tr_d_mem.dat,
tr_d5mis.dat, tr_d5ser.dat, tr_d5tra.dat, tr_d5mem.dat,

Transfer block sizes / prefetch algorithms.

Directory:
sr_trprf

Parameter files:
tr32_pref#.par: dynamic simulations, block size = 32.
tr64_pref#.par: dynamic simulattions, block size = 64.

is the prefetch algorithm: 1, 2, 3, 4, 5, 6 (see .par files).
The simulations are as function of: transfer block size.

Simulation batch files (.par files, .trc file, .ppb file):
  tr32_pref.sim (tr32_pref#.par, t100000a.trc, tr32_pref.ppb)
  tr32_pref.sim (tr32_pref#.par, t100000a.trc, tr32_pref.ppb)

Generated .dat files:
  tr_32_p_s.dat, tr_64_p_s.dat: service_times.
  tr_32_p_m.dat, tr_64_p_m.dat: miss ratios.

Transfer block size / latency.

Directory:
  sr_trlat

Parameter files:
  tr_lat#.par

is: 0, 1, 2, 3, 4, which corresponds to the latency: 0, 100, 200, 300, 400.
The simulations are as function of: transfer block size.

Simulation batch files (.par files, .trc file, .ppb file):
  tr_lat.sim (tr_lat#.par, t100000a.trc, tr_lat.ppb)

Generated .dat files:
  tr_lat_s.dat: service_times.

Prefetch algorithm.

Directory:
  sr_pref

Parameter files:
  pref#_s.par: static simulations.
  pref#_d.par: dynamic simulations.

is the prefetch algorithm: 1, 2, 3, 4, 5, 6 (see .par files).
The simulations are as function of: cache size.

Simulation batch files (.par files, .trc file, .ppb file):
  pref_s.sim (pref#_s.par, t100000a.trc, pref_s.ppb)
  pref_d.sim (pref#_d.par, t100000a.trc, pref_d.ppb)

Generated .dat files:
  pr_s_mis.dat, pr_s_tra.dat, pr_s_mem.dat,
  prF_d_ser.dat, pr_d_mis.dat, pr_d_tra.dat, pr_d_mem.dat.

Buffers.

Directory:
  sr_buf

Parameter files:
  buf#_t1.par: prefetch_never
  buf#_t2.par: prefetch_lookup
  buf#_t3.par: prefetch_never
  buf#_t4.par: prefetch_lookup
  buf#_t9.par: prefetch_lookup, smaller access_time interval.

For buf#_t1, buf#_t2 and buf#_t9 no wrap around, no fetch bypass, no prefetch_stop and no demand_fetch stop are used. These features are used for buf#_t3 and buf#_t4.

is the buffer usage: 1, 2, 3, 4, 5 (see .par files).
The simulations are as function of: access_time.

Simulation batch files (.par files, .out files, .trc file, .ppb file):
  buf_t1a.sim (buf#_t1.par, buf#_t1a.out, t100000a, buf_t1a.ppb)
  buf_t1b.sim (buf#_t1.par, buf#_t1b.out, t100000b, buf_t1b.ppb)
  buf_t2a.sim (buf#_t2.par, buf#_t2a.out, t100000a, buf_t2a.ppb)
  buf_t2b.sim (buf#_t2.par, buf#_t2b.out, t100000b, buf_t2b.ppb)
  buf_t3a.sim (buf#_t3.par, buf#_t3a.out, t100000a, buf_t3a.ppb)
  buf_t3b.sim (buf#_t3.par, buf#_t3b.out, t100000b, buf_t3b.ppb)
  buf_t4a.sim (buf#_t4.par, buf#_t4a.out, t100000a, buf_t4a.ppb)
  buf_t4b.sim (buf#_t4.par, buf#_t4b.out, t100000b, buf_t4b.ppb)
  buf_t9.sim (buf#_t9.par, buf#_t9.out, t100000b, buf_t9.ppb)

t100000b is trace B.
Generated .dat files:

ser_t1a.dat, ser_t1b.dat, ser_t2a.bat, ser_t2b.bat, ser_t3a.dat, ser_t3b.dat, ser_t4a.bat, ser_t4b.bat,: service_times.
mis_bfp1.dat: miss and hit ratios, trace B, prefetch_never (buf#_t1).
mis_bfp5.dat: miss and hit ratios, trace B, prefetch_lookup (buf#_t2).
cyc_bfp1.dat: number_of_cycles, trace B, prefetch_never (buf#_t1).
cyc_bfp5.dat: number_of_cycles, trace B, prefetch_always (buf#_t2).

Number of cycles between requests.

Directory:
sr_cycle

Parameter files:
cyc#_b1.par: no buffers.
cyc#_b4.par: both buffers.

is the number of cycles between requests: 0, 1, 2, 3.
The simulations are as function of: access_time.

Simulation batch files (.par files, .trc files, .ppb file):
cyc_b1.sim (cyc#_b1.par, t1e5a0.trc, t100000a, t1e5a2.trc, t1e5a3.trc, cyc_b1.ppb)
cyc_b4.sim (cyc#_b4.par, t1e5a0.trc, t100000a, t1e5a2.trc, t1e5a3.trc, cyc_b4.ppb)

Generated .dat files:
cycb1_se.dat, cycb4_se.dat: service_times.

Demand_fetch_stop and prefetch_stop

Directory:
sr_fstop

Parameter files:
fs#_t1.par: No buffers, no wrap around, block size = 32, tr_block size = 8.
fs#_t2.par: Both buffers, no wrap around, block size = 32, tr_block size = 8.
fs#_t4.par: Both buffers, wrap around, block size = 32, tr_block size = 8.
fs#_t5.par: No buffers, no wrap around, block size = 64, tr_block size = 16.
fs#_t6.par: Both buffers, no wrap around, block size = 64, tr_block size = 16.
fs#_t7.par: Both buffers, wrap around, block size = 64, tr_block size = 16.

is fetch stop method: 1, 2, 3, 4 (see .par files).
The simulations are as function of: access_time.

Simulation batch files (.par files, .trc files, .ppb file):
  fs_t1.sim(fs#_t1.par, t100000b, fs_t1.ppb)
  fs_t2.sim(fs#_t2.par, t100000b, fs_t2.ppb)
  fs_t4.sim(fs#_t4.par, t100000b, fs_t4.ppb)
  fs_t5.sim(fs#_t5.par, t100000b, fs_t5.ppb)
  fs_t6.sim(fs#_t6.par, t100000b, fs_t6.ppb)
  fs_t7.sim(fs#_t7.par, t100000b, fs_t7.ppb)

Generated .dat files:
  fs_t1.dat, fs_t2.dat, fs_t4.dat, fs_t5.dat, fs_t6.dat, fs_t7.dat: service_times.

Fetch bypass

Directory:
  sr_bypas

Parameter files:
  by_t#.par: demand fetch stop, wrap around.

The simulations are as function of: access_time.
# is:
  1 = No buffers, no fetch bypass.
  2 = No buffers, fetch bypass.
  3 = Fetch buffer, no fetch bypass.
  4 = Fetch buffer, fetch bypass.

Simulation batch files (.par files, .out files, .trc files, .ppb file):
  bypass.sim(by_t#.par, by_t#b.out, t100000b, bypass.ppb)

Generated .dat files:
  bypass.dat: service_time.

Index.