MASTER

A stack cache for the C-processor

Paalman, J.A.H.

Award date:
1990

EB 269

Eindhoven, january - august 1990

Master thesis:

A stack cache for the C-processor

by J.A.H. Paalman

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

Coach: ir. W.J. Withagen

Eindhoven University of Technology
Department of Electrical Engineering
Digital Systems Group

Abstract

This master thesis discusses the data cache(s), which are to be used on the C-processor. The C-processor is being developed at the Digital Systems group of the department of Electrical Engineering of the Eindhoven University of Technology. The processor is a high performance processor, so caches have to be used to provide instructions and data on the speed of the processor. The C-processor uses a stack to store the local data of procedures and functions. A stack cache is used to provide fast access times to the stack. In this thesis several options for the data cache are discussed. Besides the normal cache parameters like cache size and (transfer) block size, some special features for a stack cache are provided. A linear cache can be used. This type of cache can use some special replacement policies. Besides the cut back K replacement strategy found in literature, a hybrid replacement policy is suggested, which will behave better on task switches. A direct mapped cache is proposed for extremely high task switching frequencies. The miss penalties of the cache can be reduced by using read and write buffers. These buffers can also be used as an extension of the cache when the buffers are almost empty. A special strategy is proposed to speed up the writes and thus producing a better access time.

All cache parameters can influence the performance of the cache. The only way to estimate the performance of the cache realistically is by simulating it. A start has been made to write a simulator for the stack cache. A data format for the traces is provided. To simulate the cache realistic address traces are required. These traces, however, are currently not available. Traces generated by a trace generator will give some insight in the effects of the cache parameters, but the many parameters of an stack address trace make it difficult to produce a realistic performance prediction.
A stack cache can give high speed access to the stack, but global data will still be serviced at main memory speed. Therefore the a global data cache is proposed. Especially if the stack cache has a high hit ratio. An inventory is given of coherence protocols for the global data cache.

The data caches are far from ready. The proposed architectures for the stack cache will have to be simulated to get performance statistics. With these statistics an optimal architecture can be chosen and a decision can be made if a global data cache has to be developed.

Contents

List of terms used

base pointer:	lowest address stored in a linear cache
a block:	data ,stored in an aligned, fixed-sized region of (cache) memory.
coherence:	the problem of keeping shared data consistent
hit ratio:	defined as the ratio of the number times the processor finds a memory reference in cache to the total number of memory references.
linear cache:	a cache containing all memory locations from a certain point to the TOS
local data:	definition 1:the continuous part of the stack that fits in the stack cache is considered local, the rest global.
	definition 2: all data on the stack is considered local, all other data global.
	definition 3: all data on the stack from the TOS to a certain point is local, all other data global.
PIN:	The process identification number of the process
prefetching:	fetching data before the data is wanted by the processor
pseudo-writes:	write strategy in which tha address is issued before the data is ready (section 4.2)
Quad:	4 bytes, standart data type on the C-processor
replacement strategy:	strategy guiding all data transfers in a cache
set size:	The number of banks or the number of addresses in a cache on which a memory location can reside.
stack frame:	piece of the stack used to store the local variables of a procedure
traffic ratio:	defined as the ratio of the number of bytes transferred between cache and main memory and the number of bytes transferred between processor and the cache.
Transfer block:	The unit of data transfers.

Chapter 0 Introduction

In the micro processor world many new and fast processors are marketed. As both the addressing space and the speed of the processors grows, it becomes impossible to manufacture large memories which match the speed of the processor and still are cost effective. Therefore cache memories are used. These are small memories which can keep up with the speed of the processor. Replacement strategies try to allocate these memories dynamically to the data or instructions the processor needs at each moment. If this is done successfully, these cache memories give the impression of being large, fast memories. As the VLSI technology moves on, more and more parts of a computer are integrated on a chip. Cache memory is one of the first candidates, since it can also reduce the number of pins of a chip. At the Digital Systems Group of the Department of Electrical Engineering of the Eindhoven University of Technology a high speed processor, called the C-processor, is being developed. This processor too will be equipped with on chip cache memory. The processor will use a split cache architecture. An instruction cache has been developed by Jos Bormans, Yun Chao Hu and Joep Pernot. In this report the cache structure for the data will be discussed.
The C-processor will store the local data on a stack instead of in registers. A specialised stack cache will speed up references to this stack and can be compared to the register sets used in RISC processors. A second data cache for the global data can be necessary if the stack cache is very efficient. In the next chapter the basics of caches are discussed. Chapter 2 gives an inventory of techniques to maintain coherency in the global data cache. In chapter 3 the parameters of the stack cache for the C-processor are discussed and this is followed by a description of a few possible architectures of the stack cache. Also some new features are introduced. To manufacture a optimal cache for the workload, simulations are required. The fifth chapter gives the basics to design a simulator for the stack cache. In the last chapter some conclusions are given and some recommendations for further research are made.

Chapter 1 Caches on the C-processor

The C-processor is designed to execute high level programming languages, such as the C programming language, at high speed. This is partially done by tailoring the instruction set to the high level languages. Secondly, the architecture has to support some constructions of the high level languages. The C-language is characterised by the use of procedures and functions. To provide easy function calling and to ease the passing of parameters, two approaches for storing local variables and passing parameters are mostly used. One can use multiple register sets and pass parameters by overlapping the register sets. The alternative approach is to use a stack to pass the parameters. In this last approach a stack cache is commonly used, so stack references can be serviced most of the time by fast cache memory or cache registers, rather than by slow main memory.

Characteristics of the register sets approach are the fixed set size and a fixed number of sets. A fixed number of locals can be in registers at the same time, a fixed overlap; a fixed number of parameters can be passed through registers and the registers can be addressed by a few bits. The allocation of local variables to registers can be a mayor task for the compiler.

The use of a stack cache permits a variable number of locals per procedure and the number of parameters also is not fixed. The size of the sets become variable, and the number of sets in the cache too. A compiler can use exactly the number of memory locations (registers) needed for a frame. Thus a frame always fits in a register set, there are no free registers in a set, nor are there to few. The register set size can vary from 0 to the cache size. The stack cache has to be addressed like a memory. This implies that long addresses have to be used. The local variables simply can be put on stack by the compiler. The only allocation problem arises when the stack cache is too small to hold all the locals of a procedure. But even then a register frame larger than the cache size can be chosen, which, depending on the cache replacement strategy, only results in an increase of cache misses.

The stack cache has no fixed frame size, thus register allocation needs no attention. As it is addressed as a memory, all memory addressing modes can be used and, depending on the cache architecture, the penalty for task switching can be lower. The stack cache also is transparent to the software. However, registers can be addressed by a smaller number of bits, are faster and are less expensive. The C-processor will be a stack based machine and will use a stack cache to hold local variables and to pass parameters to procedures and functions.

Besides a stack cache to hold the local variables the C-processor will be equipped with an instruction cache and (probably) a data cache for global data (global data cache) to speed up the processor. There are separate instruction and data caches, because separate instruction and data busses are used (Harvard architecture). To use both busses effectively, the traffic on the busses should be almost equal. In this case the traffic of the instruction cache should be equal to the traffic of both the stack cache and the global data cache. If the traffic on one of the busses is higher, the processor will wait for that bus. One of the caches can be reduced, without reducing the processor throughput, so there is a waste of silicon. The matching of the traffic can be managed by manipulating the miss ratios and the traffic ratios of the caches. The traffic on the data bus depends on both the stack data traffic and the global data traffic. Although the stack cache, and the global data cache can be designed separately, they need to be examined together in the final design phase.

There are two parameters related to cache performance:

1) hit ratio (R_{hit}), defined as the ratio of the number times the processor finds a memory reference in cache to the total number of memory references.
2) traffic ratio (R_{traf}), defined as the ratio of the number of bytes transferred between cache and main memory and the number of bytes transferred between processor and the cache.

A global data cache is useful when the hit ratio of the stack cache approaches the percentage of global data references in the data stream.

Example: *Hit ratio stack cache 90%. 10% global references. Of every 100 references, the stack cache now produces 9 references to main memory, while all 10 global references refer to main memory. A higher stack cache hit ratio would not have a large effect on the processor speed.*

The goals of the caches are (in order of importance):

- high hit ratio, most of the processor memory requests should be serviced by the caches.
- low cache access time (T_{acc}), data should be provided to the processor in a minimal time, because cache access is in the critical path.
- low miss penalty (T_{miss}), in case of a miss, the extra access time should be small.
- low traffic ratio; minimize the traffic between memory and cache, but the traffic on both busses should be balanced.
- small area, don't use to much silicon.

The order of importance is only relative. If, for example, the hit ratio is above a certain offset, a further increase in the hit ratio can require so much silicon, that the design is no longer feasible.

Before going into detail about the actual design, the next section will describe some fundamentals of caches.

1.1 Why do caches work ?

A cache is a small but fast memory, which contains a part of the contents of main memory. Caches can speed up a processor considerably because of the principle of locality. In a program, instructions near the present instruction are likely to be executed and data near the data being referenced are likely to be referenced (e.g. array structures, stack). This is called the spatial locality of a program. Besides spatial locality, there is temporal locality. Instructions in a loop will soon be used again, often several operations are done on the same data, etc. If the cache memory contains instructions and/or data near those presently in use and holds instructions/data that were used in the near past, the cache memory is likely to contain the next instruction(s)/data. Thus the processor can operate on the speed of the cache memory.

1.2 What parameters can be used to reach the goals?

There are several architectural choices that gravely affect the performance of the (relatively) small cache memory. Some of these issues are the organisation of the cache, the size of the cache, the use and the size of (transfer) blocks, the replacement policy and, for data caches, the coherence scheme.

1.2.1 Cache organisation

Since the cache memory is a small memory, it can only contain a small part of the instructions or the data of a program. There are three ways to organise a cache :

- a direct mapped cache
- a fully associative cache
- a set associative cache

For each of these organisations the memory of each entry is divided in two parts. A data part which contains the actual data and a tag part, which contains the virtual or real address of the data and some information regarding the validity of the data. The address part of the tag is used to check if a memory location is cached. If data is not in the cache, or if the data is not valid, a cache miss occurs.

1.2.1.1 A direct mapped cache

In a direct mapped cache (fig. 1.1), normal memory can be used for a cache. The address of each block can be mapped to only one place in the cache. Caches are usually divided in blocks to save tag memory. The tag is only kept for a block (see section 1.2.3). A part of the address (n bits) of the request is used to specify the entry in the cache. This means that on a cache miss, only one cache entry can be replaced; no replacement

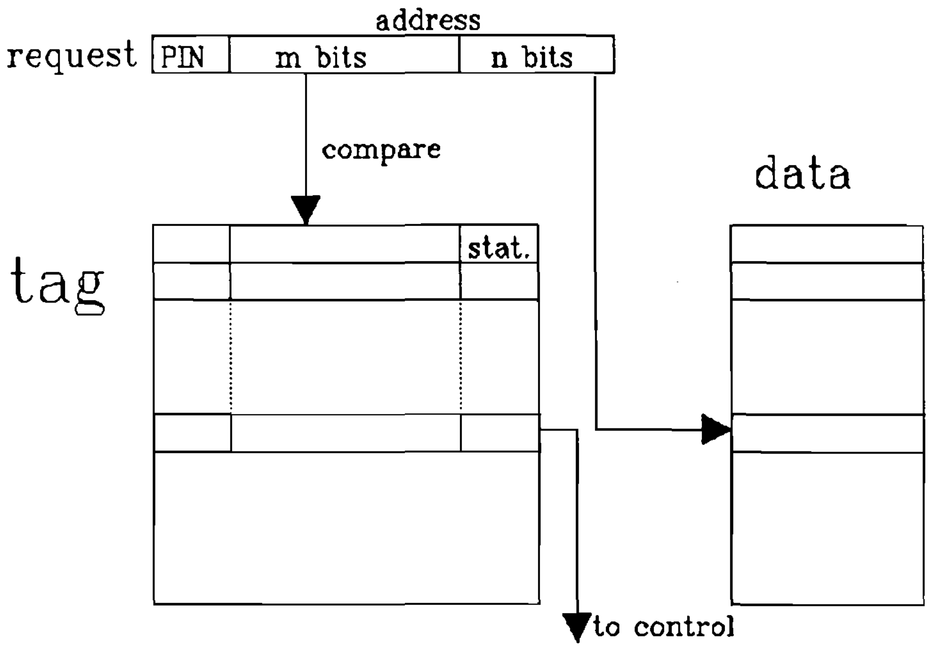

Figure 1.1 A direct mapped cache

strategy can be used. The absence of a replacement policy can have a negative effect on the cache performance. If two often used memory locations map to the same cache location, they will replace each other. This will decrease the hit ratio and increase the traffic ratio. The absence of a replacement policy also has a positive effect. No time is spent on determining which entry has to be replaced.

1.2.1.2 A fully associative cache

In a fully associative cache (fig. 1.2) each memory location can be mapped to each cache location. An associative memory has to be used. The entry that has to be replaced on a miss can be chosen by a replacement strategy. An entry with a low probability of being

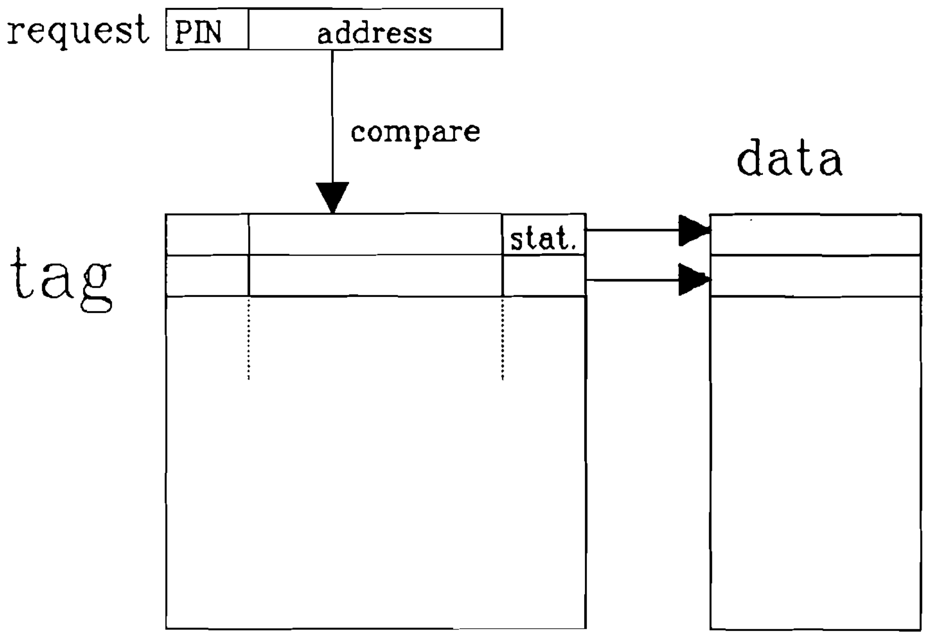

Figure 1.2 A fully associative cache

used soon, can be replaced, so the hit ratio can be high and the traffic ratio low. However, it takes some time to find this entry. Furthermore associative memory will be larger than normal RAM memory.

1.2.1.3 A set associative cache

A compromise between a direct mapped and a fully associative cache is the set associative cache (Fig. 1.3). In a set associative cache a memory entry can reside in a few direct mapped cache banks. A memory location can be mapped to only one location in each bank; to a set of a few addresses. The number of banks or the number of addresses in a cache on which a memory location can reside, is called the set size. The

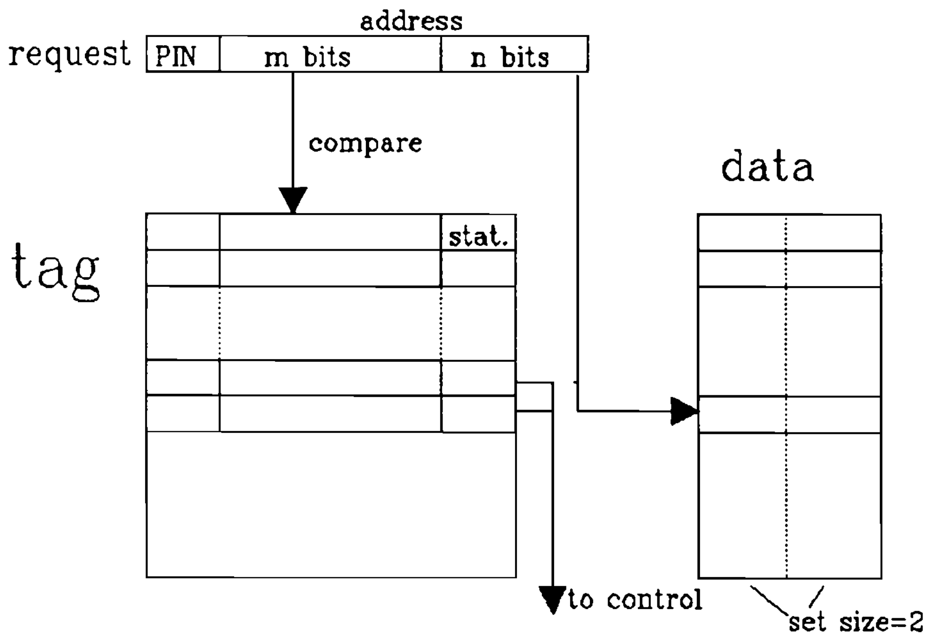

Figure 1.3 A set associative cache

number of sets is equal to the size of each cache bank. The associative search now only has to be performed in one set. The fully associative cache and the direct mapped cache are special cases of the set associative cache. In a fully associative cache the set size is equal to the cache size, and there is only one set; a direct mapped cache has a set size of one, and the number of sets is equal to the cache size. In general, the cache size is equal to the set size times the number of sets. As the set associative cache has associativity, a replacement algorithm can be used. The set size is mostly small, so the access time will be only little longer than the access time of a direct mapped cache.

1.2.1.4 Comparison of the cache types

The advantages of high associativity are important for small caches. As the caches become larger, the hit ratio will increase. The average access time for data (T) is $T = R_{hit} * T_{acc} + (1 - R_{hit}) * T_{miss}$. As the R_{hit} approaches unity, the access time will be more important than the miss penalty. A large direct mapped cache can outperform an equally large (set) associative cache. The choice of associativity not only is a simple comparison of hit ratios and complexities, but also depends on timing aspects for larger cache sizes.

For a stack cache a fourth type of cache can be used, the linear cache. This type of cache contains a contiguous part of the memory. Because this cache only contains one continuous part, it is not useful for an instruction cache, since each jump or function call can cause a replacement of a large amount of instructions in the cache. Because the stack is a continuous area, the cache can contain a continuous part of the stack, containing the top of the stack too. It then suffices to remember the address of only one entry, and calculate the addresses of the other entries, using an offset. The tag memory then consists of validity data only and will be very small. This type of cache is further discussed in section 3.2.1 about stack caches.

1.2.2 What size should the cache have ?

The size of the cache has a mayor effect on the hit ratio and the traffic ratio. It also influences the access time. As the cache becomes larger, it can contain more information, thus the chances of a hit increase. This causes an decrease of the traffic ratio, as more hits require fewer bus accesses. Because of the locality of programs, the increase of the hit ratio decreases with the increasing cache size. The extra items in the cache will be older or further away in memory, and are less likely to be referenced.

The main disadvantage of a large cache is the increase in access time, since larger memories have longer access times. Secondly large caches require a large chip surface so there also will be a physical limit to the cache size. The traffic on the data bus must be about the same as the traffic on the instruction bus, so simulations can give information about the cache size needed. The guidelines given by Smith [Smi85(B31)] can be used for a first approximation.

1.2.3 What set size to chose ?

The misses in a set associative cache caused by the mapping of various memory locations to the same cache location(s) are called collision misses. If the set size increases from one (direct mapped cache) to two, most of the collision misses are avoided. A further increase of associativity (the set size), further reduces the collision misses. But this decrease becomes less and less. Various studies [Smi82(B24)], [Aga89(P31)], [Hil89(P56)] have shown that a set associative cache with a set size of 8 has almost the same hit ratio as a fully associative cache.

1.2.4 And what about the block size ?

In a cache some memory locations can use the same tag. Such a block consists of data stored in an aligned, fixed-sized region of (cache) memory. One reason for using blocks is that it saves tag memory. The tag for a block requires only one address part (see fig. 1.4). Furthermore, using blocks, the cache can make use of the spatial locality: by fetching a block at a time, some memory locations near the desired memory location are fetched too. These locations have a high probability of being used in the future. However not all locations will be used. A large block increases the number of bytes fetched at a

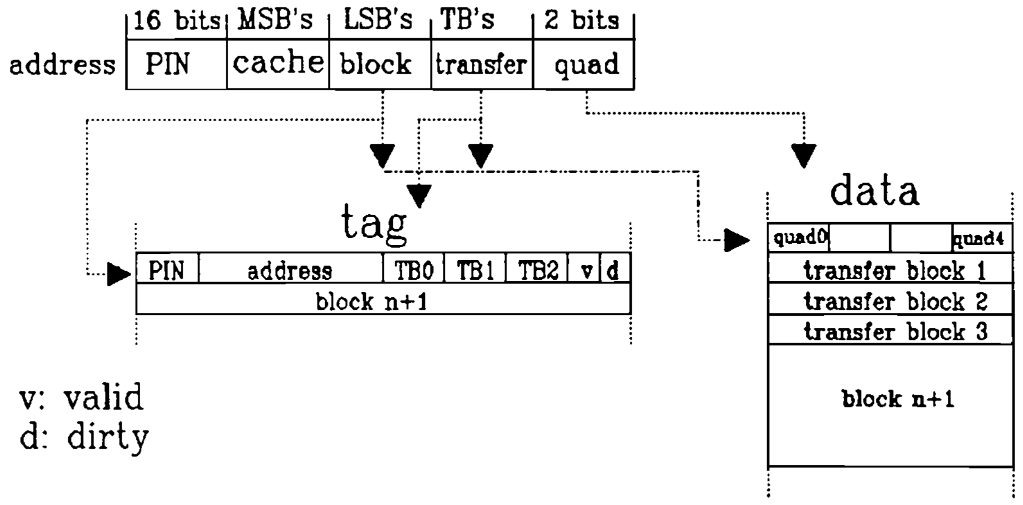

Figure 1.4 Cache memory organisation

read. The data near the requested data have a high probability of being referenced in the near future. For data further away in the block, this probability is smaller, so the ratio of useful data in a block decreases with increasing block size. This effect is due to the spatial locality. Each block fetched, will replace another block. The probability that this block contained useful data increases as the number of blocks that fit in the cache decreases (the cache size remains constant), since it only uses spatial locality. If only one block fits in the cache, at least one data item was referenced in the past. If two blocks of half the size are used, two items were referenced in the past. As the principle of temporal locality dictates, both items are likely to be referenced again. Thus, using temporal locality , one would prefer small blocks, since each block will contain data referenced in the past, and spatial locality prefers large blocks, so all data near the referenced data will be present.

A cache with a small block size, a lot of the data present was used in the past, but probably a large portion of that data was used long ago and has small chances of being used again. The data near the data in use has a small probability of being present if it was not used before. As the block size increases, some of the data used long ago will be replaced by data near the data used at the moment. The hit rate will increase, since the cache will contain data near the presently used data instead of data used long ago. If the block size further increases, the average age of the data will decrease. The cache will now contain data further away from the data in use, but data used recently will not be present. Thus after reaching an equilibrium, the hit ratio will decrease again.

The size of the blocks also has effects on the miss penalty and the transfer ratio. On a cache miss in a full cache a whole block has to be replaced. The cache can use the new data only when the replaced block is written back to the main memory and a whole block is fetched. Thus, an increasing block size results in a larger transfer of data and the miss penalty increases. Secondly if the block size increases, a block is more likely to be dirty (since it contains more data), and needs to be written back, even if the rest of the block does not have to be written back. These two negative effects of large blocks can be minimised by introducing transfer blocks.

Transfer blocks are the units of data transfers. The tag of a block containing transfer blocks, consists of the address for the whole block and valid and dirty bits for the transfer blocks only (see fig. 1.4). The miss penalty is decreased because only dirty transfer blocks have to be written back, and the transfer block containing the required data can be fetched before the other transfer blocks in the block, and thus be used by the processor before the rest of the block is present.

The optimal block and transfer block size are dependent on the cache size and organisation. Smith [Smi87(P17)] gives guide lines for block sizes (called line sizes by Smith) to get a desired hit rate / traffic ratio. But simulations will, once again, be necessary.

1.2.4.1 The address format for the cache

The use of blocks and transfer blocks has some effects on the addresses used to access the cache. Parts of the address are used to identify quads in transfer blocks and transfer blocks in blocks. In a direct mapped cache, each block can only reside on one place, so a part of the address can be used to identify the block in the cache.The addresses used for the cache have five fields (fig. 1.5):

- PIN: This is the process identification number of the current process
- cache: These bits are used to see if a block is in the cache
- block: These bits determine the place of a block in a linear or direct mapped cache
- Transfer: Select a transfer block within a block
- Quad: Select a quad within a transfer block.

These parts are only used if they are needed; i.e. if the size of the block equals the size of a transfer block, the transfer field will not exist.

	16 bits	MSB's	LSB's	TB's	2 bits
address	PIN	cache	block	transfer	quad

Figure 1.5 A cache address

1.2.5 Replacement policies !

Replacement policies in caches include all operations on cache memory. This includes the replacement of blocks by other blocks, writing back or fetching blocks in advance, and operations when the active process changes.
When the cache is full and there is a miss, a block in the cache has to be replaced by the block wanted by the processor. The problem at hand is which block to replace. Obviously, the best block in the cache to replace, is the block that will be used last of the blocks in the cache or even never again. Unfortunately, the cache has no knowledge of the future, so an educated guess has to be made. This is done by replacement policies. A first policy is to replace a random block. Since there is no knowledge about the future, each block has the same chance of being replaced, so we can replace a random block. This policy has a minimal overhead, since almost no time is lost to decide what block to replace. Thus the miss penalty is minimal. However, the past can be used to predict the future. The principle of locality can be used. This is done by the FIFO and the LRU replacement policies. In FIFO the block that was fetched the longest time ago is replaced, in LRU the block that was least recently used is replaced. As far as the hit ratio is concerned, LRU performs best, followed closely by random and FIFO is by far the worst. The bad performance of FIFO is not surprising since the block fetched the longest time ago is replaced, even if it is a frequently used block, or was the block last referenced. The closely matched performance of LRU and random can be a surprise. In the following example random even outperforms LRU. (there are of course counter examples, this example just indicates the possibility of a random replacement scheme out[performing the LRU replacement scheme.)

Example: *Cache size 3 blocks, in a loop blocks A, B and C are referenced sequentially. In LRU each block fetched always replaces the block to be referenced; A and B in cache, miss on C, C replaces A; B and C in cache, miss on A; A replaces B; etc. If random is used, the chance that the block to be referenced is replaced is 50%, so the random replacement strategy will give half the number of misses of LRU.*

The overhead of LRU however, can be high. The set size (n) is equal to the number of blocks that can be replaced, so n blocks have to be checked and updated at each reference. In the tag field of each block $\log_2 n$ bits are needed to contain the LRU information. Especially for a fully associative cache the cost can be high. For two or four way associative however, LRU is commonly used. The LRU replacement policy is used in the instruction cache. This policy is also an obvious choice for the data cache.

1.2.6 Coherency and task switching

The C-processor will be used in a multi-processor environment where it has to cooperate with other processors (i.e. a DMA processor). In a multi-processor environment, several copies of the same data can exist (in cache, main memory). This can even happen to data on the stack (which is private to a process) if processes can migrate between processors. Global data can be shared between processes too. The C-processor has a very strict data protection mechanism, which uses process identification numbers (PIN). The only way to share data is to use the real address of the data. If a virtual cache is used, there even can be two copies of the same data in the cache. This last problem can be solved by using a special mapping or by also putting the real address in the tag. The assurance that all of the copies are identical is called data coherence or data consistency. Coherence problems occur when the data in the cache and the data in main memory need not be the same. For example when DMA writes something to main memory, or when the C-processor writes something to cache.

A few of the algorithms used to solve the coherence problem are :

- write through
- write once
- Dragon
- Firefly
- Berkeley
- Illinois
- selective invalidation

These algorithms, and some more, will be evaluated in chapter 2. The coherence problem will only occur in the stack cache if processes can migrate between processors. This problem can occur in the global data cache and has to be solved.

When several processes are running on the same processor, task switching will occur frequently. At a task switch, (almost) all data in the cache will no longer be useful. In some cases, in a linear stack cache where only the base address is available, this means that the cache has to be flushed. The effects of task switching are especially important for the stack cache.

1.3 The stack cache and the C-processor

In the C-processor, two caches are placed in the two data paths between main memory and the processor. In the instruction data path an instruction cache is placed [Bor89]. The stack cache is placed in the data path, between the execution unit and the Memory Management Unit; the connection with main memory (fig. 1.5). The execution unit can issue several stack data requests at the same time. The execution unit is multi ported. The execution unit can also issue some control signals to move the TOS, or to switch off the cache. The cache responds with status signals, to indicate when the data transfer is completed. When some data does not reside in the cache, it must be fetched from main memory. The execution unit and the stack cache use virtual addresses. Main memory is

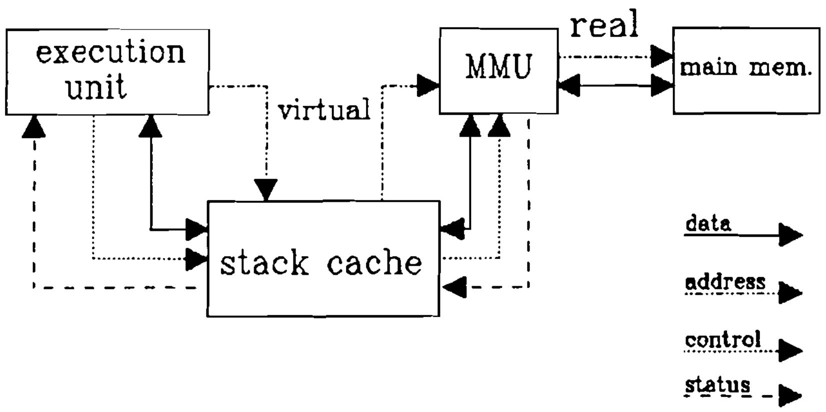

Figure 1.6 The stack cache in the C-processor

addressed with real addresses. The MMU translates the virtual addresses to real addresses. Between the cache and the MMU the same status and control signals can be used as between the execution unit and the cache. A more complete model of the C-processor is given in [Wit88] and [Bud88].

The stack cache will always handle local data, placed on the stack. The stack cache will contain at least a continuous part of the stack, containing the top of stack. If a global data cache is used too, the division between local data and global data has to be clear. The definition of local data can be difficult; local data in a procedure can be global data for a function called by the procedure. For the distinction between global and local data three options are given below, each having implications for the design of the data cache.
The first definition of local data is the most narrow: the continuous part of the stack that fits in the stack cache is considered local, the rest global. Thus, for a 1 kB cache, data from 1024 bytes below the TOS to the TOS are considered local, all other data are considered global.

Definition 2: all data on the stack is considered local, all other data global.

Definition 3: all data on the stack from the TOS to a certain point is local, all other data global. One could also opt to make the offset the current stack frame, but the size of the frames can vary. Most of the frames are small, but some can be quite large. Thus most times many frames would fit in the cache, sometimes one frame can be to large. Such a dynamic distinction between local and global data is most difficult to implement, and will not be chosen.

A request to the stack cache is considered a stack cache miss, when the request does not fall within the definition of local data. The stack cache miss must not be confused with a miss: data that is not in the cache, but inside the definition of local data. In the next chapters definition 2 will be used. Since it is not yet decided that a global data cache will be present, a stack cache based on this definition will cache more global data and can give better results. The implications on the design will be discussed further in section 4.7.

The performance of the data caches of the C-processor has to match the performance of the instruction cache, specially where the bus traffic is concerned. The design of the instruction cache is almost complete [Bor89], [Hu89]. This will be a two-way associative 1024 quad (=4 bytes) cache with a block size of 32 quads and a transfer block size of 8 quads. The predicted hit rate is 86% [Bor89].

Chapter 2 Data coherence

When data is distributed over several resources, contradictory information can be present at several places. Problems will occur when two processors operate on different copies of the same data. These coherency problems mainly affect the global data cache. Certain realisations of the stack cache can be affected by coherency problems too. In chapter 3 about stack caches, we will take a closer look at this problem. In the rest of this chapter coherence problems will be discussed in general. There are several ways to solve coherence problems.

The most simple techniques are to disallow private caches for the processors or to declare shared writable data not cachable. These techniques reduce performance, because the processor is either completely dependent on main memory (no caches) or dependent on main memory for all shared writable data. Besides this, the second approach is transparent to neither the user nor the compiler.

A third technique is to flush the cache each time the processor exits a critical section. This scheme can introduce a high overhead (lots of critical sections, large caches) and is difficult to implement on write-back caches.

A fourth scheme employs a centralised global table to store the status of the memory blocks. Cache enforcement signals can be generated on the basis of the block status. When a processor wants to use a block, it has to access the table. To limit the accesses to the global table, local status flags can be provided in the cache directories for the blocks that reside in the cache. Depending on the local flags and the type of the request, the processor may use the cached data, or has to consult the global directory.
In this scheme the global directory can be the bottleneck. Because of the distance between the processors and the directory, the latency of the table will limit the performance and increase the miss penalty.

The directory can also be distributed among the various processors. This approach takes advantage of the broadcast capability of the bus. In the schemes now to be discussed, the consistency is maintained by a bus watching mechanism, also called a snoopy cache controller. The schemes are known as copy back or snoopy cache coherence protocols [Arc86(P59)]. The bus is the limiting factor for the number of processors to be used. Furthermore, the complexity of the bus interface unit will increase, because it has to watch the bus.

For the C-processor with its on chip caches, the snoopy algorithms are more desirable for the following reasons :

- The caches are on chip. The time to consult the on chip tags will be much shorter then the time to consult a central directory in main memory.
- The number of processors will probably be small. There will be no problems with the limited bandwidth of the bus.
- The on chip caches will be small, so the miss ratio will be rather high. Any increase in the miss penalty will seriously affect the performance.

The higher performance of snoopy schemes is confirmed by simulations in [Aga88(B41)] for small multi-processor systems. In the following section the various snoopy coherence schemes will be discussed.

2.1 Snoopy coherence schemes

To ensure the consistency of the data in caches, a cache has to store some information with respect to the other copies of each entry. First one has to know if the data is correct and still up to date. This is called the validity of the data. Secondly the data can be shared with another cache, so information about the exclusiveness has to be in cache. Third a cache can be the owner of an item. Then even main memory can have an invalid copy and the cache has to respond instead of main memory if someone tries to access that data.

A model in which the various states of the data is described uniformly is needed to compare the snoopy coherence protocols. In the so called MOESI (the first letters of the states described below) state model [Swe88(P12)] the attributes validity, exclusiveness and ownership are combined to form five states for the data in a cache. These five states are :

M	modified, the data in the modified state is not shared by any other cache, it has changed since it was read from main memory and main memory has not been updated. The processor may read and write this data.
O	owned, data in the owned state may be shared by other caches and main memory is not updated. The processor may only read owned data.
E	exclusive, data in the exclusive state is not shared by other caches, and main memory is up to date. The processor may read the data.
S	shared, data in the shared state can be shared by other caches, and it is owned by either main memory or another cache. The processor can only read this data.
I	invalid, the data in the cache is not consistent with data elsewhere, and may not be read by the processor.

A block that is not in the cache is a sixth state : not present. This state is not included in the MOESI model.
With some of these states (not all are necessary) a cache coherence protocol can be formed. In table 2.1 the various coherence protocols are described in terms of their MOESI states. As can be seen in the table, there are almost no protocols that use the same states. Furthermore they use different states or state changes to write back data to main memory. The protocols are described in detail in the next section.

Table 2.1 MOESI states for various coherence protocols

	M	O	E	S	I
write through	.	.	.	valid	invalid
write once	dirty	.	reserved	valid	invalid
Synapse	dirty	.	.	valid	invalid
Berkeley	owned exclusively	owned non-exclusively	.	unowned	invalid
Illinois	exclusive modified	.	exclusive unmodified	shared unmodified	invalid
Firefly	not shared dirty	shared dirty	not shared not dirty	shared not dirty	
Dragon	dirty	shared dirty	valid exclusive	shared clean	

2.2 Description of the various protocols

In this section a description of some snoopy protocols will be given. The first protocol, write through, is a very simple protocol which can be implemented at low cost but has very low performance. It is mainly included for comparison of the various schemes.

2.2.1 Write through, states: valid(S) and invalid(I) [Yan89(P32)]

A write can give rise to a hit when the block is in cache and valid, or a miss otherwise. On a miss, the block is loaded from main memory in the VALID state. On a hit, the block is supplied by the cache and the block remains VALID. When a write occurs, the block is written through to main memory. All other caches watch the bus, and invalidate all data in the cache to which a bus write occurs.

2.2.2 Write once, states : dirty(M), reserved(E), valid(S), invalid(I) [Yan89(P32)]

On a read request of the processor the cache responds with a hit or a miss. If the requested word is not in the cache or is invalid the result is a miss. Otherwise the data is supplied without changing the state of the block. On a miss, the data is supplied by another cache if a dirty copy exists, otherwise main memory will send the data. A block is loaded with state VALID. If the block is loaded from main memory, and a cache has a RESERVED copy of the block, this cache will set its state to VALID. If the block is supplied by a cache, this cache will write the block back to main memory and set its state to VALID.

A write of the processor also gives rise to changes of states. If the block is VALID in the cache, its state is changed to RESERVED, all other caches are invalidated and the block is written through to main memory. If the block is RESERVED, the state is changed to DIRTY, and the data are written to the cache. If the block already is DIRTY, all that has to happen is to write the block to cache. The state remains the same. If the block is not in the cache, the block is loaded like on a read miss, all other caches are invalidated, and the block is updated in the cache. The state is set DIRTY.

2.2.3 Synapse, states : dirty(M), valid(S), invalid(I) [Fra84(P37)]

Like some other protocols, this protocol uses ownership to resolve the coherency problems. The main memory itself has an OWNED tag for each block.

A read hit causes no bus activity. On a read miss a public read signal is put on the bus. If main memory is the owner, the block is loaded in the valid state. If another cache is the owner (has a DIRTY copy), it writes back the word to main memory and invalidates its copy. After this transaction main memory is the owner, and the public read is handled as before.

A write hit on a DIRTY block can proceed locally. On a write hit on a VALID block or on a miss, a private read request is issued, and the same actions are taken as on a public read. After the block is updated in the cache, the state is set to DIRTY, so the requesting cache becomes the owner of the block. All caches with a VALID copy monitor the bus and invalidate their copy if they monitor a private read.

2.2.4 Berkeley, states : owned exclusively(M), owned non-exclusively(O), unowned(S), invalid(I) [Kat85(P38)]

When a processor read results in a hit, the appropriate word is provided to the processor. On a miss, a block has to be flushed. If this block is OWNED, the block is written to main memory, and to other caches if the state is OWNED NON-EXCLUSIVELY. A write without invalidation is used. Then it reads the block with status UNOWNED. It uses a special read signal; a read-shared. If the block is OWNED by another cache, this cache will provide the data, instead of main memory, and set the state in the cache to OWNED NON-EXCLUSIVELY. If the block is NOT OWNED, it is supplied by main memory. A block is always read in the state UNOWNED.

On a write, the following procedure is used. If there is a hit on an OWNED EXCLUSIVE block, then the processor writes only to the cache. If the hit is in an OWNED NON-EXCLUSIVE or an UNOWNED block, then all other caches must be invalidated, using a write for invalidation. On a miss, some data will be flushed using the protocol of a read miss, and the block is read using a read for ownership signal. On this signal the owner sends the block and all caches (except the new owner) invalidate their copy of block. Then the cache updates the block. After the block is written to the cache the state is set to OWNED EXCLUSIVELY.

2.2.5 Illinois, states : exclusive modified(M), exclusive unmodified(E), shared unmodified(S), invalid(I) [Pap84(P39)]

In this protocol too a read hit is only handled by the cache with no status change. On a read miss, the highest priority cache will give the data (daisy chain). If no cache has the data, then memory will provide the block. All caches that match the data will set the status to SHARED UNMODIFIED. If the status in one of the caches was EXCLUSIVE MODIFIED, the block is written back to main memory. The requesting cache sets the status to SHARED-UNMODIFIED if the block came from another cache, and to EXCL-UNMODIFIED if the block came from main memory. On a write miss the same read cycle is done to cache the block. But an invalidate signal accompanies the read request, on which all caches invalidate their copies of the block. The requesting cache sets the status of the block to EXCLUSIVE-MODIFIED. On a write hit, the cache is updated and the status is set to EXCLUSIVE-MODIFIED. Only if the status of the block was SHARED UNMODIFIED an invalidation signal is sent.

2.2.6 Dragon, states : dirty(M), shared dirty(O), valid exclusive(E), shared clean(S) [McC85(P52)]

This protocol looks like write once, but it uses a shared line to see if a block resides in more caches. A cache that monitors a read or write request for a block it contains, raises the shared line. The shared states can be used to minimise the number of writes.

On a read miss, the data are read. If another cache has a dirty copy, it supplies the data, otherwise the block comes from main memory. All caches that contain a copy of the block, mark their copy as SHARED. Otherwise the cache marks the data as VALID EXCLUSIVE. If the shared line is raised, the block is loaded as SHARED CLEAN, otherwise as VALID EXCLUSIVE.

A write to a DIRTY or VALID EXCLUSIVE block can proceed locally. The state of the block is set to DIRTY. On a write to a shared copy, the data are written to all caches. If a cache has a copy it reads the block as SHARED CLEAN and raises the shared line. By observing the bus, the cache performing the write can determine if the block is still shared. If the shared line is raised, the block is marked SHARED DIRTY, else the state is set DIRTY. Main memory is NOT updated.
On a write miss, the block is read as on a read miss. The block is loaded from the bus as DIRTY or SHARED DIRTY depending on the shared line. If the block is SHARED, the update is written through to the other caches.

2.2.7 Firefly, states : not shared dirty(M), shared dirty(O), not shared not dirty(E), shared not dirty(S), invalid(I) [Tha88(P58)]

This protocol, which is similar to the Dragon scheme, depends on a shared-line. A snoopy controller sets the shared line high if it finds an address on the bus that resides in the cache. The controller that provided the address watches the shared-line to see if the data also resides in another cache.

On a write that hits in non-shared data, no bus traffic is needed, and the data is marked NOT SHARED DIRTY. If the data is shared, the cache writes through the data to maintain coherence and update main memory. If the data is in another cache too (shared line is set by another cache) it marks the data SHARED NOT DIRTY, otherwise NOT SHARED NOT DIRTY. On a write miss, the same actions are taken as on a write to shared data.

A read hit requires no further action. On a read miss, the data is read from the bus. The received data are marked clean and its shared bit is set to the value of the shared-line. If the data are provided by a cache, this cache sets the shared bit.

2.3 Performance of the various coherence schemes.

The various coherence schemes can be compared in various ways. The first, and probably least reliable comparison is given by the authors of articles about new schemes (their scheme is better than...). A second approach is a comparison on the basis of certain models. The main problem with models is the validity of the models. A last method of comparing coherence schemes is using traces. This is most common practice. There are several articles in which the various schemes are compared using traces. The first comparison of the coherence schemes will be done on the basis of the various comparisons in the various articles.

A comparison on the basis of a queueing model can be found in [Yan89(P32)]. Yang uses a performance parameter called system power (processor utilisation * the number of processors), to compare the coherence schemes. The highest system power is given by the Dragon and the Firefly protocols. The system power of the Illinois, Write-once, Berkeley and Synapse schemes are almost the same, but considerably lower than the first two schemes. Of these schemes, the Illinois scheme performs best and the Synapse scheme worse. The Write-once and the Berkeley owner scheme perform similar. Write through results in the lowest system power. The other coherence protocols are not evaluated by Yang.

A simulation of the same six protocols is performed by Archibald and Baer in [Arc86(P59)]. It also uses the system power as a performance measure. The highest system power, in their simulations, is obtained by the Dragon scheme, followed respectively by Firefly, Berkeley, Illinois, Write once, Synapse and, far behind, write through. The performance of the Synapse scheme is in all simulations more than two times higher than write through. The relative performance of the schemes depends highly on the degree of sharing, the percentage of reads and the cache and block size.

The various coherence schemes have a different cost. The Dragon and Firefly scheme require a shared line, and a cache in the Illinois scheme has to see whether a block comes from memory or from a cache. In the schemes where a cache can provide blocks, the extra accesses to the cache can slow down the processor of that cache. The synapse scheme requires an extra memory bit for each block. Besides, all protocols, except write through, need a snoopy controller. The costs of the protocols and the system power both have to be taken into consideration before the choice of a coherency protocol can be made. At the moment, some new snoopy coherence schemes are being developed, the so called competitive snooping schemes [Kar88(P57)]. Their authors claim that their schemes outperform the normal snoopy schemes. It may be necessary to modify a protocol to meet the specific demands of the C-processor. These include the size of the cache, the traffic on the bus, the number and kind of additional processors, second level caches, the way of sharing data, etc.

Chapter 3 The Stack Cache

In this chapter the basics of a stack cache will be discussed. First the stack of the C-processor will be discussed. Then several organisations of caches that are designed specially for a stack will be discussed. This is followed by a discussion of the effects of the (block)size, the replacement strategy and coherency on the stack cache types.

3.1 The C-processor stack

On the stack of the C-processor all parameters used in and passed between procedures and functions are stored (return addresses can be seen as parameters passed between functions). Such a stack is shown in figure 3.1.

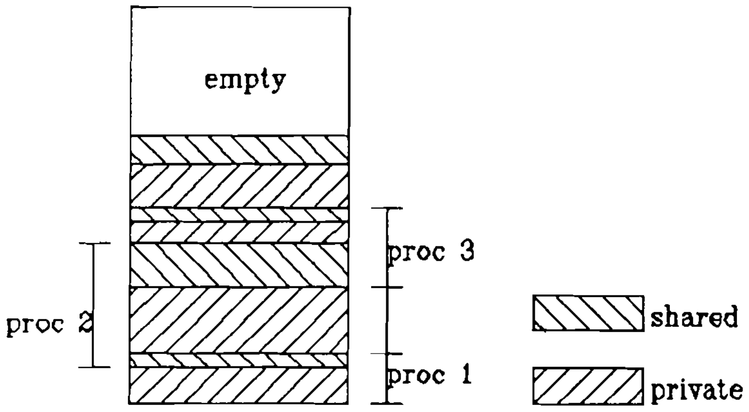

Figure 3.1 The C-processor stack

Most procedures will only require a small number of parameters and a small number of local variables, so the typical stack frame (#locals + # parameters) will contain about 10 variables (80% < 11 words [Fur88(A12)], 95% < 13 words [Laz89(A13)], 94% < 13 words

[Sta88(P5)]). Most references to the stack will be to the frame containing the TOS, since that procedure is being processed. The number of frames on the stack will probably vary slowly, and will be about 8 (a window containing 8 frames needs to shift on less than 1% of the calls and returns [Sta88(P50)]). However, some programs and procedures will require more stack space.
The C-processor can access various addresses at a time. At a certain moment, it is possible that the two operands for the next instruction are fetched from the stack, while the result from the last instruction is written on the stack. The execution unit has multiple ports to write to and read from the stack, so the stack cache has to support these multiple ports. Since it will have a high hit rate, the cache will give less memory requests to main memory. Thus the number of ports to main memory can be reduced.
So besides the speed of a cache, it has a second advantage : it can avoid multiple ports to main memory.
The stack cache will have the following requirements besides the requirements from chapter 1 :

- it has to contain the highest stack frame
- it has to support multiple requests in one clock cycle
- it must be transparent to the software

3.2 Stack cache organisation

The stack cache will contain (a copy of) the memory locations near the top of the stack (TOS). The choice to have an extension of the memory stack or a copy of the memory stack is not very difficult. If the cache is to contain a copy of the locations near the TOS, the cache has to be write through to keep main memory up to date. This involves a lot of traffic. If an extension is chosen, the write back strategy (write back dirty blocks when they are replaced by another block only) can be used to solve consistency problems, and the probability of stack references falling in the cache is very high for a reasonable cache size. The penalty for task switches will be higher if an extension is used (write back dirty

blocks vs. no dirty blocks to write), but can be limited by using buffers. Besides, since task switches will be relatively rare, the performance loss in normal use for the caches using a copy, will be larger than the performance loss on the task switches for the extension caches. So the cache will contain an extension of the stack.
Only those places below the TOS are of interest, places above the TOS are invalid (the stack is growing upward). There are no coherence problems between processes, because a stack is private to a process (each process has its own stack, identified by the PIN). Since the stack is a contiguous area, the stack cache can best be organised as a contiguous area. Only two options for the cache memory organisation remain : the cache can be linear, or it can be direct mapped. The (set) associative cache will not be discussed here. In an associative cache some replacement algorithm must be used, which can replace blocks near the TOS. This behaviour is undesired for a stack cache because, although these blocks may be least recently used, the probability that these blocks will be used in the (near) future is high. The reason for this high probability is that these blocks are near the TOS. In other words: on a stack spatial locality is higher than temporal locality. A compiler can make the spatial locality even higher. The linear cache and the direct mapped cache will be discussed in the following sections.

3.2.1 A linear cache.

A linear cache contains all memory locations from a certain point, the base pointer, to the stack pointer (see fig. 3.2). The base pointer points to the lowest valid address in the cache. This need not be the frame pointer, since the cache may contain more than one frame. It is possible that this does not fill the cache. The cache can be organised as a linear buffer. When the stack grows, data at the base of the cache are threatened to be overwritten by the TOS and have to be written back to main memory (if they are modified), before more space can be allocated to the stack. No data has to be retrieved from main memory. The resulting free block is allocated to the TOS. This situation is called cache overflow.

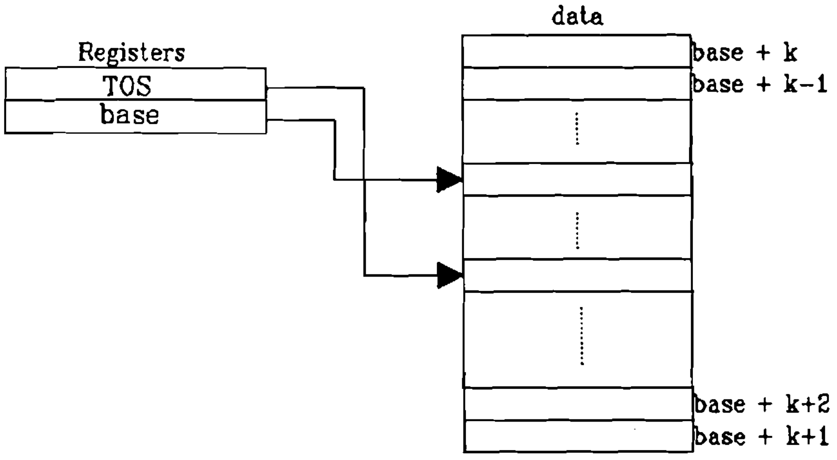

Figure 3.2 A linear stack cache

Underflow occurs, when the stack is retreating. Again the stack pointer approaches the base, but this time from the other side. The cache then becomes empty, and new data has to be cached from main memory. Because the data beyond the TOS is invalid, no information has to be written back to main memory.

When an overflow or an underflow occurs, some action has to take place. Several replacement strategies deal with this situation. Some policies even try to avoid under- and overflow situations (some kind of prefetching) These policies, specially designed for stack caches, are dealt with in section 3.3.3.

When another process interrupts the running process, a task switch occurs and the whole cache has to be flushed; the data that has been changed has to be written to main memory and the contents of the cache has to be made invalid. A flush is necessary because the only addresses remembered are the base and the TOS. If any data remained valid in the cache, this would be used erroneously as valid data for the stack of the new

process. At the time of a task switch, the bus will be heavily used, and the processor will be waiting for data from the stack of the new process. The amount of data transferred depends on the replacement algorithm.
A way to reduce the number of task switches is the introduction of a second cache, to be used by the operating system or supervisor. To achieve the same hit ratio as the normal stack cache, the supervisor cache should be larger. If the same amount of data ram would be used, each of the split caches should be half the unified cache. Simulations in [Smi82(b24)] show that a split cache has a lower hit ratio than a unified cache. The larger penalty of the unified cache on supervisor calls can be reduced by architectural measures (chapter 4).

There are no coherence problems in a linear cache. The stack is private memory for a process and the whole cache is flushed on a task switch. Thus it is impossible that two different copies of the stack can be used at any time.

3.2.2 A direct mapped cache.

Instead of a linear cache, a direct mapped cache can be used. As long as all references are made to memory locations near TOS, this cache behaves like a linear cache.

Because each block in cache has its address in the tag, it is not obligatory to invalidate the whole cache on a task switch. If the processes can not migrate from one processor to the other, no flushing is needed at all! So if a process is interrupted by another process, which makes almost no use of the stack, a major part of the stack of the interrupted process will stay intact. This remaining part of the stack will probably not be a continuous part. Still, the parts that remained in cache will be valid and will not have to be read again. This minimises the penalty of task switches. When a process is interrupted by a large process, this process will fill the whole stack cache, so, though the cache was not flushed at the task switch, all data is replaced by the new process. The net result is the same as a flush. The overhead of flushing the cache however, is not at the time of the task switch, but is smeared out over the beginning of the new process.

A disadvantage of a direct mapped cache is that the virtual addresses and the PIN-codes of all blocks have to be kept in the tag memory. So direct mapped caches will be larger than linear caches.
The same replacement strategies as in a linear cache can be used. However, care has to be taken to remove no blocks of other processes, and thus reduce the advantages of a direct mapped cache over a linear cache on task switches. Simulations will have to provide information about the behaviour of the cache organisations.

In a direct mapped cache, coherence problems can occur if a process can migrate from one processor to another. If a process migrates form processor A to processor B, changes some of the data on the stack and moves back to processor A, the contents of the parts of the stack which are still in the cache of processor A might not be the same as in the cache of processor B! If the migrating of processes is allowed, a cache coherence algorithm has to be used (see chapter 2).

3.3 The effects of cache parameters on a stack cache.

In this section the cache parameters will be discussed. First the cache size and the block size will be handled, then the replacement policies for stack caches will be compared. Finally the effects of task switching and cache coherency will be taken into account.

3.3.1 The cache size.

In general larger caches give smaller miss ratios, longer access times, smaller traffic ratios and, depending on the cache structure, a larger or smaller penalty for task switching. Although the hit ratio increases with the cache size, the increase of the hit ratio will decrease as the cache becomes larger. In a linear cache the penalty for task switching will grow with increasing cache size since more data has to be flushed. In a direct mapped cache the probability that after a task switch some parts of the stack of the resumed process will still be present increases with the cache size and so the penalty of task switching decreases.

The task of the stack can be compared to the register sets in a RISC processor, so a cache size comparable to the number of registers in a RISC processor will probably suffice. The number of registers in a RISC processor ranges from several dozens to several hundreds (Risc I: 128, AM29000: 192, C/70: 1024). A processor with a stack cache , the CRISP uses 32 registers, each register is one 32 bit word wide. Simulations by Ditzel et al. [Dit87(P60)] proved this size to capture a sufficient amount of data for certain C-programs (using a non optimising compiler). However this included only the data of one procedure or function so "a larger cache size would reduce the amount of flushing on procedure or function calls and returns". Although these numbers are dependent on the instruction set, compilers and optimisers, they give an indication on the number of locals that need to be cached (in register sets) to give a high processor performance. Simulations have to provide data to find a cache size with a good trade-off between stack cache size and hit ratio.

3.3.2 The (transfer) block size.

The block size has a great influence on the performance of the cache. In a linear cache, the block size determines the quantity of data that is written to or read from main memory at overflow or underflow. Since a whole block is fetched on an underflow, a whole block will reside in the cache. The TOS will now reside on one block from the bottom of the cache. A large block size will thus increase the time between two underflow situations. A large block size will move the TOS further away from the border of the cache which was crossed. If the block size is to large, the traffic ratio will be high, and too much data will be replaced or fetched. Simulations are necessary to find the optimal block size.

In a direct mapped cache the tag of a block contains its virtual address and information about the validity of the data. Because the address of the data now also resides in the tag, a larger block size can reduce the size of the tag memory with a serious amount.

A block can be divided in transfer blocks. The tag of the block then provides information about the validity of each of the transfer blocks. The address is stored for the whole block. In case of an overflow only those transfer blocks that have been changed have to be written back to main memory. The introduction of transfer blocks thus reduces the amount of write back traffic. Transfer blocks can also reduce the miss penalty. As soon as a transfer block is in the cache, it can be marked valid, and the data can be transferred to the execution unit. Transfer blocks of more than one quad (= bus size) can make use of burst transfers to main memory which improves the average time to fetch a byte. However, large transfer blocks will increase the miss penalty and the amount of data written back.

3.3.3 Replacement policies

There are several replacement strategies which have to deal with or prevent overflow and underflow of the stack cache. Since most stack caches discussed in literature are linear caches, a few of the replacement policies for this type of caches will now be discussed. These policies are also discussed in [Sta87(B36)].

- The cut back K algorithm (see also [Has85(B36)]) is the most simple algorithm and acts only on demand.
- The barometer pointer algorithm tries to avoid both overflow and underflow.
- The two-pointer algorithm tries to keep the cache full and to avoid overflow.

In a modified form these can also be used for a direct mapped cache.

3.3.3.1 The cut back K algorithm

If the stack cache overflows (TOS moves above the cache) or underflows (TOS moves below the base), one or more blocks (containing K transfer blocks) are written back to

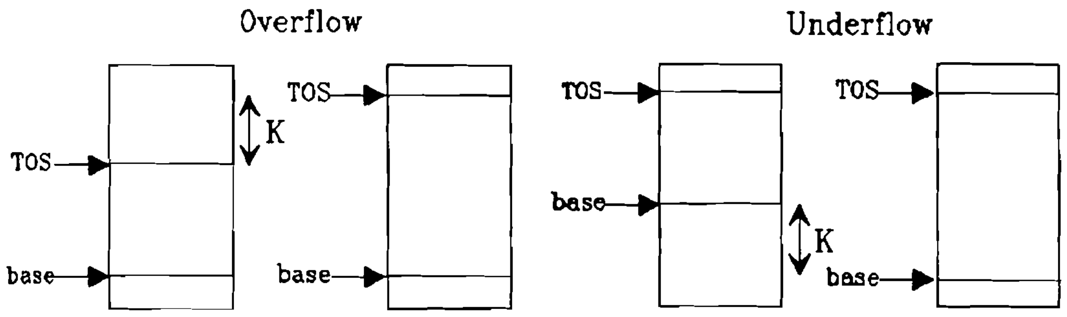

Figure 3.3 The cut back K algorithm

main memory in order to free cache space, respectively read from main memory to obtain the TOS (figure 3.3). The algorithm makes no attempt at predicting the future needs of the stack cache. On the average, the cache will be more than half full. Besides, the number of over- and underflows can be high. This can be offset by using larger blocks. However, this will increase the number of bytes transferred. More bytes that are not used are fetched, more bytes are replaced on an overflow.

3.3.3.2 The barometer pointer algorithm

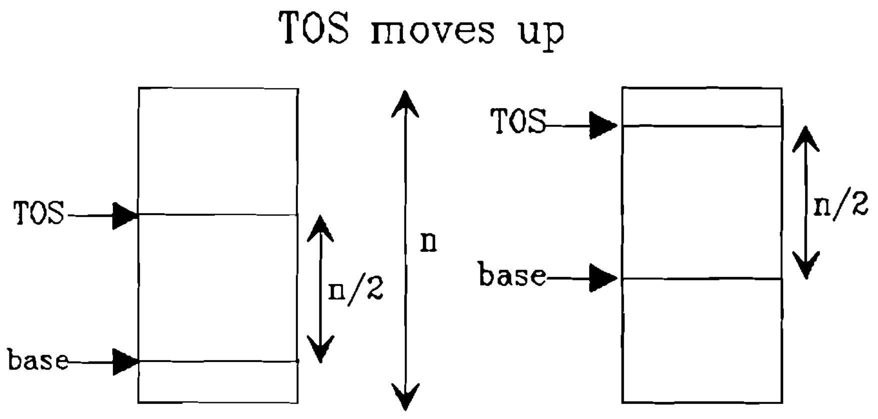

Figure 3.4 The barometer pointer algorithm

This algorithm keeps the TOS pointing to the middle of the stack cache (figure 3.4). It is possible to let the TOS point to another place in the cache. [Sta87(B36)] describes a simple barometer pointer algorithm. On an overflow the block to be cleared is always written back. Thus the penalties for overflow and underflow are equal (the main memory access for one block). The chance that the stack will grow is about equal to the chance that the stack will shrink. This makes the middle of the cache a logical place for the TOS to point at. The frequency of overflow and underflow is thus reduced. If the penalty for overflow differs from the penalty for underflow, the TOS can be kept pointing to another place in the cache, such that the overall cache miss penalty is minimised. If the TOS is greater than the middle of the cache, blocks are read from memory until the TOS is in the middle, if the TOS is less than the middle of the cache, blocks are written to memory. A special pointer points at the middle of the cache. By keeping the cache half-full, the usage density is independent of the growth of the stack, but on the average it will be no more than half. Some extra buss traffic is introduced if the stack grows and shrinks without overflow or underflow, compared to the cut back K algorithm. As long as the cache would not overflow or underflow with the cut back K algorithm, these extra bus transfers are useless.

3.3.3.3 Two-pointer algorithm

This algorithm keeps the cache full (figure 3.5). It uses the cache to keep a copy of the stack in main memory, and uses a write back strategy (only write to main memory if a block is replaced) to keep main memory up to date. Besides the stack pointer, it uses two

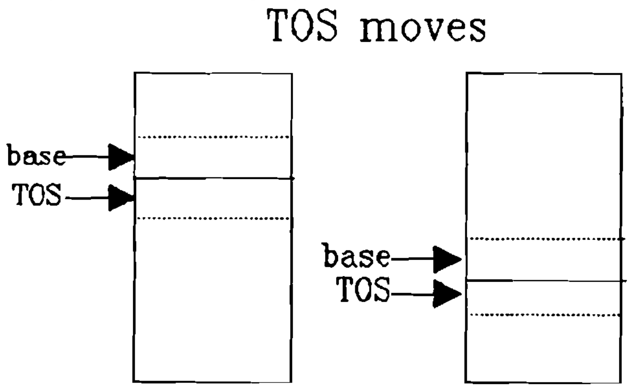

Figure 3.5 The two pointer algorithm

pointers to indicate how many blocks need to be written back to main memory (the PUT pointer) and how many blocks need to be fetched (the GET pointer). A detailed description is given in [Sta87(B36)]. If both pointers are zero, the cache is at rest. If a push is performed, the PUT pointer is incremented to indicate that a block has to be written back to main memory to make room in the cache for some new data.

This strategy also introduces a lot of useless bus traffic compared to the cut back K algorithm. Besides, if the stack is growing there is a continuous overflow situation. The mayor advantage of this scheme is that the whole cache is always full.

3.3.3.4 Comparison of the algorithms.

In general, some strategies try to minimise over- and underflow by keeping the cache half filled (two pointer algorithm), or try to detect trends (barometer pointer). Although they can succeed in their purpose, they also increase the traffic ratio, because not all space freed by writing blocks back to main memory will be allocated, and not all blocks cached from main memory will be referenced. The strategies can also work against themselves. Especially at the start and end of processes when the stack is continuously growing or shrinking. In general these strategies reduce overflow and underflow a little, while requiring a lot more bandwidth.

Other algorithms (like cut back K) only move blocks when necessary. This can result in a high penalty if a new procedure is started or finished if the cache is full resp. empty. The traffic ratio is completely dependant on the block size. If a fixed frame size is used for procedures, and the block size is equal to the frame size, the stack cache can be compared to the multiple register sets used in RISC (although there are some differences). One of the disadvantages of using a fixed frame size, is that the utilisation of cache memory will be far from optimal. Furthermore the allocation of variables in frames will become a task of the compiler, with a major influence on cache performance. A consequence of this is that the cache is no longer transparent to the software.

In [Sta87(B36)] three TOS management schemes are compared for a TOS buffer. The study uses single pops and pushes to move the TOS. This is not the case in the C-processor, where the TOS moves more than one memory location on a call or a return. Although the situation is not the same, the study can be used to give an indication for the

C-processor. The conclusions of [Sta87(B36)] support the conclusions given above. Their simple algorithm (comparable to the cut back K algorithm) greatly reduces the memory bandwidth by requiring little TOS management. The demand TOS management is nearly as high as the best of the other algorithms. The usage density is better than for the barometer pointer algorithm, but worse than the two pointer algorithm.
The barometer-pointer algorithm reduces the demand TOS management but requires a higher memory bandwidth and the usage density is low.
The two-pointer algorithm has the highest usage density, but requires a high memory bandwidth, the demand TOS management is high and it is the most difficult one to implement.

The study of Stanley clearly indicates that:

> The simple algorithm (comparable to the cut back K algorithm) combines a high usage density with low bus traffic. Besides it requires the least extra hardware. This algorithm seems to be a good choice for a linear cache. The barometer pointer algorithm is second best.

However, as stated before, the results only give an indication for the C-processor. Besides, the results the effect of task switches has not been taken into account in [Sta87(B36)]. The effects of task switching will now be discussed.

3.3.4 Task switching and stack caches.

The linear cache only keeps track of the lowest and the highest address of the data in the cache. If another process wants to use the cache, this information will be lost, so all data in the cache has to be written back to main memory before the new process can use the stack. For large caches these cache flushes can cause a lot of bus traffic and a large time penalty for task switches.
For the replacement algorithms, the effects are most severe for the two pointer algorithm, which keeps the cache full, and the cut back K algorithm. The barometer pointer cache

is only half full, so only half the data has to be flushed compared to the two pointer algorithm. In the barometer pointer and the two pointer algorithm a secondary effect occurs that reduces the penalty of a task switch. Each time the TOS moves, some data is moved to or from the cache. So a part of the dirty data is written back to main memory before a task switch. This reduces the amount of data to be written back at a task switch. Although the cut back K algorithm is a good choice for an environment with no task switching, the barometer pointer algorithm can be a good choice if task switching occurs frequently. The two pointer algorithm, which has even more write back traffic, contains twice as much data as the barometer pointer algorithm. The performance of this algorithm on task switches is difficult to estimate. It can be as bad as the cut back K algorithm, or even better than the barometer pointer algorithm.

The low bus traffic and the good performance at task switches can be achieved if a mix

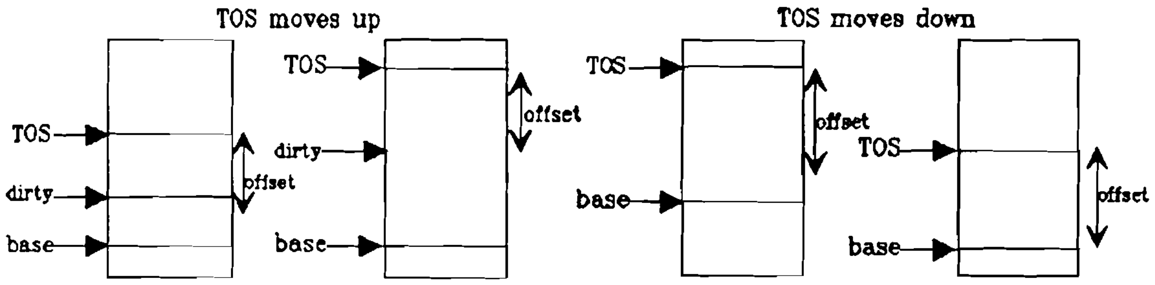

Figure 3.6 The hybrid algorithm

of the cut back K and the barometer pointer algorithm is used (figure 3.6). In the hybrid algorithm, blocks that are below a certain offset from the stack pointer are written back to main memory, but are not cleared. As soon as the stack pointer and the base pointer are within a certain distance, a block will be prefetched. The offsets at which blocks are fetched or written back can be chosen to match the characteristics of the memory and

the data requests. A dirty pointer will be used to keep track of the lowest dirty block. This pointer must be updated on each write. After each write and each increase of the TOS the distance between the TOS and the dirty pointer is checked against the offset. If the offset is exceeded, write backs are issued. A prefetch pointer will keep track of the distance between the base and the TOS. Each time a block is fetched (base changes) or the TOS changes, this pointer is updated. A special situation occurs when the TOS approaches the bottom of the stack. Even if the prefetch pointer is smaller than the offset, no prefetches must be issued, since blocks below the bottom of the stack may not be fetched. A check of the dirty and the prefetch pointer can be done when no memory requests are pending. This ensures that the interference with demand fetches and write backs is small. The interference can be reduced further by interrupting the prefetch memory accesses when a demand fetch or write back occurs. The prefetch part of the algorithm has to be separated from the demand part as much as possible to reduce the overhead for the demand part, which determines the speed of the cache.
This algorithm requires a more intelligent and complicated controller than the cut back K algorithm. This can also have effects on the speed of the cache. The access time of the cache on a hit can be increased, so that it is no longer possible to handle one request in a clock cycle. Therefore the cut back K algorithm can still outperform the hybrid algorithm if the frequency of task switches is low.

If the frequency of task switches is high, a direct mapped cache can have advantages over a linear cache. A direct mapped cache keeps the complete address of each entry in the tag memory. This type of cache does not need to be flushed on a task switch. It is possible that parts of the stack are still in the cache if an interrupted process is resumed after a few task switches (for large caches). If only little time has past between the task switches, the chance that the cache contains something useful rises. In the direct mapped cache the same replacement strategies can be used as in a linear cache. However, prefetching replacement strategies can now replace useful parts of the stack of other processes. Prefetching algorithms will also be more complicated than in linear caches, because task switches now can create holes in the cache (fig. 3.7). Besides,

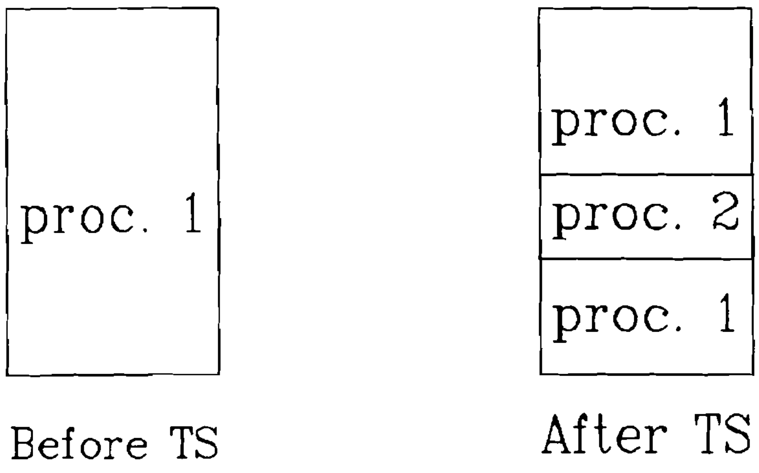

Figure 3.7 Task switching in a direct mapped cache

replacement strategies will only keep track of the pointers of the current process. Remembering the pointers of other processes has no use, since each process can change the contents of the cache. After a task switch the pointers will point to the first block. Only when the cache fills up, the algorithm becomes aware of other blocks of the current process residing in the cache. Of course, this can be eliminated by a time and/or hardware consuming search process of the cache...

To compare the replacement strategies and the two cache designs in a multi tasking environment, the linear cache will be simulated with the cut back K algorithm and the hybrid algorithm. The direct mapped cache will be simulated with a replace on demand algorithm (equivalent to the cut back K algorithm) and an algorithm comparable to the hybrid algorithm. These algorithms are almost the same as the algorithms for the linear cache. To determine if e request is a miss, they not only have to check the boundaries, but also have to check if the requested data already resides in the cache. All caches have to be simulated for various amounts of task switching.

3.3.5 Coherency

As said before, there are no coherence problems in a direct mapped cache without process migration or in a linear cache. In a direct mapped cache with process migration the coherence problem does exist. It can be solved by using a linear cache, prohibiting migration or by using a cache coherence algorithm. It is not useful to flush a direct mapped cache on a task switch, because the effects of a task switch would then be equal to those of a linear cache, which has a lower complexity and has a smaller tag field (so it should be used instead).

It is unlikely that a coherence algorithm, with its overhead, will result in an improved performance, so no coherence schemes will be used on the stack cache. The various coherence schemes have been discussed in chapter 2.

Chapter 4 Architecture of the data caches

A cache can be divided in several blocks (fig. 4.1) according to the functions the cache has to provide. One block will hold the actual data. This block is called the data ram. Another block will hold information about the data in the data ram. This block, called the

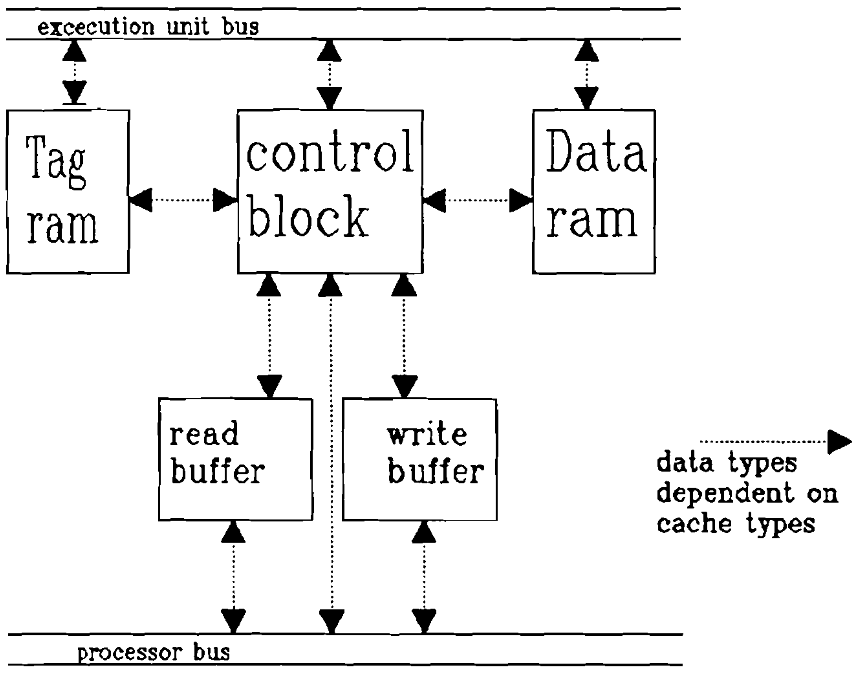

Figure 4.1

tag block, consists of a tag ram, and, optionally, some registers. Furthermore some control logic is needed. The control block controls all accesses to the cached data. The address of the requested data is issued at the same time to the data ram and to the tag ram. Thus the data will be ready for transport to the execution unit at the same time as the hit information is ready. This parallelism speeds up the cache operations. It can be useful to use some buffers in the design. Although these buffers improve the cache performance, especially in case of a cache miss, the extra control logic will increase the complexity of the cache. The extra checks that have to be performed can increase the access time of the cache. If necessary (global data cache) a coherence controller can be included in the control block. As stated in chapter 1, the most wide difinition of local data will be used. In this chapter only stack references will be discussed. How a stack cache miss is detected is discussed in section 4.7.

All these blocks will be discussed in the following sections. Section 4.1 describes the blocks that are used in all types of data caches. Section 4.2 handles the communications to the MMU. In section 4.3 the architecture of a linear stack cache is discussed, section 4.4 deals with a direct mapped stack cache and in section 4.5 some fundamental architectural considerations of the global data cache are discussed. Section 4.6 discusses multi port issues and proirit, section 4.7 revisits the definition of local data and finally the performance of the cache is discussed in section 4.8.

4.1 General cache blocks.

Although some blocks have a specific implementation for a cache type, most blocks of the cache are the same for all caches, or differ only slightly between the caches. If a cache type uses a completely different implementation of a block, this will be discussed in the section on that cache type.

4.1.1 The data ram

This part of the cache contains the actual data (fig. 4.2). The organisation of the cache ram is the same for a direct mapped and a linear cache. A normal single- or multi- ported ram can be used. In a set associative global data cache multiple ram banks have to be used, and some selection for the correct bank has to be used. Multi ported Ram can be used to service more than one data request at a time (i.e. two reads and one write per

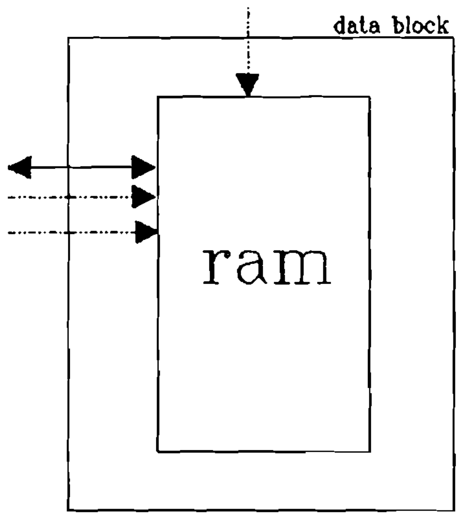

Figure 4.2 The data block

instruction). It can also be used to write some (pre)fetched data to the Ram, or read some write back data from the ram, while servicing a data transaction with the execution unit. The allocation of the ports will be handled by the control block.

The data can be accessed by the execution unit and the control block. The execution unit gives read and write requests to the data ram, which can be blocked by the control block, depending on the TAG information. The control block accesses the data ram to control the replacement of blocks in the data ram.

4.1.2 The Tag Block

The tag block consists of a ram part, which contains the virtual (and in the global cache in some implementations also the real) addresses and the status information of the blocks in the data ram. The tag ram of the linear cache only holds the status information of the data blocks. Information about the addresses is kept in some special storage (registers or Ram).

The tag ram will be accessed by the execution unit and the control block. The execution unit provides the least significant bits (LSB's) of the address of the request to the tag ram, so this ram can provide the status information and the most significant bits of the address of the block in cache to the control block. The control block uses this information to determine whether there is a hit or not.

The control block accesses the tag ram to update the status information. The status has to be updated when the block in the data ram is written back to main memory (is no longer dirty), when the block is changed by the execution unit (becomes dirty), and when the block is replaced by another block. In the last case, the address has to be updated too.

In the global data cache, the snoopy coherence controller may access the tag ram too. Because many controllers can demand access to the tag ram at the same time, this ram has to be multi ported. The number of ports depends on the characteristics of the data requests to the cache.

4.1.3 The read buffer

A read buffer can be applied when the block size is larger than the number of bytes (quads) requested by the execution unit or the number of bytes transferred between the cache and main memory in one data transfer cycle. The buffer increases the performance of the cache because the execution unit can access data, fetched from main memory, as soon as it is available, and does not have to wait until the whole block is in the data ram (this can be imitated by using valid bits for transfer blocks or even quads). Secondly, the access time to the data ram in case of a prefetch can be limited. The access time is set by the buffer and the data ram, and no longer by main memory. This makes it easier to perform prefetches in the background.

The read buffer can also decrease the miss penalty of a write miss. When the execution unit wants to write data to a block which is not in the cache, the data can be written to the read buffer before the rest of the block is fetched from main memory. The control unit must take care that this data is not overwritten by the data fetched from main memory. This strategy can reduce the miss penalty for writes to zero, since a write to the buffer takes no more time than a write to the data ram.

The status information and the addresses of the data in the buffer can reside in the tag ram. Thus all status information is kept in one block. The buffer can be seen as an extension of the data ram. The size of the buffer is dependent on the size of a block. If the cache is multi-ported, a buffer that contains more blocks can be advantageous, since misses to different blocks can occur at the same time (i.e. two read misses and one write miss on two read ports and one write port). The advantages of a read buffer are dependent on the size of the cache (with a low miss ratio the miss penalty will become less important), the block size (small blocks can be transferred to the data ram with no extra cost) and the size of the buffer.

The buffer can be accessed by the control block. The access can be demanded by the execution unit if a part of a block is already in the buffer, but not in the data ram (this can be a read or a write request!!). The control block itself can access the buffer for prefetch activities. Because the control block has to check if the data are in the read buffer before it can access the buffer, the buffer must have a very short access time. The buffer will also be accessed by the Memory Management Unit (MMU), when parts of a block are moved from main memory to the cache.

4.1.4 The write buffer

Besides a read buffer to speed up cache misses, a write buffer can used to speed up cache misses in a full cache, when one block is replaced by a new block. If a dirty block needs to be replaced, it has to be written back to main memory, before the new block can take its place in the data ram. When no buffer is used, the miss penalty for the replacement of a dirty block will be about twice as large as the miss penalty for a non-dirty block. Both miss penalties can be equalised using a write buffer. On a miss on a dirty block, the block is written to the buffer before the new block is fetched. Only when the transfer of the block from main memory is completed, the dirty block is written to main memory. Since main memory is slow, the read request can even be issued before the block is written to the write buffer. Thus, the miss penalty will be dependent only on the time needed to fetch a block. Problems can arise however, when the write buffer is full when a dirty block has to be replaced. A replacement strategy which uses prefetching (linear cache), or a buffer with a size of more than one block will solve this problem. Prefetching will limit the number of misses, and thus limit the number of times that the write buffer is full when a new dirty block needs to be replaced. When a large buffer is used, more than one block is needed to fill the buffer, so the buffer will be full less often. The status information and the address of the data in this buffer can reside in the tag ram too.

The write buffer needs some control logic to avoid conflicts between the control block and the MMU. If something is written to a transfer block (access via the control block) while this data is moved to main memory, the write has to be cancelled, and started again when the write from the execution unit is completed (see also section 4.2).

The write buffer can be accessed by the control block. Because the tag will reside in the tag ram, the execution unit can still modify or read the data in this buffer. The data can also be moved back to the data ram. The buffer can also be accessed by the MMU (either directly or via the control block) to move data to main memory when the MMU indicates that data can be transferred (dependent on the access protocol between the cache and main memory/MMU). If a block consists of transfer blocks, only the dirty transfer blocks have to be written back. When the MMU indicates that all dirty blocks are written back, the whole block will be marked invalid (the space is freed).

4.1.5 Intelligent buffers.

The read and the write buffers are used to minimise the miss penalty. To work correctly, they almost never will be filled. An intelligent buffer can use this free space, in a linear cache or a direct mapped cache with a replacement strategy, as an extension of the cache. This is done by introducing a controller and a small tag memory for the buffers, containing the destination of the data (fig. 4.3). There are three possible destinations for

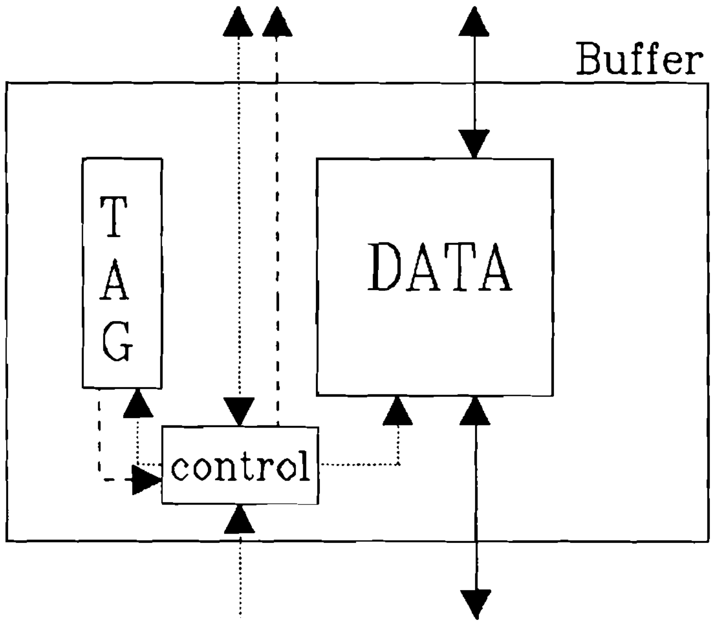

Figure 4.3 An intelligent buffer

the data, each giving certain "rights" to the data.

Destination:

RAM: This data mostly will reside in the read buffer. It arrives there by a (pre)fetch. As soon as a whole block has arrived, it will be moved to the data ram. It can not be replaced by another block before it is moved to the ram.

Data of this type can also reside in the write buffer, in case the TOS moves up and down in quick succession. The data will be moved to the data ram as soon as possible. In a linear cache there will always be place in the ram, but in a direct mapped cache it can replace a dirty block from another process. In the latter case the data will be moved to the read buffer first. Data with destination RAM may be dirty.

MAIN: This data will be dirty and has to be written back to main memory. It normally resides in the write buffer. It can not be replaced by other data before the write back has completed. A block with destination MAIN in the read buffer will be moved to the write buffer as soon as possible.

BUFFER: Data with destination buffer only resides in the intelligent buffer. It is data which is requested by the processor, but does not fit in the RAM. The data may not be dirty and can be replaced by other data at any time. It can be reside in each of the buffers (not simultaneously in both). A buffer is not considered full if it contains data with this destination.

Changes of the destination of the data blocks can be made by the control block and by the buffers themselves. The following transitions exist:

RAM -> BUFFER: Non-dirty blocks for which there is no longer a place in the ram. This can happen when the TOS increases while a block is (pre)fetched. When the block is completely valid (in the buffer), the buffer sends a request to move the block to the control block. This block can send back a signal that it no longer has a place reserved for the data in the ram. The destination is then set to buffer.

RAM -> MAIN: Dirty blocks for which there no longer is a place in the RAM.

MAIN -> BUFFER:	If all dirty transfer blocks from the dirty block are written back, the destination is changed to buffer. In a linear cache with the hybrid replacement algorithm however, the block will be invalidated if the write was dictated by the dirty pointer (a copy of the block will still reside in the ram). This invalidation will require an extra state (MAIN, DIRTY POINTER) or some communication with the control block to check for a copy.
MAIN -> RAM:	This transition is dictated by the control block. It occurs when the block is situated below the base and the TOS decreases. The control block can then force the transition by issuing a (pre)fetch.
BUFFER -> RAM:	This transition is also dictated by the control block, but now the data is dirty.
BUFFER -> MAIN:	This transition automatically happens when something is written to the block. Since it becomes dirty, it has to be written back.

The use of intelligent buffers will result in a lower miss ratio but also some extra hardware. The buffers can each have a controller to govern the changes in destination, or share one controller.

4.1.6 The control block

The control block (fig. 4.4) controls the dataflow in the cache. It avoids conflicts between the requests which want to access the various stores and buffers at the same time. Since most stores are multi-ported, it can resolve conflicts by appointing different ports to different requests. This can be a fixed allocation; each port is dedicated to a specific task, or a dynamic allocation, where the ports are allocated depending on the demand. A fixed allocation will require more ports, but less logic than dynamic allocation. The type of

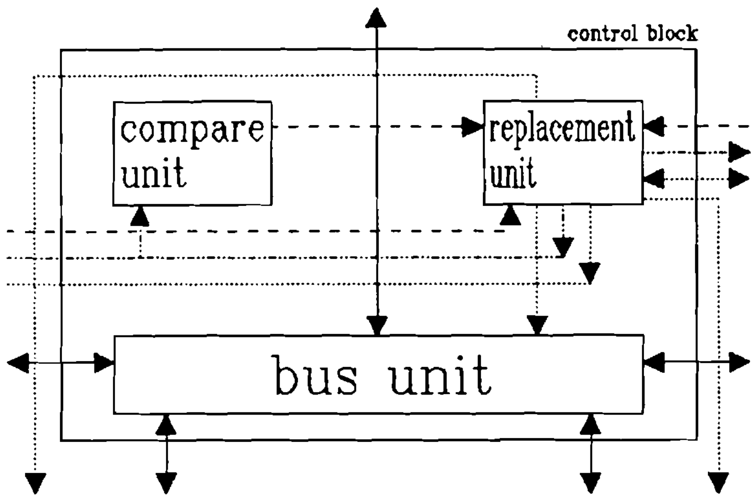

Figure 4.4 The control block

allocation and the number of ports for each ram or register block is dependent on most of the cache parameters. A good choice can only be made when simulation results are known. A few ports are needed to serve requests simultaneously. By using some extra ports, data can be written from the read buffer to the data ram, while the execution unit fetches data from the data ram. The number of these extra ports also depend on the extra bus traffic introduced by the replacement strategies.

The actions that take place in the control block are shown in fig. 4.5. Horizontally actions that can be executed in parallel are shown, the serial stages are placed vertically.

Stage 1:
On a request from the execution unit, the tag block responds with hit information (address or address and offset) and the status of the block which matches the (least significant) bits of the address. The comparison of the most significant bits to determine a hit can either be done in the tag block or in the control block. The status info of the buffers will

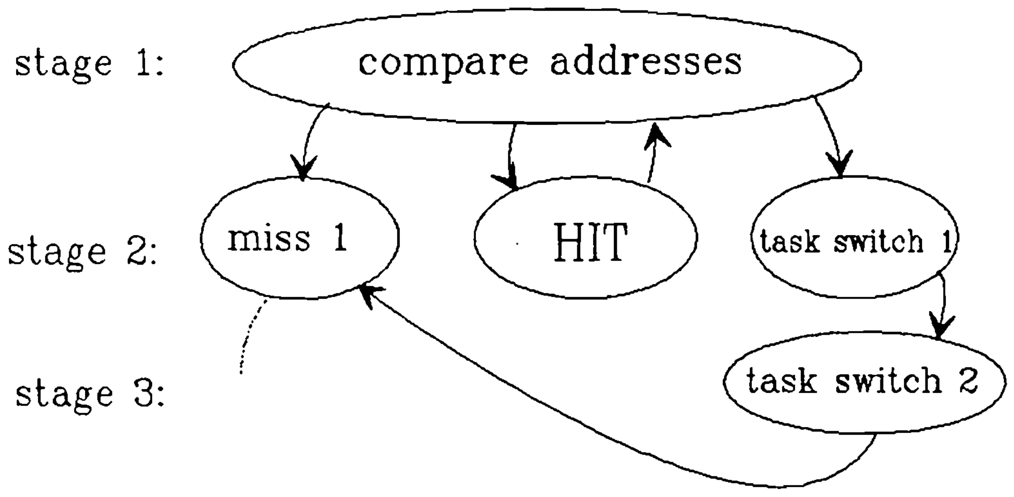

Figure 4.5 Cache actions on a reference

reach the control block on a separate bus, so no interference occurs between the buffers and the ram. A hit is detected if the PIN and the address match and the block is valid. If the RAM nor any of the buffers produces a hit, there is a cache miss. If the PIN of the RAM also does not match, a task switch has occurred.

The various comparisons can be performed simultaneously. The PIN consists of 16 bits, so a 16-bit comparator is needed to check the PIN. The size of the comparators for the addresses is dependent on the block size and the cache size. The C-processor address space is 4 gigabyte, addressed in quads (4 bytes), so 30 bit addresses are used. The address of a block in the buffers does not include the bits used to select a quad in a block. For a block size of 2^m quads the address contains (30 -m) bits. The buffers require (30-m) bit comparators. For the Ram a distinction has to be made between linear caches and direct mapped caches. In a linear cache the address of the request has to be checked against the borders of the cache (base, TOS). If the complete address is used to address the cache, the address comparison requires two (30-m) bit adders

(subtractors). A cache, which is addressed with an offset from the TOS, needs only a (30-n) bit adder. (cache size = 2^n quads)
A direct mapped cache requires a (30-n) bit comparator to determine if there is a hit. The n-m remaining bits are used to select a block in the cache.
A p-ported cache requires an address check for each port to the ram, and for each buffer entry (fully associative buffers).

Example: *linear cache, 5-ported (p=5), block size 16 quads (m=4), cache size 512 quads (n=9), both buffers contain 4 quads.*

PIN comparison RAM: 1	*1 16 bit comparator*
PIN comparison buffers	*40 16 bit comparators*
RAM address check	*10 26 bit adders*
(rel. addressing	*10 21 bit adders)*
buffer address check	*40 26 bit comparators*

This makes a total of 91 signals, which are used to produce 5 hit signals. Most of the hardware is used for the buffers. It may be awarding to reduce these costs (see also section 4.7).

These results are used in stage 2, where the data are moved to the correct part of the cache on a hit, and the replacement algorithm starts working on a miss.

Stage 2:
Since the replacement algorithms differ for the various cache designs, the actions on a miss are treated in the sections on the specific control blocks (4.3.1 and 4.4.1).

On a read hit, the data is moved from the hit place (RAM or buffer) to the correct port. On a write hit, the data is written to the hit place and the tag of the (transfer) block is marked dirty. The control block of the hybrid replacement algorithm will also update the dirty pointer. A hit on the write buffer requires some extra actions. The replacement algorithm now has to decide if the block will be moved back to the data ram (actions like on a miss, but fetch from write buffer), or if the write back will proceed. The hybrid algorithm requires some special attention here. In this algorithm, a block can reside in both the cache ram and the write buffer. When it is written back because the distance between the TOS and the dirty pointer is to small, it will be written to the write buffer first. Then a copy of the same data resides in both the data ram and the write buffer, until the block is actually written back to main memory. The write will then proceed to both the ram and the buffer. The block in ram will remain not dirty.

In case of a write miss, it is possible to reduce the miss penalty. The data can be written to the data ram or the read buffer and the rest of the block can be filled in from main memory later. Thus the write can take place immediately and the rest of the block can be fetched later. This results in no write miss penalty. This option will be discussed further in the next sections, since the difficulties implementing it vary for the different caches.

The handling of task switches is different for a direct mapped cache and a linear cache. The direct mapped cache treats a task switch like a normal cache miss. Only if a replacement algorithm is used, the pointers have to be initialised for the new stack.
A linear cache on the other hand, needs to be flushed. In stage 2 all dirty blocks in the cache ram are moved to the write buffer, and from the write buffer to main memory. The time needed for a task switch depends on the number of dirty blocks and the size of the write buffer. It varies between no time (no dirty blocks) to some memory cycles (not all dirty blocks fit in the buffer). The time also depends on the use of burst writes. It can be faster to write back a whole block (burst) instead of only the dirty transfer blocks (several normal writes). When all dirty blocks have been written to the buffer, the data ram is invalidated, the new PIN is put in the PIN register and the pointer registers in the tag are initialised (stage 3). Finally the new process can start with a miss.

The stages 1) and 2) have to be completed in the same clock cycle, so that the cache can work on the same speed as the execution unit. The write request will probably be in the critical path of the cache. The dirty bit in the tag has to be set and the data has to be written to the cache. These writes can only start when the hit data is present. The write has to be finished before a read of the tag and the data can be issued in the next clock cycle.

The control block accesses all other blocks of the cache, and also gets information of all these blocks and the execution unit.

4.2 Communications to the MMU.

The MMU is placed between the cache and the main memory and plays an important role in the communications between the two parts when data are transferred. The control of this traffic introduces some communication between these two units. The MMU communicates with the control of the buffers to provide the correct data to the requesting buffer as fast as possible. On a miss, the control for the read buffer (the read control) and the control for the write buffer (the write control), can initiate the data transfer protocol with the MMU. At a read request, the address of the wanted transfer block is provided to the MMU, which will transfer this part of the block first, and sends the rest of the block later (the standard data types cannot overlap (transfer) block boundaries). The read control will update the tag of the buffer each time a transfer block arrives. As soon as the MMU signals that the whole block is sent, the read control can initiate the transfer from the buffer to the data ram. The read control has to keep track of bytes that are written by the execution unit to the read buffer to avoid that these data are overwritten by data from main memory. If a complete transfer block is written, the transmission of this block can be cancelled by the read control.
The communications with the write control are a little more complicated, since they can be interrupted by hits to data in the read block. Only the dirty transfer blocks of a block have to be transferred. The total amount of data to be transferred now is dependent on the size of the transfer blocks and the number of dirty transfer blocks. Large transfer blocks will result in more data per block and a larger probability of dirty transfer blocks.

The transfer of a transfer block can be suspended or cancelled by a write to that transfer block. This requires some extra signals like 'halt' and 'cancel'.

Write control can be elaborated further with the introduction of 'pseudo-writes'. In an instruction there are three addresses : two for the operands and one for the result (addresses can be implicit; a:=a+b). The address of the result is known to the execution unit before the result is available. If the address of the result is given to the cache in a pseudo write before the data itself, the control unit can use the address to get the hit information, and issue a prefetch for the requested block on a miss. The pseudo write can be labelled with a request number, which can be used to transport the result to the cache. A write then acts as:

```
pseudo_write <address>, <request_number>
calculate result, get hit information
write <request_number>, <data>
```

The maximum request number depends on the maximum number of cycles it can take for an instruction to calculate the result. The prefetch can be stopped if an exception occurs in the execution unit which eliminates the need to store the result (divide by 0). This system can also be used for the data transport between the cache and main memory. It will require a more elaborated communication between the cache control and the MMU.

4.3 The linear stack cache.

In a linear stack cache, the tag area requires special attention. First of all, the stack cache needs only a minimal tag ram. The data in the cache is a continuous part of the stack. Therefore only the lowest or base address and the highest (TOS) address of valid data in the cache need to be stored. The only information that needs to be stored for each block, regards the write-back or dirty status. The second effect of the continuous data in the stack is that some special replacement or 'prefetch' algorithms can be used (section

3.3.3). To implement some of these algorithms several extra pointers have to be used. The functions of the blocks will now be described.

4.3.1 The control block

The control block has to coordinate the transport of the data in the cache. The unit has to avoid conflicts between data moving between the buffers and the data ram, and the servicing of the data request. It will also implement the replacement strategy of the linear cache. Two replacement strategies will be implemented (simulated) for the linear cache: the cut back k algorithm and the hybrid algorithm. The control block itself consists of several units (fig. 4.4).

The compare unit compares the address given by the execution unit with the boundary addresses in the registers in the tag block. It also checks if the PIN numbers match. This block provides this information to the replacement unit.

The replacement unit implements the replacement policy of the cache. The actions for a hit are discussed in section 4.1.6. Since there are two replacement policies to be simulated for the linear stack cache, two replacement units will be discussed. The unit for the cut back k algorithm has to provide the following replacement functions on a miss (fig. 4.6).

Stage 2)
First a check is made to see if there is enough space to store the block(s) to be fetched. If the ram is full, the data will not be placed in the ram, but will reside in the buffers. In the ram it would replace data closer to the TOS, which is more likely to be used. On a read, the data will be stored in the read buffer. On a write request, the data is written to the write buffer, if the size of the data is equal to a transfer block. The data can be written back without the rest of the block being fetched. Otherwise the data will be stored in the

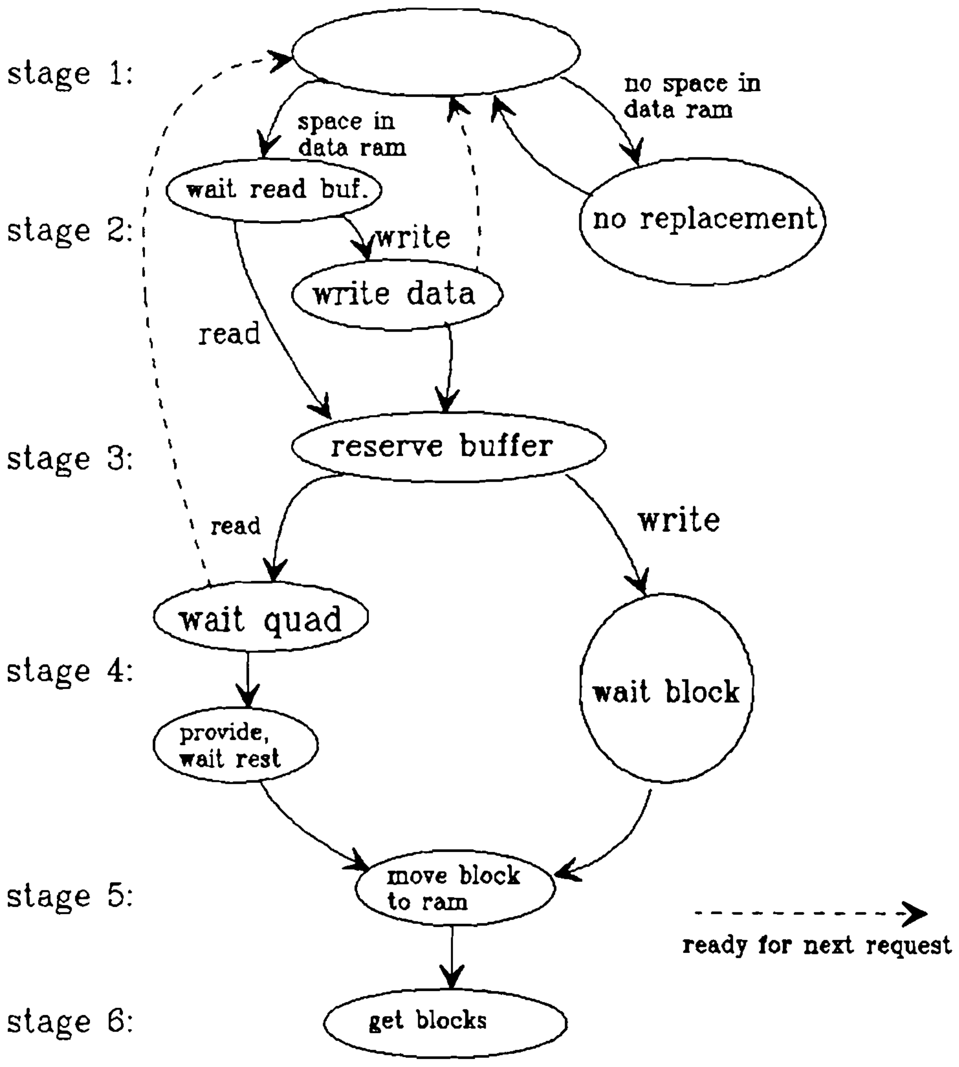

figure 4.6 Cut back K cache, actions on a miss

read buffer. The rest of the transfer block is fetched and the transfer block is moved to the write buffer. If the buffer is full, the request will be hold until a block has been moved to (write buffer) or from (read buffer) main memory.

If the ram does not have enough space to store the requested block (and the blocks between this block and the base), the control block has to wait until a block is free before the read request can be issued (stage 2a).

On a write request the data will be written to the read buffer, and the tag is updated (stage 2b).

stage 3)
After the fetch is issued, some free space for the requested block is reserved in the buffer.

stage 4)
On a write this state consists of the waiting for, arrival of and storing of transfer blocks if the fetched block.
The fetch request for a read first fetches the requested transfer block. As soon as this data has arrived, it is provided to the execution unit. Afterwards the control block waits for the rest of the block to arrive.

stage 5)
When the whole block is stored in the read buffer, the block is moved to the data ram. The tags are updated. The blocks between the old base pointer and the new base are marked invalid (except the fetched block). The base pointer is updated.

stage 6)
The blocks between the old and the new base pointer are fetched. The tags of these blocks are updated (=marked valid). The number of blocks fetched will be a multiple of K.

The miss penalty for a write consists of stage 2). This can take from 1 cache cycle if space is available, to several memory cycles if a block has to be moved to free some buffer space. Everything after stage 2) can be performed in the background, i.e. a new request can be serviced.

The penalty for a read miss is more severe, since it also includes stages 3) and 4a). It now takes at least one memory cycle before the request can be serviced. Everything from stage 4b) on can be performed in the background.

A second series of actions is started in the control block when the TOS starts to move (fig. 4.7).
If the TOS moves down, all data between the old TOS and the new TOS are no longer of interest (even if some blocks are dirty). The valid bits in the tag are reset to make room for data below the base (stage 1). If the new TOS is below the base pointer, the block containing the new TOS is fetched and the base pointer and the TOS are updated (stage 2).

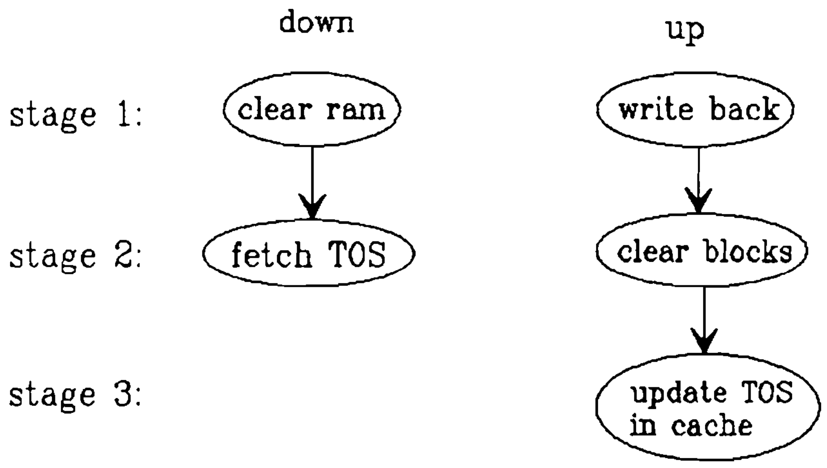

Figure 4.7 Cut back K cache: TOS movement

On an up movement space has to be reserved for use of the stack. If enough space is available (new TOS < base + cache size) only the TOS register needs to be updated (stage 3). Otherwise the base has to be moved up to free this space. The dirty blocks are written back to the write buffer (free space, stage 1). Then the blocks between the old TOS and the new TOS will be cleared. This consists of setting the valid bit and, optionally, clearing the blocks by resetting them (stage 2). Finally the TOS and the base are updated in stage 3). Normally clearing the blocks is not necessary since usually newly allocated cache space is written first. Cache blocks can be cleared to avoid the possibility of reaching unwanted data by moving the TOS up after, for example operating system calls.

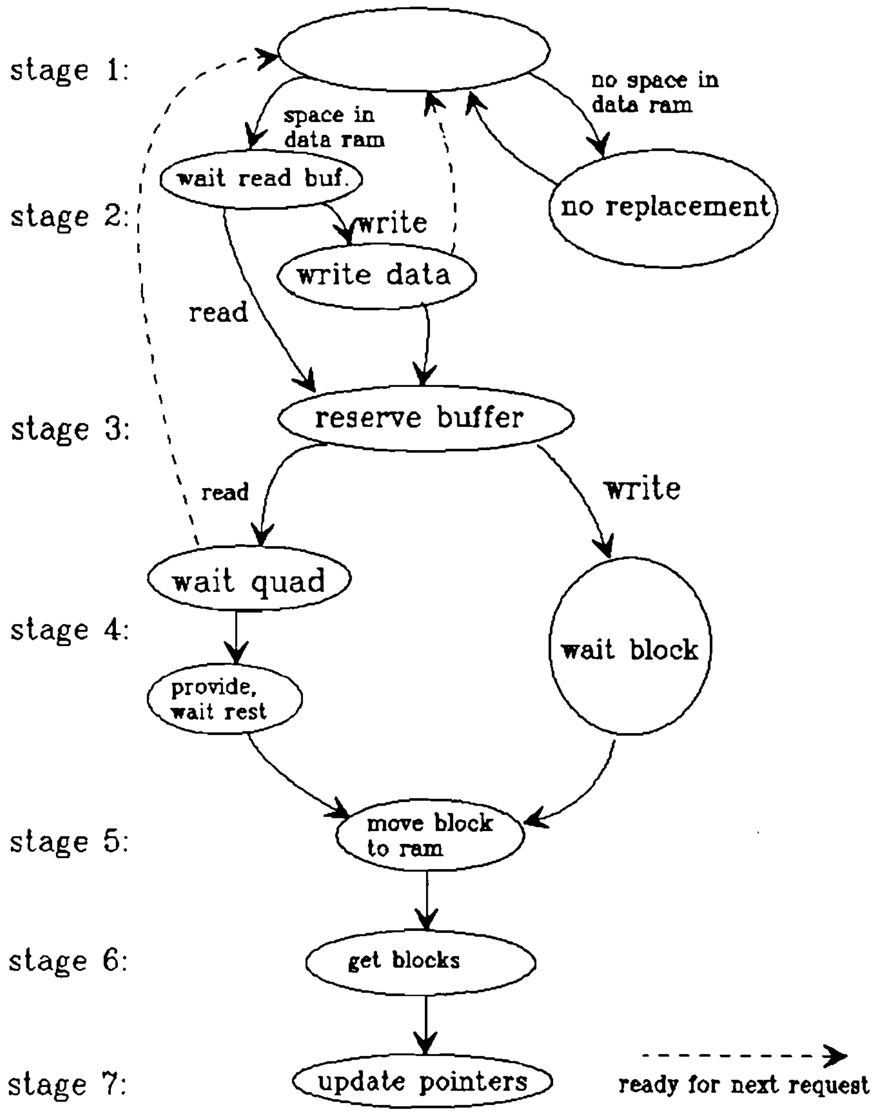

Figure 4.8 Hybrid algorithm, cache miss

An extra clear bit in the tag ram can be used to identify addresses that are appointed to the cache, but never have been written to.

The hybrid algorithm uses an extra pointer, the dirty pointer, to keep track of the lowest dirty block in the cache. A second extra pointer is used to record the distance between the TOS and the base. The state diagram of the algorithm looks very much like the state diagram of the cut back K algorithm (fig. 4.8). However, an extra state is used to update the dirty and the prefetch pointer.

On a miss, the 7th state is introduced. If some data below the base is fetched, the distance between the base and the TOS increases. The prefetch pointer will be updated, which can eliminate the need to prefetch blocks.
A write miss results in a dirty block below the base. The dirty pointer now has to point to this block. This will probably cause the dirty transfer block in this block to be written back to main memory. The distance between the dirty pointer and the TOS is likely to exceed the offset.
As can be seen in the figure, the demand replacement algorithm is active from stage 1) to 6), while from stage 7) on the prefetch part of the algorithm becomes active again. In this way the prefetch algorithm does not interfere with the demand algorithm.

When the TOS moves, the pointers have to be updated (fig. 4.9). As the TOS moves down, the prefetch pointer has to be updated (the distance between TOS and base decreases). If the TOS passes the dirty pointer, the pointer will be set to point to the new TOS. This assures that no more blocks are written back (TOS - dirty = 0, < offset) since there are no dirty blocks below the new TOS in the cache (stage 3).

On an up-movement of the TOS, the prefetch pointer has to be updated; the distance between the base and the TOS increases. If the TOS passes (base + cache size), the base pointer has to be updated too. Finally, if the TOS even grows beyond (dirty + cache size), the dirty pointer has to be updated, after the dirty blocks between the old and

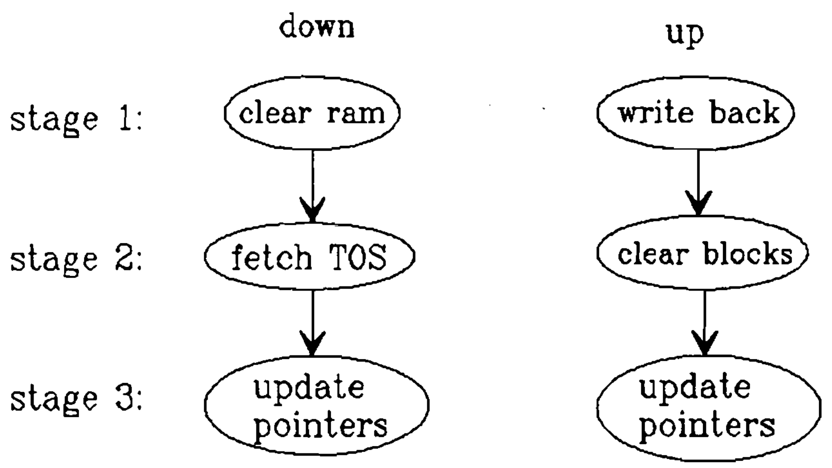

Figure 4.9 Hybrid algorithm: TOS movement

the new dirty pointer are written back.

If the demand replacement algorithm does not require any bus cycles (i.e. on hits), the hybrid algorithm can use these cycles to move the pointers to within their offsets. Normally, the offsets are chosen thus, that the dirty pointer and the prefetch pointer can not be both outside their offsets. In an almost empty cache, only prefetches will be issued, while in a full cache only write backs are useful. The prefetches and dirty pointer write backs can always be interrupted by demand reads and writes. The blocks written back remain valid in the cache; since they are no longer dirty, the dirty bit in the tag is reset. After a dirty block is written back, the algorithm will have to search the next dirty block. This is done by reading the dirty bits, either through the normal ports on the tag, or through an extra port. Of the parts of the tag, the dirty bit is used least; it is only written on a write. Only on a write or on a write back this bit has to be accessed. The construction of the tag thus can help the implementation of this algorithm.

Finally there is one occasion when the prefetching of blocks has to be suppressed. When the TOS approaches the bottom of the stack normal use of prefetches would result in fetching data below the stack space. This can be avoided by a check before the prefetch is issued.

The last unit to be discussed is the bus unit (fig. 4.4). It decides which port and bus have to be used for each data transfer in the cache. The allocation of the busses can be static or dynamic.

The control block for the hybrid algorithm is more complex than the control block for the cut back k algorithm. This can result in longer service times for the hybrid cache. Since there are more accesses to the data ram, more write backs, prefetches, an extra port for the data ram or the tag ram might be needed to avoid bottlenecks. The performance of this algorithm might be better, mainly at task switches. In normal operation the expected higher hit ratio can be offset by a higher bus usage (see section 3.3.4). If the resulting service times for the hybrid algorithm result in a cache that can not service requests in one clock cycle, the cut back K algorithm should be chosen or the performance of the hybrid linear cache should be improved (introduction of the 'pseudo write'). If the bus bandwidth proves to be the bottleneck, the least demanding algorithm should be chosen (cut back K).

4.3.2 The tag block

In the tag ram of the linear cache the status of all data in the cache and the addresses of the data in the buffers (if present) will be stored (fig 4.10). Secondly, there are several registers present which contain the addresses of the various boundaries needed for the replacement algorithm. For the cut back k algorithm, only the base register, which contains the lowest valid block in the cache, and the TOS register, which contains the highest valid address or the TOS, are necessary. The hybrid algorithm requires two additional registers, one containing the address of the lowest dirty block in the cache and

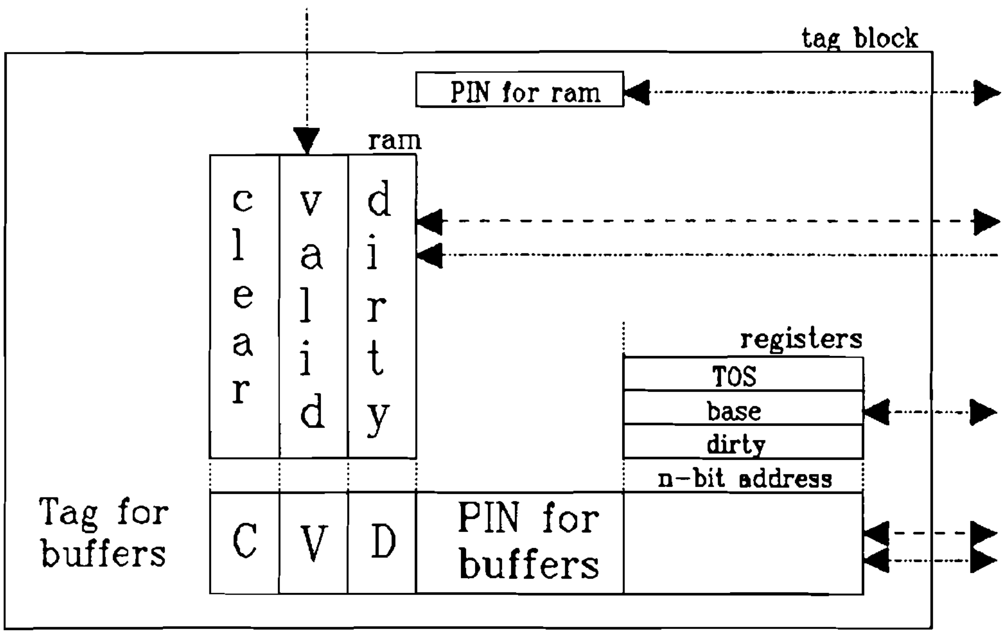

Figure 4.10 The tag block for a linear cache

one containing the distance between the TOS and the base address.

The ram contains two status bits for each (transfer) block in the cache and in the read and the write buffer. These bits are a valid bit, which indicates that the block in the cache is still valid, and a dirty bit, which is set when a write to the block occurs.

The base register and the TOS register contain the complete address of the boundaries of the stack cache. The addresses of the blocks in the buffers are also in separate registers. The dirty register only needs to store the LSB's of the address, since it will always point at a stack location between the base and the TOS. A problem can arise when the TOS and the base do not point to the same page. Extra logic is required if this passing of the page boundary causes a non continuous address space for the stack. The PIN of the current process can either be stored in the boundary registers, or in a separate register.

The logic to compare the address issued by the execution unit and the addresses of the contents of the buffers and the registers, can be placed in either the control block or the tag block. This logic gives hit/miss information, and indicates when a block has to be written back to main memory, or has to be replaced.

To implement the hybrid algorithm, two extra adders are necessary to see if the two pointers exceed the offsets. For a cache size of 2^n quads and block size 2^m quads, n - m bits have to be stored and compared (thus 2 (n - m) bit adders required). If a block needs to be stored or fetched, this information is passed on to the control block. In case of a write back, the control unit then accesses the tag to find the dirty transfer blocks to be stored.

The tag ram needs to have the number of ports of the execution unit, plus some ports to use for the replacement algorithm and to move data between the buffers and the data ram while the data requests are serviced. The cut back k algorithm requires no additional ports, since it operates on demand (a block is fetched/stored on demand). The hybrid algorithm requires at least one extra read modify write port to update the dirty pointer when the TOS moves up. This will be a one bit port, dedicated to the dirty bits. To check more than one block at a time, more read ports can be used. The write port can also be used if new data is fetched when the TOS moves down.

In the hybrid algorithm, the control block accesses the dirty bits in the tag to find the transfer blocks to replace when the dirty pointer is smaller than the offset. The dirty bits don't have to be accessed on a read, only on writes and write backs. If the dirty bits are placed in a separate ram block (fig. 4.11) this can reduce the number of ports otherwise needed for the tag ram. However, two rams require two selection parts.

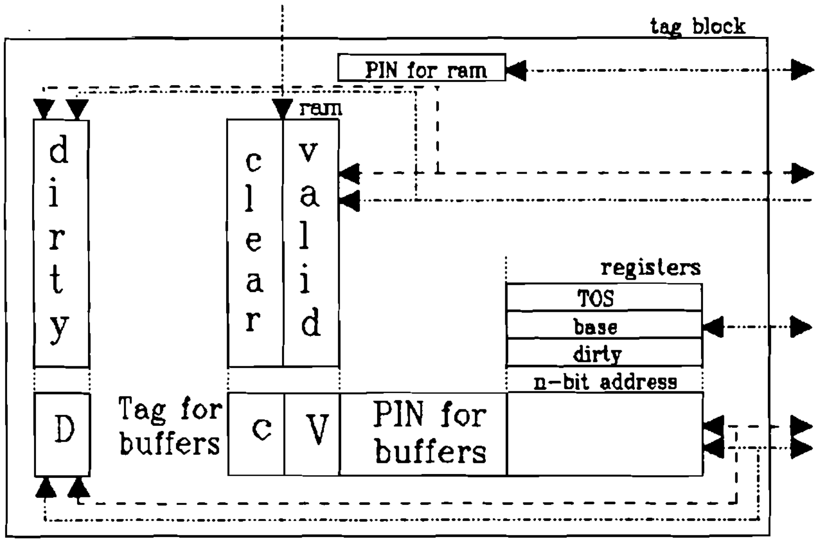

Figure 4.11 Tag for the linear cache with dirty bits separated

The part which contains the addresses and the status of the data in the buffers has to be associative. All entries of buffers have to be checked for all ports. So each entry needs a comparator for each port of the execution unit. Again extra ports for the tags have to be used to move data between the buffers and the data ram (requires an update of the tag) at the same time as data requests are serviced. The implementation of the replacement strategies will also be helped by these extra ports.

4.4 The direct mapped stack cache

The direct mapped cache needs a complete tag; the address of the data needs to be present for all entries. The control block looks a lot like the control block of the cut back k linear cache. Since the direct mapped cache is chosen for its low penalty on task switches, it will not flush the cache on a task switch. The control block and the tag block will be discussed in the following sections.

4.4.1 The control block

As mentioned in chapter 3, the direct mapped cache will be used when the task switching frequency is high. So the replacement strategy of the direct mapped cache can be simple: only the blocks that are requested by the execution unit have to be fetched. There is no kind of prefetching (so the cache can be compared to the cut back k linear cache). Prefetching can be included by implementing the same (hybrid) algorithm as for the linear cache, but is more difficult to implement. It has to check if data below the base resides in the ram (on a miss). The stages are almost the same as for the cut back K algorithm. Only stage 2) is different.

Stage 2)
A block that is fetched now always replaces an other block. If the block to be replaced is dirty, the new block can only be fetched if space is available in the read buffer to store the block and if space is available in the write buffer to store the dirty block.

A movement of the TOS now requires a check of all places where blocks between the old and the new TOS can be stored because these blocks have to be invalidated if they belong to the current process.

4.4.2 The tag block

The tag block of the direct mapped cache (fig. 4.12) holds the same status information as the tag block of the linear cache. But instead of keeping border addresses in registers, the MSB's of all the blocks in the data ram and the complete addresses are kept in the tag ram.

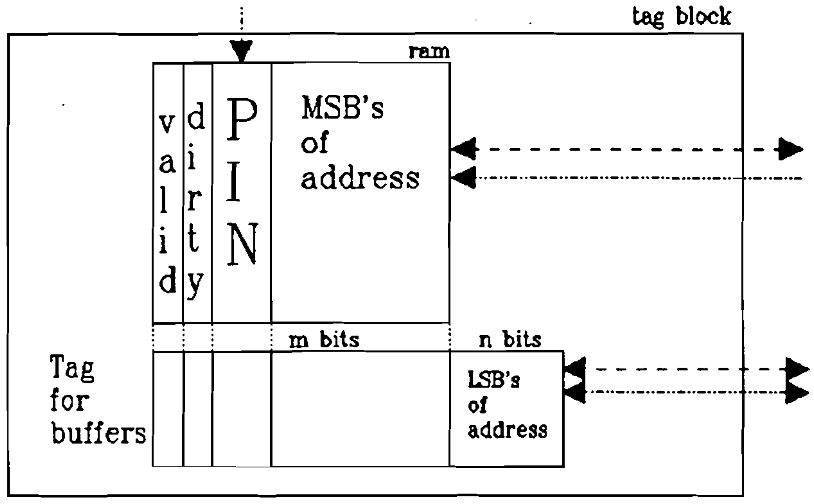

Figure 4.12 The tag block for a direct mapped cache

4.5 The global data cache

In this section the blocks of the global cache are briefly discussed. The cache will use the LRU replacement algorithm.

4.5.1 The data ram

The set size of the global data cache does not need to be one. A set associative cache will be used, probably with set size 2. The data ram now consists of two ram banks, addressed by the same bus (fig. 4.13). When an address is provided, both banks provide their data, and the data corresponding to the complete address is selected by the control

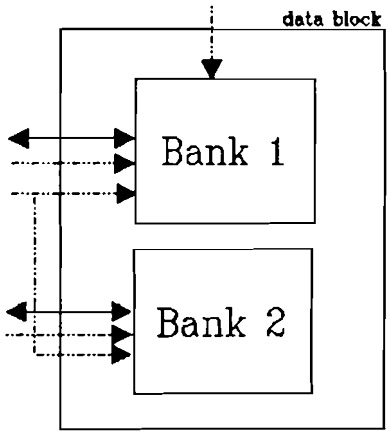

Figure 4.13 The data block of the global data cache

block.

4.5.2 The tag block

The tag block again holds the virtual and the real addresses and the status information of all the blocks in the data ram (fig. 4.14). The real address is kept in the tag ram to avoid real to virtual transformation in coherence issues (see section 4.4.3 about the control block). The tag ram is addressable per set. If an address is given to the ram, the ram provides the tag of both blocks of the data ram. Thus two addresses, two pieces of status information and two PIN's are given to the control block. The status information now includes coherence information and replacement information besides the valid and the dirty bit. These last two bits can be a part of the coherence information. The replacement information will be one bit per set, which indicates which element of the set is least recently used.

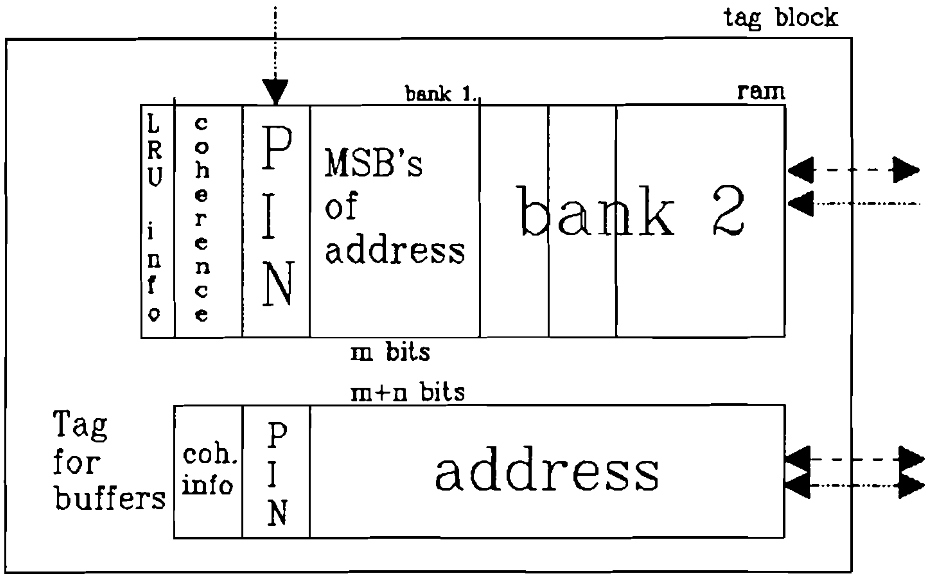

Figure 4.14 The tag block of the global data cache

4.5.3 The control block

The control block consists of several parts.

The control block checks if one of the elements in the referenced set, or the contents of the buffers is wanted by the processor and checks its validity. It sends the correct data to the processor. If the wanted address is not in the cache, the data can be in the cache with an other address, used by an other process (i.e. PIN is different). This so called alias can map to only one set, if the number of sets is less than or equal to the page size and if the LSB's remain the same in the virtual to real address mapping. Aliases can be found by keeping the real addresses in the tag ram too. On a miss, the real addresses of the data in all the sets are compared with the real address provided by the virtual to real address translator' (usually situated in the MMU). If there is an alias, the data is provided and the virtual address in the tag ram is changed. On a miss it signals the replacement unit.

The replacement unit decides on basis of the LRU bit which element of the set has to be replaced. If this element is dirty, it will be written to main memory via the write buffer. It only replaces blocks on a request from the control unit.
The snoopy unit monitors the address bus and compares the addresses on the bus with the real addresses in the tag ram. If the real address was not in the tag ram, a real to virtual translation would be needed for each monitored bus access. This could slow down the bus, because the snoopy unit must have reacted before the bus transfer is completed. The unit reads and writes in the tag block and on the bus according to one of the coherence protocols discussed in chapter 2.

4.6 Multi port issues and priority

Using more than one port for the cache introduces some problems but can give some extra options to the cache. The cache needs logic to detect hits for each port to the execution unit. This logic does not need to communicate with the logic of the other ports, as long as it produces hits. A miss will be handled by a central part of the control block. The ports can give some priority to the requests as only one miss at a time will be handled. The priority of the ports will probably be fixed (i.e. port 1 first, then port 2, etc.). As long as the read buffer is not full and there is a write miss, the execution unit can write to the read buffer as if there was a hit on the read buffer. The rest of the block will then be fetched after possible read misses have been serviced. If there is no place in the read buffer, the writes will be serviced first.
Secondly, some hardware has to check that prefetches and data movements between buffers and the data ram do not interfere with data requests (avoid the use of blocks requested by the execution unit). This last issue mainly affects the extra ports introduced to do these extra transfers.

The minimal number of ports needed for the cache blocks has to ensure that the accesses to the blocks by the control block do not interfere with the demand requests. Moving a block from a buffer to the ram may not prohibit access to the ram for a processor read or write. The data ram needs one extra read port to transport data from the data ram to the write buffer and an extra write port to receive data from the read buffer. The tag needs an extra read port and a write port to let the control block read the tag of the block that is to be replaced, and to write the address of a fetched block. The tag of the buffers can also be accessed by the MMU. Because the changes in the buffers are controlled by the control block (to avoid conflicts), the tag does not need more ports. The buffers themselves need extra ports to exchange data with main memory and the data ram.

Example: *Three ports to the execution unit; 2 read an 1 write port.*
Data ram: 3 read ports + 2 write ports
Tag block: 3 read ports + 2 write ports
Read buffer: 3 read ports + 2 write ports
Write buffer: 3 read ports + 2 write ports
Two read ports and one write port are dedicated to service requests of the execution unit. All blocks together have 12 ports for this task, of which only three can be used simultaneously. Each block has one read and one write port for other tasks. All these 8 ports can be used simultaneously.

4.7 The definition of local data revisited

In chapter one, three definitions of local data were given. In the above, the second, most wide, definition was chosen: all data on the stack is local. Data requests outside this definition will be handled by the global data cache (if present) or will be presented to main memory. If all references to the stack are relative to the TOS, or use a flag, this algorithm makes it possible to send only stack requests to the stack cache, and all other requests to the global cache (if present). If no special addressing modes are used, an additional

check against the lowest address of the stack has to be made to determine a stack cache miss.
The main advantage of this definition is, that stack data can only reside in the stack cache. This simplifies the communications between the processor parts (execution unit, both data caches, MMU, main memory). If intelligent buffers are used, some of the global data in other definitions can be stored in the buffers. At the cost of larger buffers, more of this data can be stored. The buffers will only contain a small portion of the global data. A dedicated global data cache can outperform these buffers.
When no global data cache is used, this definition of local data will probably give the highest performance.

The definition no. 1 given in chapter 1 (only data fitting in the cache are local), almost begs for a global data cache. There is a big penalty for data just outside the cache if the global data cache is not present. Each change of the TOS would cause data to change between local and global. Data that becomes global no longer fits in the cache and can be invalidated, or moved to the global data cache. If this data is placed in the global data ram, the global cache could be used as buffer for the stack cache. The buffers on the stack cache will be obsolete. Since stack data just below the stack cache can reside in the global cache, the global cache must be searched on each miss (global data can become local if the TOS moves down). All misses to the stack cache can be handled by the global data cache; there are no buffers in the stack cache; some of the extra features can not be used: the stack cache could become quite simple. If buffers are present, more data can be stored in the stack cache, and since data from only one address can reside on each place in the read buffer, this buffer can be a linear buffer. However the communications between the parts of the processor become more complicated. The global cache will contain stack data, which will not be flushed on task switches. This does not cause any complications since the global data cache is equipped with an algorithm to maintain coherence. Since stack data can reside in both data caches, coherence problems between both caches can occur. A write buffer in the stack cache can contain global data (replaced dirty data).

The buffer strategies used for the stack cache can now be used for the global data cache.

The definition no. 2 gave a small restriction: data below a certain offset from the TOS is no longer local. To implement such an algorithm an extra check is required for each reference. A cache based on this algorithm can use all the features mentioned in this chapter. The communications between the stack cache and the global data cache will be complicated by the fact that stack data can reside in both the global data cache and the stack cache.

The definition of local data is still difficult. However, since no decision has been made about a global data cache, definition number 2, the most wide one, seems the best solution. It has the best performance when no global data cache is used, and communications with the global data cache are most simple when such a cache is present. If the stack references posses a high locality to the most recently used stack frames, most misses will be global references. With a less local reference pattern, one of the other definitions can give a better performance if a global data cache is present. The locality of stack data is highly compiler dependent (what should be placed on the stack?). In definition 1 and 3 (stack data in the global cache), stack data has to compete with other global data for a place in the global data cache. If most global data (in either of these 2 definitions) is non stack data (which can be caused by a high hit ratio in the stack cache), the performance of a stack cache based on these definitions will not be (much) higher than for a cache based on definition number 2. Simulations are needed to decide which definition of local data gives the best performance. Definition number 2, all stack data is local, could be used as a default.

4.8 The performance of the cache

The size of the cache has to ensure a stack cache hit ratio comparable to the instruction cache. The expected hit ratios can only come from simulations. The optimal block size

and the transfer block size (resp. 32 quads an 8 quads in the instruction cache) will be smaller, since data can be written to the cache. The speed on a write can be critical for the performance of the cache. The introduction of pseudo writes (section 4.2) can eliminate this bottleneck. Splitting the dirty bit from the rest of the tag will reduce the number of ports to this part of the tag (the dirty bit is not needed for read requests, so no port needed for read requests) and thus speed up the cache response on writes. The read and write buffers will reduce the miss penalty, while intelligent buffers will use the buffer space to reduce the miss ratio. As stated before, the cut back K algorithm can perform quite well when the task switching frequency is low. The hybrid algorithm can perform even better and will certainly outperform the cut back K algorithm as the task switching frequency increases. A direct mapped cache can be used when the task switching frequency is extremely high.
However, the more the hit ratio is increased by using more complicated algorithms, extra buffers, more complicated write strategies etc, the more space and is used by control logic. One has to keep in mind that this space could also be used to store more data. It is difficult to translate the complexity of the control block to an amount of silicon, but extra complexity must give an obvious increase in performance (in simulations) before the extra cache features are actually implemented.

Chapter 5 Cache simulations

In this chapter an answer is given to the questions Why, and How the cache is simulated; what has to be simulated; an outline of a simulator is given; what data is required to simulate the cache and the various ways to generate traces are discussed.

5.1 Why does a cache have to be simulated?

A cache will get a number of data requests from the processor or execution unit, which it has to service in the shortest time possible. These data requests follow a certain reference pattern, which can be used to adapt the various cache parameters in such a way that the cache performs best. An optimal way to test the behaviour of a cache is to observe a hardware cache in a hardware environment. However, most parameters are fixed in hardware, so a change in the parameters would require the manufacturing of a new cache in hardware. This approach is too expensive. Another approach is to model the dependency of the cache characteristics on the cache parameters, and the characteristics of the reference patterns, in a mathematical model of equations. Here the validation of the model is the gravest problem. It appears that the characteristics of programs are not similar, and that the effects on cache behaviour are serious [Aga89(DDV89AGA)].

An intermediate approach is to model the cache in a computer program and to evaluate the cache behaviour by feeding this simulator with a series of references. These references, or traces, can be obtained from an existing processor by using a hardware monitor. Another way to obtain traces is to run a workload on a software model of the processor. A last way to get traces is to manufacture them artificially by using trace characteristics, but in this method the trace characterisation has to be validated again.

The first method is mostly used when a cache for an existing processor is designed. The third method gives the least reliable results and can be used for a rough estimate of the cache behaviour, and will be used when the second method is not available. Traces from software are widely used (the Architect's Workbench [Mit88(P61)]) because it gives accurate results, and can be used when a compiler and/or assembler is available for the C-processor.

5.2 What has to be simulated?

The behaviour of the cache has to be caught in some numbers. Two parameters which are widely used for cache characterisation are the miss ratio and the traffic ratio. These parameters describe the effectiveness of the cache and the use of the bus between the cache and main memory. These parameters do not use any timing information. This information can be used to calculate a service time. The service time is the number of cycles a computer (built with the execution unit, the cache and some main memory) needs to run a certain trace.

With these parameters the performance of the cache can be measured. The next decision to be made is which cache parameters will be included in the model. Partly this is dictated by the cache models of chapter 4 and the general parameters discussed in chapter 1. Included are :

- architectural parameters like
    - cache organisation (linear/direct mapped)
    - replacement policy
    - size characteristics of the buffers
- general parameters like
    - cache size
    - block size and transfer block size
    - number of ports

- number of cache (processor) cycles per main memory access (latency and access time.

The simulator uses these inputs and the information from the trace to produce the hit and traffic ratio and the service time. Some extra information can be calculated to measure the performance of the replacement strategy, the buffers, etc. This leads to improvements in the design of the cache.

5.3 Traces for the stack cache

The traces, which are used to simulate caches, need to store all information necessary to get useful characteristics from the simulations. The traces will contain the minimal amount of data required to get useful results, since traces tend to be long. Especially if the effects of multi-tasking are to be included, the traces can require a lot of storage. A minimal trace for a simple cache is a list of successive addresses of data requests. The timing in this kind of traces is minimal, only the order of the references is recorded.
The stack cache requires more detailed timing information. Other information that is required, regards the TOS movement, read/write information, the port issuing the request and information on task switches.

The timing information for the stack cache has to be more detailed, to be able to give estimates of the service time and the miss penalty. The stack cache uses buffers, which can be filled. The replacement algorithm needs time to prefetch blocks. The amount of time passed between the prefetch of a block and a request of data in the block can determine whether the request results in a hit or a miss.
The timing information can be stored in two ways. First, one can store for each request the amount of time (clock cycles) passed since the previous request was serviced. This method is useful when more than one cycle elapses before a new request is issued. Secondly, all clock cycles can be recorded. A flag is used to indicate that no requests are

issued on a clock cycle. This approach requires less storage when one (or more) requests are issued on almost every clock cycle.
The C-processor will be a high performance processor, equipped with a few ports between the execution unit and the stack cache. This makes it very likely that at least one request is issued on each clock cycle. Therefore the second approach of recording traces will result in the least overhead in traces for stack cache simulation.

The movement of the TOS is used by several replacement strategies to optimise the usage of cache memory. Information regarding these TOS movements should therefore be included in the traces. The information is only needed when the stack moves. The current TOS will be stored in the simulator.

The execution unit can use several ports to read from and write to the cache. The read ports and write ports are separated. The ports will have a fixed priority. All requests can be serviced in parallel, as long as they result in hits. When there is one miss, all hits will be handled, the data from the miss will be read or written later. When there are more misses at the same time, the priority determines which miss will be serviced first. Since there is only one data port to main memory, only one miss can be handled at a time. Priority and read/write information can be stored by appointing a port to a certain request. The priority of the ports will be fixed in the simulator, so this information must be available to the manufacturer of the traces.
The number of ports used in the trace will have to fit the number of ports used in the cache. This can be done by using an optimising compiler for each of the wanted port configurations. The trace then can be generated by using a tool like the Architect's Workbench [Mit88(P61)]. As long as the exact number of ports between the stack cache and the execution unit is not determined, one trace can be used to store the information for several port configurations. This will reduce the number of traces to be stored. A possible way to achieve this is given in appendix A.

The behaviour of the stack cache on task switches determines the type of stack cache to be used. The information about task switches has to be included in the traces. Task switches on the C-processor are indicated by a change of PIN. As task switches are even more rare than changes of the TOS, the PIN's only have to be recorded on task switches.

A suggestion for a trace organisation in which all the above characteristics are implemented, is given in appendix A. This format is supported by the simulator, discussed in the next section.

The traces can be compacted if basic blocks are used [Mit88(P61)]. The trace then consists of basic block descriptions and a file in which the order of the basic blocks is stored (order file). The description of the blocks can use the format of appendix A. The stack cache is used to store local variables. The addresses of these variables on the stack can vary for various procedure calls. If all addresses and movements of the TOS are recorded relative to the current TOS, the same basic block can be used each time a procedure is called. Problems can arise when a reference is made to some data outside the stack frame of the procedure. This address can differ for various procedure calls, even relative to the TOS. A possible solution could be to split the basic block in two parts, and store the references which give problems in the order file. It is also possible to ignore the problem and only put one address in the basic block. This will reduce the accuracy of the simulation results.

The traces will have to match the actual reference pattern of the C-processor as close as possible to get valid cache characteristics. With traces that match the reference pattern less it is possible to gain insight in the effects of the cache parameters and the replacement strategies. For this purpose traces generated by the workbench could be used. To fine tune the stack cache to the C-processor, however, realistic traces are still necessary. The number of these traces and the number of simulations with these traces can be limited if other traces are used first. The final simulations with realistic traces can be used to examine the need of a global data cache and the matching of the traffic on the instruction and the data bus.

5.4 The simulator

The simulator consists of three parts. In the first part the input parameters are read, the second part reads the trace and does the actual simulation, and the last part generates the statistics. If basic blocks are used, the basic block file is read in the first section of the program, while the file containing the order of the blocks is read in the second part.

In the first part the cache configuration is read. This can be from a file in batch mode, or interactive. All cache options as discussed in the previous chapters can be included. These options are:

- architecture (Linear/Direct mapped)
- replacement strategy (Cut back K/Hybrid)
- the number of blocks fetched on a miss (K of the cut back K algorithm)
- the dirty offset (distance between dirty pointer and TOS when blocks are written back in the hybrid algorithm, < cache size)
- the prefetch offset (distance between base and TOS at which blocks are prefetched in the hybrid algorithm, < cache size)
- cache size (in quads, power of 2)
- block size (1..cache size, power of 2)
- the use of intelligent buffers (y/n)
- the use of pseudo reads (y/n)
- transfer block size (1..block size, power of 2)
- read buffer size (quads)
- write buffer size (quads)
- the number of read ports and the number of write ports
- main memory latency and access time (cycles)

The number of read and write ports can also be passed in the header of the trace file. The number of ports in the trace has to match the number of ports used in the simulator.

The simulation part will first translate the data from the trace to the data types used in the simulator.

The second function of the simulation part is the simulation itself. The simulator has to act as the control block of the cache (the control blocks are given in pseudo pascal in appendix B). It has to keep track of the number of hits and misses and has to keep track of data transfers to, from and within the cache. Finally it has to keep track of the number of elapsed processor cycles. To be able to operate as a control block, the program will need to know which data resides in the cache. This requires a data structure which resembles the tag block in the cache. The data itself is not important and will not be recorded.

The simulator stores the following information, which is used by the third part of the simulator to produce the cache statistics.

1) the number of read and write requests
2) the number of read and write misses on the ram and the buffers
3) the number of elapsed clock cycles
4) the number of bytes transferred to and from the execution unit and main memory
5) the number of times and clock cycles the buffers are full
6) The number of conflicts between the demand and the prefetch part of the algorithm

On each clock cycle, the simulator first determines the status of the references. If all references are hits, only the statistics above have to be updated. If a miss, a task switch or a TOS movement occurs, a procedure has to be called to implement the replacement algorithm. This happens after the hits have been recorded. This procedure will also record the times at which blocks will arrive in the buffers (if a block is (pre)fetched) and when transfer blocks are written back. These times will have to be checked each clock cycle, to see if some data has arrived. The recording of these times can be used to make the simulator event driven. The procedure will have to check too when conflicts between the prefetch and the demand algorithm arise.

The third part of the simulator calculates the statistics like hit ratio, transfer ratio, execution time, etc. from the raw material gathered in the second part.

Chapter 6 Conclusions recommendations

6.1 Conclusions

The C-processor makes use of a stack cache to store local variables and pass parameters between procedures and functions. In this report two possible architectures for such a cache are discussed. Each of these architectures can use dedicated replacement strategies and buffers to speed up cache misses. The linear cache with the cut back K replacement algorithm is the most simple option. It uses the least control logic and produces the smallest amount of bus traffic. It will have a higher miss ratio than the linear cache with the hybrid algorithm and suffer at task switches when the cache has to be flushed.
The more complicated hybrid linear cache will have a smaller penalty at task switches since more of the data will be written back already at the moment a task switch occurs. The usage of the bus will be less on task switches,but this algorithm will cost a higher usage of the bus when no task switches occur.
A direct mapped cache can be used when the frequency of task switches becomes extremely high. This cache type does not have to be flushed on a task switch. It requires a complete address in the tag for all data in the cache, while the linear cache only needs two boundary registers. The same replacement policies can be used as for the linear cache but have to be adapted.

All cache types can profit from read and write buffers. These buffers between the cache and main memory can reduce the miss penalty by fetching and writing blocks of data in a background process. They require extra ports to the data ram and the tag to avoid interference with the normal data requests. The buffers can also be used as an extension of the cache when they are not completely filled with partly fetched blocks and data that is not yet written back. This can further reduce the miss ratio.

Writing data to the cache can cause problems for the speed of the cache. The data can only be written after the control block has established that the data hits in the cache (or not). The introduction of pseudo writes (i.e. provide the address as soon as the instruction is decoded and the data when available) can reduce the time required for a write to the cache.

Since more than one request can be issued in one clock cycle, all parts of the cache will be multi ported. They will even get some extra ports used by the control logic to implement the replacement algorithm.

The performance of the caches has to be measured. A simulator has to be made, which implements the suggested stack caches. The format for trace data has been defined. The simulator has to use realistic traces to give a realistic estimation of the cache performance. These traces are not yet available. At this moment work is done on the architects workbench to provide these traces. To compare the behaviour on task switches, these should be included too. Besides the normal performance parameters for caches (i.e. hit ratio, traffic ratio and service time) the performance analysis should include both cache timing (the cache should answer data requests within one clock cycle) and complexity (= mm^2 silicon) too. Silicon used for control logic for a complicated cache could be used to store more data in a simple cache. An increase of the data ram will double the size (power of 2). The extra space can also be used to enlarge the buffers or the global data cache.

6.2 Recommendations

First of all, the simulator has to be created and realistic traces have to be produced. These should include task switches and timing information. The timing information consists of the time between the servicing of one request and the arrival of the next request. When a simulator is available, the cache architectures can be simulated and compared.

If the simulations are completed, the cache can be modeled on a lower hierarchical level. This could end in the construction of a stack cache generator, which could tailor caches to processors.

The simulation results can also show the need of a global data cache. A global cache is suggested when the number of stack cache misses is smaller than the number of global references. In the global cache the coherence problem has to be solved. In this report an inventory of coherence algorithms and a general outline of the global data cache have been given.

Literature

Besides the references used in this thesis, a lot more was found by Jos Bormans (especially about instruction caches) and me (data caches). The whole list is available at the Eindhoven University of Technology. The numbers between brackets are used for inventarisation; B12 indicates article no. 12 in the articles gathered by Jos Bormans.

Aga88(B41)
Agarwal, A, R. Simoni, et al.
An evaluation of directory schemes for cache coherence.
15th annual international symposium on computer architecture 1988. p. 280

Arc86(P59)
Archibald, James & Jean-Loup Baer
Cache coherence protocols: evaluation using a multiprocessor simulation model.
ACM Transactions on computer systems, Vol. 4, No. 4, p.273

Bor89
Bormans, Jos
Instruction cache for the C-processor
Master thesis, Eindhoven University of Technology, Department of Electrical Engineering, 1989

Bud 88
Budzelaar, Frank P.M.
The structured design of a processor for the language C.
Master thesis, Eindhoven University of Technology, Department of Electrical Engineering, 1988

Dit87(P60)
Ditzel, David R, Hubert R. McLellan & Alan D. Berenbaum
Design trade-offs to support the C programming language in the CRISP microprocessor.
ACM Computer architecture News, Vol. 15, No. 5, p. 158
Proceedings of the symposium on architectural support for programming languages and operating systems ASPLOS II, 1987

Fra84(P37)
Frank, Steven J.
Tightly coupled multiprocessor system speeds memory-access times.
Electronics, Januari 12, 1984, p. 164

Fur88(A12)
Furht, Borivoje
A RISC architecture with two-size, overlapping register windows.
IEEE Micro, April 1988, p. 67

Has85(B30)
Hasegawa, M. & Y. Shigei
High-speed top-of-stack scheme for VLSI processor: A management algorithm and its analysis.
12th annual international symposium on computer architecture 1985. p. 48

Hu89
Hu, Y.C.
An instruction cache in IDaSS
Master thesis, Eindhoven University of Technology, Department of Electrical Engineering, 1989

Kar88(P57)
Karlin, Anna R, Mark S. Manasse, Larry Rudolph & Daniel D. Sleator
Competitive snoopy caching.
Algorithmica, Vol. 3, p.79

Kat85(P38)
Katz, R.H, S.J. Eggers, D.A. Wood, C.L. Perkins, R.G. Sheldon
Implementing a cache consistency protocol.
12th annual international symposium on computer architecture 1985. p. 276

Laz89(A13)
Lazzerini, Beatrice
The RISC approach
IEEE Micro, February 1989, p. 57

McC85(P52)
McCreight, Edward M.
The dragon computer system
Microarchitecture of VLSI computers, ed. P. Antognetti et al, NATO ASI series E 96, 1985, Martinus Nijhoff, Dordrecht

Mit88(P61)
Mitchell, Chad L. & Micheal J. Flynn
A Workbench for computer architects.
IEEE Design and test of computers, February 1988, p. 19

Pap84(P39)
Papamarcos, Mark S. & Janak H. Patel
A low-overhead coherence solution for multiprocessors with private cache memories.
11th annual international symposium on computer architecture 1984. p. 348

Smi82(B24)
Smith, Alan Jay
Cache memories
Computing surveys, Vol. 14, No.3, p. 473

Smi87(P17)
Smith, Alan Jay
Line (block) size choise for CPU cache memories
IEEE Transactions on computers, Vol. C-36, No. 9, p.1063

Sta87(B36)
Stanley, Timothy J. & Robert G. Wedig
A performance analysis of automatically managed top of stack buffers.
14th annual international symposium on computer architecture 1987. p. 272

Sta88(P50)
Stallings, William
Reduced instruction set computer architecture.
Proceedings of the IEEE, Vol. 76, No. 1, p.38

Swe88(P12)
Sweazey, Paul
VLSI support for copyback caching protocols on Futurebus.
1988 IEEE international conference on computer design. p. 240

Tha88(P58)

Thacker, Charles P, Lawrence C. Steward & Edwin H. Satterthwaite jr.
Firefly: A multiprocessor workstation.
IEEE Transactions on computers, Vol. 37, No. 8, p. 909

Wit88

Withagen, Willem Jan
Definition and high level descriptioin of the C-processor
Master thesis, Eindhoven University of Technology, Department of Electrical Engineering, 1988

Yan89(P32)

Yang, Qing, Laxmi N. Bhuyan & Bao-Chyn Liu
Analysis and comparison of cache coherence protocols for a packet-switched multiprocessor.
IEEE Transactions on computers, Vol. 38, No. 8, p. 143

Appendix A A trace format

Each trace will consist of a header and a trace body. The header contains information regarding the cache configuration(s) for which the cache is intended and some trace properties. The body of the trace contains the actual trace data.
The header (table A.1) consists of a main_configuration, optionally followed by one or more alternative_configurations. Each configuration consists of 2 numbers; the number of read ports (no_read_ports) and the number of write ports (no_write_ports). Traces are most useful if the number of ports in the main configuration is a multiple of the number of ports in the alternative configurations.

Table A.1 The trace format; the header

```
trace = header body
header = main_configuration {alternative_configuration} trace_properties "."
main_configuration = configuration
alternative_configuration = configuration
configuration = no_read_ports "," no_write_ports ";"
no_read_ports = integer
no_write_ports = integer
trace_properties = trace_size pseudo_write?
trace_size = integer
pseudo_read? = "y" | "n"
```

The number of ports are separated with a comma, the alternative configurations with a semicolon. The trace properties now include the size of the trace (trace_size) and the inclusion of information about pseudo writes (see chapter 4). The header is finished with a dot.

There are two ways to put timing information in the trace. First one simply puts in all information for each clock cycle. A second option is to record only the clock cycles on which requests are given, and pass the timing information in a parameter which tells how many clock cycles have passed since the last request was serviced. If the simulated processor is pipelined and uses more ports, there are few clock cycles on which no requests are issued. So the first approach is chosen. The body (table A.2) of the trace has to store all timing information and the information about the ports that can be used in parallel for the various configurations. This information will be organised as follows.
The trace body consists of a number of clock cycles. A clock_cycle can contain one of three possible reference types: a task_switch, a change of the TOS or a list of references. The type is determined in the status field.
Since the PIN only changes at task switches, this part of the address is given only when a task switch occurs. The PIN will be a 16 bit word. The same is true for a change of the TOS. The TOS will not change very often, thus only needs to be provided when necessary. The TOS is a normal address, thus will be a 32 bit quad.
Each reference_list consists of a list of addresses issued at the ports of the cache (address_list), optionally followed by some alternative lists (alternative_list).
In the address_list the addresses at the ports are given. Since some ports can issue no requests, information about empty ports (no request) is given in a p-bit present_list (p = total number of ports). The bit sequence 110110 indicates that ports 1 and 4 of the 6 ported cache issue no request. The present_list can be proceeded by zero's to make empty fit in an integer number of bytes (to ease reading by the simulator). The empty byte will now become 00110110. The addresses for the read ports are given first, followed by the addresses for the write ports. The addresses will be 32 bit quads. To accommodate pseudo writes (chapter 4), the addresses for the write ports can be followed by the number of clock cycles between the time the address is issued and the time the data is available to the cache (delay). The delay will be placed in an extra byte.
The order of the addresses determines the priority: the first read address has the highest priority of the read ports. The priority of the read ports in respect to the write ports is dependent on the cache organisation as defined in the simulator. This priority is partly

dependent the free space available in the buffers and the replacement strategy (see chapter 4).

The information for the alternative configurations follows the information for the main configuration. Since these configurations contain fewer ports, it takes more clock cycles to handle them. These cycles are placed in the alternative_list. In each of these alternative clock cycles (alt_clock_cycle) the addresses issued on the ports are passed. The addresses are passed by the port number (port_no) of the address in the address_list. The ports numbering starts with the read ports, followed by the write ports. A port_no of zero indicates an empty port. Each port number is stored in a byte. A byte filled with "1"'s as the first port number in an alt_clock_cycle indicates the end of an alternative_list.

Table A.2 The trace format; the body

body = clock_cycle {clock_cycle}
clock_cycle = status TOS | PIN | reference_list
status = byte
TOS = quad
PIN = word
reference_list = address_list {alternative_list}
address_list = empty read_address {read_address} write_address {write_address}
empty = {"0"} present_list
present_list = bit {bit}
read_address = address
write_address = address delay
address = quad
delay = byte
alternative_list = alt_clock_cycle {alt_clock_cycle} "11111111"
alt_clock_cycle = port_no {port_no}
port_no = byte

Appendix B Control Block

In this appendix the replacement algorithms for the control block are described in a pseudo pascal. All actions in a stage can take place in parallel.

```
Program control_block

BEGIN
 WITH each_reference DO
 BEGIN
        {stage one, check TAG information }

        hit_info := miss
        IF ram_pin = request_pin AND ram_address = request_address AND ram_valid
               THEN hit_info := hit;
        WITH each_buffer DO
               IF buffer_pin = request_pin AND buffer_address = request_address
                      AND buffer_valid THEN hit_info := hit;
        IF ram_pin <> request_pin THEN hit_info := task_switch
        {end of stage one}

        {stage two, on a miss various routines are called. these routines vary for the different
        algorithms and will be specified later}

        IF hit_info = hit THEN
        BEGIN {hit}
         IF read_request THEN get_data_from_hit_place
         IF write_request THEN
         BEGIN
               write_data_to_hit_place
               mark_tag_dirty
               IF hybrid_replacement THEN update_dirty_pointer
         END
        END {hit}

        IF hit_info = miss THEN
        BEGIN {miss}
         IF cut_back_K then cut_back_K_miss
         IF hybrid_replacement THEN hybrid_miss
         IF direct_mapped THEN direct_miss
        END {miss}

        IF hit_info = task_switch THEN
        BEGIN {task switch}
         IF direct_mapped THEN direct_miss
         IF linear THEN
         BEGIN {linear}
               move_dirty_blocks_to_main_memory
               {end of stage two, begin of stage three}
               IF all_blocks_in_buffer THEN invalidate_all_ram_tags
               put_new_pin_in_tag
               {end of stage three, begin of stage four}
               IF cut_back_K THEN cut_back_K_miss
               IF hybrid_replacement THEN hybrid_miss
               {end of stage four}
         END {linear}
        END {task switch}
 END {all references, computer dead}
END.
```

The various procedures are:

```
PROCEDURE cut_back_K_miss

BEGIN
  {begin stage 2}
  IF no_empty_space_in_ram THEN
  BEGIN
          IF read_request THEN read_without_replace
          IF write_write_request THEN write_to_buffer_without_replace
  END {no_empty_space_in_ram}
  {begin stage 2a}
  IF read_buffer = full THEN wait_until_space_free
  issue_fetch
  {begin stage 2b}
  IF write_request THEN
  BEGIN
          write_data_to_read_buffer
          mark_tag_quad_dirty
  END {write_request}
  {end of stage 2, begin of stage 3}

  reserve_space_for_requested_block_in_buffer
  {end of stage 3, begin of stage 4}

  IF write_request THEN wait_for_block_to_arrive
  IF read_request THEN
  BEGIN
          {begin of stage 4a}
          wait_for_required_quad
          {begin of stage 4b}
          provide_data
          wait_for_rest_block
  END {read_request}
  {end of stage 4, begin of stage 5}

  move_block_from_buffer_to_ram
  update_base
  invalidate_blocks_between_old_base_and_new_base
  {end of stage 5, begin of stage 6}

  get_blocks_between_old_base_and_new_base
  updat_tag
  {end of stage 6}
END {cut_back_K_miss}


PROCEDURE hybrid_miss

BEGIN
  cut_back_K_miss {the first 6 stages are the same}
  {end of stage 6, begin of stage 7}
  IF write_request THEN update_dirty_pointer
  update_prefetch_pointer
  {end of stage 7}
END {hybrid_miss}
```

```
PROCEDURE direct_miss

BEGIN
  {begin stage 2}
  IF no_empty_space_in_ram THEN
  BEGIN
        IF read_request THEN read_without_replace
        IF write_write_request THEN write_to_buffer_without_replace
  END {no_empty_space_in_ram}
  {begin stage 2a}
  IF block_to_replace_not_dirty THEN wait_until_space_free
  IF block_to_replace_dirty THEN wait_until_space_in_both_buffers
  issue_fetch
  {begin stage 2b}
  IF write_request THEN
  BEGIN
        write_data_to_read_buffer
        mark_tag_quad_dirty
  END {write_request}
  {end of stage 2, begin of stage 3}

  reserve_space_for_requested_block_in_buffer
  {end of stage 3, begin of stage 4}

  IF write_request THEN wait_for_block_to_arrive
  IF read_request THEN
  BEGIN
        {begin of stage 4a}
        wait_for_required_quad
        {begin of stage 4b}
        provide_data
        wait_for_rest_block
  END {read_request}
  {end of stage 4, begin of stage 5}

  move_block_from_buffer_to_ram
  update_base
  invalidate_blocks_between_old_base_and_new_base
  {end of stage 5, begin of stage 6}

  get_blocks_between_old_base_and_new_base
  updat_tag
  {end of stage 6}
END {direct_miss}
```

The control block also has to work on movement of the TOS. This requires some special procedures.

```
PROCEDURE tos_down_cut_back_K

BEGIN
  {stage 1}
  invalidate_ram_between_old_TOS_and_new_TOS
  {can be done by setting the invalid bit}
  {end of stage 1, begin of stage 2}

  IF new_TOS < base THEN
  BEGIN
```

```
          fetch_TOS_block
          update_base
  END {TOS below base}
  {end stage 2}
END {tos_down_cut_back_K}


PROCEDURE tos_up_cut_back_K

BEGIN
  {stage 1}
  IF new_TOS >= (base + cache_size) THEN  {new TOS will overwrite part of data near base}
  BEGIN
          write_back_dirty_blocks
          {end of stage 1, begin of stage 2}

          clear_blocks  {can be done by setting the clear bit}
          update_base
  END {overwrite base}
  {end of stage 2, begin of stage 3}

  update_TOS_in_cache
END {tos_up_cut_back_K}


PROCEDURE tos_down_hybrid

BEGIN
  {stage 1}
  invalidate_ram_between_old_TOS_and_new_TOS
  {can be done by setting the invalid bit}
  {end of stage 1, begin of stage 2}

  IF new_TOS < base THEN
  BEGIN
          fetch_TOS_block
          update_base
  END {TOS below base}
  {end stage 2, begin of stage 3}
  update_dirty_pointer
  update_prefetch_pointer
  {end of stage 3}
END {tos_down_hybrid}


PROCEDURE tos_up_hybrid

BEGIN
  {stage 1}
  IF new_TOS >= (base + cache_size) THEN  {new TOS will overwrite part of data near base}
  BEGIN
          write_back_dirty_blocks
          {end of stage 1, begin of stage 2}

          clear_blocks  {can be done by setting the clear bit}
          update_base
  END {overwrite base}
  {end of stage 2, begin of stage 3}
  update_TOS_in_cache
  update_prefetch_pointer
  IF new_TOS > (dirty_pointer + cache_size) THEN update_dirty_pointer
END {tos_up_hybrid}


PROCEDURE tos_down_direct_mapped

BEGIN
  {stage 1}
  FOR all_blocks_between_old_TOS_and_new_TOS do
          IF request_PIN = ram_pin THEN invalidate_ram
  {can be done by setting the invalid bit}
  {end of stage 1, begin of stage 2}

  IF new_TOS < base THEN
  BEGIN
          fetch_TOS_block
          update_base
  END {TOS below base}
  {end stage 2}
END {tos_down_cut_back_K}


PROCEDURE tos_up_cut_back_K

BEGIN
  {stage 1}
  IF new_TOS >= (base + cache_size) THEN  {new TOS will overwrite part of data near base}
  BEGIN
          FOR base TO (new_TOS - cache_size) do
            IF PIN_request = PIN_ram THEN
            BEGIN
                    write_back_dirty_block
                    {end of stage 1, begin of stage 2}

                    clear_blocks  {can be done by setting the clear bit}
            END {PINs match}
          update_base
  END {overwrite base}
  {end of stage 2, begin of stage 3}

  update_TOS_in_cache
END {tos_up_cut_back_K}
```