TECHNICAL REPORT

COMPUTER SCIENCE

DEVELOPING THE HATFIELD SUPERSCALAR ARCHITECTURE CACHE SIMULATOR

Report No 318

Daniel Tate

June 1998

Developing the Hatfield Superscalar Architecture Cache Simulator

Contents

1 Introduction

A great deal of the current research into computer architecture is directed at Multiple Instruction Issue (MII) processors. These processors have the ability to issue, process and retire more than one instruction per cycle. Multiple instructions can be simultaneously issued when there are no dependencies between them.

MII architectures can be split into two diverse types: VLIW and superscalar. These types are differentiated by the time at which the instructions are scheduled into groups that could be issued in parallel. A VLIW processor relies on the compiler to generate fixed sized groups of instructions, while a superscalar processor relies on the processor to dynamically generate groups of independent instructions.

Current work at the University of Hertfordshire is focused on developing a processor that combines the best features of both VLIW and superscalar processors. The Hatfield Superscalar Architecture (HSA) is a minimal superscalar that uses aggressive static scheduling. The static scheduling allows code motion with a global program view, while the minimal superscalar provides a variable issue rate without the complexities of dynamic scheduling. The current HSA software consists of two advanced instruction schedulers and an instruction level processor simulator. Each of these ongoing project activities is a large software development in its own right.

All of the software systems are highly parameterised and can be configured to emulate a large range of processor models. Probably the main limitation of the current HSA project is the assumption that the HSA simulator achieves a 100% cache hit rate. Therefore, a cache simulator has been developed and integrated into the current HSA simulator.

The cache simulator will maintain the ongoing HSA idiom of extensive parameterisation. All of the cache structure parameters are configurable at run-time, and can also be saved into retrievable set-up files. The cache structure will have zero or more levels of Instruction, Data or Unified cache; however, there must be one Main Memory level. Each level can be independently parameterised. The cache parameters dictate the size, associativity, number of sets, block size, pipelining, block replacement policy, latencies,

number of ports and buffer sizes. The only parameters to the Main Memory will be its latencies, number of ports and the size and ports of the Data Write Buffer.

The cache will therefore be an *n*-level multi-ported non-blocking cache structure. It will incorporate an optional Data Write Buffer (not Instruction caches) and an Outstanding References Buffer (not main memory) at each cache level. The former will take writes that occur in bursts and send them to the cache when required, while the latter keeps a track of all blocks currently being fetched into the cache, so that a block will only be fetched once.

This document is not intended to provide an introduction to caches. A full knowledge of the workings and trade-offs involved in caches is assumed. Throughout this document the idea of **higher** will be used to indicate cache levels towards the processor; i.e. the primary caches are higher than all other caches. The idea of **lower** will be used to indicate cache levels towards the main memory; i.e. the second level cache is a level lower than the primary cache.

Section 2 contains a detailed account of how the current HSA simulator processes a memory access. The process as a whole is introduced, and then the specific parts that are altered will be examined in depth. The current data structures, for data and instruction memory are presented because they will form the basis of the cache simulators main memory.

Section 3 introduces the external view of the cache simulator. The parameters that the simulator will need are introduced and briefly explained. The HSA simulator's new cache selection screen will also be introduced along with how the current menus will be amended. The block view of the different cache levels are given and briefly explained. The overall features are given, and then each of the four cases of top and lower level instruction and data caches are described individually.

Section 4 concerns the Data Write Buffer and Outstanding References Buffer. These are perhaps the most important features of the cache design. They are therefore presented in detail, with a clear description of both their attributes and actions for all possible inputs. The priorities are then discussed. This involves choosing which input line is selected to transmit data when more than one line is ready to send.

Sections 5 and 6 describe how the reading and writing to the cache will be simulated respectively. The handling of cache writes depends on the settings of the two parameters: *Allocate on Write Miss* and *Write Back.* To avoid unnecessary complexity, the operations involved for each of the four combinations of the parameters are described separately. Priority and timing issues are also addressed thoroughly.

Section 7 provides detailed information on the cache pre-fetch instruction. Information on this subject is spread throughout this document. This chapter is designed to bring all of the information together.

Finally, section 8 contains a summary and overview of the HSA cache simulator project.

2 Current HSA cache

2.1 Overview

The HSA simulator provides multiple functional units (FU's) that can be optionally pipelined. When an instruction is dispatched it occupies an FU, or a stage of the FU if it is pipelined, until the instruction has been completed. On completion, the result is forwarded to the relevant forwarding register file and the FU is freed for re-use.

The current HSA simulator memory model is in actuality either a super-fast main memory or a very large primary cache. It has the access time of a primary cache (1 or 2 cycles) and the hit rate of main memory (100%). There are three distinct cache structures, one for the instruction cache and two for the data cache. The two data cache structures contain the same information, but in a different form.

2.2 The Instruction Cache

When the HSA simulator is executed, the instruction cache is created with a pre-defined number of records. From the simulator's main menu, a test program can be loaded into this cache. An instruction cache record holds the label and memory address of the instruction and a pointer to the instruction record itself. The array index of the instruction cache is the same as the instruction record's memory address field.

```
struct icache_rec:
    instruction label
    memory address
    Instruction pointer

icache_rec icache[size]
```

Figure 2-1 Icache structure

2.3 The Data Caches

There are two structures containing the data cache information. One structure is a block of contiguous memory with the data residing at the appropriate address. The other structure is a set of data cache records similar to the instruction cache above. There are two main ways of addressing the data caches that are

```
struct dcache_rec:
    optional data label
    byte memory address
    integer value
    floating point value
    double fp value
    jump table value
    type of the data

dcache_rec *dcache[size]
```

Figure 2-2 Dcache structure

interchangeably used by the simulator namely *byte* and *word* orientated addresses.

Each data cache record contains an optional label for the data, its memory address in byte form, its value and its type. As the type of the data is not known in advance, the data record provides a separate field for each data type. Each data field, except for the one selected by the data type field, is set to NULL.

The linear form of the data cache is of no interest to this specification. Any further reference to a data cache will therefore refer to the record based structure.

2.4 Reading from the Instru ction Cache

An instruction read is issued at the end of every cycle. Instructions are read into the instruction buffer via a pipelined fetch unit, Figure 2-3. The maximum number of instructions moved each cycle is defined by the *Fetch Width*. The instructions are loaded from the cache into the first pipeline stage of the fetch unit during the first cycle. After the specified instruction read latency, they are transferred from the last pipeline stage of the fetch unit into the Instruction Buffer.

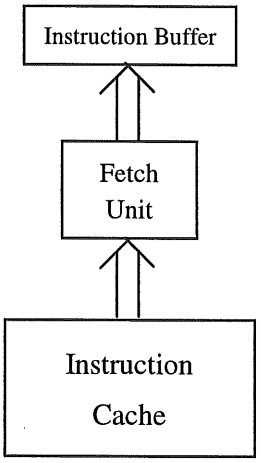

Figure 2-3 Instruction Fetch

At the end of each cycle, the instructions in the fetch unit are advanced one position through the pipeline, space permitting. Instructions in the last pipeline stage of the fetch unit, are transferred into the instruction buffer, again space permitting. Space is freed in the instruction buffer when instructions are 'squashed' or issued to FU's. The HSA simulator supports Boolean guards, if these guards are known to evaluate to false while the instruction is still in the Instruction Buffer, then the instruction is removed from the buffer, this is termed as squashing. Each cycle, all the valid instructions in the Instruction Buffer are compacted as far down the buffer as possible. The instruction fetch procedure is now complete.

2.5 Reading from the Data Cache

A data load is a program instruction and is decoded, issued and executed by the simulator in much the same way as any other program instruction.

When a data load instruction is encountered, it first obtains a load FU. The memory address is then compared with the address currently being accessed by the write FU's. If the content of the data address is being updated by a store FU, then the latest data is forwarded directly from the write FU. Otherwise the data is obtained from the perfect cache. Each cycle, the instruction progresses through the load pipeline, unless a stall is encountered.

When the read operation is completed, a result bus of the appropriate type is requested and the data is written to the relevant forwarding register file. If a result bus is unavailable, the pipeline is stalled. A stalled pipeline will halt the progression of instructions in subsequent pipeline stages, until an empty stage is encountered.

After successfully acquiring a result bus, the FU is made available to new instructions.

2.6 Writing to the Data Cache

A data store instruction is very similar to the above load instruction. When a store instruction is encountered it requests a store FU, which is assigned if available. Each cycle, the store instruction progresses through the pipeline until it has completed.

After the allotted write latency, the write is assumed to have successfully written to the perfect cache, and the FU is made available to new instructions.

3 Cache Set-up and Parameters

3.1 Requirements

A cache simulator integrated into the existing design of the HSA simulator.

An independent module, called only by designated external procedures.

A highly parameterised model that can be amended at run-time.

Robust and functional code so that the finished product is maintainable and upgradeable.

3.2 Program Parameters

The use of external global parameters is avoided unless absolutely necessary. The cache parameters are set through an initial menu structure and are held in a set of records global to just the cache module. To facilitate parameter capture, the following procedures are provided in the cache module:

- Display the current parameters
- Amend the current parameters
- Load/Save parameters to/from a file

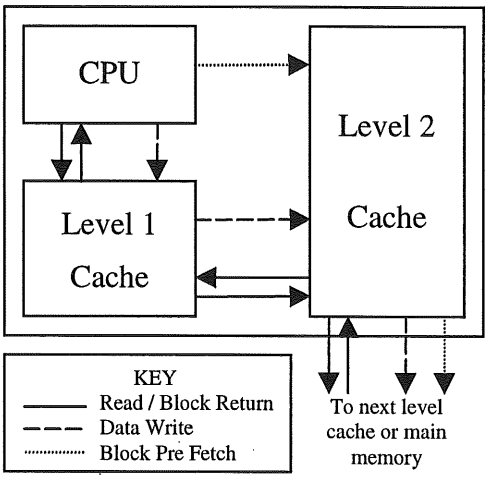

Figure 3-1 Overview of Cache Set-up

Initial cache parameters are loaded during initialisation; these parameters match the assumptions of the original perfect cache.

The original HSA simulator contained many cache specific references and global variables. Wherever possible, the global variables have been removed, and the function calls have been redirected to procedures within the cache module.

3.3 Cache Parameters

3.3.1 The 'global' parameters th at affect all caches and levels are:

Allocate on Write Miss - When a write miss occurs, the block is fetched from the lower levels, or the data just written to the next level down. If this value is set to true, the cache block would be loaded and a subsequent read would generate a 'hit'. However, the new block that is fetched may be useless, whereas the block that is purge may be in more frequent use.

Cache Write Back Policy - Can be either *write-back* or *write-through*. Write-back attempts to keep the lower levels as free as possible so the necessary accesses are not stalled. Write-through keeps all cache levels valid, so that all cache levels have the same value for the same data.

Pre-fetch Top Level - The initial level a pre-fetch instruction is sent to. If pre-fetch's are sent directly to the second level, then an entry is simultaneously made in the top level Outstanding References buffer. Pre-fetching has not yet been implemented.

Fetch From Just One Block - Fetches can ignore block boundaries, and make a greater number of possibly inefficient instruction fetches. Fetches can alternatively, only fetch up to the end of the block and therefore generate fewer actual fetches, but averaging a greater number of instructions per fetch, and also minimising port usage.

Return Requested Sub-block First - When a block is being fetched in more than one sub-block, the first sub-block to be returned may not be the one required, hence forcing additional delay waiting for the required sub-block. If this feature is enabled, the sub-block containing the requested data will be returned before all other sub-blocks.

Data Write Buffer Latency - The latency of the Data Write Buffer will depend on hardware implementation and its associated features, therefore a suitable value can be assigned. A zero delay entails a write being assigned to and checked by the Data Write Buffer, then arbitrating for a cache write port, all in one cycle.

Data Write Buffer Merge Level - Merging of records with different progression levels, or merges of writes to the same sub-block can occur. Records in the Data Write Buffer can be either waiting for a cache write port, currently accessing the cache, waiting to write to the next level or waiting for a block to be fetched. There are three types of merging available; there are the two extremes of all and none. The third type merges all records, except those with the cache currently accessed.

Data Write Buffer Merge Sub-blocks - The three above merge types can be used to merge either records accessing the same address; or record accessing the same sub-block.

3.3.2 The local parameters that affect only the cache to which it is assigned are:

Total Cache Size - The size (in bytes) of the current cache being defined.

Block Size - Size in bytes of the main cache building block.

Sub Block Size - Size of the data transmitted, a block may contain one or more sub-blocks.

Associativity - The number of possible spaces a cache contains for any given block.

Replacement Policy - On a block load, which block is replaced (Random, FIFO or LRU).

Pipelined - This is a Boolean denoting whether the cache is to be pipelined or not.

Cycles to Read Cache - The length of time to access cache. At end of this time either the data is ready on the return line, or a request to the next cache level is being initiated.

Cycles to Sequentially Read - The time taken to read the non-initial sub-block in a block (the first read normally takes more cycles to read than any subsequent sub-blocks).

Cycles to Write Cache - The number of cycles for the cache to be written to from its buffer.

Cycles to Bypass Level - Cycles to bypass the cache, and be ready on the data return line.

Cache Read Ports - The number of parallel reads allowed for the cache level.

Cache Write Ports - The number of parallel writes allowed for the cache level.

General Ports - Number of cache ports that can be either read or write at any given time. A cache has specific read and write ports OR general ports.

Return Buffers - The number of parallel data returns. This is either in total or at a pipeline stage depending on whether the cache level is pipelined.

Data Write Buffer Size - The number of record entries available in the Data Write Buffer.

Outstanding References Buffer Size - The number of record entries that are available in the Outstanding References Buffer.

Data Buffer Write Ports - The number of parallel writes allowed to the Data Write Buffer.

3.4 Integrating the Cache Simulator

The original ICACHE and DCACHE modules have been merged into a unified program called MEMORY that combines current functions. A new module called CACHE has been created to contain all of the cache accessing facilities and data structures presented in this document.

```
Display Main Memory Option

DCACHE valid address range is 0 to xxx
ICACHE valid address range is 0 to xxx

1.  View Data Memory from Base Address
2.  View Instruction Memory from Base Address
3.  Specify Start Address for Data Display
4.  Specify Start Address for Instruction Display
5.  Display Current Allocation for Data Memory
M.  Return to Main Menu
```

Figure 3-2 Combined Data and Memory options screen

There were originally two menus for the perfect cache system, one for the instruction cache, and one for the data cache. These menus have been combined into one menu, accessible from the main menu. The menu is called 'Display Main Memory Option', Figure 3-2.

A further menu has been created to provide all of the cache facilities. This menu is also accessible from the main menu, and is formatted in the same way as current HSA menu screens ('M' for previous menu). The new menu will be called 'Cache Set-up Option' (an example format can be seen to the right in Figure 3-3).

```
Cache Set-up Option

V.  View Current Cache Set-up
I.  Amend Instruction Cache(s)
D.  Amend Data Cache(s)
U.  Amend Unified Data and Instruction Cache(s)
Y.  Amend Main Memory
G.  Global Cache Parameters
L.  Load / Store Cache Structure
M.  Return to Main Menu
```

Figure 3-3 Cache Set-up options screen

Each of the Amend Cache options will lead to a standard Add, Update, Delete screen, with a list of currently defined caches of the appropriate cache type. The 'global cache parameters' screen facilitates the setting of the variables presented in 3.3.1. When a cache structure is loaded or saved, the global cache parameters are kept with the cache level information. The Load / Store option will lead to a list of standard and user defined cache

structure files which can be utilised. Each user-defined file has a 'title' that is displayed alongside the file number. The user sets the title when a cache structure is saved.

The options screens are defined in the OPTIONS module, but all of the procedure calls, including those to display the currently defined caches, are external procedures located in the CACHE module.

3.5 Cache Architecture

The basic design of the three types of cache can be seen in the following figures:

Primary Instruction Cache	*Figure 3-4*
Lower Level Instruction Cache	*Figure 3-5*
Primary Data Cache	*Figure 3-6*
Lower Level Data Cache	*Figure 3-7*
Unified Cache	*Figure 3-7*

There are diagrams of the primary instruction and data caches, as well as examples of the lower levels of both types. The primary caches include specific return buffers that would map directly onto the processor's components, such as the instruction buffer and register file.

3.5.1 Common Features

The dotted lines at the top and bottom of the figures indicate the limits of the cache level. The buffers at the bottom of the primary cache map onto the buffers at the top of the secondary cache.

There are two Multiplexers in most of the figures. These will be differentiated by their function. The Multiplexer to the left of the diagrams is involved in Block Returns and will therefore be called the *Return Multiplexer.* The Multiplexer to the right of the diagrams is involved with requesting blocks when a cache miss is encountered and will therefore be called the *Request Multiplexer.*

The method of fetching a block, when required by a cache miss involves both the Outstanding References Buffer and the Request Multiplexer. The Outstanding References Buffer is checked for an outstanding block fetch in parallel with the cache access. Therefore,

as soon as a cache miss is registered, a block fetch can be issued. Any block fetch must arbitrate for access to the next level via the Request Multiplexer.

The bus widths are equal to the sub-block sizes of each level. These sizes are frequently different. The cache level can be split into three sections depending on the bus widths. The read and pre-fetch lines just carry addresses and are therefore only an address word in width. The write lines and the return lines after (above, in the diagrams) the Return Multiplexer are the previous levels sub-block size in width, width '*p*'. The return lines before (below, in the diagrams) the Return Multiplexer are this levels sub-block size in width, width '*q*'.

When information is returned, the higher level only requires a sub-set. The Return Buffer holds the sub-block returned, and the portion(s) of the block that are required by the higher level are selected by the relevant Outstanding References Buffer record(s). The size of the sub-set is dependent on the relation ship between 'p' and 'q'. Width 'p' must always be less or equal to width 'q'. If there is a one to one relationship, then the Return Multiplexer is redundant. More formally, the following relationship must hold true:

$$q = x * p = y * \textit{instruction length}\ (x, y \bullet \exists : N)$$

3.5.2 Primary Instruction Cache

This is the most complex of the possible configurations due to the inclusion of the pre-fetch facility. Pre-fetches can be sent directly to the second level, assuming the primary misses.

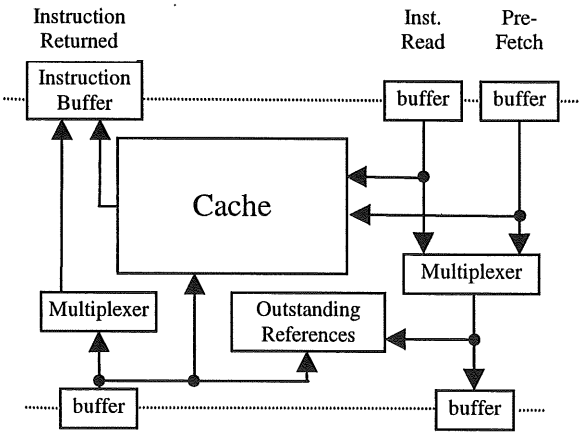

Figure 3-4 Top Level Instruction Cache

The return line from the Return Multiplexer to the Instruction Buffer has a width of 'p' bits; this correlates to the processors fetch width. The line to the Return Multiplexer from the lower level has a bus width of 'q' bits, which corresponds to the sub-block size of the primary cache.

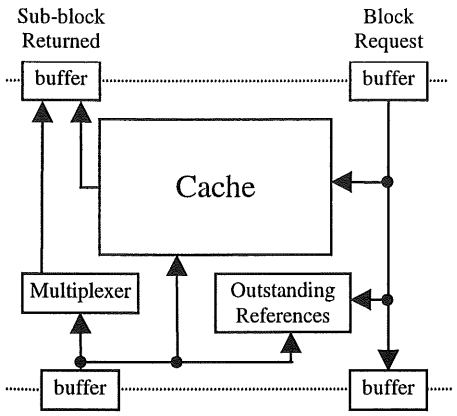

Figure 3-5 Lower Level Instruction Cache

3.5.3 Lower Level Instruction Cache

The lower levels of the instruction cache only have one request line. This line is activated by cache and Outstanding References Buffer misses from either the instruction read or pre-fetch lines (*see Figure 3-4*).

A Request Multiplexer is not present on the Block Request line. This is because there is only one line, therefore no contention.

If the bus widths of 'p' and 'q' are equal, then both the Outstanding References Buffer and the Request Multiplexer are not required.

3.5.4 Primary Data Cache

The primary data cache adds two further features, a Data Write line and a Data Write Buffer. For the functions of the pre-fetch line see section 7 on page 30.

During a data read, both the cache, the Data Write Buffer and the Outstanding References Buffer are checked for the data, or a block fetch, in parallel. Either the Data Write Buffer or the cache may return the data, with the Data Write Buffer having priority.

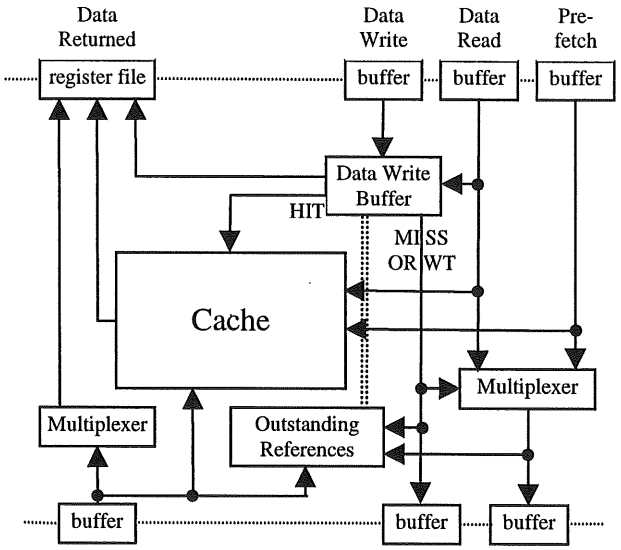

Figure 3-6 Top Level Data Cache

If a read miss occurs at both the Data Write Buffer and the cache, and if the result of the Outstanding References Buffer check is a miss, a block fetch is generated via the Request Multiplexer. There are three lines into the Request Multiplexer; arbitration is discussed in *section 4.4*.

A data write places data into the Data Write Buffer. A cache write port is accessed when available. If a cache miss occurs and *Allocate on Write Miss* is used, then a block fetch is initiated, otherwise the data is written to the next level.

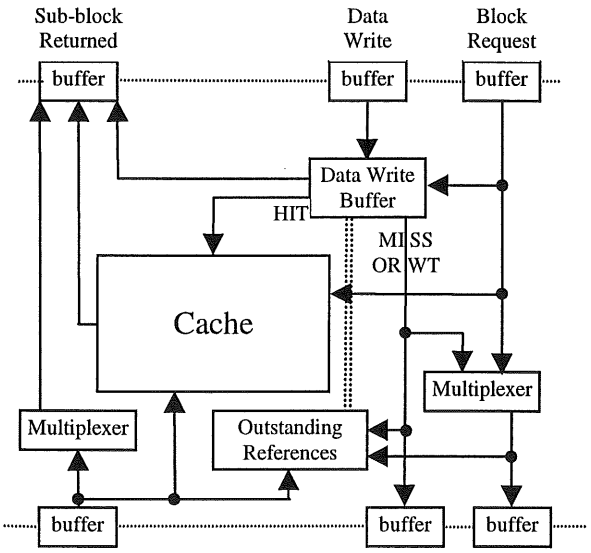

Figure 3-7 Unified OR Lower Level Data Cache

3.5.5 Unified and Lower Level Data Cache

The lower data cache levels are, as with the instruction cache, just a simplification of the primary cache. As the instruction cache is also just a simplification of the data cache, the unified cache is identical to the lower level data cache architecture.

If *Write Back* and *Allocate on Write Miss* are used, then the data write line will be a block write line, as the only time data will be written to a lower level cache is when a block update purges a dirty block.

The block request line can be activated by a read, pre-fetch, or a write (if the *Allocate on Write Miss* option is used).

4 The Cache Buffers

4.1 Overview

The Data Write Buffer and the Outstanding References Buffer contain small amounts of high-speed memory and control logic. There is a Data Write Buffer above each data cache, unified cache and main memory. There is an Outstanding References Buffer at each cache level. The Data Write Buffer receives writes from the level above; the Data Write Buffer above the primary cache receives data writes at the end of the ID pipeline stage. The Outstanding References Buffer sits between the cache and the next level down and determine which misses should generate block fetches from the next cache level.

The main idea behind the Data Write Buffer is to prevent unnecessary processor stalls from occurring when there are insufficient cache ports available to service all the pending requests for data reads and writes. The main idea behind the Outstanding References Buffer is to avoid unnecessary block requests from wasting the limited bandwidth of lower cache levels.

If the Data Write Buffer is not used, when data is written to a cache, it immediately has to request a write port. If all of the cache write ports are busy, then the processor is stalled until the write can be issued. This is an unnecessary stall as no instruction waits for the result of a write. In contrast, processor stalls caused by a stalled cache read are generally unavoidable in an in-order-issue processor. Using a Data Write Buffer, virtually no write stalls occur. Therefore at the end of the pipeline ID stage, the processor can usually forget about a write and assume it has completed.

4.2 The Data Write Buffer

4.2.1 Introduction

The Data Write Buffer is organised as a FIFO queue. It contains a list of records, each consisting of data, an address and a status flag. The data is a sub-block in size and can contain anything from a bit up to its capacity. The status flag indicates whether the write is waiting for a cache port, accessing the cache, stalled or waiting for a Data Write Buffer port from the level below. The size, number of ports and write latency of the Data Write Buffer are defined at run-time by the 'Cache Set-up' option from the main menu.

Data is written to memory sporadically. The primary data cache therefore needs multiple write ports to avoid unnecessarily stalling the processor. However, adding ports to a cache becomes exponentially more expensive and complex, and could even increase the access latency. Adding more write ports to a Data Write Buffer is much simpler and cheaper due to its smaller size. Therefore, all writes can be successfully accepted by the Data Write Buffer, before being sent on to the cache at a more uniform rate. Thus the cache need only support the mean writes per cycle write ports, instead of the maximum writes per cycle write ports.

4.2.2 Functionality

During the load latency of the Data Write Buffer, the new record is checked against the existing contents of the buffer and the contents of the Outstanding References Buffer. The Data Write Buffer is either not checked at all, checked for a record referencing the same address, or an address in the same sub-block depending on the value of the global *Data Write Buffer Merge Level.* The Outstanding References Buffer is checked for a block update.

If a match is found in the Data Write Buffer, the two records are merged with any new data taking precedence. If a matching block fetch is only found in the Outstanding References Buffer, then a new record is added to the Data Write Buffer with a status flag of 'blocked'. If no matches are found, a new record is added to the Data Write Buffer with a status flag of 'waiting for a cache port'. A new record can also request a cache write port in the same cycle, if it is at the front of the queue.

When a record reaches the front of the Data Write Buffer it must arbitrate for a cache write port (see *section 4.4.1*). When a cache write port has been successfully obtained then the cache access can begin. If a hit occurs, then after the write latency, the record can be removed from the buffer. If *Write Through* is used then the data is also sent to the next level. However, if a miss occurs, using *Allocate on Write Miss*, the record remains in the Data Write Buffer with its status flag set to 'blocked', and a block fetch is initiated. When the relevant block has been loaded into the cache, all Data Write Buffer records referencing the block have their status flag reset to 'waiting for a cache port', and can then request a cache write port at the end of the cycle. Without *Allocate on Write Miss*, the data is simply sent to the next level and the record is removed from the Data Write Buffer.

A record will be stalled if data is sent to:

- A Data Write Buffer with no free ports.
- A Data Write Buffer which is full.
- A cache with no free ports
- An Outstanding References Buffer that is full.
- Either of the buffers to the next level when they are busy.

If the first two cases were at the primary cache, then the processor would be stalled.

4.3 The Outstanding References Buffer

4.3.1 Introduction

The Outstanding References Buffer is a fully associative array of records. Each record contains an address and the destination register. A record is loaded into the Outstanding References Buffer if a read, write (with *Allocate on Write Miss*) or a pre-fetch misses in the cache, or if a pre-fetch instruction is sent directly to the second level. When a write misses and requests a block fetch, it is added to the Outstanding References Buffer as a pre-fetch.

The Outstanding References Buffer can be accessed by 1 (*see Figure 3-5*) to 3 (*see Figure 3-6*) block request lines and the block update line simultaneously. The most important of these lines is the block update (bottom left line in the diagrams), as this line reduces the

entries in the buffer by at least one at each access. Outstanding References Buffer priority issues are discussed in *section 4.4.3*.

4.3.2 Functionality

When the Outstanding References Buffer is scanned during a cache read or write; a match indicates that the associated cache block is already being retrieved from a lower level. This avoids duplicate requests.

When the Outstanding References Buffer receives a new record it is stored in any available slot. For each memory address, there will be at most one record as a result of a write record, and possibly multiple read records. All the reads would pre-date the write, as any subsequent read would see the write, blocked in the Data Write Buffer, as a cache hit. Thus, as soon as a write record for an address has been added to the buffer, no more records for that address will be added.

When the Outstanding References Buffer receives a block return, all of the records in the buffer are associatively searched using the block address. Any pre-fetch records found are removed from the buffer, its job complete. At the primary cache, all read records move data/instructions into the Register File/Instruction Buffer. At all other levels, block requests generate block returns to the next level up.

A block request will be stalled if data is sent to an Outstanding References Buffer with no space free.

4.4 Cache Read and Write Port Priorities

There are multiple lines going in and out of all of the major components in the cache structure. This section will show how the arbitration policies work to select the most important information first. The Request Multiplexer is on the right of *Figure 3-4* to *Figure 3-7*, while the Return Multiplexer is on the left of the figures.

4.4.1 The Cache

The cache can be accessed by up to four different units at once (*see Figure 3-6*), two read units (read and pre-fetch) and two write units (data write and block update). Block

updates always have priority over all other accesses, but do not require a cache port. Assuming general ports, priority will be given to the read/block request line, then the data write line and finally, lowest priority, the pre-fetch line.

4.4.2 The Return Buffer

The return buffer is positioned at the top left of Figure 3-4 to Figure 3-7, in Figure 3-4 the return buffer is mapped onto the Instruction Buffer, in Figure 3-6 it is mapped onto the Register File. It is accesses by two (Figure 3-4 and Figure 3-5) or three (Figure 3-6 and Figure 3-7) data return lines. Priority is always given to the oldest access, which will be from the Return Multiplexer. A read that hits both at the cache and the Data Write Buffer will only try to return data on the Data Write Buffer return line. Therefore, priority between the cache and the Data Write Buffer is given in chronological order, so the oldest outstanding access is completed first.

4.4.3 Outstanding References Buffer

The Outstanding References Buffer can be written to by up to four lines at once; it could also be checked by up to three (*see Figure 3-6*). Priority in writing is given to the block update line as it reduces the number of records in the buffer. The priorities for the remaining three lines apply to both the writing and checking of the buffer. The highest priority is given to reads, then writes and finally pre-fetches.

4.4.4 The Request Multiplexer

This is the Multiplexer on the right of Figure 3-4 to Figure 3-7. It can receive block requests from up to three lines (see Figure 3-6), the priority list is again: read, write, pre-fetch.

4.4.5 The Return Multiplexer

This is the Multiplexer on the left of *Figure 3-4* to *Figure 3-7*. A block fetch can facilitate the generation of many block returns by the Outstanding References Buffer. The return buffer, positioned below the Return Multiplexer, remains marked as busy until all of the Outstanding References have been forwarded. The Outstanding References Buffer sorts the references in reads, writes and then pre-fetches. Multiple returns in one group as ordered in chronological order, so the oldest return is sent first.

4.5 Performance Aspects

In addition to the speed up in writing and the removal of all write stalls, the Data Write Buffer and the Outstanding References Buffer have many additional advantages.

With conventional caching systems, there is no facility to have more than one outstanding cache access to the same address when one of them is a write. This would force a processor stall until the outstanding write or read(s) are completed. Using the two buffers, any memory reference instruction can be dispatched (in order) as soon as its source registers are available, buffer space permitting.

If a data write misses in the cache, a read to that data would normally also miss. However, with this structure, the data write record would be blocked in the Data Write Buffer (unless an *Allocate on Write Miss* policy is NOT being used). Thus, any subsequent read would register a hit. The buffers do not adversely affect reads, as they are accessed in parallel with the normal cache read.

If two outstanding memory operations reference the same cache block, then two block fetches could be generated. As the lower cache levels are slower and have less cache ports, the possibility of a cache access being blocked is increased. However, using the Outstanding References Buffer, only a single block fetch will be initiated, thus reducing unnecessary memory traffic at the lower levels.

5 Reading from the Cache

5.1 Overview

Reading from memory is one of the most common instructions. Therefore, to make the common case fast, the read path is optimised for speed. The Data Write Buffer and Outstanding References Buffer are therefore placed off this critical path. This is achieved by accessing the two buffers in parallel with the cache read.

5.2 Basic Procedure

The processor sends the address of the instruction/data and optionally the destination register to the primary cache during the ID stage of the pipeline. The read can arbitrate for a cache read port in that cycle, and so commence reading in the next cycle. If a data read fails to obtain a cache read port, then no more instructions can be issued in that cycle.

When a cache read port has been acquired, the cache, the Outstanding References Buffer and the Data Write Buffer are read in parallel. The read port is held for the length of the read.

If, after the cache read latency, a hit is obtained in either the Data Write Buffer or the cache, the data can be returned.

Data Write Buffer HIT $\Rightarrow$ If a block is requested and the whole block is not present, then the read is stalled until the write is completed. Otherwise, the data is successfully returned. The cache and Outstanding References Buffer results are ignored.

Cache HIT $\Rightarrow$ If a hit is also registered at the Data Write Buffer, then the cache access is ignored. Otherwise, the data/block is returned and the Outstanding References Buffer result is ignored.

If, after the cache read latency, a hit is not obtained, an entry is made in the Outstanding References Buffer containing the address required and the destination register. If the Outstanding References Buffer access also registered a miss, then a block request is sent

to the next level via arbitration at the Request Multiplexer. Once the Request Multiplexer grants the block request, the read at the next level functions in the same way.

When the block is returned, the Outstanding References Buffer is accessed to see what sub-block(s) need to be returned to the previous level/processor. The block, buffered in the return buffer is written into the Cache. Sub-blocks, selected from the return buffer by relevant records in the Outstanding References Buffer, are returned to the previous level/processor. When all outstanding references to the block have been forwarded, the return buffer is free to receive another block.

Returned data can come from one of three sources at each cache level (*see Figure 3-5 or Figure 3-6*), the Data Write Buffer, the cache itself, or straight from a lower cache level. Priority is given to results from lower levels (*see section 4.4 on page 20*).

When reads and writes to the same address are issued, two problems are created. A write followed by a read to the same address (the read should pick up the new data written), also, a read followed by a write of the same address (the read should ignore all writes issued after the read). The resolution of these problems is presented in *section 6.6 page 28*.

6 Writing to the Cache

6.1 Overview

The main advantages of this new cache design concern writing to the cache. Memory writes constitute a significant proportion of memory references, which are considered to be a major limiting factor in high performance processors. Multi-ported caches have made an improvement, but write ports are harder to duplicate than read ports. Processor stalls caused by writes are avoidable; they have therefore been given a great deal of attention.

Writing to a cache should be done in the background, therefore taking the responsibility of completion away from the processor. The Data Write Buffer is designed to accept all writes as they are issued. The cache structure will then ensure successful completion of the write.

Write policies have an effect on processor performance, and have therefore been parameterised. Four possible combinations are created by the *Write Back/Write Through* and the *Allocate on Write Miss*, parameters. Each combination will be considered independently.

6.2 Write Through but not Allocate on Miss

The processor will request a Data Write Buffer write port at the end of the ID stage of the instruction pipeline. Writing to the Data Write Buffer will commence in the next clock cycle. If there are no top level Data Write Buffer write ports free, then no more instructions can be issued that cycle.

1. The write is blocked until it acquires a Data Write Buffer write port. Once acquired, the write may then commence during the next cycle.

2. When a Data Write Buffer write port is acquired, the Data Write Buffer and Outstanding References Buffer are associatively searched. The Data Write Buffer is searched for a record addressing the same data or sub-block depending on the global parameter 'Data Write Buffer Merge Level'. The Outstanding References buffer is searched for a record addressing the same sub-block. The latency of the write to the Data Write Buffer is

defined by the global parameter 'Data Write Buffer write latency'. After the write latency, if a match is found in:

The Data Write Buffer ⇒ The matching record is updated with the new data. Therefore, a new record does not need to be added.

The Outstanding References Buffer ⇒ A new record is added to the Data Write Buffer with its status set to 'Blocked', hence the write is stalled until the block is fetched into the cache.

Neither buffer ⇒ A new record is added to the Data Write Buffer. If the new record is at the front of the queue, it will, in the same cycle, take part in arbitrating for a cache write port (for port priorities, see *section 4.4 page 20*). A record can thus pass through the Data Write Buffer in zero or more cycles.

3. Each cycle, all free cache write ports are assigned to records in the Data Write Buffer that have a status of 'waiting for a cache write port'. Write ports are assigned to Data Write Buffer records in FIFO order, therefore the oldest record with the correct status is assigned the first free cache write port.

4. When a cache write port has been acquired, the cache is written and the Outstanding References Buffer is associatively searched simultaneously. The ports are held for the length of the cache write.

5. If a cache HIT occurs, the data is written to the cache after the cache write latency. Irrespective of a hit or miss, the data and address are sent to the next cache level, and the Data Write Buffer record is removed. The write to following level is explained by reapplying this procedure from *step 1*. After the write to the main memory, the write procedure is complete.

6.3 Write Through and Allocate on Miss

This procedure is the same as section 6.2 for steps 1 to 4.

5. After the cache write latency, a HIT will have occurred in:

   The Cache ⇒ The data is written to the cache, the data and address are sent to the next level, and the record is removed from the Data Write Buffer. Writes to the following levels are explained by reapplying this procedure from *step 1*. After the write to the main memory, the write procedure is completed.

   The Outstanding References Buffer ⇒ The record is kept in the Data Write Buffer, but marked as 'stalled'. A block request is already in progress, therefore nothing need doing until the block is returned.

   Neither ⇒ The block needs to be fetched from the next level. An entry is made in the Outstanding References Buffer to indicate the address of the block and that it is a pre-fetch. No more processing is needed until the block is returned.

6. The block fetch is explained by applying the read procedure to the next cache level, explained in section 5.2 on page 23.

7. When the block is returned, the pre-fetch record is removed from the Outstanding References Buffer, if present. Any relevant records in the Data Write Buffer have their status flags set to 'waiting for a cache port'. This procedure is then reapplied from *step 1*.

6.4 Write Back but not Allocate on Miss

This procedure is the same as section 6.2 for steps 1 to 4.

5. After the cache write latency, a cache HIT or MISS will have occurred:

   Cache HIT ⇒ The data is written to the cache, and its dirty bit is asserted. The record is then removed from the Data Write Buffer.

   Cache MISS ⇒ The data and address are sent to the next level, the writes to the following levels are covered by reapplying this procedure from *step 1*. The record is then removed from the Data Write Buffer.

6. The write procedure is now completed.

6.5 Write Back and Allocate on Miss

This procedure is the same as section 6.2 for steps 1 to 4.

5. After the cache write latency, a HIT will have occurred in:

   The Cache ⇒ The data is written to the cache, and the record is removed from the Data Write Buffer. The write procedure is completed.

   The Outstanding References Buffer ⇒ The record is kept in the Data Write Buffer, but marked as 'stalled'. A block request is already in progress, therefore nothing need doing until the block is returned.

   Neither ⇒ The block needs to be fetched from the next level. An entry is made in the Outstanding References Buffer to indicate the address of the block and that it is a pre-fetch. No more processing is needed until the block is returned.

6. The block fetch is explained by applying the read procedure to the next cache level, explained in section 5.2 on page 23.

7. When the block is returned, the pre-fetch record is removed from the Outstanding References Buffer, if present. Any relevant records in the Data Write Buffer have their status flags set to 'waiting for a cache port'. This procedure is then reapplied from step 1.

6.6 Preserving Coherency

The cache is most at risk of loosing coherency when there is an outstanding write. A write must make sure that it updates every level of the cache hierarchy required, and provides data for any subsequent reads, but does not it must not corrupt any older reads. This data management is achieved by intelligent use of the Data Write Buffer and the Outstanding References Buffer. There are four possible methods of writing to the cache, controlled by two variables: *Allocate on Write Miss* and *Write Back/Write Through.*

6.6.1 The problem of a write then a read

This concerns a read request at any level of the memory system. A read is not allowed to pass a write to the same address in the primary cache, or to the same sub-block at any other

level. A read will pick up a prior write still sitting in a Data Write Buffer as a hit unless the cache is a lower level and the whole block is not present in the Data Write Buffer.

If a read is sent to a lower level cache containing a write to the same address/sub-block that is waiting for a write port, then the read is refused a read port to that cache level. If a read completes before a prior write to the same address/sub-block, then the read is again stalled until the cycle after the write has completed, to allow the signals to settle.

6.6.2 The problem of a read then a write

Writes can not pass reads to the same address (primary caches) or sub-block (lower level caches). The same rules apply as outlined in the previous sub-section.

After a lower level cache read hit, the correct data is sent to the processor, up the memory hierarchy. Any writes that were issued after the read will only alter the records in the Outstanding References and Data Write Buffers. The cache entries will not be entered as the block can not be present in the higher level caches. The data itself, will bypass the cache levels via the Return Multiplexer (*see Figure 3-6 and Figure 3-7*) and up to the processor uncorrupted.

7 Cache Block Pre-fetching

7.1 Overview

The action of pre-fetching cache blocks into the primary cache is an old trick previously employed by assembly language programmers. The idea is to remove the latency involved in accessing a memory block for the first time. This will remove some of the so-called 'unavoidable' cache misses during a program run when a new working set is loaded. The aim of this section is to show how the art of pre-fetching can be related to current developments in instruction scheduling technology.

A new pre-fetch instruction is introduced, which differs in several respects from a basic load instruction. The main difference is that the data at the location referenced is only fetched into the primary cache and not into the register file/instruction buffer. The cache will be parameterised so that the pre-fetch instruction may optionally bypass the primary cache, assuming a miss.

7.2 How Pre-fetching can be achieved

There are two problems involved in pre-fetching. The first is what memory blocks are going to be needed by the processor, and the second is, which implementation of the pre-fetch instruction will be most effective.

The first problem is particularly relevant to the current HSA project. The memory blocks that are needed by the processor are identified when the program is compiled. Pre-fetch instructions are then placed far enough in front of memory referencing instructions so that the required memory block will be available in the primary cache. To know which memory references will cause a cache block conflict, the compiler will need to know the block size, associativity and number of sets in the cache.

The second problem concerns the relative advantages and disadvantages of different implementations of the pre-fetch instruction. Parameters will be included in the simulation to allow the different methods to be evaluated. Parameters will include facilities to ignore all

pre-fetch instructions, issue only if a read port is free, and send directly to the secondary cache without checking the primary.

7.3 The Basic Procedure

A pre-fetching compiler needs a program, and a model of the caches to be used. The model can be general, but the best results will always be obtained with the correct model. The compiler will analyse the code and insert pre-fetching instructions in their optimal positions. This process will reapply existing HSA instruction scheduling technology to a new problem. The program can now be run on any machine using the same instruction set.

When the simulator encounters a pre-fetch instruction, a cache block pre-fetch is sent directly to either the primary or secondary cache depending on the parameter. While misses occur, a low priority (see *section 4.4*) block fetch is issued to the next level cache, until a hit occurs. The block is then returned to the primary cache. This completes the pre-fetch instruction, and so it is removed from the Outstanding References Buffer.

When the instruction(s) referencing this block are executed, they now obtain a primary cache hit!

8 Summary

This cache design is meant to combine the benefits of sound, well known design structures with the new buffering ideas to avoid write, and to some extent read stalls.

The design is highly parameterised, and therefore the benefits of the two buffers can be assessed on a multitude of cache types and at each cache level. In essence, the cache structure is a multi-level, non-blocking cache with a parameterised number of ports and latencies. The Data Write Buffer size and number of write ports is parameterised, as is the size of the Outstanding References Buffer.

Processor stalls due to blocked reads can be avoided in this cache structure. When a read needs to access a cache to check for a hit, it could be blocked by a write updating the cache. This type of stall is alleviated by the buffering structure. Any block or data that is being written is also present in the Data Write Buffer. Therefore, the parallel check of the cache and the Data Write Buffer will show any data in that level cache, and in the specified read time of that level.

The reduction in processor stalls due to blocked writes is the main benefit of this new cache structure. Multiple cache write ports are complex and expensive, thus their number is generally very low. An intermediate buffer with more ports would be cheaper and simpler. The bursts of reads can be accepted by the buffer, without the need to stall the processor, and the cache is only sent a small number of writes per cycle. Thus the cache need only have write ports to match the mean number of writes per cycle as opposed to the maximum to avoid all write stalls.

Finally, a facility for the pre-fetching of memory blocks into the primary cache is presented. A special compiler that uses a model of the cache hierarchy to schedule the memory references will insert the pre-fetch instructions. This area has not been well examined by current research; therefore different methods of dealing with the instructions will be incorporated and parameterised. This will facilitate the examination of the relative merits of the different methods.