TECHNICAL REPORT

COMPUTER SCIENCE

THE IMPACT OF A REALISTIC CACHE STRUCTURE ON A HIGH PERFORMANCE SUPERSCALAR ARCHITECTURE

Report No 319

Daniel Tate

August 1998

The Impact of a Realistic Cache Structure on a High Performance Superscalar Architecture

Abstract

Despite the widely held belief that the most limiting factor in processor performance is the memory hierarchy, much of the recent research into multiple instruction issue techniques assumes a perfect cache structure with a 100% hit rate. This paper attempts to rectify this imbalance by quantifying the performance impact of incorporating a realistic cache structure into a high-performance superscalar architecture.

A highly parameterised cache simulator is integrated into a minimal superscalar architecture, the Hatfield Superscalar Architecture (HSA), that uses static instruction scheduling and in-order instruction issue. Two main studies are presented. First, the impact of a cache on unscheduled code is compared to the impact of a cache on scheduled code. Second, the speedup achieved through static instruction scheduling with a perfect cache is compared to the speedup achieved with a series of cache sizes.

Contents

Appendices

Table of Figures

1. Introduction

It is widely believed that the most limiting factor in current architectures is the memory sub-system [Patt97]. Despite this, much of the research into Multiple Instruction Issue (MII) processor design assumes a perfect cache and memory system. There are two diverse schools of thought on MII research; dynamic instruction scheduling relies on complex hardware to uncover instruction parallelism at run-time, while static instruction scheduling relies on a specialised compiler to uncover the instruction parallelism at compile-time. In this paper we use the Hatfield Superscalar Architecture (HSA), a statically scheduled architecture developed at the University of Hertfordshire, to examine the impact of caches on MII architectures. The main reason for this selection is that the code expansion inherent in static instruction scheduling will place increased pressure on the cache structure. Furthermore, it will not be possible for the variable load latencies caused by cache misses to be masked by dynamic out-of-order instruction issue. Cache performance is therefore particularly crucial in a statically scheduled architecture.

Our purpose is to quantify the impact of a realistic cache on MII processor performance. All too easily, stalls due to cache misses and memory port unavailability can partially offset any speed-up gained through instruction scheduling. The performance of scheduled and unscheduled code will therefore be compared, both with realistic sized caches and with a perfect cache model. A range of cache implementations will be tested on two high-performance superscalar machine models; a Maximal Model will represent an idealised cache memory model, while a Standard Model will provide a more realistic number of cache ports.

Two areas are of particular interest to the HSA project. Firstly, we wished to quantify the effect of a realistic cache structure on both scheduled and unscheduled code. Secondly, we expected a cache miss to have a disproportionate impact on scheduled code and wished to quantify the extent of this effect.

This study uses a highly parameterised cache simulator [Tate98] that has been developed to model a wide variety of cache structures. This cache simulator has been integrated into the processor simulator for the Hatfield Superscalar Architecture.

2. The Hatfield Superscalar Architecture (HSA) Project

The two main schools of thought on instruction scheduling led to two types of architecture, VLIW and superscalar. Because empty VLIW instruction slots must be filled with NOPs, VLIW architectures suffer from excessive code expansion. Furthermore different implementations of a VLIW architecture tend to be incompatible because of the need to fill empty instruction slots. Superscalar processors can only extract parallelism from within a limited window of instructions and tend to suffer from hardware complexity.

The HSA [StevG96] was developed as a progression from HARP [StevG91], a VLIW processor that was developed and fabricated at the University of Hertfordshire. As with all VLIW architectures, HARP suffered from a pre-determined constant instruction issue rate, and from code expansion. The HSA project attempts to combine the best features of VLIW and superscalar architectures. The problems of incompatibilities within a processor family and fixed VLIW instruction widths are averted by dynamically issuing variable sized instruction groups. The problems of superscalar complexity and a fixed-sized instruction window are avoided by using extensive instruction scheduling at compile-time, combined with in-order instruction issue at run-time. The HSA has therefore been described as a minimal superscalar architecture [StevG96].

The HSA includes speculative and guarded execution and a generalised branch delay mechanism. To ease the pressure on the instruction issue rate, any instruction whose Boolean guard fails prior to instruction issue can be ‘squashed’ in the instruction buffer. However, this will not ease the pressure on the instruction cache.

The HSA uses a simple four-stage pipeline:

IF	Instruction Fetch
ID/RF	Instruction Decode / Register Fetch
EX	Execute
WB	Write Back

At the first pipeline stage, instructions are fetched from the instruction cache into the Instruction Buffer. At the second stage, instructions in the Instruction Buffer are decoded and register operands are fetched from either the integer, floating point or Boolean register files. Alternatively, the operands can be forwarded from other functional units. At the third and fourth pipeline stages, instructions are processed and results written to the destination registers and made available to other functional units through forwarding.

The HSA instruction set contains a simplified addressing mechanism [StevF93], which does not need to pass through the ALU to generate an effective address. Memory references can therefore be dispatched to the data cache at the end of the ID/RF pipeline stage. As a result, with one-cycle instruction fetch and data cache accesses, a LOAD instruction can be executed in four cycles, with no load delay.

To generate code that is executable by the HSA, a Gcc compiler was generated for the HSA instruction model. The HSA instruction set was originally developed as part of the HARP project. Extensions to the instruction set were implemented to facilitate the new superscalar nature of the processor. The HSA code is executed by a highly parameterised simulator [Coll93] that allows an immense variety of processor models to be configured and simulated. An equally flexible cache simulator has now been integrated into the original processor simulator.

3. The Cache Simulator

Like most high-end computer architecture research, the HSA initially assumed a 100% cache hit rate for all memory references. This study removes this simplification, and quantifies the effect of a realistic cache on the speedups achieved by the HSA static instruction scheduler.

The cache simulator is completely configurable at run-time with the facility for the number and configuration of each data, instruction and unified cache to be set interactively before each simulation run. A facility to load and save entire cache models from a file is also included for convenience. At run-time, each cache level, excluding main memory, can have the following parameters configured: Cache size, Block size, Sub-block size, Associativity, Number of sets, Block replacement policy, Pipelining, Read / Write / Data Write Buffer latencies, Number of read / write / Data Write Buffer ports, Data Write Buffer size, Outstanding References Buffer size and Number of return (cache bypass) paths. At each level the cache is accompanied by three other structures of note: a Data Write Buffer, an Outstanding References Buffer and a data return (bypass) line (Figure 1).

The Data Write Buffer has a basic record queue structure. It is positioned above each data cache, including the first-level data cache, and also above main memory. This position facilitates the staging of writes to the cache. Each record contains the data, destination address and a status flag. The status flag is used in conjunction with an '*allocate on write miss*' policy, to indicate a record which has missed in the cache and is stalled waiting for a block to be fetched from a lower level. The size, number of ports and write latency of the Data Write Buffer can all be configured at run-time. The Data Write Buffer has numerous benefits to the cache structure. It is a small fast block of memory, therefore a write latency of zero is feasible with a minimum sized structure. However, in this study a latency of one is assumed to allow numerous checks and self-configuration logic to function. This buffer also has the ability to combine writes to the same address or sub-block, although this facility is not activated in this study.

The Data Write Buffer ahead of the first-level data cache is used to provide multiple virtual write ports for the processor and to minimise the number of processor write stalls. The Data Write Buffer's small size allows these write ports to be provided without excessive costs in terms of either time or silicon area. To provide a similar number of write ports in the cache itself would be far more costly. To avoid write stalls and thus gain maximum write throughput, the Data Write Buffer is usually provided with sufficient write ports to match the peak processor demand. In contrast, the cache itself only provides sufficient write ports to meet the average demand for writes, typically no more than one per cycle.

Another significant benefit of the Data Write Buffer is that data written to the cache is always kept in a 'visible' structure. If a write is stalled because of insufficient write ports, or the write is in the process of accessing the cache, then a following read from the same address is in danger of being stalled until the completion of the write. Using the Data Write Buffer, the data is visible at all times; therefore a read will register a Data Write Buffer hit and complete in its normal latency.

The Outstanding References Buffer is a fully associative structure. Each record contains the address referenced and internal configuration information. The size of the structure is configurable at run-time. One of the major problems uncovered by cache research is that a major cause of stalls at the top level is the overuse of memory bandwidth at the lower levels. The function of the Outstanding References Buffer is to keep track of all references to the next cache level in the hierarchy. If a request occurs for a block that is already being accessed, the Outstanding References Buffer stores the request as usual, but does not issue a block request to the next cache level. When the block is returned, all outstanding references to that block are activated. This facility significantly reduces the amount of lower level memory traffic, and thus the time it takes to fetch important blocks.

The data return line, which bypasses the cache, facilitates improved performance by returning requested data or instructions directly to the previous level. This facility is used to remove the latency involved in updating a block in the cache and then re-issuing a read. With the return (bypass) line, the block is written to the cache as usual, but in parallel the part of the block that is the subject of the read is forwarded directly to the processor or the previous cache level.

Although it includes some novel ideas, the cache structure implemented is not intended to be a 'state-of-the-art' cache design. The cache simulator was implemented to be a highly parameterised cache template that would allow almost any basic cache structure to be specified. The cache simulator has been successfully designed and implemented as an extension to the HSA processor simulator, and is used in this paper to simulate the effects of a cache structure on a statically scheduled superscalar architecture.

4. Results

For all tests, the well-established Stanford integer benchmarks were used. The successful examination of the performance of a cache relies on the careful selection of the cache sizes. With too large a cache, a benchmark's working set will completely reside in the cache; with too small a cache, the large number of block replacements will completely dominate the results. Four cache models between 256 bytes (64 words) and 2K (512 words) were chosen to reflect both the instruction and data working sets of the benchmarks. The tests consisted of all permutations from three delay slot models, five cache models and three processor models.

The first two branch delay slot models simulate cache access times of one and two cycles. An instruction cache access time of one cycle results in one branch delay slot, while a two cycle instruction cache access time gives two branch delay slots. The final branch delay slot model specifies zero delay slots; this simulates a perfect branch prediction model. The data cache latencies used mirror the instruction cache read latencies. Multi-cycle cache models have pipelined cache reads and writes.

The five cache models simulated differ in the sizes of their instruction and data caches. The data and instruction cache sizes in the first four models range from 256 bytes (4 sets) through 512 bytes (8 sets) and 1K bytes (16 sets) up to 2K bytes (32 sets). The final cache model provides a 100% hit rate. The number of cache ports depends on the processor model, but all have a single cache return (bypass) line. The main memory contains one, non-pipelined, read/write port which can satisfy a single read or write in ten processor cycles. Blocks are 64 byte (16 word), and no use is made of sub-blocks. The cache structure is direct mapped, while write policies are 'write back' and 'allocate on write miss'. The number of Data Write Buffer ports and cache write ports is always equal, and the Data Write Buffer and Outstanding References Buffer both hold 40 records. These figures were chosen to ensure that the size and configuration of these buffers did not limit performance. The instruction cache has a fetch width of 16 instructions. In this study instruction fetches are allowed to cross cache block boundaries. A later study will quantify the performance impact of confining instruction fetches to a single block.

The three processor models we simulate consist of a Baseline, a Standard Superscalar and a Maximal Superscalar model. All instructions have unit latencies, except for multiply which has a latency of three and divide which has a latency of sixteen. The Baseline Model simulates a standard RISC processor that fetches and issues one instruction per cycle and contains one functional unit of each type (arithmetic, load, store, etc.), and executes the code that is produced directly by our gcc compiler. The Standard Model simulates a high-performance superscalar architecture and contains 16 functional units of each type. The cache structure, however, only contains two data read and two data write ports. The Maximal Model simulates an ideal processor; it contains 16 functional units of each type and also 16 data read and 16 data write ports. Both of these superscalar models execute code that has passed through the HSA scheduler [StevF95].

The same unscheduled code is executed on each of the three delay slot models. However, scheduled code is optimised by the scheduler for a specific number of delay slots. There are therefore, six scheduled benchmarks for every benchmark, three for each delay slot model on the processor's Standard Model, and another three optimised for the processor's Maximal Model.

Most of the results are presented as averages of the performance of the individual Stanford benchmark programs (Figure 2 to Figure 4, and Figure 6 to Figure 7). Figure 5 and Figure

8 are included to illustrate the spread of figures achieved by the individual benchmarks. This allows examination of the effect of a cache on different types of programs.

Our first set of results compares the impact of a cache on scheduled code to the impact on unscheduled code. Figure 2 to Figure 4 show the performance impact of the cache for zero, one and two delay slots respectively, with Figure 5 showing how the cache impacts each benchmark individually when using the Maximal Model with one branch delay slot. Using the zero delay slot chart in Figure 2 as an example, the unscheduled code is executed using the perfect and finite cache models for the zero delay slot model on the processor's Baseline Model. The results show that using a cache size of 256 bytes, the code runs at 91% of the speed achieved with a perfect cache. If a cache size of 2K is used, then the performance rises to just over 98%. The suite of benchmarks scheduled for zero delay slots and the Standard Model are also executed with perfect and finite cache models. In this case using a cache size of 256 bytes gives 73% of the speed of a perfect cache, while a cache size of 2K gives almost 96% of the speed. For the Maximal Model, the execution speed using a cache size of 256 bytes results in 65.5% of the perfect cache's speed, while a cache size of 2K gives almost 94% of the speed.

The performance of individual benchmarks is shown in Figure 5. These figures are a break down of the average figure shown for the Maximal Model in the graph using zero branch delay slots. It can be seen that there is a vast difference in the impact of the cache on the performance of individual programs. Bubble is hardly affected by a finite cache, achieving 99.99% of a perfect cache's performance with 1K direct-mapped caches and over 94% with 256 byte caches. In contrast, Tree achieves only 80% with 2K caches, dropping to 53% with 256 byte caches. Perm is affected drastically by the size of the cache, with performance falling from over 99% with 2K caches, to less than 38% with 256 byte caches.

Our second results set compares the speedup achieved with a perfect cache model to the speedups achieved with finite cache sizes. The speedups are calculated by dividing the execution time into the execution time of the Baseline Model. The Baseline Model is always a single instruction fetch and issue RISC processor. Figure 6 and Figure 7 use values averaged over the suite of benchmarks to show the speedup for the Standard Model and the Maximal Model respectively. Figure 8 details how the figures shown in the graph for the Maximal Model with one branch delay slot are made up from the results for individual benchmarks.

Using the Maximal Model, Figure 7, as an example, it can be clearly seen that there is a steady gain in performance as the cache size increases. The speedup obtained using zero branch delay slots with data and instruction caches of 256 bytes is only 2.53 (equivalent to a 60% decrease in cycle count), while using a cache size of 2K raises the average speedup to 3.62 (A 72% decrease in cycle count). The average speedup obtained by using a perfect cache is 3.76.

To examine how the average figures are made up, we must select one block of figures. Figure 8 contains the individual benchmark breakdown of the average figures given for the Maximal Model with one branch delay slot. This figure shows two distinct sets of values. Bubble and Perm's speedups are both very high. Bubble ranges from a speedup of 4.8 for 256 byte caches to 5.2 for 2K caches, while Perm, although more affected by cache size, ranges from 3.65 for 256 byte caches to 5.65 for 2K caches. In contrast, the speedup for the remaining benchmarks is less spectacular. Finally, Queens displays some interesting performance characteristics. With 256 and 512 byte caches the speedups are 1.34 and 1.36 respectively, while for 1K, 2K and perfect caches the speedups jump to 2.7, 2.9 and 3.1 respectively.

5. Discussion and Conclusions

When initial results were first obtained, two anomalies were uncovered. Firstly, zero branch delay slot models should always run much faster than one branch delay slot models, which should in turn always run faster than two branch delay slots models. Secondly, it should not be possible for the Standard Model to run faster than the Maximal Model. However, both of these perfectly logical premises were found to be incorrect.

On closer inspection, both of these problems have a common cause; the size of the code directly effects the cache usage for smaller cache sizes. An example of this is Perm when it is scheduled for zero branch delay slots and run with caches of 256 bytes. With the Maximal Model Perm executes in 118,000 cycles, while on the Standard Model this figure falls to 98,000 cycles. The cache utilisation statistics show that while the data cache usage for the two runs is almost identical, the instruction cache usage is very different. With the Maximal Model there are twice as many instruction cache misses (40,000 versus 20,000). This increase is a direct result of the size of the scheduled code, which contains almost 40% more instructions for the Maximal Model. Confirmation that the extra code size is the cause of the disparity between logical and actual run-times can be found by interrogating the sets occupied by the extra instructions. On the Minimal Model, sets 0 and 1 are updated 10,000 times, with sets 2 and 3 updated just 3 times. In contrast, the Maximal Model has set 0 updated just 4 times, but the other three sets are updated over 10,000 times. This demonstrates the importance of instruction fetching to the performance of a processor. It has been said that instruction delivery is the most important problem [Patt97] when talking about high-performance superscalar processors, and these figures support that hypothesis.

The overall results produced charts that looked much as expected. As the size of the cache decreases, the performance also decreases. In Figure 2 to Figure 4, the performance of the Maximal Model looks worse than the Standard Model. However, it must be remembered that these figures are related to the performance with a perfect cache. The benchmarks actually run faster using the Maximal Model, but the caches have more of an impact because of the increased code size.

When the individual benchmarks are examined, five of the seven perform as expected, but Tree performs disproportionately badly; conversely, Bubble performs disproportionately well. The reason for Tree's inability to cope with a cache is not due to the static instruction size of 338, which fits straight into a 2K cache with no contention. The problem is the size of the data working set. The rest of the benchmarks use arrays to hold their data, thus providing localised data. However, Tree uses nodes and pointers to sort its data. This creates loosely coupled data, and proves impossible for even a 2K data cache to cope with. Using the example given in Figure 5, zero branch delay slots on the Maximal Model, Tree suffers from data cache miss rates of 40% for reads and 57% for writes when using a cache size of 2K. No particular cache set is affected more than another; figures range from 28 (set 25) to 2621 (set 8).

Bubble performs almost as well with a cache size of 1K as with a perfect cache (99.99%) for the same reason that Tree performs so badly. Both the instruction and data working sets are fully located in their respective caches. Only one block was even updated, data cache set 16, and then only ten times. As the cache size reduces, the performance impact increases, but due to Bubble's small instruction and data sets, the impact is less than for the rest of the benchmarks.

Introducing a cache into the Baseline Model and superscalar models (Figure 6 to Figure 8), produces the expected results. Implementing a superscalar processor with a realistic cache will incur a significant penalty. A Baseline Model incorporating a 2K cache and zero delay slots achieves 98% of the performance achieved with a perfect cache, a Maximal Model using instruction scheduling and superscalar execution only achieves a speedup of 3.62. This figure is significantly slower than the speedup of 3.76 achieved with a perfect cache. On viewing Figure 6 and Figure 7 it is clear that scheduled code suffers more from the introduction of a realistic cache than unscheduled code, and the higher the performance of the processor, the greater the impact of the cache.

The speedups achieved by Queens using zero and one delay slots are very interesting (Figure 9). It has been previously demonstrated that code expansion is a major impediment when using a finite cache size. When Queens is scheduled using the Standard

Model and zero delay slots code expansion is 26%. However, when scheduled for one delay slot code expansion increases drastically to 139%. A lot of this code expansion can be attributed to aggressive scheduling filling branch delay regions, of 16 instructions, with speculative instructions. The effect of this is to put increased pressure on the instruction cache especially when the current working set can not be fully held by the cache (see Figure 8 with cache sizes of 256 and 512 bytes). However, when the cache is large enough the performances of the two models converges; the performance gained by speculative instruction execution outweighs the cache stalls caused by increased code size.

The increased importance of maintaining a flow of instructions for a high-performance superscalar processor is highlighted by these results. Using more advanced caching structures can reduce the impact of an instruction cache. Set associative caches are well known, but may increase cache access times. Alternatively multi-level, trace and victim caches are also likely to prove effective.

To conclude, it has been shown that when the current working set can not be held in either the instruction or data cache, then a significant reduction in performance is encountered. The challenge is to regain the speedup that has been lost to cache stalls as a result of the increase in code size caused by instruction scheduling. Our objective is to develop the HSA idiom to maximise the extraction of parallelism at compile time. We will also determine whether any further speedup can be extracted from scheduled code through the introduction of out-of-order instruction issue.

6. References

[Coll93] Collins R - *Developing a Simulator for the Hatfield Superscalar Processor*, University of Hertfordshire Technical Report No. 172, December 1993.

[Egan97] Egan C, Steven F L, Steven G B - *Delayed Branches Verses Dynamic Branch Prediction in a High Performance Superscalar Architecture*, Euromicro97, Budapest, September 1997.

[Patt97] Patt Y N et al. - *One Billion Transistors, One Uniprocessor, One Chip*, Computer, September 1997, pp 51-57.

[Roten96] Rotenberg E, Bennett S, Smith J E - *Trace Caches: A Low-Latency Approach to High-Bandwidth Instruction Fetching*, Proceedings of the 29th Annual International Symposium on Microarchitecture, Los Alamitos, California, 1996, pp 24-34.

[StevF93] Steven F L, Adams R G, Steven G B, Wang L, Whale D J - *Addressing Mechanisms for VLIW and Superscalar Processors*, Microprocessing and Programming, Vol.39, Numbers 2-5, December 1993, pp 75-78.

[StevF95] Steven F L, Steven G B, Wang L - *Using a Resource Limited Instruction Scheduler to Evaluate the iHARP Processor* - IEE Proceedings - Computers and Digital Techniques Vol. 142, No.1, January 1995, pp 23-31.

[StevG91] Steven G B, Adams R G, Findlay P A, Trainis S A - *iHARP: A Multiple Instruction Issue Processor*, Proceedings of the 18th Annual International Conference on Computer Architecture, Toronto, May 1991.

[StevG96] Steven G B, Collins R - *A Superscalar Architecture to Exploit Instruction-Level Parallelism*, Euromicro96, Prague, September 1996.

[Tate98] Tate D, Steven G - *Developing the Hatfield Superscalar Architecture Cache Simulator*, University of Hertfordshire Technical Report, No. 318, June 1998.

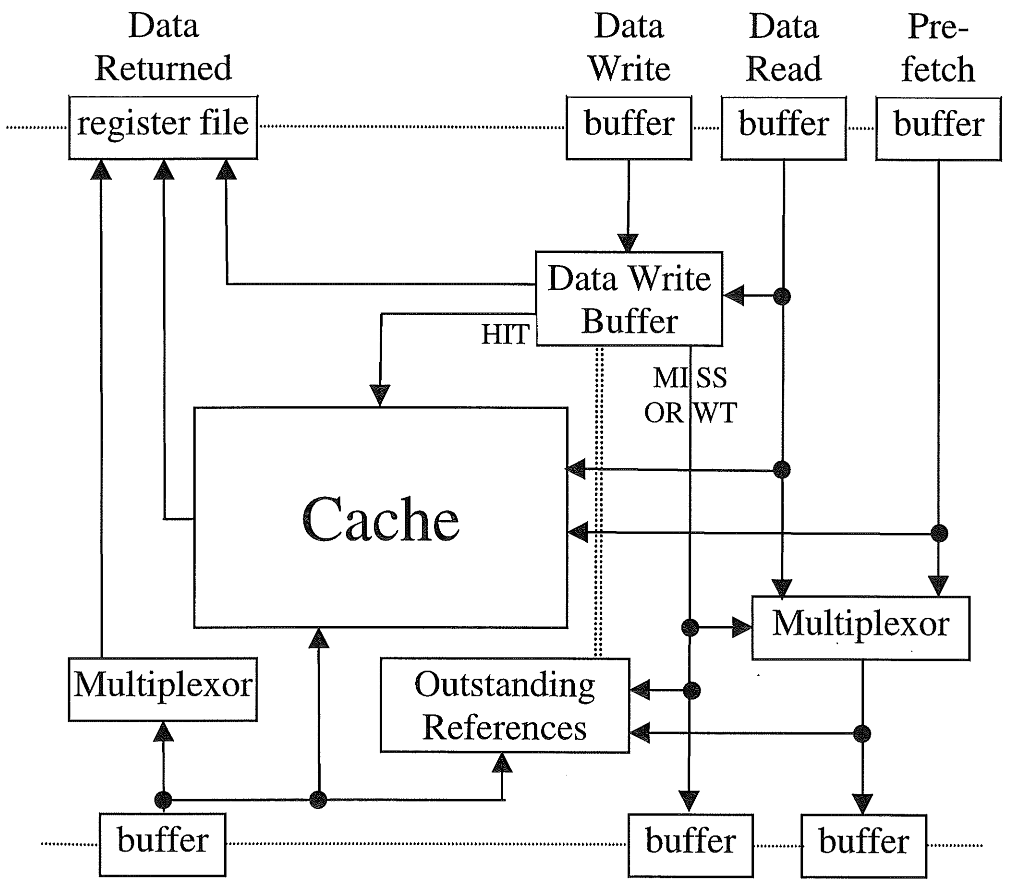

Figure 1 Realisation of Cache Structure

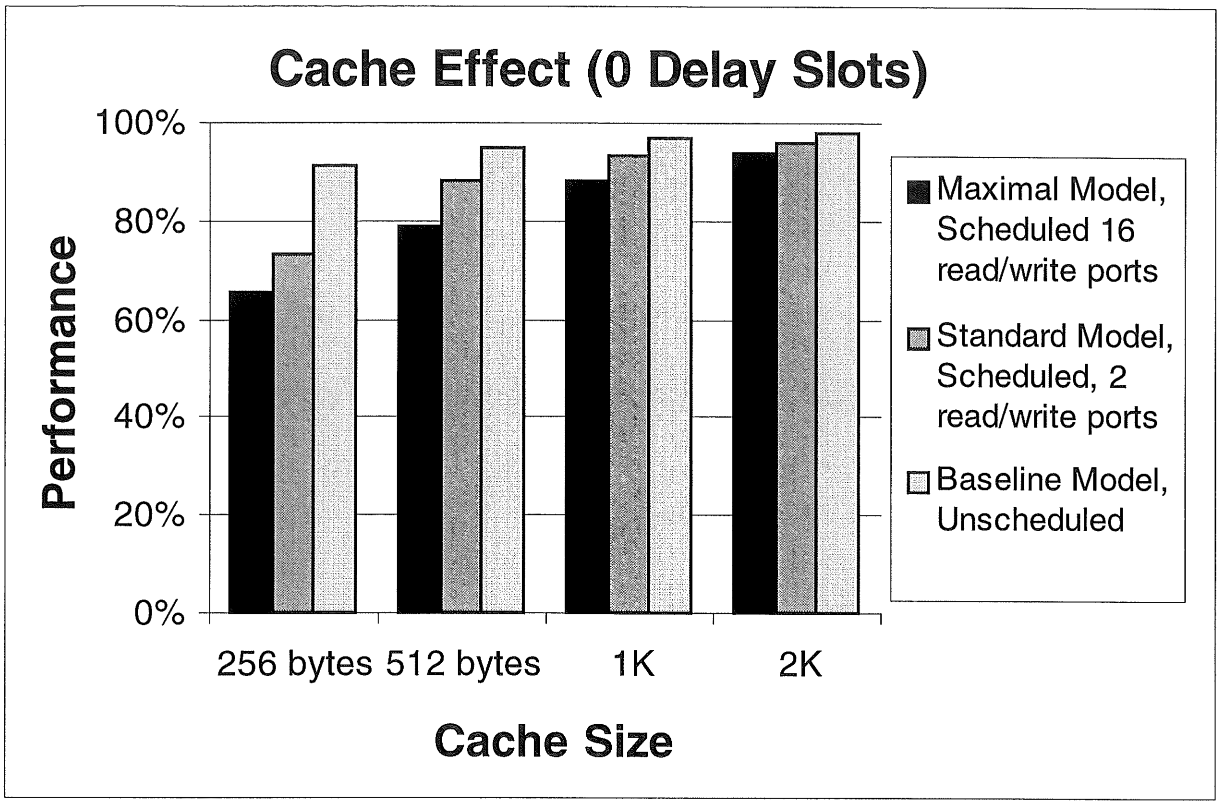

Figure 2 Impact of cache on performance when using 0 Delay Slots

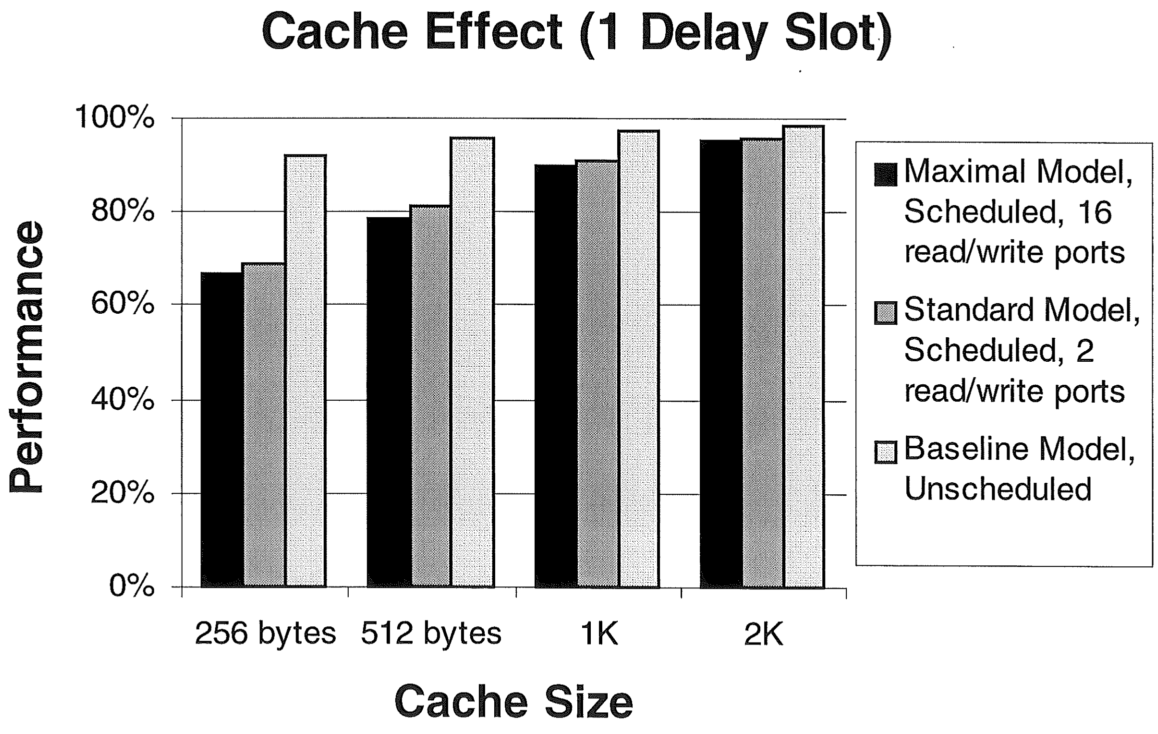

Figure 3 Impact of cache on performance when using 1 Delay Slot

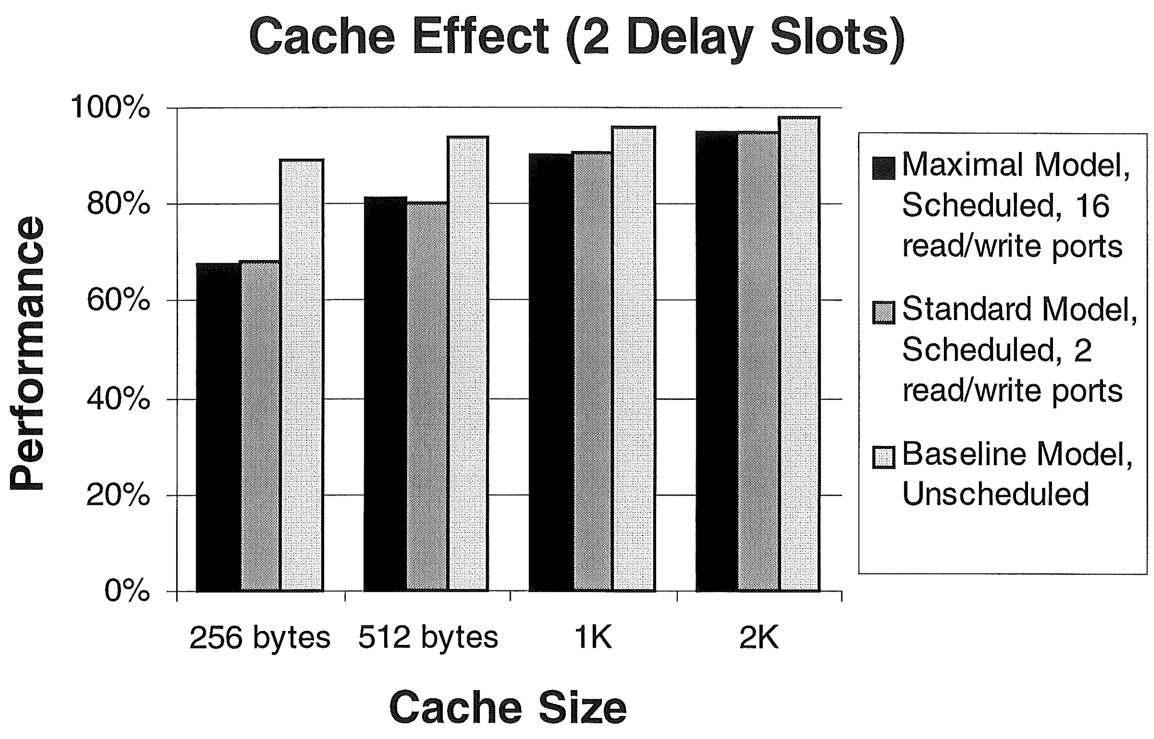

Figure 4 Impact of cache on performance when using 2 Delay Slots

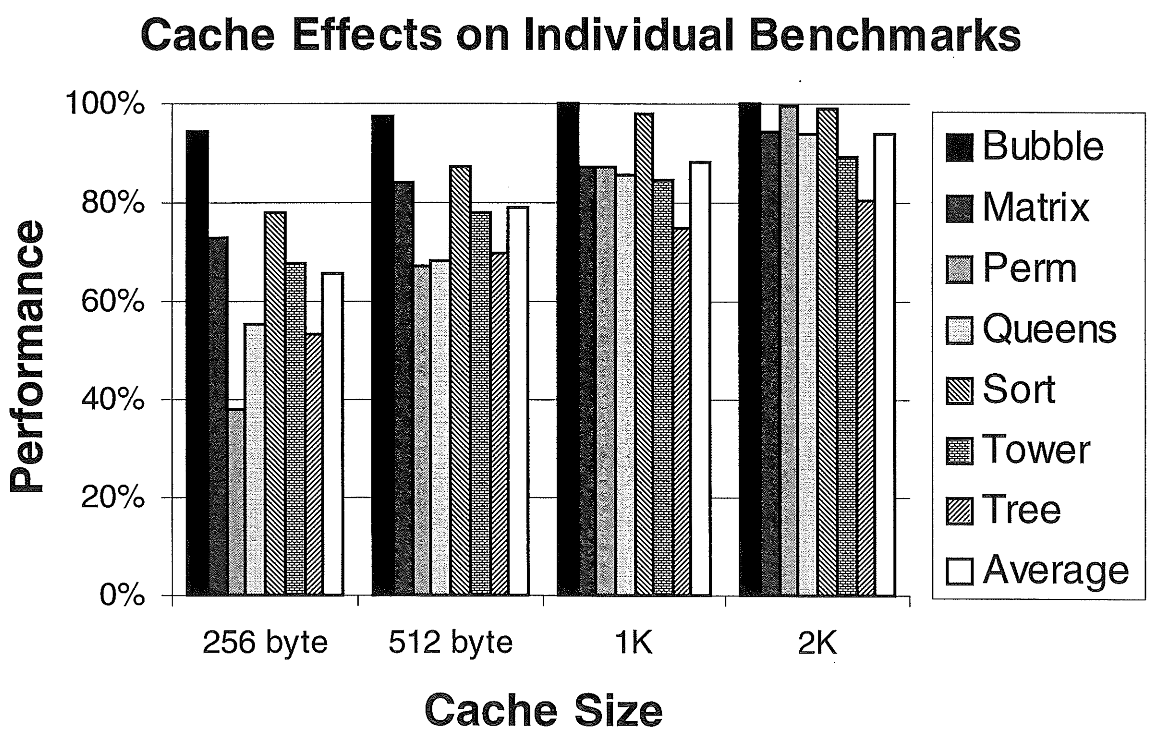

Figure 5 Cache impact using the Maximal Model with 0 Delay Slots

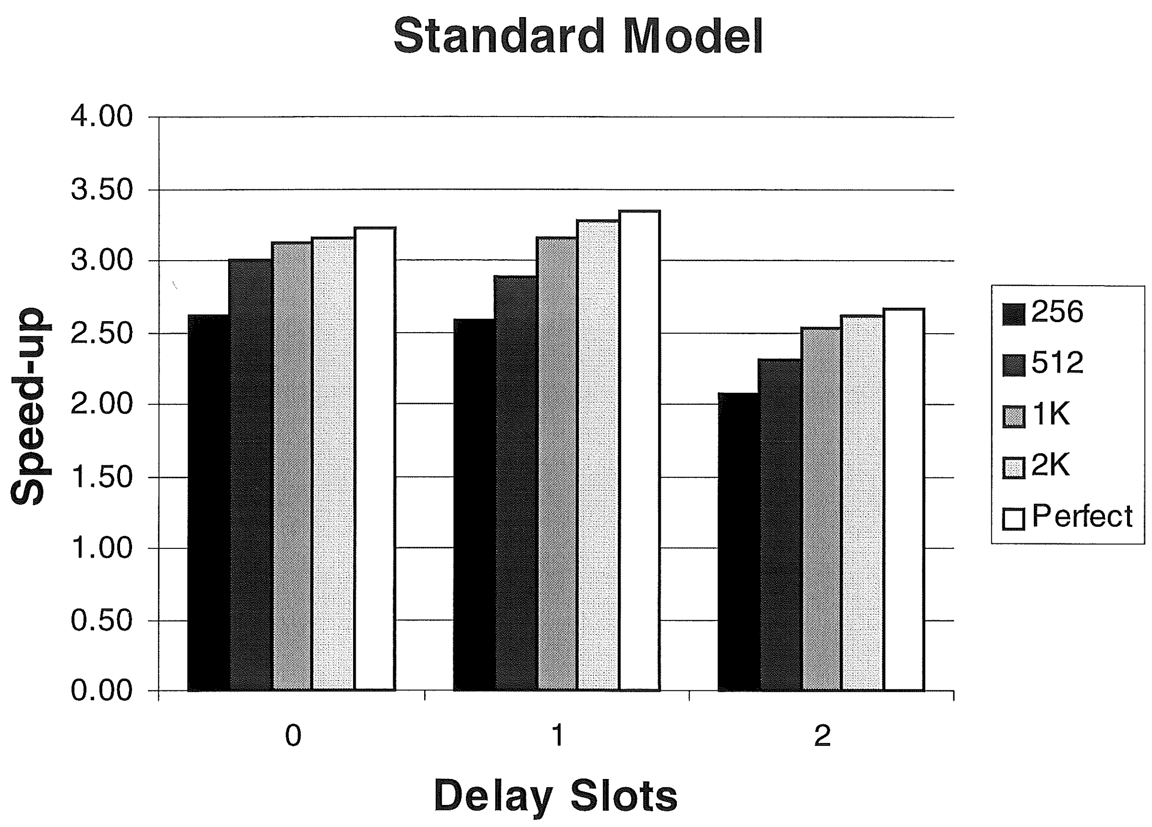

Figure 6 Speed-ups on the Standard Model using variant cache sizes

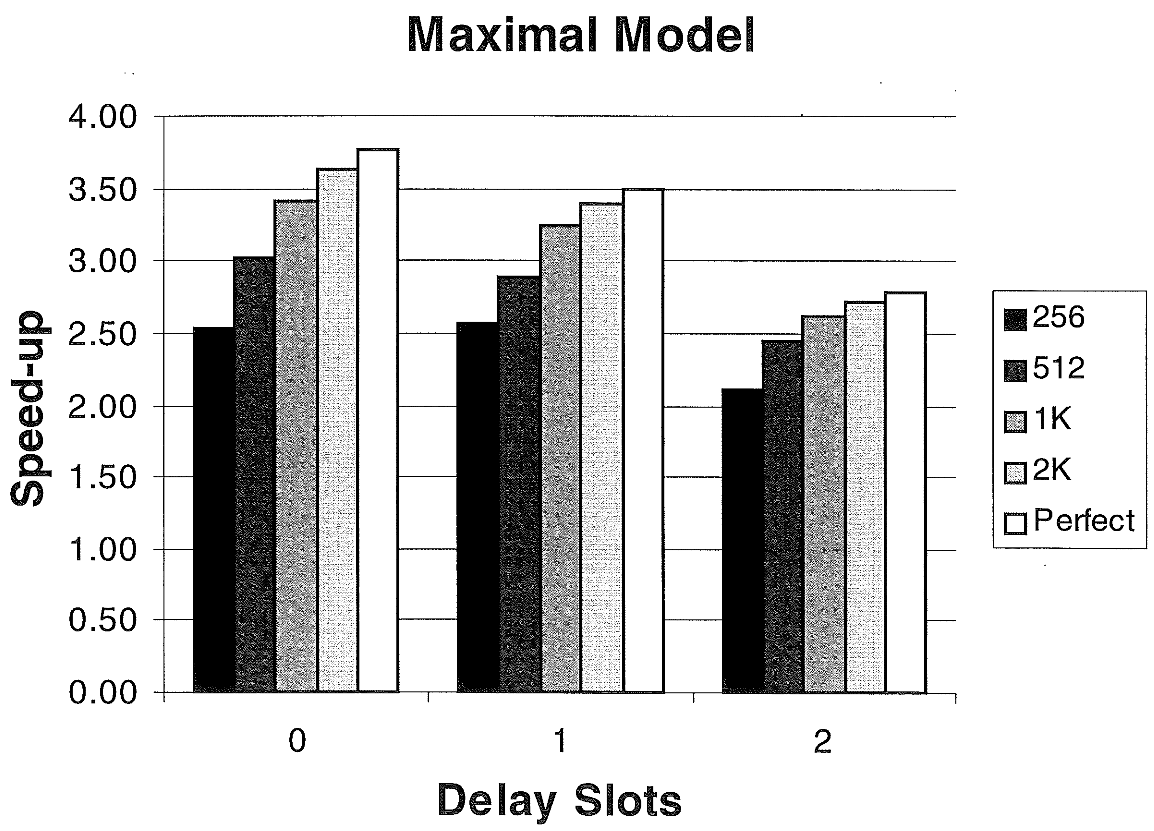

Figure 7 Speed-ups on the Maximal Model using variant cache sizes

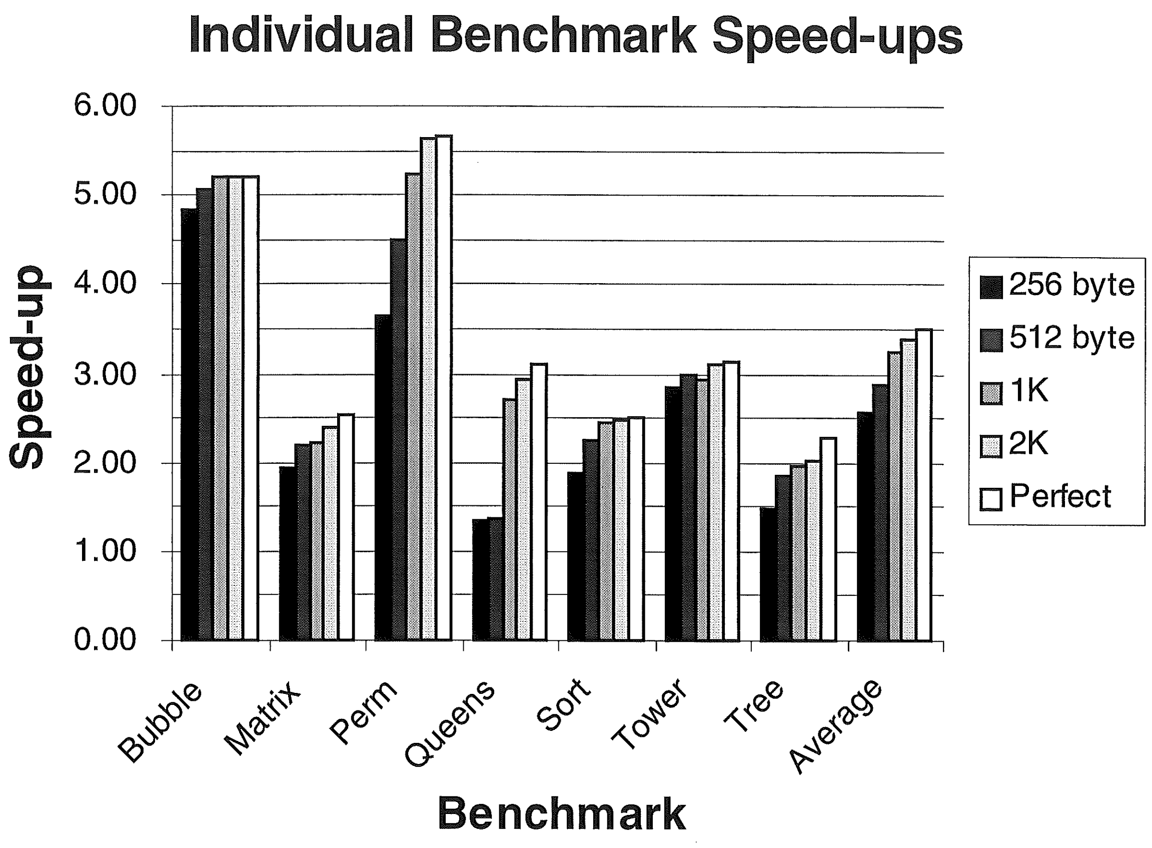

Figure 8 Benchmark speed-ups using the Maximal Model and 1 Delay Slot

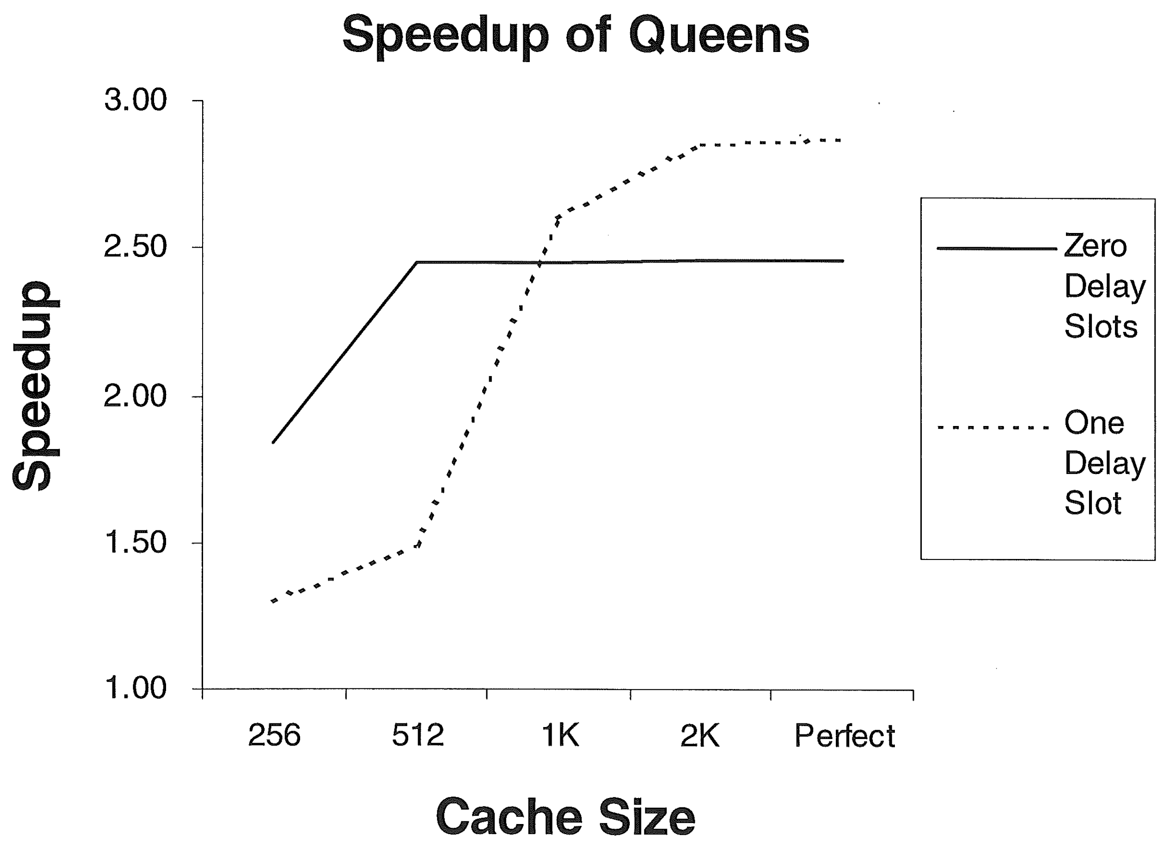

Figure 9 Speedup of Queens with zero and one Delay Slots

Appendix

A. Paper Presented at Euromicro '98 - Sweden

The Impact of a Realistic Cache Structure on a Statically Scheduled Architecture

Daniel Tate D.1.Tate@herts.ac.uk
Gordon Steven G.B.Steven@herts.ac.uk
Paul Findlay P.A.Findlay@herts.ac.uk
University of Hertfordshire, Hatfield, Hertfordshire AL10 9AB, UK.
Telephone: 01707-284-000, Fax: 01707-284-303

Abstract

Memory hierarchy has been accepted as the most limiting factor in current MII processors [1]. *For architectures that employ static instruction scheduling, memory performance is of increasing importance, since the instruction scheduling process tends to increase code size. This paper looks at the impact of a realistic memory hierarchy on a minimal superscalar processor model which uses aggressive static instruction scheduling techniques. The divergent performance impact of a cache on scheduled and unscheduled code is quantified, as well as the resultant effect on overall scheduling speed-up.*

1. Introduction

It is widely believed that the most limiting factor in current architectures is the memory sub-system [2]. Despite this, much of the research into Multiple Instruction Issue (MII) processor design assumes a perfect cache and memory system. There are two diverse schools of thought on MII research: dynamic instruction scheduling relies on complex hardware to uncover instruction parallelism at run-time; static instruction scheduling relies on a specialised compiler to uncover the instruction parallelism at compile-time. In this paper we use the Hatfield Superscalar Architecture (HSA), a statically-scheduled superscalar architecture developed at the University of Hertfordshire to examine performance gains of aggressive static scheduling [3]. It is well known that the code expansion inherent in the scheduling process [4] make statically-scheduled architectures more reliant on cache performance. In this paper we wish to compare the performance impact of a realistic cache structure on scheduled and unscheduled code, and examine the impact of code expansion. Also, we expected cache misses to have a disproportionately high impact on scheduled code, and we quantify the extent of the loss of speed-up.

To facilitate this and future studies, a highly parameterised cache simulator [5] has been developed to model a wide variety of cache structures. The cache simulator is integrated into the Hatfield Superscalar Architecture instruction level simulator [6].

2. The Hatfield Superscalar Architecture

The HSA evolved as a progression from HARP [7], a VLIW processor that was developed and fabricated at the University of Hertfordshire. As with all VLIW architectures, HARP suffered from a pre-determined constant instruction issue rate, and from code expansion. The HSA project attempts to combine the best features of VLIW and superscalar architectures. The problems of incompatibilities within a processor family and fixed VLIW instruction widths are averted by dynamically issuing variable sized instruction groups. The problems of superscalar complexity and a fixed-sized instruction window are avoided by using extensive instruction scheduling at compile-time, combined with in-order instruction issue at run-time. The HSA has therefore been described as a minimal superscalar architecture [8].

The HSA uses a generalised branch delay mechanism [9] to hide the latencies caused by fetching a new stream of instructions after a taken branch. To avoid the introduction of NOPs in empty branch-delay-slots, the number of instructions in the branch delay region is encoded directly into each branch instruction. The instruction scheduler is responsible for filling the branch delay region with useful instructions. However, a recent study [10] describes how a branch prediction mechanism can be added to the HSA. Perfect branch prediction can be simulated using a zero branch-delay-slot model. In this paper we therefore simulate processor models with zero, one and two branch-delay-slots.

3. The cache simulator

Like most high-end computer architecture research, the HSA initially assumed a 100% cache hit rate for memory references. This study removes the simplification, and quantifies the effect of a realistic cache on the speed-ups achieved by the HSA static instruction scheduler.

The non-blocking cache structure was developed to support multiple cache accesses per cycle, see figure 1. Although the primary intention was not to develop a new cache structure, the design does include some novel ideas. A Data Write Buffer is positioned before the cache at each

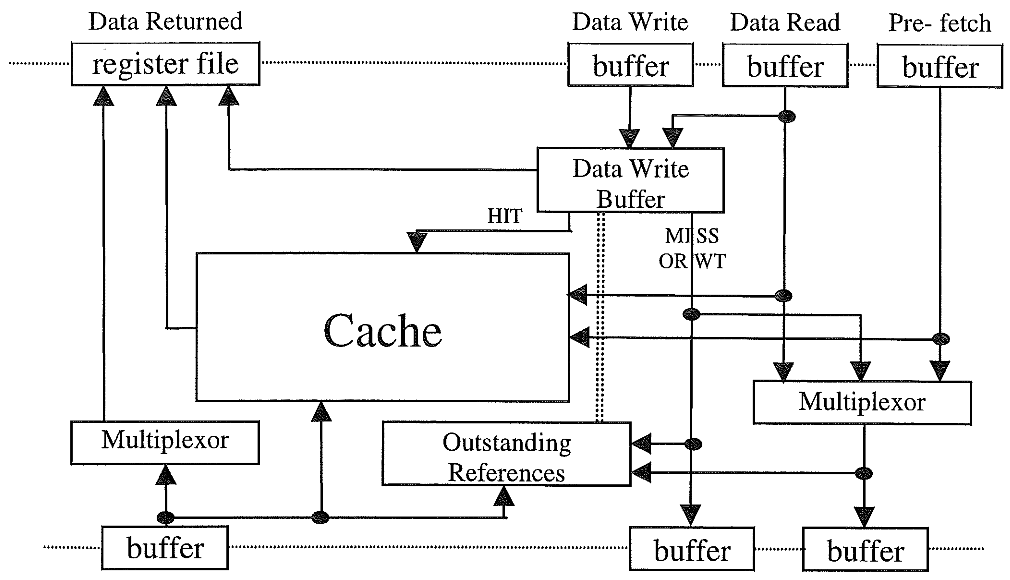

Figure 1 - Basic cache structure

cache level and acts as a small associative store that accepts all writes, and then passes them on to the cache when write ports are free. The write latency of the buffer can be varied, and it can optionally merge writes to the same cache block. The Outstanding References Buffer supports multiple outstanding memory references, while preventing duplicate block requests.

The cache simulator was implemented as a highly parameterised template that allows almost any basic cache structure to be specified. The cache simulator has been successfully designed and implemented as an extension to the HSA processor simulator, and is used in this paper to simulate the effects of a cache structure on a statically-scheduled superscalar architecture.

4. Test parameters

For all tests, eight integer benchmarks were used. Run-times on the baseline model with a perfect cache range between 200 000 and 11 000 000 simulated clock cycles. The successful examination of the impact of a cache relies on the careful selection of the cache sizes. With too large a cache, a benchmark's working set will completely reside in the cache; with too small a cache, the large number of block replacements will completely dominate the results. Four cache models between 512 bytes (128 words) and 4KB (1024 words) were chosen to reflect both the instruction and data working sets of the benchmarks. The tests consisted of all permutations from three branch-delay-slot models, five cache models and three processor models.

Two branch-delay-slot models simulate fast and slow cache access times of one and two cycles. The zero branch-delay-slot model simulates perfect branch prediction. The cache read and write times are 1 cycle for the zero and one branch-delay-slot models, and 2 cycles for the two branch-delay-slot model.

The cache models are all direct-mapped, with a block size of 64 bytes (16 words). No use is made of sub-blocks. The five sizes of instruction and data cache modelled are 512 bytes (8 sets), 1KB (16 sets), 2KB (32 sets), 4KB (64 sets), and a perfect cache.

All cache models have a single cache return (bypass) line. Write policies are 'write back' and 'allocate on write miss'. The instruction cache has a fetch width of 16 instructions. In this study instruction fetches are allowed to cross cache block boundaries. The main memory contains one, non-pipelined, read/write port which can satisfy a single read or write in ten processor cycles.

The three processor models consist of a Baseline, a Standard Superscalar and a Maximal Superscalar model. The Baseline Model simulates a standard RISC processor that fetches and issues one instruction per cycle and contains one functional unit of each type. It executes code generated by our gcc compiler. The Standard Model simulates a high-performance superscalar architecture and contains 16 functional units of each type. The cache structure, however, only contains two data read and two data write ports. The Maximal Model differs only by having 16 data read and 16 data write ports. Both superscalar models execute code that has also passed through the HSA scheduler [3]. All instructions have unit latencies, except for multiply (3) and divide (16).

5. Results

Figures 2 to 4 show the performance of each cache size as a percentage of the perfect cache performance. The figures are calculated with the cycles taken for a run, by:

*100% * (perfect cache / cache of size x)*

Figures 2 and 3 show that with a 512 byte cache, the performance of the superscalar models falls to 40%, while on the Baseline Model, the performance only drops to 75%. Even with a 4KB cache the performance of the superscalar models is only 88%, while the Baseline Model achieves 95%.

Figure 4 examines the Maximal Model with zero branch-delay-slots. It shows that six of the benchmarks correlate to the average very well. The two exceptions are *Bubble*, which maintains 99.9% performance with 1KB caches, and *Tree*, which only manages 35.2% performance with 4KB caches.

Figure 5 shows the speed-ups that the Maximal superscalar model achieves over the Baseline Model. The figures are calculated with the cycles taken for a run, by:

Baseline Model Cycles / Maximal Model Cycles

The range of figures is most significant on the Maximal Model with zero branch-delay-slots. Using a 512 byte cache on both the Baseline and Maximal Models results in a speed-up of only 1.7. In contrast, using a perfect cache on both the Baseline and Maximal Models results in a speed-up of 3.8. Even with a 4KB cache on both models, the speed-up falls 11% to 3.4.

Figure 6 shows the speed-ups for each individual benchmark with the Maximal Model and one branch-delay-slot. *Perm* shows great sensitivity to the cache size; speed-ups range from 2.4 for a 512 byte cache to 8.0 for a perfect cache.

Finally, figure 7 examines the behaviour of *Perm* in more detail by comparing zero and two branch-delay-slots on the Maximal Model.

6. Discussion and conclusions

Reducing the cache size affects the Baseline, Standard and Maximal Models at a different rate. This is shown in figures 2 and 3. The difference in performance degradation between the Baseline Model and the two superscalar models can be attributed to two main factors.

The first factor is that scheduled code places more strain on the instruction cache due to code expansion. Across the eight benchmarks, each scheduled for 6 models, the average code expansion is 80%. This ranges from 9% for *Tower* on the Standard Model with one branch-delay-slot, to 200% for *Tree* on the Maximal Model with one branch-delay-slot.

The second factor is that the HSA scheduler assumes a perfect cache. It therefore schedules memory loads to take one or two cycles, depending on the branch-delay-slot model. However, a cache miss will stall instruction issue until the missing block can be fetched (for at least 10 cycles). This is more significant for an MII model.

Overall the Maximal Model has a slightly greater rate of decline than the Standard Model. This is because the scheduler aggressively fills the 16 load ports, provided by the Maximal Model, with speculative loads. With a 100% cache hit rate these loads improve performance. However, with a realistic cache structure they also significantly increase the number of data cache misses.

Figure 4 examines the two atypical benchmarks (*Bubble* and *Tree*) in more detail. *Bubble* is a linear sort program with a small array as the data set. It therefore has a small contiguous working set. In contrast, *Tree* is a tree sort program with memory allocated to nodes dynamically. It therefore has a large fragmented data set.

With a perfect cache on the Maximal Model, speed-ups of 3.7 (zero), 3.4 (one), 3.0 (two branch-delay-slot model) are achieved through static instruction scheduling. In figure 5, it can be seen that a 512 byte cache reduces the speed-ups given to 1.7, 1.9 and 2.0 respectively. However, with a 4KB cache the performance is only reduced to 88%, the speed-ups recover to 3.5, 3.2 and 2.9 respectively.

We have therefore established that the scheduled superscalar models suffer more from decreased cache size than the Baseline Model. The result of that disparity can be seen in figures 5 to 7.

It has been stated that instruction delivery is the most important problem [2]; these results support this hypothesis. Figure 7 shows a significant performance increase for the zero delay slot model between 1KB and 2KB caches. This is because, with 1KB caches, the instruction and data working sets are not fully located in their respective caches. However, increasing the cache sizes to 2KB cuts almost all block replacements to single figures, indicating that the working sets now fit into the caches.

To conclude, it has been shown that the impact of varying cache sizes are much more severe on scheduled code than unscheduled code, thus having a significant effect on overall speed-up achieved. While increasing cache size recovers most of the lost performance, increasing cache size leads to longer read latencies. This shows the increased need for high performance memory [2] and branch prediction [11] systems, and also pre-fetching techniques [12].

References

[1] Hennessy J L, Patterson D - *Computer Architecture: A Quantative Approach*, Morgan Kaufman, San Francisco, 2nd Edition 1996.

[2] Patt Y N, Patel S J, Evers M, Friendly D H, Stark J - *One Billion Transistors, One Uniprocessor, One Chip*, Computer, September 1997, pp 51-57.

[3] Steven F L, Steven G B, Wang L - *Using a Resource Limited Instruction Scheduler to Evaluate the iHARP Processor* - IEE Proceedings - Computers and Digital Techniques Vol. 142, No.1, January 1995, pp 23-31.

[4] Nicolau A, Fisher J - *Measuring the parallelism available for very long instruction word architectures*, IEEE Trans. Comput. Nov 1984.

[5] Tate D - *A Cache Simulator for the Hatfield Superscalar Project*, Hertfordshire University Technical Report.

[6] Collins R - *Developing a Simulator for the Hatfield Superscalar Processor*, University of Hertfordshire Technical Report No. 172, December 1993.

[7] Steven G B, Adams R G, Findlay P A, Trainis S A - *iHARP: A Multiple Instruction Issue Processor*, Proceedings of the 18th Annual International Conference on Computer Architecture, Toronto, May 1991.

[8] Steven G B, Collins R - *A Superscalar Architecture to Exploit Instruction-Level Parallelism*, Euromicro96, Prague, September 1996.

[9] Steven F L, Adams R G, Steven G B, Wang L, Whale D J - *Addressing Mechanisms for VLIW and Superscalar Processors*, Microprocessing and Programming, Vol.39, Numbers 2-5, December 1993, pp 75-78.

[10] Egan C, Steven F L, Steven G B - *Delayed Branches Verses Dynamic Branch Prediction in a High Performance Superscalar Architecture*, Euromicro97, Budapest, September 1997.

[11] Rotenberg E, Bennett S, Smith J E - *Trace Caches: A Low-Latency Approach to High-Bandwidth Instruction Fetching*, Proceedings of the 29th Annual International Symposium on Microarchitecture, Los Alamitos, California, 1996, pp 24-34.

[12] Kroft D - *Lockup-free instruction fetch/prefetch cache organisation*, Proc. 8th Annuu. Int. Symp. on Computer Architecture, June 1986, pp 404-413.

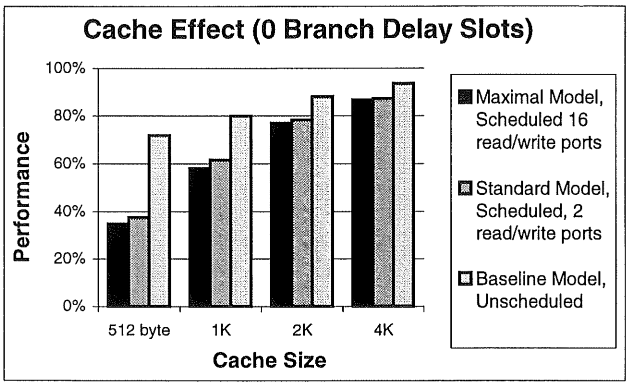

Figure 2 - Cache impact using zero-delay-slots

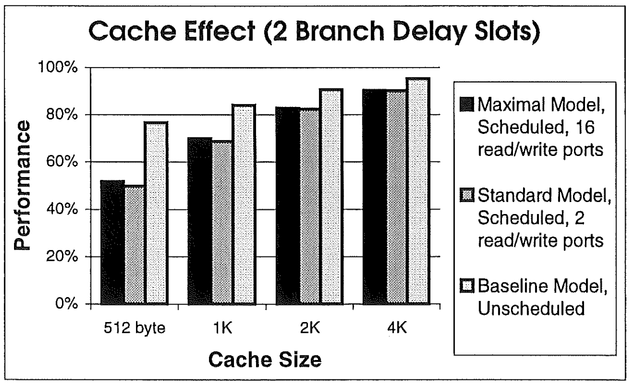

Figure 3 - Cache impact using two-delay-slots

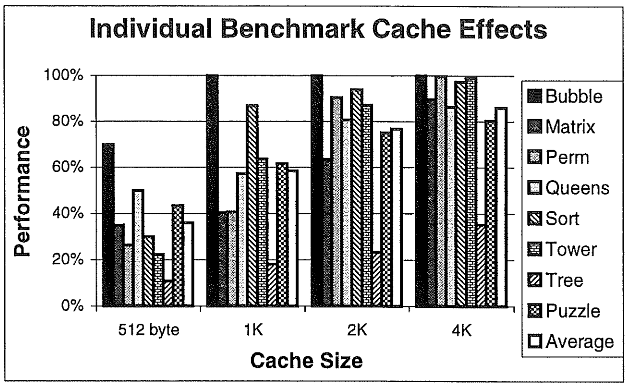

Figure 4 - Impact on Maximal Model, 0-delay-slot

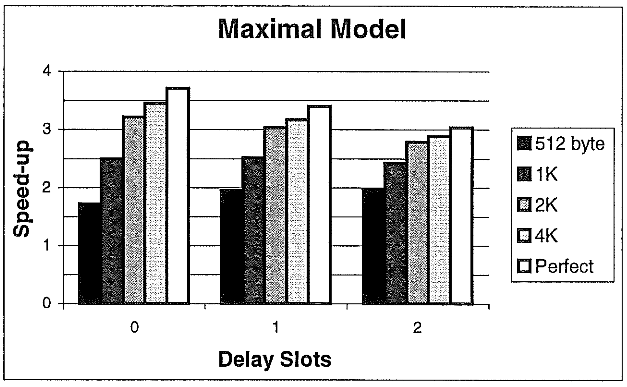

Figure 5 - Speed-ups for the Maximal Model

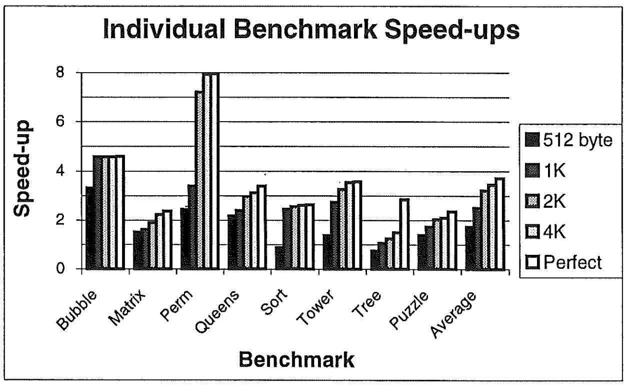

Figure 6 - Maximal Model speed-ups, 1-delay-slot

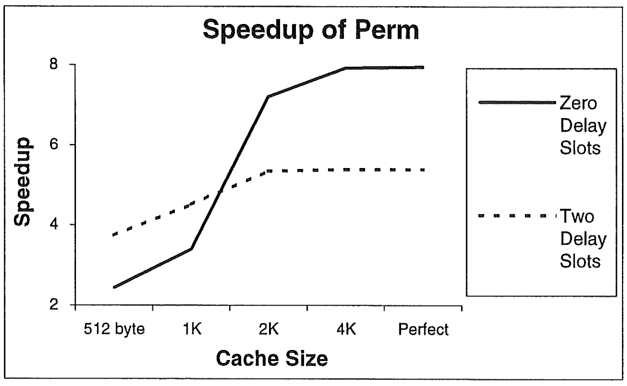

Figure 7 - Speed-up of Perm (Maximal Model)