DIVISION OF COMPUTER SCIENCE

Instruction Scheduling for a Superscalar Architecture

R Collins
G B Steven

Technical Report No.248

September 1996

Instruction Scheduling for a Superscalar Architecture

Roger Collins and Gordon B Steven
University of Hertfordshire, Hatfield, Hertfordshire AL10 9AB
Telephone: 01707 284319 Fax: 01707 284 303 email: comqgbs@herts.ac.uk

Abstract

It is increasingly accepted that superscalar processors can only achieve their full performance potential through compile-time instruction scheduling. This paper presents preliminary performance results using a Conditional Group Scheduler which targets the HSA processor model developed at the University of Hertfordshire. In particular, we show that guarded instruction execution improves performance by allowing the processor to squash instructions in the Instruction Buffer before they are issued to functional units and enables the scheduler to delete a significant number of branch instructions.

1. Introduction

Currently available superscalar processors issue up to four instructions in each processor cycle. Even higher issue rates are expected in the future. To exploit these issue rates, a processor must identify groups of independent instructions that can safely be issued and executed in parallel. Unfortunately, the processor itself has only a local view of the dynamic instruction stream it is executing. As a result, even with advanced hardware techniques such as dynamic branch prediction and out-of-order instruction issue, the amount of parallelism that can be realised is limited and may be as low as a factor of two [1].

An alternative approach is to use an instruction scheduler to reorganise code into independent groups at compile time. The processor then issues these pre-formed groups in parallel at run time. Unlike the hardware, an instruction scheduler can take a global view of a program and is therefore able to assemble large groups to satisfy high instruction issue rates. As a result future high performance superscalars are likely to rely increasingly on instruction scheduling to realise their full potential.

The Hatfield Superscalar Project aims to develop the instruction scheduling technology to support a high instruction issue rate. The research is based on the Hatfield Superscalar Architecture (HSA) that has been developed at the University of Hertfordshire. The long term aim of the project is to realise an order of magnitude speedup over traditional RISC processors that issue only one instruction per cycle. Further objectives include the development of heuristics to avoid excessive code expansion during instruction scheduling and the development of hardware mechanisms, such as guarded instruction execution, to support high issue rates.

This paper presents preliminary results generated using the first instruction scheduler developed for HSA. In particular, we measure the speedup achieved by different processor models and evaluate the impact of running code scheduled for one processor on alternative processor models. We also examine the impact of multi-cycle cache access times and of using guarded instruction execution to remove instructions prematurely from the instruction pipeline. Finally, we quantify the ability of the scheduler to remove branches during scheduling.

2. The HSA architectural model

HSA is a load and store architecture with a RISC instruction set derived from our earlier HARP project [2]. Separate integer and Boolean register files are provided. The one-bit Boolean registers are used to store branch conditions and to implement guarded instruction execution. Functional units include arithmetic, relational, shift, multiply, memory reference and branch units. A simple four-stage pipeline is used:

- IF: Instruction Fetch
- ID: Instruction Decode and register fetch
- EX: Execute
- WB: Write Back

In the first stage a fixed number of instructions is fetched from the instruction cache into an Instruction Buffer. One or more processor cycles may be required. In the case of multiple cycle fetches it is assumed that the cache accesses are pipelined and that a new instruction access can begin in each cycle. In the instruction decode stage one or more instructions are issued to functional units. Instructions then spend a variable number of cycles in the execution stage before returning results to a register file in the write-back stage.

The Instruction Buffer decouples instruction fetch from instruction issue. Typically, the fetch rate of an HSA processor exceeds the maximum issue rate to allow the processor to benefit from its ability to remove or squash instructions from the Instruction Buffer before they are issued to functional units. HSA always issues instructions to functional units in program order. There is little to be gained from out-of-order issue if the instruction

stream has already been re-ordered for parallel instruction issue by an instruction scheduler. In contrast, many superscalar designs use scoreboarding or Tomasulo's algorithm to provide out-of-order instruction issue. HSA avoids this complexity and the resultant pressure on processor cycle time.

Branches are resolved in the ID stage. With a single cycle instruction cache the branch delay is therefore one. Load and store instructions use register indirect addressing or the ORed indexing addressing mechanism developed for HARP [3]. These simple addressing mechanisms allow memory addresses to be made available at the end of the ID stage and avoid a load delay with a single cycle data cache.

Other major features of HSA include a generalised delayed branch mechanism, guarded instruction execution and hardware support for speculative instruction execution.

2.1 Delayed branch mechanism

In a RISC pipeline, branches are typically resolved in the second pipeline stage giving a branch delay of one. This latency is often hidden by using a delayed branch mechanism in which a fixed number of instructions following the branch is always executed, irrespective of the outcome of the branch.

The classic delayed branch mechanism is too inflexible for a superscalar architecture. HSA therefore generalises the traditional mechanism by allowing each branch instruction to specify explicitly the number of instructions that must be executed in its branch delay slots [4]. The flexibility provided by this mechanism allows it to adapt to different instruction issue rates and to different branch latencies. Compatibility is ensured as long as each processor can execute the new branch instructions. Any HSA processor can therefore execute code scheduled for any other HSA processor.

In contrast some recent superscalar architectures have abandoned the delayed branch mechanism [5,6] in favour of using a branch target cache (BTC) to predict the outcome of branches. While a perfect BTC will result in no performance degradation, in practice there is a performance penalty proportional to both the BTC miss rate and the instruction issue rate. As a result, as the issue rate of superscalar processors increases, the performance penalty of using a BTC will also increase. Using a BTC also involves a significant increase in hardware complexity. In addition to the cost of the BTC itself, the processor must be able to recover rapidly from incorrect branch predictions by squashing any instructions that have been speculatively issued after the branch prediction.

2.2. Guarded instruction execution

Guarded or conditional instruction execution has been proposed by a number of researchers [7,8] and has been implemented in several processors including the Acorn ARM [9] and the HARP processor [2] developed at the University of Hertfordshire. All HSA instructions are guarded. Guarded execution is implemented by associating one or more Boolean guards with each instruction. For example consider:

```
TB1 ADD R1, R2, R3
```

The addition will only return a result to R1 if the value held in Boolean register B1 is true at run time. The Boolean values themselves are generated by relational instructions that return a Boolean value to one of the Boolean registers.

Guarded instructions provide a simple mechanism for percolating instructions into the preceding basic block. For example consider:

```
EQ B1, R1, R2      /* B1 := (R1 = R2) */
BF B1, label       /* branch if B1 = false */
ADD R6, R3, R4
```

The instruction scheduler can always safely move the ADD instruction in parallel with the branch instruction by attaching the guard condition TB1:

```
EQ B1, R1, R2
BF B1, label;      TB1 ADD R6, R3, R4
```

Guarded execution also allows the scheduler to remove some conditional branch instructions. For example, consider the following code that has been generated for an *if then* construct:

```
NE B6, R1, R15     /* B6 := (R1 ≠ R15) */
BF B6, continue    /* branch if B6 = false */
ADD R1, R2, R3     /* then-code */
continue:
```

Scheduling produces the following code:

```
NE B6, R1, R15
TB6 ADD R1, R2, R3;
```

Branch removal is particularly important in superscalar processors that predict the outcome of branch instructions dynamically, since removing branches from the program will also reduce the number of branch prediction failures.

Another advantage of guarded execution is that the pressure on functional units and other processor resources can be reduced. Consider the following example with three parallel instruction groups:

```
BT B1, label (#6);  TB1 instr1;  FB1 instr4
TB1 instr2;  FB1 instr5
TB1 instr3;  FB1 instr6
```

Two branch delay slots are assumed and six instructions following the branch must be executed before the branch is taken. All six will be issued to functional units but three of them will be squashed in their functional units without returning a result to a register. Consequently the pressure on result buses and register file write ports will be reduced.

Instructions can also be squashed in the Instruction Buffer [10]. HSA will squash an instruction in the ID stage, if the relevant Boolean guard becomes available and the instruction has remained in the Instruction Buffer for a full cycle without being issued. In the above example, the instructions in the two branch delay groups satisfy these conditions; so only two of the final four instructions need

be issued to functional units.

3. Instruction scheduling

HSA relies on compile-time instruction scheduling to achieve high execution rates. Scheduling techniques were originally developed to pack independent operations into microinstructions to reduce the size of microprograms and to enable them to execute more quickly. Recognition that limited parallelism was available between branches led to attempts to schedule code globally. Initially efforts were directed towards combining basic blocks to create large groups of instructions that could then be scheduled using the techniques that had been developed to compact microcode.

One of the first global scheduling techniques developed was Trace Scheduling [11]. In Trace Scheduling, the scheduler identifies a series of paths or traces through a procedure that are highly likely to be followed at run time. The most important trace is then selected and scheduled as if it were a single basic block. Finally, code is added at the entry and exit points of the scheduled trace to preserve the program semantics. This process is repeated until all the traces in a procedure have been scheduled.

While spectacular speedups were achieved with highly numeric programs, Trace Scheduling has a number of limitations. Firstly, there is the problem of identifying traces that are likely to be followed at run time. Accurate trace identification depends on knowing the likely outcome of branches and, in general, this information can only be obtained by running a program and collecting the required branch statistics. To reduce this difficulty, loops are often unrolled to obtain a long trace consisting of several loop iterations. Secondly, the execution time of the trace path is optimised at the expense of off-trace paths, which may be slowed down. Thirdly, the addition of semantic preserving code together with aggressive loop unrolling can lead to dramatic code expansion. Finally, and perhaps crucially, instructions will usually only be executed in parallel if they were originally in the same trace. Branches off and into a trace therefore become barriers to code motion. In the case of loops, instructions from one loop iteration will never be overlapped with instructions from a subsequent iteration to achieve so-called software pipelining. This disadvantage can be partially offset through aggressive loop unrolling, but only at the cost of dramatic code expansion.

Professor Hwu's group at Illinois University also increases the scope for instruction scheduling by combining basic blocks to form larger scheduling units called superblocks [12]. A superblock consists of a series of basic blocks with a single entry point but multiple exit points. Superblocks of maximum size are systematically created through basic block duplication called Tail Duplication. They are then scheduled as single enlarged basic blocks.

Superblocks provide larger scheduling units and additional speedup at the cost of some additional code duplication. Although speedups between 2 and 7 are reported with an eight instruction-issue processor model [12], superblocks retain one of the major disadvantages of Trace Scheduling. Superblocks, like Traces, erect artificial boundaries to code motion, that prevent code in one superblock from ever being scheduled for parallel execution with code from a different superblock. Similarly, code motion across loop back edges is precluded. Trace Scheduling and the formation of superblocks therefore both remove the barriers to code motion between basic blocks, only to create new barriers between either traces or superblocks. As a result, neither technique, on its own, can ever realise the full-potential of instruction-level scheduling.

A more attractive approach is to move or percolate instructions systematically between basic blocks, duplicating code where necessary to preserve program semantics. The code motion primitives involved were first codified by Fisher [11] who viewed branches as fixed points in the code and allowed non branch instructions to flow in both directions across branch edges. Nicolau [13] describes alternative primitives where all code motion is in an upwards direction and where all instructions, including branches, move upwards past both branch and non branch instructions. Downwards code motion is therefore replaced by upwards motion of branch instructions. For our purposes, Nicolau's primitives are the more useful since they directly support algorithms in which operations are consistently percolated upwards until every operation is executed at the earliest opportunity.

Enhanced Percolation Scheduling [14] is an elegant scheduling algorithm that has been developed by Kemal Ebcioglu's group at IBM. Innermost loops are scheduled first, followed by the surrounding code. Starting with the first instruction in a loop, parallel instruction groups are constructed by searching forward through the code for instructions to add to the group. All possible paths are searched in turn and suitable candidates are moved forward into the current group using modified versions of Nicolau's code motion primitives. Where necessary, compensation code is added off the percolation path. When the current instruction group is full, or there are no further candidates to consider, immediately following instructions are used as the basis for further parallel groups. The algorithm therefore recursively fills instruction groups on all possible control paths through the loop.

The execution of successive loop iterations is overlapped by allowing code motion across loop back edges. Percolation across a back edge requires a copy to be inserted ahead of the loop to ensure correct program operation, so a loop prologue is also generated automatically. As a result, Enhanced Percolation Scheduling achieves software pipelining with arbitrarily complex loops. Furthermore, providing there are neither resource restrictions, nor loop carried dependencies, a new loop iteration can be initiated in every processor cycle. This is the best that can be achieved without loop replication.

IBM apply Enhanced Percolation Scheduling to an unconventional VLIW architecture with uniform unit

instruction latencies and achieve speedups between 2.6 and 10.6, with a geometric mean of 5.1 [14]. The IBM model assumes that each VLIW instruction can be fetched and used to generate the immediately following instruction address in a single processor cycle. Although the processor only has to select one of several alternative addresses embedded in each instruction, Ebcioglu estimates [8] that the processor cycle time will be degraded by 30%. However, in the case of a two-cycle cache access time, the penalty would be over 100%. A major objective of the HSA project is to achieve similar speedups using more realistic processor models with multi-cycle cache access times and non unit latencies for the more complex instructions such as multiply and divide.

4. A conditional group scheduler

HSA uses a Conditional Group Scheduler (CGS) to reorganise code for parallel execution at run time. The primary goal of CGS is to achieve the maximum speedup while minimising code expansion. A secondary aim is to allow code scheduled for one processor to run on different processor models with minimal performance loss.

CGS [10] is a global code scheduling algorithm that applies powerful low-level mechanisms within the framework of a high-level algorithm. The high-level algorithm determines the order in which sections of code are to be scheduled and which low-level mechanisms are to be enabled during the scheduling phases. CGS provides two sets of parameters that significantly effect the overall scheduling algorithm. One set of parameters controls the individual low-level scheduling features applicable for the duration of the entire scheduling process. A second set determines the type and scope of code motions that are allowed during each individual scheduling phase.

The basic CGS scheduling unit is loop. However, branch instructions impose an unnecessary lower bound on the time required to execute each loop body. For example, with a branch delay of two, the minimum loop length is three instruction groups, and no loop iteration can start less than three cycles after the previous iteration. This limit on the Iteration Interval can only be removed by replicating the loop body.

To minimise code expansion, CGS only replicates loops if the loop execution time can be reduced. This is achieved by first compacting the loop body into the minimum number of instruction groups. Then if the natural length of the scheduled loop is less than the total execution time of a branch instruction, the loop body is duplicated. Guarding replicated loops avoids increasing the number of conditional branches in the loop [4]. The replicated loop body is then rescheduled to combine code from different iterations. As a result, assuming a branch delay of two, a loop body with only one instruction group will be replicated twice to give an Initiation Interval of one, while a loop body with two instruction groups will be replicated once giving an Initiation Interval of at most two.

The scheduling process is divided into four phases. In the first phase each instruction group is filled in turn, starting at the head of the loop. Instructions are moved into each group by recursively searching through the loop control structure for suitable candidates. Software pipelining is achieved by allowing instructions to be moved across loop back edges from previously scheduled groups. To preserve program semantics, instructions moved across loop back edges are also added to a prelude ahead of the loop.

If no restrictions are placed on code motion across loop back edges, the scheduler performs excessive code motion, eagerly pre-computing successive values on the faster paths through the loop, sometimes many iterations ahead of the point where the values are used. Successive values of the loop count, for example, can often be pre-computed many cycles ahead without speeding up the computation. To avoid the resultant code explosion, a number of heuristics were developed. Firstly, code motion across a loop back edge is terminated as soon as the scheduler is unable to move an instruction into the instruction group currently being assembled. The objective of this restriction is to synchronise the operations involved in each iteration and to avoid premature computation of values many iterations ahead. Code motion that requires the duplication of instructions within the loop or renaming of any previously scheduled instruction is also prohibited in this phase.

Although the first scheduling phase minimises the number of instruction groups occupied by the loop body, a lower bound is set by the delayed branch mechanism. As a result, the code will be distributed across all the available instruction groups, including any branch delay groups. A loop with a natural size of one or two may therefore be distributed across three instruction groups, including the two branch delay groups. The second scheduling phase therefore compacts the code groups at the top of a loop and allows the minimum size of the loop to be determined. During this phase code motion across the loop back edge is disabled. At the end of the second phase, loop replication is performed if the natural length of the loop is less than the total latency of the branch instructions.

In the third phase, the replicated loop body is rescheduled with code motion across loop back edges enabled once more. This phase allows code from multiple loop bodies to be overlapped. If no code replication has occurred virtually no code motion takes place in this phase. In the final phase, code is compacted again with the back edge disabled. Earlier limitations on code replication and register renaming can now be safely removed in the absence of code motion across back edges.

Scheduling proceeds from inner to outer loops until each procedure is scheduled. During the later stages, code moves both into and across previously scheduled inner loops, providing the execution time of a previously scheduled loop is not increased. Finally, after all the procedures in a program have been scheduled, code is percolated across procedure calls. This code motion halts as soon as one of the instruction groups in the called procedure fails to move successfully into the calling

routine. This allows the program semantics to be preserved by simply adjusting the procedure entry point.

5. Experimental results

This section presents results that demonstrate the performance gains to be had by combining a powerful scheduling algorithm with a simplified but sophisticated superscalar architecture. Eight general-purpose programs known collectively as the Stanford Integer Benchmark Set [Table 1] are used throughout. All of the test programs are written in 'C' and compiled by a GNUCC generated compiler that targets the HSA instruction set. After scheduling, the programs are executed on a highly parameterised HSA simulator [10].

Table 5.1 Stamford Benchmark Set

1. Bubble:	Bubble sort.
2. Matrix:	Matrix multiplication
3. Perm:	Permutes 7 elements.
4. Puzzle:	A cube packing problem.
5. Queens:	The 8 queens chess problem.
6. Sort:	Quicksort .
7. Tower:	Towers of Hanoi problem.
8. Tree:	Binary tree sort .

The dynamic instruction distribution of the benchmarks is unremarkable with 40% arithmetic, 30% load & store, 13% relational and 17% branches. However, the distribution of the branch instructions is unusual with 26% of all branches being procedure calls and returns. This distribution reflects the highly recursive nature of three of the programs: *perm, tower* and *tree*.

Two versions of the HSA architecture are compared, a Slow Cache Model and a Fast Cache Model. In the Slow Cache Model, both the instruction and data cache require two cycles to perform a read operation. As a result all branch instructions have two branch delay slots. Also, data loaded from the cache by a load instruction can not be used by an immediately following instruction without introducing a stall of one cycle. The load delay is therefore one. In the Fast Cache Model, the cache access time is reduced to one cycle, giving a single branch delay slot and eliminating the load delay. In both models multiplication requires three cycles and division 16 cycles.

Many of the results involve measurement of the overall speedup of scheduled code over unscheduled code. A Baseline Model is provided by running unscheduled code through a single-instruction-issue version of the simulator and recording the total number of machine cycles required. Since the compiler makes no attempt to fill branch delay slots, the Baseline Model effectively predicts that all branches are not taken and incurs a branch penalty whenever a branch is taken.

5.1. Unscheduled code performance

Since HSA is a superscalar rather than a VLIW architecture, some speedup will be achieved when unscheduled code is run on a multiple-instruction-issue model. The speedups obtained with the Fast Cache Model are given in Figure 1. With an issue rate of eight a 60% speedup is achieved, reducing to 34% with an issue rate of two. These figures conceal significant variation between individual programs. *Perm*, for example, speeds up by 118% while *puzzle* only improves by 36%.

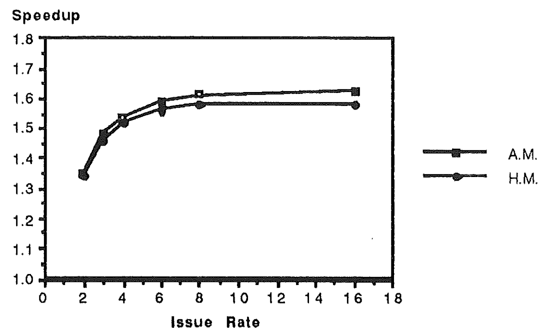

Fig. 1 Speedup for unscheduled code

5.2. Maximum parallel performance

Instruction scheduling results in a further significant performance improvement. With the Slow Cache Model, the overall arithmetic mean speedup is 3.1 with a harmonic mean of 2.9 [Fig 2]. Again the average figures conceal major variations between individual programs, with speedups varying between 2.3 and 5.0. The low figures for *matrix* and *tree* reflect the high proportion of long latency operations in these programs.

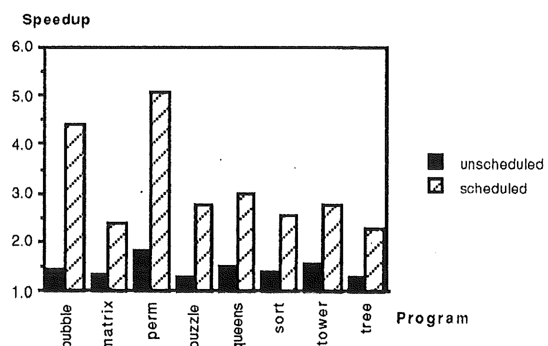

Fig. 2 Speedup for slow cache model

With the Fast Cache Model speedups range from 2.4 to 6.2 with the arithmetic mean increasing to 3.6 and the harmonic mean to 3.2 [Fig 3]. These speedups are 60% better than those achieved on the HARP project with identical benchmarks and cache latencies and reported at Euromicro94 [15].

Finally, if all instruction latencies are reduced to one, the arithmetic mean speedup is increased to 3.9 and the harmonic mean to 3.6, a 24% improvement over the slow cache model. This difference emphasises the dangers inherent in using over simplified processor models.

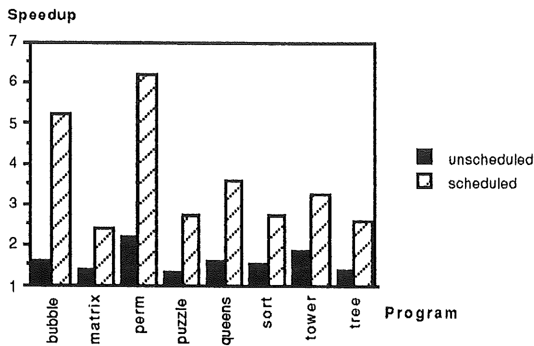

Fig. 3 Speedup for fast cache model

These performance improvements are achieved at the cost of increasing the static code size [Fig 4]. The mean code expansion is a modest 1.9 for the Fast Cache Model, increasing to a factor of 2.3 for a three cycle cache model.

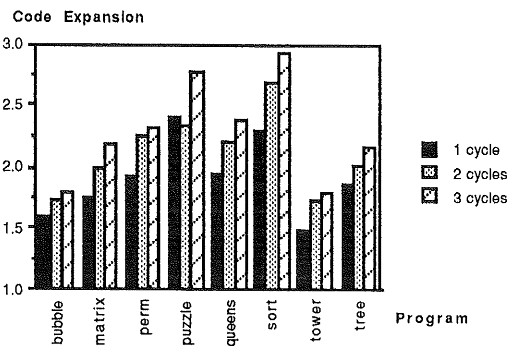

Fig. 4 Code size vs. cache access time

5.3. Impact of issue rate on performance

The results in the previous section assumed processor models with infinite resources. Figure 5 shows how the performance of the Fast Cache Model degrades when code scheduled for a machine with no resource constraints is run on a model with progressively lower instruction issue rates.

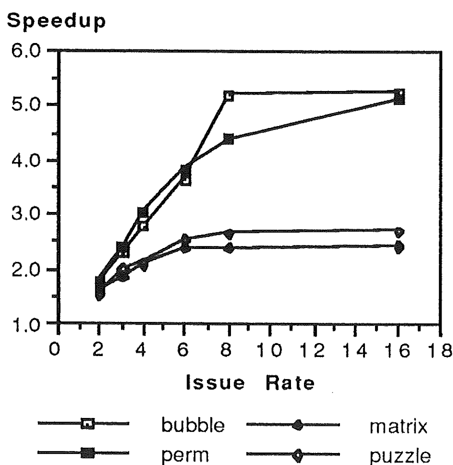

Fig.5a Speedup vs issue rate (fast cache]

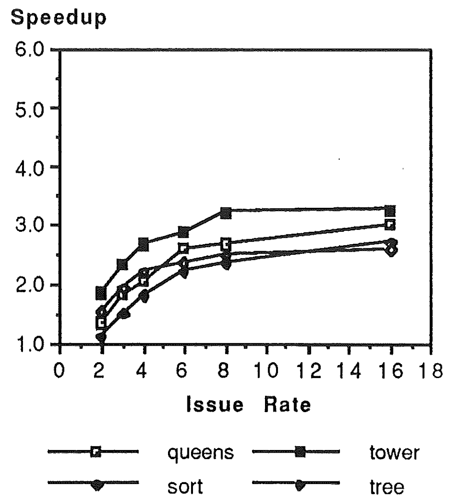

Fig. 5b Speedup vs issue rate (fast cache]

In general, performance degrades gracefully as the issue rate is reduced. A mean speedup of 3.0 is maintained with an issue rate of eight and the speedup is still 2.0 with the highly restrictive issue rate of three.

This graceful degradation in performance as the issue rate is reduced can be partly attributed to the ability of an HSA processor to squash code in the Instruction Buffer. Typically, guarded code will be scheduled into branch delay slots from both successor paths. Squashing allows much of this code to be removed from the Instruction Buffer before it is issued to a functional unit. Squashing therefore becomes increasingly beneficial as the instruction issue rate is reduced.

The same code was re-run with squashing in the Instruction Buffer disabled. With low issue rates, some programs now have speedups less than one [10]. In these cases the scheduled code is running more slowly than the unscheduled code. Over-optimistic code promotion has resulted in time being wasted sequentially executing instructions whose results were never used. With higher issue rates, these speculative results were executed in parallel and therefore did not degrade performance.

Figure 6 compares the speedups obtained for the benchmark programs with and without the ability to squash code in the Instruction Buffer.

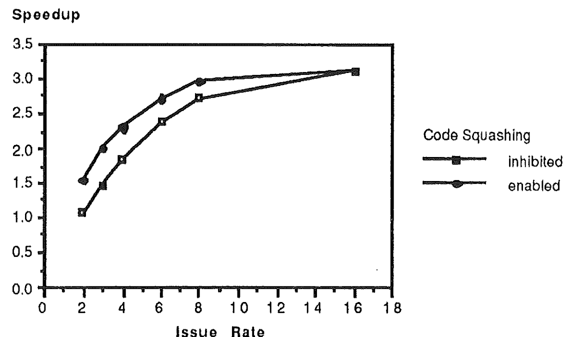

Fig. 6 Effect of disabling squashing

These figures confirm that squashing becomes progressively more important as the issue rate is reduced, with a 44% improvement in speedup for the 2-issue model. In general, implementations require a 50% increase in processor resources to compensate for an inability to squash instructions in the Instruction Buffer.

5.4. Scheduling for specific issue rates

Up to now we have looked at the speedup of code scheduled for infinite resources. Although squashing allows this code to perform reasonably well as the issue rate is reduced, additional speedup can be achieved by scheduling code for each issue rate [Fig 7]. Code scheduled for a two issue model is 18% faster, while code scheduled for an issue rate of eight achieves 94% of the speedup realised with infinite resources.

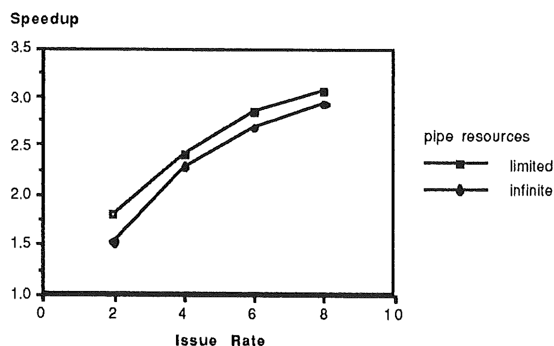

Fig. 7 Scheduling for fixed issue rates

The reduction in static code size is more dramatic than the impact on speedup [Fig 8]. Code size is reduced by 6% for an issue rate of eight and 33% with an issue rate of two. Scheduling code without regard for resource restraints therefore results in significant over scheduling at low issue rates.

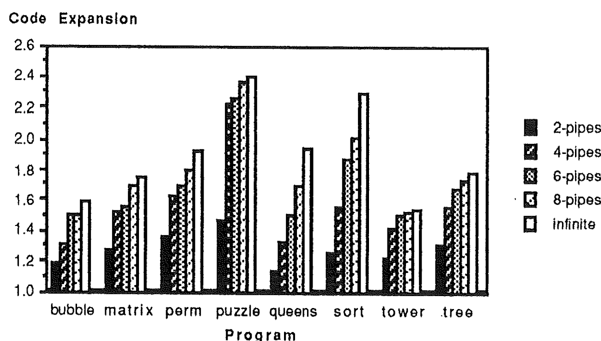

Fig. 8 Reducing the code expansion

5.5 Reducing the cost of branches

An important objective of the HSA project is to use instruction scheduling to minimise the impact of multi-cycle cache access times. Further speedups were therefore measured as a function of the Instruction Cache access time. The Data Cache access time was held constant at one cycle to avoid any impact on the results from increased load latencies.

Figure 9 shows two sets of harmonic mean speedups for the scheduled benchmarks. The first set indicates the speedup achieved by scheduling code for each model, while the second set normalises the speedups with respect to the Fast Cache Model. While the first set of figures suggests that CGS performs well as the Instruction Cache latency is increased, the normalised figures reveal that the total execution time does increase significantly.

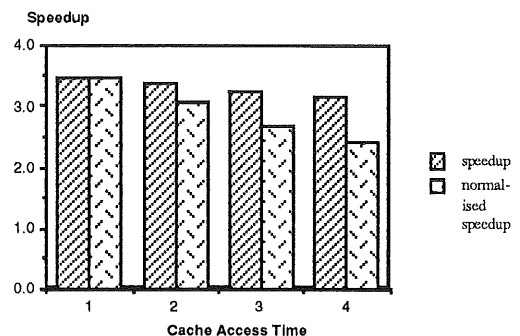

Fig. 9 Speedup vs cache access times

This loss of performance is caused by the schedulers inability to fill all of the branch delay slots as the instruction cache access time is increased. However, detailed examination of the scheduled code suggests that much of the performance loss is caused by the inability of the scheduler to deal adequately with all types of branches. In particular, CGS currently makes no attempt to promote conditional branches into the scope of other conditional branches or to replicate conditional branches. As a result these figures should improve as CGS is enhanced.

5.6. Branch removal

Finally, it is interesting to examine the total numbers of each type of instruction that are executed in the scheduled code and to compare these figures with those taken for runs of the unscheduled programs. With the Fast Cache Model, although the total number of instructions executed increases by 23%, the number of branch instructions executed actually falls by 35%. As a result the percentage of executed instructions which are branch instructions falls from 17% to 10%. The reduction is particularly dramatic in *perm* where 17327 conditional branches are reduced to just five. Similar results were observed with the Slow Cache Model. The total instructions executed increasing by 32%, while the number of branches executed fell by 31%.

6. Concluding remarks

These results demonstrate the validity of the HSA model and show that compile-time scheduling can achieve significant speedups whilst retaining code compatibility across a range of processor implementations. By transferring much of the superscalar's work to the scheduler, the processor has also been greatly simplified. Nonetheless, our minimal superscalar design is still able

to find significant levels of parallelism without the aid of a scheduler, achieving a speedup of 1.6.

Currently scheduling realises speedups in the range 2.9 to 3.6 depending on the processor model. These figures are 60% better than those achieved on our earlier HARP project and compare favourably with results reported by other groups.

We also show that code scheduled for a processor with infinite resources can achieve satisfactory performance over a wide range of processor models, with performance degrading gracefully as the issue rate is reduced. Much of this graceful degradation is achieved through HSA's ability to squash instructions in the Instruction Buffer. In general, implementations require a 50% increase in processor resources to compensate for the ability to squash instructions. Nonetheless maximum speed is achieved by scheduling code for a specific processor.

Figures are also presented showing the impact of cache access time on speedup, with a 25% degradation as the cache access time is increased to three cycles. Future work with CGS will attempt to reduce this figure.

Finally, scheduling was found to reduce the number of branches executed by over 30%. Similar results are reported by Hwu's group which used conditional execution specifically to remove branches and achieved a 27% reduction in dynamic branch counts [16]. In contrast, CGS reduces the number of branches as a side effect of instruction scheduling. These results suggest that guarded execution can significantly improve the performance of Branch Target Caches in a superscalar processor.

The initial thrust of our scheduling work was to realise ever greater parallelism. However, it was found that the unrestrained application of CGS's powerful code motion primitives demanded excessive resources and, on occasion, even slowed up program execution. Restraining mechanisms were therefore developed to avoid code explosion. An important conclusion of this work is therefore that the application of code motion primitives must always be tempered by heuristics which take into account the likely costs and benefits of the code motion.

Many questions regarding instruction scheduling remain unanswered. We still do not know much parallelism can be realised through instruction scheduling. Although the speedup achieved by CGS is encouraging, our scheduler is still incomplete in many respects. Furthermore, analysis using trace driven simulation [17] suggests that significant additional parallelism is theoretically available. Further work is also required to quantify the benefits of guarded execution and instruction squashing. It is therefore likely that our scheduler will be continually enhanced for some time to come.

REFERENCES

1 Johnson M *Superscalar Microprocessor Design* Prentice Hall, 1991.

2 Steven G B, Adams R G, Findlay P A and Trainis S A 'iHARP: A Multiple Instruction Issue Processor' *IEE Proceedings, Part E, Computers and Digital Techniques* Vol. 139, No. 5 (September 1992) pp 439-449.

3 Steven F L, Adams R G, Steven G B, Wang L and Whale D J 'Addressing Mechanisms for VLIW and Superscalar Processors' *Microprocessing and Microprogramming* Vol. 39, No. 2-5 (December 1993) pp 75-78.

4 Collins R and Steven G B 'An Explicitly Declared Delayed-Branch Mechanism for a Superscalar Architecture' *Microprocessing and Microprogramming* Vol. 40, No. 10-12 (December 1994) pp 677-680.

5 McLellan E 'The Alpha AXP Architecture and 21064 Processor' *IEEE Micro*, Vol. 13, No. 3 (June 1993) pp 36-47.

6 Song S P, Denman M and Chang J 'The PowerPC 604 RISC Microprocessor' *IEEE Micro* Vol. 14, No. 5 (October 1994) pp 8-17.

7 Hsu P Y T and Davidson E S 'Highly Concurrent Scalar Processing' *Proceedings of the 13th Annual Symposium on Computer Architecture* (June 1986), pp 386-395.

8 Ebcioglu K 'Some Design Ideas for a VLIW Architecture for Sequential-Natured Software' IFIP *WG 10.3 Working Conference on Parallel Processing*, Pisa, Italy (April 1988) pp 3-21.

9 Furber S *VLSI RISC Architecture and Organization* Marcel Dekker, 1989.

10 Collins R 'Exploiting Instruction-Level Parallelism in a Superscalar Architecture' PhD thesis, University of Hertfordshire (October 1995)

11 Fisher J A 'Trace Scheduling: A Technique for Global Microcode Compaction' *IEEE Transactions on Computers*, Volume C-30, No.7 (July 1981) pp 478-490

12 Hank R E, Mahlke S A, Bringmann R A, Gyllenhaal J C and Hwu W W 'Superblock Formation Using Static Program Analysis' *Micro 26* (December 93) pp 247-255.

13 Nicolau A 'Uniform Parallelism Exploitation in Ordinary Programs' *Proceedings of the International Conference on Parallel Processing*, 1985, pp 614-618.

14 Moon S and Ebcioglu K 'An Efficient Resource-Constrained Global Scheduling Technique for Superscalar and VLIW Processors' *Micro25* (December 1992) pp 55-71.

15 Steven F L, Steven G B and Wang L 'An Evaluation of the iHARP A Multiple-Instruction-Issue Processor' Euromicro94, Liverpool, (September 1994).

16 Mahlke S A, Hank R E, Bringmann R A, Gyllenhaal J C, Gallagher, D M and Hwu W W 'Characterizing the Impact of Predicated Execution on Branch Prediction' Micro27 (November 1994) pp 217- 227.

17 Potter R and Steven G B 'Investigating the Limits of Fine-Grained Parallelism in a Statically Scheduled Superscalar Architecture' Europar96, Lyon, (August 1996).