DIVISION OF COMPUTER SCIENCE

AN INTRODUCTION TO THE HATFIELD SUPERSCALR ARCHITECTURE

G Steven
B Christianson
R Collins
R Potter
F Steven

Technical Report No.253

August 1996

AN INTRODUCTION TO THE HATFIELD SUPERSCALAR ARCHITECTURE

Gordon Steven, Bruce Christianson, Roger Collins, Richard Potter and Fleur Steven
University of Hertfordshire, Hatfield, Herts, AL10 9AB
Telephone: 01707 284319 Fax: 01707 284303 email: comqgbs@herts.ac.uk

To be published by Microprocessors and Microsystems under the title: *A Superscalar Architecture to Exploit Instruction-level Parallelism.*

Abstract

If a high-performance superscalar processor is to realise its full potential, the compiler must re-order or schedule the object code at compile time. This scheduling creates groups of adjacent instructions that are independent and which therefore can be issued and executed in parallel at run time. This paper provides an overview of the Hatfield Superscalar Architecture (HSA), a multiple-instruction-issue architecture developed at the University of Hertfordshire to support the development of high-performance instruction schedulers. The long-term objective of the HSA project is to develop the scheduling technology to realise an order of magnitude performance improvement over traditional RISC designs. The paper also presents results from the first HSA instruction scheduler that currently achieves a speedup of over three compared to a classic RISC processor.

Keywords

Superscalar Multiple-Instruction-Issue Instruction Scheduling

Introduction

During the last few years, computer architects have turned to multiple instruction issue (MII) to boost processor performance beyond the one instruction per cycle level. A processor is classified as MII if it is capable of issuing more than one instruction in a single processor cycle. MII architectures can be divided into two classes: superscalar architectures and Very Long Instruction Word (VLIW) architectures. In a superscalar processor [1] the hardware decides which instructions to issue in parallel at run time while in a VLIW processor [2] the compiler re-orders the original sequential code into fixed sized instruction groups that are then fetched and issued in parallel at run time.

This paper introduces the Hatfield Superscalar Architecture (HSA) [3], a statically-scheduled architecture being developed at the University of Hertfordshire. The objective of the HSA project is to develop a processor that will sustain an order of magnitude performance improvement over conventional RISC processors that issue only one instruction in each processor cycle. Issue rates of 16 - 32 instructions per cycle are therefore envisaged.

We believe that a processor with the desired performance must use a hybrid superscalar - VLIW approach that draws heavily on both hardware and software technology. A pure superscalar processor can only realise parallelism from within a limited window of pre-fetched instructions. Such a window is unlikely to deliver the parallelism we are seeking. High-performance superscalar designs also require excessively complex hardware that ultimately limits the instruction issue rate that they can sustain [1, 4]. Johnson, for example, suggests that hardware complexity limits the sustained instruction issue rate of a high-performance superscalar processor to around two.

An order of magnitude increase in performance is only likely to be achieved through the aggressive run-time instruction scheduling that is characteristic of VLIW architectures. However, VLIW architectures in their turn have major disadvantages. In particular, traditional VLIW architectures require code to be recompiled for each new processor implementation. We believe that such incompatibility is unacceptable in a commercial environment. VLIW architectures can also lead to excessive expansion in code size. Much of this code expansion is caused by the addition of redundant no-op instructions to pad out the fixed sized instruction groups that are fetched and issued in parallel at run time [5].

HSA therefore seeks to realise an order of magnitude performance improvement through a minimal superscalar architecture with in-order instruction issue and aggressive instruction scheduling. The superscalar approach guarantees object code compatibility over a range of implementations and removes any requirement for no-ops, while static instruction scheduling provides a vehicle for achieving the high speedup we desire. At the same time the opportunity is taken to reduce the hardware complexity significantly by specifying in-order instruction issue.

Limits of Instruction-Level Parallelism

Any project seeking to realise a speedup of ten through multiple instruction issue is doomed to failure if insufficient parallelism is available. It is therefore important to know how much parallelism is inherently available in a typical program.

Early studies focused on the parallelism available within basic blocks and concluded that speedups in the range of 1.5 to 2.0 could be achieved. More recently Wall[4] used trace driven simulation techniques to explore the parallelism available to superscalar processors in 17 benchmarks, including the SPEC benchmarks. Wall was primarily interested in the upper bounds placed on the performance of superscalar processors using dynamic instruction scheduling and concluded that even with an "impossibly good" superscalar implementation, the average parallelism rarely exceeded seven, with five being more common. These results support our belief that high sustained instruction issue rates can only be achieved through aggressive instruction scheduling.

Wall also obtained parallelism ranging from 6 to 60 using a "perfect" superscalar model. It is important to realise that these figures are strongly influenced by Wall's assumptions and therefore do not, as one might suppose, represent ultimate limits. Three assumptions will be mentioned. Firstly, Wall never allows more than 64 instructions to execute in parallel, so no program can exhibit parallelism greater than 64. Secondly, Wall assumes an instruction pre-fetch "window" size of 2K instructions, effectively preventing any instruction from executing in parallel with an instruction more than 2K ahead of it in the dynamic instruction stream. This prefetch window size is unrealistically large for a superscalar processor using dynamic instruction scheduling. However, in a statically scheduled MII processor such as HSA, parallel execution of instructions which are initially widely separated can be achieved through aggressive code motion. For example, the instruction scheduler might move an instruction from after a loop to before the loop and as a result move it many instructions forward in the dynamic instruction stream.

Finally, Wall makes no attempt to collapse chains of dependent instructions which repeatedly add a constant to a variable. As a result a loop body that is executed 1000 times may require 1000 cycles to increment a loop count, even though each iteration of the loop performs a self-contained operation and all the loop bodies could be executed in parallel. We conclude that while Wall's study accurately reflects the limitations affecting a dynamically scheduled superscalar processor, his figures for a "perfect" superscalar model do not represent the ultimate limits for a statically-scheduled MII processor.

Similar trace driven studies [6] have been performed by Yale Patt's group at the University of Michigan. Patt's group was also concerned with the speedup that can be obtained using a high-performance dynamically scheduled processor and concluded that issue rates between 2.0 and 5.8 instructions per cycle could be sustained. The group also computed the ultimate limits of achievable parallelism using an Unrestricted Dataflow Model in which instruction issue was only constrained by true data dependencies, effectively removing the first two restrictions noted in Wall's model. Using the SPEC benchmarks, figures from 17 to 165 instructions per cycle were obtained.

Finally, Lam and Wilson [7] used trace driven simulation to investigate the limits of instruction level parallelism but imposed far fewer restrictions on their processing model. As in earlier studies, instruction execution was only constrained by true data dependencies. However, in addition, perfect procedure in-lining and loop unrolling were assumed. Perfect in-lining was simulated by ignoring all instructions associated with procedure entry and exit, including all instructions that manipulate the stack pointer. Perfect loop unrolling was simulated by ignoring all instructions that manipulated the loop index and any associated induction variables. As a result, variables that were incremented in each loop iteration no longer created true data dependencies between loop iterations. Lam and Wilson therefore not only remove all three restrictions noted in Wall's model but also make further idealistic assumptions about procedure entry and exit overheads. Using this idealised "Oracle" model, they obtained figures for parallelism ranging from 47 to a staggering 188,470, with a harmonic mean of 158 for their integer benchmarks.

Since there is considerable overlap in the benchmarks used in the above three studies, the figures can be combined in a single table (Table 1). As can be seen, the amount of parallelism detected by the simulations is dramatically affected by the architectural model used. Nonetheless, we conclude that there is more than enough parallelism inherent in most programs to support an order of magnitude speedup through static instruction scheduling. This conclusion is supported by trace driven simulations using the HSA model which demonstrate that speedups of over 26 are potentially available in the Stanford benchmarks [8].

HSA Architectural Model

HSA is a load and store architecture with a straightforward RISC instruction set derived from the earlier HARP project [5, 9] at the University of Hertfordshire. Separate integer, Boolean and floating-point register files are provided. The one-bit Boolean registers are used to store Boolean branch conditions and to implement guarded instruction execution.

The following basic functional units are postulated: arithmetic, relational, shift, multiply, memory reference and branch. Similarly, floating-point add, relational, multiply and divide units are provided. The number of each type of unit is configuration dependent. An unusual feature is the provision of relational units to compute Boolean conditions. Dedicated relational units have two advantages. Firstly, since the number of arithmetic units can be reduced, fewer result buses and register file write ports are required. Secondly, a specialised relational unit can generate Boolean results at an earlier point in the processor cycle than a general-purpose ALU [10].

A compact 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 for each cache access. 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.

HSA always issues instructions to functional units in program order. As a result the processor will, in general, issue the instruction groups previously assembled by the scheduler. Furthermore, 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 [11] to provide out-of-order instruction issue. HSA avoids this complexity and the resultant pressure on processor cycle time.

All 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 [12]. These simplified addressing mechanisms allow all effective addresses to be made available at the end of the ID stage and avoid a load delay with a single cycle data cache.

Precise interrupts are achieved by requiring each functional unit to detect a potential overflow condition during its first cycle of operation. A functional unit can therefore only start a second execution cycle if it can guarantee that overflow will not occur. If there is any possibility of an overflow, the unit must stall the processor until the presence or absence of an interrupt is confirmed. Intel's Pentium [13] uses a similar mechanism.

Limits to Code Motion

HSA seeks to exploit fine-grained parallelism through aggressive compile-time scheduling. Ideally each instruction in a program should be able to percolate or float up through the code flow graph until an immediately preceding instruction generates one or more of the operands required by the instruction. Ultimately code motion is limited by data dependencies between individual program instructions. Three classes of data dependencies can be identified: Read-after-write (RAW), write-after-read (WAR) and write-after-write (WAW). WAR dependencies are also called anti-dependencies while WAW are called output dependencies. In the following example a read-after-write data dependency exists between the two instructions:

```
DIV R1, R2, #6      /* R1 := R2 / 6 ;      writes R1 */
SUB R4, R1, R5      /* R4 := R1 - R5 ;     reads R1 */
```

Since no instruction scheduler can change the order of these instructions, RAW data dependencies ultimately limit the performance of all MII processors. In contrast, WAR and WAW data dependencies can both be removed using register renaming. For example, in the fragment below, the second instruction has an anti-dependence on the first instruction.

```
DIV R5, R6, R7              /* R5 := R6 / R7 ;     reads R6 */
SUB R6, R8, #4              /* R6 := R8 - 4 ;      writes R6 */
```

However, there is no need for the subtraction to wait for the outcome of the long latency divide operation. The dependence can be removed by using a different register to hold the result of the subtract. This renaming allows the subtract to be moved ahead of the divide in the instruction schedule:

```
SUB R16, R8, #4     /* R16 := R8 - 4 ;     R6 replaced by R16 */
DIV R5, R6, R7      /* R5 := R6 / R7 */
MOV R6, R16         /* R6 := R16 */
```

The MOV instruction is added to restore the result of the subtraction to R6. This additional instruction need not inhibit further code motion. Later instructions that are percolated past the MOV can replace any use of R6 as a source operand by R16 and continue percolating through the code structure until a true data dependency is reached. This subsequent code motion often results in R6 becoming dead after the MOV instruction, allowing the MOV to be deleted.

Support for Speculative Instruction Execution

An instruction is executed speculatively if it is executed before it is known whether the path originally containing the instruction will be followed. Consider the following example:

```
NE B1, R1, R2               /* B1 := ( R1 ≠ R2 ) */
```

```
BT B1, Label            /* If B1 is true go to Label */
LD R6, 8(SP)            /* R6 := contents of (SP + 8) */
```

The LD instruction could be moved ahead of the branch instruction and executed speculatively giving the following code:

```
NE B1, R1, R2
LD R6, 8(SP)            /* Speculative execution */
BT B1, Label
```

However, if R6 is live at the branch target, its value on this path will have been corrupted by the code motion. Register renaming can be used to avoid this problem:

```
NE B6, R1, R2
LD R20, 8(SP)           /* R6 replaced by R20 */
BT B6, Label
MOV R6, R20             /* Result of load returned to R6 */
```

As before, a MOV instruction is added to copy the contents of R20 into R6.

The above code illustrates a further problem introduced by the speculative execution of instructions. Suppose the load instruction in the previous example generates an invalid memory address. If the path originally containing the load instruction is not subsequently followed, the instruction will generate a spurious exception that will incorrectly terminate the program. To solve this problem, all non-branch instructions in HSA exist in two forms. In the first form, an exception generated by an instruction is immediately taken in the usual way. In the speculative form an exception will simply mark the result register of the instruction as invalid.

In the above example, the speculative version of the LD instruction is therefore used. If the LD instruction raises an exception, R20 will be marked as invalid. An exception will only be taken if the non-speculative MOV instruction attempts to use the invalid value held in R20. Note that this is the earliest point in the code where it is certain that the speculative load should be executed. In contrast, if the branch is taken, the exception flag will be reset when a new value is loaded into R20, and the exception will be ignored.

To support speculative execution an extra tag bit is added to all processor registers, including the Boolean registers, to identify invalid values. This hardware support allows loads and other instructions, such as additions that generate an exception on overflow, to be executed speculatively. Unfortunately, even with this additional hardware support, store instructions can still not be executed speculatively. A store instruction can therefore not be percolated into a preceding basic block that ends with a conditional branch instruction.

A Generalised Delayed Branch Mechanism

Program execution time is also limited by branch instructions. If, for example, a conditional branch is resolved in the third pipeline stage, two cycles are potentially lost whenever a branch is taken and instructions have to be fetched from the branch target address. In a traditional RISC pipeline, branches are typically resolved in the second pipeline stage giving a branch delay of only one. This latency is often disguised by using a delayed branch mechanism [11] in which the instruction following the branch is always executed irrespective of the outcome of the branch instruction.

The classic delayed branch mechanism specifies that a fixed number of the instructions after each branch instruction will be executed irrespective of whether the branch is taken or not. This mechanism is too inflexible for a superscalar architecture which issues a variable number of instructions in each processor cycle. HSA therefore generalises the traditional mechanism by allowing each branch instruction to specify the number of instructions to be executed in its branch delay slots [14]. This variable count is implemented by adding a count field to all branch instructions. Initially the compiler sets all count fields to zero. Then, after instruction scheduling, the scheduler sets the counts to equal the number of instructions that have been successfully moved into the branch delay slots.

The flexibility inherent in this mechanism allows it to adapt to different pipelines and instruction issue rates. Machine compatibility is therefore ensured as long as each processor can execute branches that specify a variable number of delay slots. Any HSA processor can therefore execute code scheduled for any other HSA processor.

In contrast, recent superscalar architectures [15, 16] have tended to abandon the delayed branch mechanism in favour of using a branch target cache (BTC) to predict the outcome of branches dynamically. If the outcome of a branch is successfully predicted then the branch penalty is zero. Significant progress has recently been made in improving the accuracy of dynamic branch prediction with success rates as high as 97% being reported by Yale Patt's group at the University of Michigan [17].

While a perfect BTC will result in zero performance degradation, in practice there will be a performance penalty every time a branch is mispredicted:

$$\text{Penalty cycles / useful cycle} = IR \,.\, BF \,.\, (1 - H_{BTC}) \,.\, BP$$

where

IR =	Issue rate or average number of instructions issued per cycle
BF =	Branch Frequency
BP =	Branch Penalty of a misprediction in cycles
H_{BTC} =	Branch Target Cache successful prediction rate.

A superscalar processor using dynamic branch prediction will therefore suffer a performance loss proportional to the sustained instruction issue rate, the branch frequency, the miss rate in the BTC and the branch penalty. Crucially as aggressive instruction schedulers achieve ever higher average instruction issue rates, the branch penalty will increase. For example, with an effective issue rate of eight, a BTC success rate of 95% and a three cycle misprediction penalty, performance is likely to be degraded by approximately 20%. In contrast, since HSA does not incur a branch misprediction penalty, it has the potential to sustain an issue rate of IR instructions per cycle.

The use of a BTC also significantly increases the complexity of the hardware. In addition to the cost of the BTC itself, the processor must be able to recover rapidly from incorrect branch predictions by undoing the effect of any instructions that have been speculatively issued after the branch prediction. Rapid recovery from mispredicted branches is usually achieved by providing a reorder buffer [18] or an equivalent mechanism. The main concern with these recovery mechanisms is the increased complexity of both the operand fetch and result write back operations. Even without a reorder buffer, a significant number of register operands have to be read and written during each processor cycle. If a reorder buffer is added, all of these operand reads and writes need to be performed associatively on the reorder buffer. The danger is that this complexity will result in either additional pipeline stages or a longer cycle time. An additional stage will increase the branch penalty, while increasing the cycle time will, of course, affect all instructions.

The HSA branch mechanism also has disadvantages. Firstly, a delayed branch mechanism generates each branch target address one or more cycles later than a BTC. Performance can only be maintained by speculatively executing instructions from both paths after the branch. HSA can therefore only avoid the penalty cycles incurred by a BTC by increasing the instruction bandwidth. Secondly, additional bits are required for a delay count field in all branch instructions, restricting the branch range. Thirdly, the count mechanism itself requires careful implementation. Finally, some researchers view delayed branches as an artefact that is too closely tied to specific hardware implementations.

Guarded Instruction Execution

Guarded or conditional instruction execution has been proposed by a number of researchers[19-21] and has been implemented in several processors including Cydra5 [22], the Acorn ARM [23] and the HARP processor [24] developed at the University of Hertfordshire. All HSA instructions are executed conditionally. Guarded execution is implemented by associating one or more Boolean guards with each instruction. For example consider:

```
TB1 FB2 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 and the value in B2 is false. The Boolean values themselves are generated by relational instructions that return a Boolean value to one of the Boolean registers.

Guarded instructions have a number of advantages. Firstly, renaming can be avoided when an instruction is percolated into a preceding basic block. For example consider:

```
EQ B1, R1, R2            /* B1 := (R1 = R2) */
BT B1, label             /* Branch if B1 = true */
ADD R6, R3, R4
```

The instruction scheduler can safely move the ADD instruction in parallel with the branch instruction as long as the guard condition FB1 is attached:

```
EQ B1, R1, R2
BT B1, label;        FB1 ADD R6, R3, R4
```

In the absence of guarded execution, the register R6 must be renamed if the contents of R6 are live on entry to the basic block starting at "label". Guarded execution therefore reduces the pressure on register usage. Since renaming involves the insertion of additional MOV instructions, guarded execution also reduces the pressure on code size. However, guarding can not be used if the add instruction is percolated into or beyond the instruction group generating the Boolean guard.

Secondly, guarded execution allows store instructions to be percolated into the preceding block. Since store instructions can not be executed speculatively on HSA, it is not normally possible to move a store instruction into a basic block that ends with a conditional branch instruction. Guarded execution allows such percolation since the guard ensures that the store operation will only take place at run time if the path that originally contained the store instruction is followed. This additional movement of store instructions is important since store instructions tend to block the motion of subsequent load instructions.

Thirdly, guarded instruction execution results in the complete elimination of some basic blocks. For example, consider the following code that has been generated for an "if then else" construct:

```
             NEQ B6, R1, R15          /* B6 := (R1 ≠ R15) */
             BF B6, else-code         /* Branch if B6 = false */
then-code:   ADD R1, R2, R3
             BRA continue             /* Unconditional branch */
else-code:   ADD R4, R5, R6
continue:
```

After scheduling, the following code is produced:

```
NEQ B6, R1, R15            /* B6 := (R1 ≠ R15) */
TB6 ADD R1, R2, R3;        FB6 ADD R4, R5, R6
```

The advantage of branch removal is that code size is reduced and potentially fewer branch execution units are required. However, the execution time need not be significantly improved. Non branch instructions should percolate to the same point in the code irrespective of the intervening branch structure.

Branch removal is more important in superscalar processors that predict the outcome of branch instructions dynamically. Here removing branches from the program will also reduce the number of branch prediction failures. This reduction was recently quantified by Professor Hwu's Impact Group at the University of Illinois [25] which used guarded execution to remove conditional branches systematically from their benchmarks. In one case the miss rate of the branch target cache was dramatically reduced by almost a factor of 1000. Average results, however, were less encouraging with 27% of conditional branches being removed and branch prediction failures being reduced by 20%. Similar results have been obtained with HSA [26], where the scheduler removed over 30% of all branches executed.

A fourth advantage of guarded execution is that it provides a simple mechanism for allowing a branch instruction to be moved either into the branch delay region of an earlier branch instruction or into the same instruction group as a previous branch. For example, consider the following code fragment where only the control flow instructions are shown:

```
              BT B1, label
              :
              BT B2, label2
              :
label:        BT B3, label3
```

Guarded execution allows all three branches to be consolidated into a single instruction group and executed in parallel.

```
FB3 BT B1, label;  FB1 BT B2, label2;  TB1 BT B3, label3
```

A fifth advantage of conditional execution is that the number of functional units and other processor resources can be reduced. Consider the following example:

```
NE B1, R1, R2
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 are executed before the branch is taken. All six will be issued to functional units but three of the instructions will receive false Boolean guard operands and will therefore be squashed in their functional units without returning a result to a register. As a result the bandwidth of functional unit result buses can be reduced.

Carrying the idea of squashing further, instructions can be removed from the pipeline in the Instruction Decode Stage [27] whenever their Boolean guards are available and evaluate to false. The HSA simulator will squash an instruction in the ID stage, if the relevant Boolean guard evaluates to false and the instruction has remained in the instruction buffer for a full cycle without being issued. In the above example, the instructions in the first branch delay group are the first instructions that satisfy these conditions. This squashing mechanism operates in parallel with the instruction issue logic and therefore does not place additional pressure on the instruction decode cycle time. Such squashing can be remarkably effective with as many as 50% of all instructions fetched from the instruction cache being squashed in some simulation runs [26]. Squashing is therefore a powerful mechanism for reducing the impact of the additional instruction fetch bandwidth required to exploit the delayed branch mechanism fully.

A number of major disadvantages must be set against these advantages. Firstly, a significant amount of encoding space is required in each instruction to specify the guard conditions. Fortunately, a large number of Boolean registers is not required with eight proving more than adequate on the HARP processor chip [24]. Nonetheless four bits were used on HARP to add a single guard to each instruction. Arguably three of these bits could have been more usefully employed in doubling the size of the register file. Multiple Boolean guards will require even more instruction bits, even though a fully general mechanism can be encoded using only two bits per Boolean guard:

bit 0: Guard used/ not used
bit 1: True/false guard

Secondly, incorporating Boolean guards in the HARP processor added significantly to the time required to develop the chip [28]. There is, however, no evidence from the HARP development to suggest that guarded instruction extended the processor cycle time or degraded performance in any way.

Thirdly, using Boolean guards increases the complexity of the instruction scheduler. Instruction scheduling, like all compiler optimisation, ultimately reduces to a massive case analysis problem and introducing Boolean guards introduces significantly more cases. These additional cases can either be handled by introducing a more complex data structure to represent instruction groups [27] or on an individual basis.

Fourthly, guarded execution, like the use of delayed branches, is open to the charge of being an artefact. Future developments in technology and processor organisation could make guarded execution redundant in later versions of an architecture.

Finally, although several researchers have reported impressive speedups using architectures with guarded execution, there is little published evidence to quantify the benefits of guarded execution itself. Our own work on HARP suggests that guarded execution yields a performance advantage of about 10% with an instruction issue rate of four [5], not a significant advantage. Similar improvements of between 6 and 11% are reported by Professor Hwu's group with an instruction issue rate of eight [25].

Memory Disambiguation

Program execution time is also limited by data dependencies involving memory locations. However, while data dependencies involving registers are relatively easily handled by the instruction scheduler, data dependencies that involve memory locations are significantly harder to deal with. Consider the following example:

ST 4(R_i), R1 /* Store register R1 at address R_i+ 4 */
LD R2, 8(R_j) /* Load register R2 from address R_j + 8 */

Can the instruction scheduler safely percolate the load ahead of the store? This code motion is only safe if, at compile time, the two memory addresses can be shown to be always different. In the above example, the addresses are clearly different if the two registers are identical. Otherwise no conclusions can be drawn. HSA instruction schedulers use a disambiguating routine that compares two addresses and returns one of three values:

Different:	Addresses are always different.
Same:	Addresses are always the same.
Fail:	Addresses can not be disambiguated.

If the addresses always differ, the load can safely be moved ahead of the store. Percolation can also continue if the addresses are always the same. In this case the value required is already in the register being stored, so the load can be replaced by a register-to-register move.

Unfortunately many address pairs can not be disambiguated at compile time. Since many of these addresses will prove to be distinct at run time, disambiguation failures could seriously limit the parallelism realised by HSA. If this proves to be the case, the instruction scheduler will replace pairs of store and load instructions with code that compares the two addresses dynamically at run-time [29].

ADD R16, R_i, #4 /* Compute store address */
ADD R17, R_j, #8 /* Compute load address */
EQ B1, R16, R17 /* Compare two addresses for equality */
FB1 LD R2, 8(R_j) /* If addresses differ perform load */

```
T B1 MOV R2, R1          /* Else obtain value from register */
ST 4(Ri), R1
```

Both the LD and MOV instructions that replace the original LD instruction have been moved ahead of the ST. Note that the use of guarded instruction execution has avoided the insertion of two branch instructions. Furthermore, if its guard is removed, the LD can also be moved ahead of the new address comparison instructions.

The crucial importance of memory disambiguation was emphasised by recent trace driven simulations [8] performed using the HSA model where it was shown that forcing loads and stores to execute sequentially reduced the speedup available by a factor of 4.8.

Scheduling Results

Two instruction schedulers are currently being developed for HSA. The first HSA scheduler, a Conditional Group Scheduler [26, 28], is already at an advanced stage of development. The Conditional Group Scheduler aims to exploit the guarded instruction execution facilities provided by HSA and implements new data structures that directly support the scheduling of guarded instructions. The Conditional Group Scheduler attempts to fill each parallel instruction group in turn. Scheduling therefore consists of repeated searches through the code for instructions that can be added to the current group.

This section presents some preliminary results obtained using this scheduler. Three versions of the HSA architecture are compared, a slow cache version, a fast cache version and an ideal version (Table 2). In the slow cache version, both the instruction cache and the data cache are assumed to require two cycles to perform a read operation. As a result all branch instructions have two branch delay slots, and the load delay is one. Data loaded from the cache by a load instruction can therefore not be used by an immediately following instruction without introducing a stall of one cycle. In the fast cache model, a cache access time of one cycle is assumed, giving a single branch delay slot and eliminating the load delay. In both of these models multiplication is assumed to require 3 cycles and division 16 cycles. In contrast, in the ideal model all instructions, including multiply and divide, are assumed to execute in a single cycle. There is therefore no load delay although the branch delay remains one.

Modified versions of the Stanford benchmarks are used throughout. Each program is compiled using the GNUCC generated HSA 'C' compiler, scheduled using the Conditional Group Scheduler and simulated using the HSA instruction-level simulator [27]. Both the instruction fetch and issue rate are set at 16. In practice, however, issue rates over 8 are uncommon. Since we were initially interested in achieving the maximum possible speedup, no additional resource limitations were introduced.

Since HSA is a superscalar rather than a VLIW architecture, some parallel instruction issue is to be expected without any instruction scheduling. The speedups achieved with the Stanford benchmarks range from 1.42 with the slow cache model, through 1.58 with the fast cache model, to 1.63 with the ideal model (Table 3). HSA therefore achieves a respectable speedup without any instruction scheduling despite the restriction of in-order instruction issue.

Instruction scheduling results in significant further improvements in performance. The HSA scheduler currently obtains speedups of 2.91 with the slow cache, 3.23 with the fast cache and 3.62 with the ideal model (Table 4), more than doubling the performance of a superscalar processor without any instruction scheduling. This improvement is achieved at the cost of code expansion ranging from a factor of 1.91 to 2.12. Further results using the Conditional Group Scheduler were presented at a recent conference [26].

Conclusions

HSA is a highly parameterised processor model, which has been developed to support Computer Architecture research at the University of Hertfordshire. HSA has a number of distinctive features including strict in-order instruction issue, guarded instruction execution, a general delayed branch mechanism, hardware support for speculative instruction execution and an ability to remove or squash redundant guarded instructions in the instruction buffer.

Two high-performance instruction schedulers are currently being developed for HSA. The first scheduler has already achieved speedups of over three. Our aim is to improve significantly on this figure, our ultimate goal being an order of magnitude speed up over traditional single-instruction-issue RISC processors. Further goals of the HSA project are to limit code expansion and to quantify architectural trade-offs within the HSA model.

Acknowledgements

The authors would like to thank Steve Stott, Martin Loomes and John Davis for their continuing support of Computer Architecture research at the University of Hertfordshire. Part of this research was supported by EPSRC Research Studentships.

References

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

2 Fisher J A 'Very Long Instruction Set Architectures and the ELI-512' *10th Annual Symposium on Computer Architecture* (June 1983) pp 140-150.

3 Steven G B 'The Hatfield Superscalar Architecture: Version 2' *University of Hertfordshire Technical Report* (September 1994) pp 1-58.

4 Wall D W 'Limits of Instruction Level Parallelism'*ASPLOS-IV* (April 1991) pp176-188.

5 Steven F L, Steven G B and 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.

6 Butler M, Yeh T, Patt Y, Alsup M, Scales H and Shebanow M 'Single Instruction Stream Parallelism is Greater than Two' *18th Annual International Symposium on Computer Architecture* (May 1991) pp 276-286.

7 Lam M S. and Wilson R P 'Limits of Control Flow on Parallelism' *19th Annual International Symposium on Computer Architecture* (May 1992) pp 46-57.

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

9 Steven G B, Gray S M and Adams R G 'HARP: A Parallel Pipelined RISC Processor' *Microprocessors and Microsystems* Vol. 13, No. 9 (November 1989) pp 579-587.

10 Steven G B and Steven F L 'ALU Design and Processor Branch Architecture' *Microprocessing and Microprogramming* Vol. 36, No. 5 (October 1993) pp 259-278.

11 Patterson D A and Hennessy J L *Computer Architecture A Quantitative Approach* Morgan Kaufmann, 1990.

12 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.

13 Alpert D and Avnon D 'Architecture of the Pentium Microprocessor' *IEEE Micro* Vol. 13, No. 3 (June 1993) pp 11-21.

14 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.

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

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

17 Yeh T and Patt Y N 'Alternative Implementations of Two-Level Adaptive Branch Prediction' *19th Annual International Symposium on Computer Architecture* (May 1992) pp 124-134.

18 Smith J E and Pleskun A R 'Implementing Precise Interrupts in Pipelined Processors' *IEEE Transactions on Computers* Vol. 37, No. 5 (May 1988) pp 562-573.

19 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.

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

21 Arya S, Sachs H and Duvvuru S 'An Architecture for High Instruction Level Parallelism' *28th Hawaii International Conference on System Sciences* (January 1995).

22 Rau B R, Yen D W L, Yen W and Towle R A 'The Cydra5 Departmental Supercomputer: Design Philosophies, Decisions and Trade-offs' *Computer* Vol. 22, No.1 (January 1989) pp 12-35.

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

24 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.

25 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.

26 Collins R and Steven G B 'Instruction Scheduling for a Superscalar Architecture' *Euromicro96* (September 1996).

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

28 Trainis S A 'Architectural Trade-offs in the Design of a Multiple Instruction Issue Processor' PhD thesis, University of Hertfordshire (October 1994).

29 Nicolau A 'Run-Time Disambiguation: Coping with Statically Unpredictable Dependencies' *IEEE Transactions on Computers* Vol.38, No.5 (May 1989) pp 663-678.

Table 1 Limits of Instruction Level Parallelism

	Parallelism Potentially Available		
Benchmark	**Wall**	**Patt**	**Lam**
Eqntott	-	30	3283
Espresso	41	179	742
Gcc	27	38	175
Duduc	56	55	-
Fpppp	60	378	-
Matrix300	-	1165	188,470
Spice2g6	-	17	843
Tomcatv	60	930	3918

Table 2 HSA Architectural Models

	Slow Cache Model (cycles)	**Fast Cache Model** (cycles)	**Ideal Model** (cycles)
Instruction Cache	2	1	1
Branch Delay	2	1	1
Data Cache	2	1	1
Load Delay	1	0	0
Multiplication	3	3	1
Division	16	16	1
Remaining Instructions	1	1	1

Table 3 Speedup without Instruction Scheduling

Program	Slow Cache Model	Fast Cache Model	Ideal Model
Bubble	1.45	1.61	1.63
Intm	1.32	1.38	1.60
Perm	1.82	2.18	2.18
Puzzle	1.27	1.36	1.36
Queens	1.48	1.64	1.64
Quick	1.38	1.50	1.59
Towers	1.57	1.86	1.86
Tree	1.25	1.40	1.43
Average	1.44	1.62	1.66
Harmonic Mean	1.42	1.58	1.63

Table 4 Speedup after Instruction Scheduling

Program	Slow Cache Model	Fast Cache Model	Ideal Model
Bubble	4.41	5.23	5.61
Intm	2.38	2.42	3.77
Perm	5.04	6.21	6.21
Puzzle	2.75	2.70	2.70
Queens	2.96	3.54	3.54
Quick	2.52	2.73	3.43
Towers	2.73	3.23	3.23
Tree	2.29	2.60	2.87
Arithmetic Mean	3.13	3.58	3.92
Harmonic Mean	2.91	3.23	3.62