Comparison of two common pipeline structures

M. Golden
T. Mudge

Indexing terms: Cache memory, Interlocks, Memory system, Pipelines, RISC

Abstract: Two pipeline structures that are employed in commercial microprocessors are examined. The first is the load-use interlock (LUI) pipeline, which employs an interlock to ensure correct operation during load-use hazards. The second is the address-generation interlock (AGI) pipeline. It eliminates the load-use hazard but has an address-generation hazard, which requires an address-generation interlock for correct operation. The performance of these two pipelines on existing binaries and on applications that have been recompiled with a local code scheduler that understands the difference in the pipeline structures is compared. Under the assumption of perfect branch prediction, the AGI pipeline outperforms the LUI pipeline on the SPEC92 integer benchmarks, even on binaries that have been compiled for the LUI pipe. When branch prediction is considered the AGI pipeline performs significantly better than the LUI pipeline if branch prediction is more than 80% accurate and the data cache access time is greater than two cycles. Recompiling the benchmarks with a new local code scheduler optimised for the AGI pipeline provides little additional performance improvement.

1 Introduction

Although pipelining is a widely used technique for speeding up instruction execution the existence of dependences between instructions means that pipelines cannot run at 100% efficiency. Nevertheless, the improvement in speed through pipelining usually offsets any loss in performance [1].

This paper examines three types of hazards that can reduce the efficiency of a pipeline: branch, load, and address-generation hazards. In particular we compare two pipeline organizations employed in several commercial machines that make different trade-offs between these three hazards. The first, which we refer to as the load-use interlock (LUI) pipeline, issues and completes its instructions in order. It is subject to branch hazards and load hazards, but not address-generation hazards. The second, which we refer to as the address-generation interlock (AGI) pipeline, also issues and completes its instructions in order but differs from the LUI pipeline in that the execute stage is placed later in the pipeline to avoid load hazards. However, this difference results in address-generation hazards and increases the penalty for branch hazards. In this paper we report on experiments to determine if these penalties are outweighed by the benefits of eliminating load hazards.

© IEE, 1996
IEE Proceedings online no. 19960359
Paper first received 12th September 1995 and in revised form 1st February 1996
The authors are with the Electrical Engineering and Computer Science Department, The University of Michigan, Ann Arbor, MI 48109-2122, USA

The MIPS R2000 and R3000 use a precursor to the LUI pipeline. It did not employ hardware interlocks for loads or branches. Instead, NOPs are inserted after loads and branches, as required, to ensure correct operation. Load interlocks were added in the R6000, a short-lived ECL implementation of the MIPS instruction-set architecture (ISA) [2]. Load interlocks were also subsequently employed in the R4000, R4200, and R4400 [3]. The AGI pipeline is used in the Intel i486 and Pentium and the Cyrix M1, as well as in the R8000 [4–7]. The R8000, which was originally referred to in the literature as the TFP, also implements the MIPS ISA [8] and preserves binary compatibility with its LUI counterparts. Considerable amounts of software exists in the form of binaries optimised for the LUI pipeline structure, and it is not known how much performance is degraded when these binaries are run on the rearranged pipeline. To be acceptable, any reduction must be small to avoid the cost of recompiling applications.

There are two questions that this paper attempts to answer:

– How does the AGI pipeline affect performance on binaries created for an LUI pipeline?

– Does the AGI pipeline improve performance if the compiler performs local code scheduling specifically for this organisation?

2 Pipeline hazards and previous work

2.1 Branch hazards

We define the scope of a branch to be the number of instructions that can be issued before the branch is resolved. A branch hazard occurs when an instruction in the scope of the branch depends on the outcome of the branch. Although a processor may stall the issue of new instructions until it resolves a branch instruction, the introduction of pipeline bubbles caused by this approach can reduce performance to an unacceptable level.

Branch hazards can be eliminated statically by having the compiler schedule independent instructions in the scope of a branch. Alternatively, the pipeline may dynamically eliminate branch hazards by predicting the outcome of the branch, allowing control-dependent instructions to enter the pipeline and squashing them if the branch has been mispredicted [9]. These approaches are not mutually exclusive and it is not unusual for some combination to be employed.

Both approaches to removing branch hazards have shortcomings. It is not always possible to eliminate branch hazards by reordering code. It may be necessary to insert NOPs so that any instructions that cause branch hazards are moved beyond the scope of the branch. As noted, this is the solution taken by the R2/3000. However, the presence of NOPs in the execution stream reduces efficiency. Branch prediction can also introduce inefficiency when a prediction fails and instructions that execute as a result of mispredictions must be squashed.

2.2 Load hazards

Load hazards are a result of data dependences rather than control dependences. They occur when the instructions immediately following a load depend on the value retrieved by the load instruction. We define the scope of a load to be the number of instructions that can be issued before the data retrieved from memory by the load becomes available to later instructions. A load hazard occurs when an instruction in the scope of a load uses the value read by the load.

In a pipeline that supports out-of-order execution, an instruction that depends on an outstanding load operation can simply be buffered at a reservation station until all of its operands are available and it can be sent to a function unit. In a pipeline that only allows in-order execution of instructions, there are three approaches to tolerating a load hazard: reorder instructions so that there are no instructions that cause load hazards after the load (NOPs may have to be added); stall the pipeline when an instruction that causes a load hazard is fetched until the load is completed (load-use interlock); and use some form of load prediction to prefetch load data and effectively remove dependences that arise from the load.

All three approaches to removing load hazards have shortcomings. It is not always possible to eliminate load hazards by reordering code. It may be necessary to insert NOPs so that dependent instructions that cause hazards are moved beyond the scope of the load (the processor must still have some interlock mechanism to handle cache misses). The presence of NOPs in the execution stream reduces efficiency. The use of load-use-interlock stalls avoids the code expansion of NOPs, but it too reduces efficiency. Finally, loads are much more difficult to predict than branches and the last method is rarely used [10]. Again these approaches are not mutually exclusive.

2.3 Address-generation hazards

Address-generation hazards occur when a value is computed for a register that is used to form the address of the data retrieved by a load instruction. For the purposes of this discussion we consider only the base-register-plus-offset address mode for load instructions. In this case, the scope of address generation is the number of instruction slots between an instruction that modifies a register and its earliest availability for use as a base register in an address calculation.

As in the case of any data hazard, a machine that supports out-of-order execution of instructions can simply buffer the dependent instruction until all operands become available. For a pipeline that does not allow this model of execution, there are two approaches to tolerating an address-generation hazard: insert instructions so that there is sufficient time to finish modifying the address register before its use by the load instruction; and stall the pipeline until address generation is completed (address-generation interlock). In principle, address generation could also be predicted but it is never done. Removing address-generation hazards by stalling is, as with the other hazards, a source of inefficiency.

2.4 Other hazards

In addition to the hazards that we are concerned with in this paper, there are others that have only a small impact on the performance of LUI or AGI pipelines, or that are avoided altogether in the LUI and AGI pipelines. In the first category are instructions that store values to memory. During a store operation, the memory system does not return a value to the CPU, so subsequent instructions can usually be issued without delay. A hazard can occur if, before the store completes, a load instruction is issued that retrieves data from the memory location that is the target of the store. Microarchitectural features such as write buffers or write caches with hazard detection logic have been used to solve this problem [11]. In this paper, the effects of store hazards are ignored.

In the second category are hazards resulting from true data dependences on instructions that perform ALU operations: if the results of an instruction are required by a succeeding instruction, and if the second instruction issues before the first instruction computes its result, then a hazard occurs. The LUI and AGI pipelines avoid this class of hazards by implementing ALU operations that only require one cycle and by employing bypass paths that send a value from one pipeline stage directly to another stage.

In this paper we are not be concerned with machines that issue more than one instruction at a time, typified by superscalar or VLIW architectures. Of course, their individual execution pipes are likely to be of the LUI or AGI type, and future studies might investigate their relative merits in this setting where the matter of instruction dependence becomes much more complex. For two excellent discussions on this the reader is referred to [12, 13].

2.5 Previous work

Previous work has proposed both static and dynamic techniques of eliminating the hazards that instructions dependent on load instructions cause. Static techniques involve code scheduling in which the compiler attempts to hide the latency of load instructions by scheduling them well before their results are needed. In [14], Krishnamurthy presents a survey of techniques for local code scheduling. Global code scheduling techniques, such as superblock [15] and hyperblock [16] scheduling, allow code motion between basic blocks. Dynamic techniques involve some form of hardware support; they may also require compiler support too. Work in this area includes the fast address calculation technique of Austin, *et al.* [17]; Iliffe's 'forward looking' architecture that attempts to issue a memory load early [18]; Sohi and Davidson's structured memory access architecture [19] that has an address processing unit capable of prefetching from all addresses having a common address pattern; and Golden and Mudge's load target buffer (LTB) [10], which predicts load addresses much like a branch target buffer.

3 Two pipeline organisations

3.1 Load-use interlock pipeline

The LUI pipeline is shown in Fig. 1. This has been referred to as the 'classic five-stage RISC pipeline' [20]. Each box represents a single machine cycle and a list of the functions that are performed during that cycle. Fig. 1 labels the five stages with their primary function: IF, instruction fetch; RD, register read and decode; EX, execute the ALU operation; MEM, data cache access; and WB, write back to the register file. The bypass paths are also shown. The number of cycles spanned by the path indicates how long the bypass operation takes.

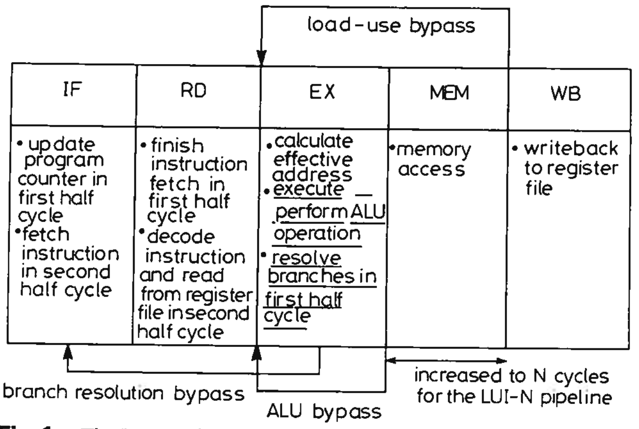
load-use bypass
IF
RD
EX
MEM
WB
• update program counter in first half cycle
• fetch instruction in second half cycle
• finish instruction fetch in first half cycle
• decode instruction and read from register file in second half cycle
• calculate effective address
• execute — perform ALU operation
• resolve branches in first half cycle
• memory access
• writeback to register file
branch resolution bypass
ALU bypass
increased to N cycles for the LUI-N pipeline

Fig. 1 *The LUI pipeline*
Five stages and bypass paths are shown. Underlined actions in EX stage are moved into MEM stage in AGI pipeline, Fig. 2. Load-use bypass spans $N+1$ stages in LUI-N case

Fig. 1 shows that conditional branches are not resolved until the end of the first half of the EX stage. This results in a branch scope of one cycle, during which a branch hazard can occur. This is solved by the inclusion of a branch-delay slot in the MIPS ISA. Correct operation requires that the instruction in the branch-delay slot must be able to execute independently of the result of the branch. If an independent instruction cannot be found, a NOP is inserted into the branch-delay slot.

During a load instruction, the effective memory address is computed during the EX stage and sent to the memory system. If the request hits in the first level cache, the result is available at the end of the MEM stage, where it may be forwarded back to the EX stage. The forwarding path spans two cycles, indicating that the MEM stage result is not available to the instruction that immediately follows it in the pipeline, but to the second instruction after the load. Any instruction immediately after a load that uses the result of that load creates a load hazard. In such cases, the pipeline stalls for one cycle. Of course, if the instruction misses in the cache, the delay is much greater and the pipeline stalls for many cycles.

In the early MIPS machines (R2000 and R3000), as noted earlier, the absence of a load-use interlock is handled by requiring that the compiler guarantee that the instruction after a load is not dependent on the load. This instruction occupies the load-delay slot. If the compiler cannot find an independent instruction, it puts a NOP instruction in the load-delay slot [3].

In high clock rate microprocessors, even the on-chip primary cache can take more than one cycle to access. This paper will also consider a generalisation of the LUI pipeline to systems with multiple-cycle data cache access times. These pipelines will contain additional MEM stages. A data cache with an access time of N cycles will be paired with a LUI pipeline with N MEM stages, and will be referred to as an LUI-N pipeline. In an LUI-N pipeline, the scope of a load is N instructions and its load-use interlocks can last from 1 to N cycles. If the first dependent instruction in the load scope is k instructions after the load, then the interlock will stall the pipeline for $(N-k) + 1$ cycles.

3.2 Address-generation interlock pipeline

The AGI pipeline is shown in Fig. 2. In this pipeline, the load-use interlock has been eliminated by delaying the EX stage by one cycle and combining it with the MEM stage. Combining the EX and MEM stages requires an extra adder, which is dedicated to computing the target address of memory operations. This address calculation is performed in the AD stage before the EX/MEM stage. In contrast, the LUI pipeline has only a single adder in the EX stage, which is used for both integer arithmetic instructions and address calculations. In the AGI pipeline, when an instruction that depends on a load in the previous cycle reaches the EX/MEM stage, the results of the load are available from the ALU bypass. However, branch resolution now occurs one stage later because a conditional branch instruction may require a result from the instruction that immediately precedes it. This result will not be available until the end of the EX stage.

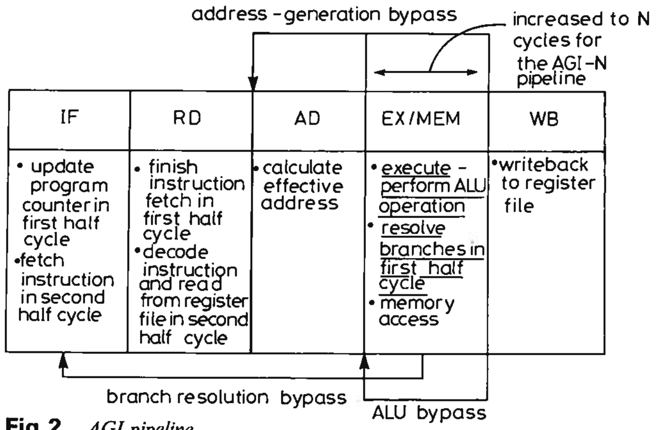
address-generation bypass
increased to N cycles for the AGI-N pipeline
IF
RD
AD
EX/MEM
WB
• update program counter in first half cycle
• fetch instruction in second half cycle
• finish instruction fetch in first half cycle
• decode instruction and read from register file in second half cycle
• calculate effective address
• execute - perform ALU operation
• resolve branches in first half cycle
• memory access
• writeback to register file
branch resolution bypass
ALU bypass

Fig. 2 *AGI pipeline*
Five stages and bypass paths are shown. Underlined actions in EX/MEM stage are moved from EX stage in LUI pipeline, Fig. 1. Address-generation bypass spans $N+1$ stages in AGI-N case. EX stage remains in last of N MEM stages, and ALU bypass still spans only one stage

There are two disadvantages to this arrangement. First, an address-generation interlock is required when a load instruction requires the register result of an uncompleted instruction to calculate the target address in memory. Secondly, the branch scope is now two cycles because branch resolution occurs in the first half of the EX/MEM stage of the pipeline. This means that in addition to the branch-delay slot, a second instruction will issue before the branch is resolved. We assume that this instruction is chosen by a prediction scheme, and that it may have to be squashed if the branch has been mispredicted. This contrasts with the LUI pipeline which, because of the branch-delay slot, needs no branch prediction strategy.

As cache access time grows beyond a single cycle, delay stages can be added to the AGI pipeline between the AD and EX/MEM stages. A processor that takes N cycles to access the cache will require N–1 extra MEM stages. We refer to this as an AGI–N pipeline. In an AGI–N pipe, N instructions must be squashed every time a branch is mispredicted, and address-generation interlocks can last from 1 to N cycles. If the first dependent load instruction is issued k cycles after the instruction that generates its base register, the interlock will stall the pipeline for $(N–k) + 1$ cycles.

The code fragment written in MIPS assembly language shown below further illustrates the difference between the two pipeline organisations:

```
I1: move  a3, a0            # move the value in register
                              a0 into register a3
I2: lw    v1, 4(a3)         # use it as the base register
                              to load register v1
I3: beq   v1, zero, 0x400328  # conditionally branch
                              on v1 == 0
    nop
I4: move  a0, v1            # v1 ! = 0, so put the value
                              in v1 into a0
I5: jal   copy_bnode        # and call copy_bnode (a0)
```

NOPs in load-delay slots have been removed; load-use interlocks are modelled instead. The code is taken from the program eqntott, a SPEC92 integer benchmark. In this example, instruction I3 depends on instruction I2, which in turn depends on instruction I1. Because the branch instruction I3 depends on I2, a load-use interlock will occur in an LUI pipeline. This interlock does not occur in the AGI pipeline. Instead, an address-generation interlock will stall the pipeline since I1 calculates a value for the base register of the load instruction I2. In addition to the address-generation interlock, the AGI pipeline may face an additional possible performance loss if the branch is mispredicted. In the case of the LUI pipeline, the NOP in the branch-delay slot covers the branch penalty. For every memory access stage in the AGI pipeline, an additional instruction must be squashed after a mispredicted branch. For example, in an AGI-2 pipeline, both I4 and I5 would be squashed if the branch instruction I3 were incorrectly predicted not-taken. Note that for both the LUI and the AGI pipeline, the instruction after the branch occupies a branch-delay slot. Only the additional instructions in the branch scope for the AGI pipeline are speculatively executed.

4 Compiler and simulator

This paper considers programs compiled for the MIPS I instruction set architecture — the version of the architecture that does not support load-use interlocks. This architecture was chosen for several reasons:

• The MIPS architecture has been implemented with a LUI pipeline and with an AGI pipeline. The R-series machines all have LUI pipelines and the TFP has an AGI pipeline.

• The Gnu C Compiler (GCC) is available for the MIPS architecture [21]. GCC is in the public domain and the source codes are easily available, so the compiler may be modified.

• The MIPS is a load/store architecture, so all memory operations are contained in explicit load and store instructions. This simplifies the creation of compilers that optimise for the two different pipeline structures.

The experiments use the SPEC 92 integer benchmarks, summarised in Table 1. All of the benchmark programs are executed to completion using one of the 'reference' input files provided by SPEC except xlisp, which uses the SPEC-provided 'short' input file owing to simulation time considerations. When several SPEC reference input files are available the experiments use the file listed in the table. The base execution time is the time required to execute the benchmark to completion on a processor with a zero-cycle cache access time. The benchmarks are compiled three times. The MIPS C compiler creates one version of each program. The MIPS C compiler heavily optimises the code and assumes a single load-delay slot. In effect, this provides a binary that is optimised for load instructions that have a scope of one cycle on a cache hit. GCC is used to create two versions of each benchmark: one optimised for the AGI pipeline and one optimised for the LUI pipeline. The versions differ in the cost function given to GCC's scheduling algorithm.

Table 1: SPEC92 integer benchmarks and their characteristics

Benchmark	Input file	Base execution time in cycles	Average basic block size
compress	reference	79 192 765	5.1
eqntott	reference	1 381 970 038	3.0
espresso	bca.in	493 384 704	5.6
gcc	stmt.i	133 778 490	5.0
sc	loadda1	436 172 261	4.6
xlisp	short	1 171 528 797	3.0

GCC's scheduler assigns a priority to each instruction in a basic block. Instructions with high priorities are scheduled first. Several factors determine the priority of an instruction, but the most important is the scope of an instruction. An instruction with a large scope that produces results used by a later instruction is assigned a high priority equal to the number of instructions in its scope. Once the instructions are prioritised, GCC attempts to schedule each instruction so that the pipeline will never interlock.

To provide a binary that optimises for load-use hazards, one version of each benchmark is produced in which GCC is told that two instructions are required between a load and its use for interlock-free execution. To create a version optimised to reduce address-generation hazards, the scheduler is told that the scope of address generation is two cycles. The study includes the MIPS C compiler version because it is the standard compiler for systems using the MIPS processors. A comparison of the code produced by GCC with the MIPS C compiler, provides a confidence check that the code that is produced by GCC for the AGI pipeline is equally well optimised.

Each version of the program is then instrumented to produce an instruction and data trace by pixie. A simulator based on the xsim tool developed by Smith consumes the trace [22]. The simulator models a machine with the following characteristics:

• There are no load-delay slots. Other delay slots, mainly those required by the MIPS architecture for integer multiply and divide instructions, are present in the machine model. This includes a single branch-delay slot for both the AGI-N and the LUI-N pipeline.

- All operations except data cache accesses complete in a single cycle.
- There is a single execution pipeline.
- All memory references hit in the instruction and data caches.
- Instruction fetch requires a single cycle.

Load-delay slots have been eliminated in newer RISC architectures, such as the Alpha, in favour of load-use interlocks. As cache access times get longer, code expansion caused by NOPs in unfilled delay slots becomes a problem [23]. Typical RISC integer instructions complete in a single cycle, except integer multiplication and division, which usually take more than one cycle. The MIPS ISA requires delay slots in the scope of these instructions, which must be filled by independent instructions or NOPs.

5 Experimental results

5.1 Experiments on an ideal pipeline

In the Figures in this Section, the x-axis shows the access time of the data cache in cycles. The y-axis shows an execution time that is normalised to the run time of code compiled by the MIPS C compiler for a machine with an LUI pipeline and a zero-cycle cache access time ($N = 0$). In other words, all memory references are immediately available so there are no load-use hazards or address-generation hazards in the reference machine. The third column of Table 1 lists these base execution times for each benchmark in cycles. The harmonic means of the experimental results for all benchmarks are shown in Fig. 3. Results for individual benchmarks can be found in [24]. High numbers indicate poor performance.

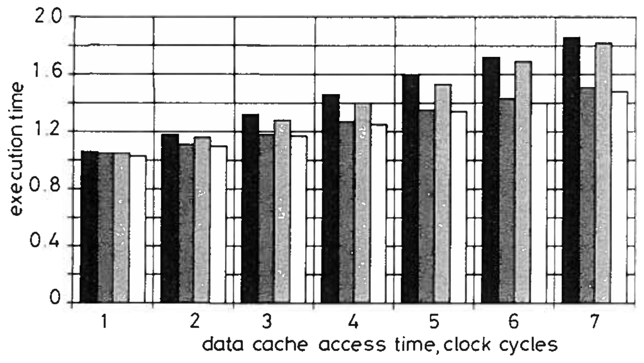

Fig. 3 *Harmonic mean of all benchmarks, 1-cycle I-cache*
Figure assumes perfect branch prediction. Improvement between GCC-LUI and GCC-AGI-Perfect is similar to the improvement between LUI and AGI-Perfect. Informing GCC's local scheduler of need to avoid address-generation interlocks has little effect. AGI pipeline shows better performance than LUI pipeline in all cases
■ LUI
▩ AGI-Perfect
▨ GCC-LUI
□ GCC-AGI-Perfect

The first experiment compares how the benchmarks perform on code compiled by the MIPS C Compiler for the MIPS R2000 performs on an LUI and an AGI pipeline for varying cache access times. The results assume perfect branch prediction in the AGI case. These bars are labelled LUI and AGI - Perfect in Fig. 3. For low cache access times there is very little difference between the two pipeline organisations. As the access time increases beyond about three cycles the performance benefit of the pipeline with an address-generation interlock begins to appear. The AGI-3 pipeline completes the benchmarks almost 10% faster than the LUI-3 pipeline. The performance gap continues to grow as the cache access time gets larger.

This first experiment answers the question about the performance of existing binaries. For our sample set of benchmarks the AGI pipeline actually performs slightly better than the LUI pipeline on binaries compiled for an LUI pipeline.

The next set of experiments considers code compiled by GCC for LUI pipelines against code compiled by GCC for AGI pipelines. The programs are run on the pipelines for which they were compiled with the assumption of perfect branch prediction. In Fig. 3, these experiments are labelled GCC-LUI and GCC-AGI-Perfect. Once again a small benefit is seen through the use of AGI pipelines for small cache access times. As cache access times increase, AGI pipelines again provide a larger speedup.

Informing GCC's local scheduler of the new pipeline structure does not seem to affect execution time to a large extent. The percentage change between the GCC-LUI experiments and the GCC-AGI experiments are similar to those between the LUI and the AGI-Perfect experiments. This may be because GCC's scheduler works only within a single basic block. For the benchmarks under consideration, the basic block size tends to be small, as small as three in the case of xlisp, so modifying the code scheduling costs may not have a large effect. The limited improvement obtained from the compiler suggests that more aggressive global scheduling techniques may be needed. However, the performance of the Gnu C compiler against the MIPS C compiler (compare LUI against GCC-LUI) makes it clear that, for our machine model, GCC is as good as one of the best commercial compilers. This gives support for our remaining results with GCC.

This set of experiments gives a limited answer to the second question posed in the introduction. Simply altering the local scheduling algorithm does not significantly improve the compiler's ability to produce efficient code for the AGI pipeline. However, the performance of the AGI pipeline is already better, as shown. More sophisticated compiler techniques may provide further improvement.

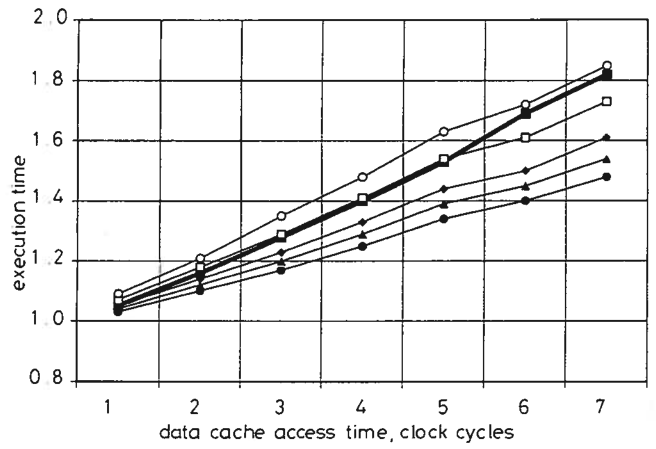

Fig. 4 *Harmonic mean of all benchmarks, I-cache access time = 1 cycle*
Figure shows results when branch prediction is taken into account. AGI pipeline suffers from reduced performance when accurate branch prediction is not available
—■— GCC-LUI
—●— GCC-AGI-Perfect
—▲— GCC-AGI-95%
—◆— GCC-AGI-90%
—□— GCC-AGI-80%
—○— GCC-AGI-70%

The final set of results, labelled GCC-AGI-X% represents AGI pipelines with X% branch prediction over all branches, including unconditional jumps and calls. These results are summarised in Fig. 4. Because the MIPS branch delay slot is included in the simulator, all of the results for LUI pipelines are valid for any branch prediction accuracy. The branch penalty is accounted for by the instruction in the delay slot, which may be a NOP. In contrast, an AGI-N pipeline must squash N extra instructions when a branch is mispredicted. A branch penalty is approximated by assessing a fixed number of cycles for each mispredicted branch and adding it to the total execution time of the benchmark. The penalty for machines with LUI and AGI pipelines are calculated with the following formulas:

$$\text{penalty}_{LUI} = (N_i - 1) \times (1 - b) \times C_b$$

$$\text{penalty}_{AGI} = (N_d + N_i - 1) \times (1 - b) \times C_b$$

where N_d is the date cache access time and N_i is the instruction cache access time in machine cycles, b is the branch prediction accuracy expressed as a probability, and C_b is the dynamic branch count of the program. In accordance with the pipeline structures described, N_i = 1 for both pipelines.

For machines with accurate branch prediction the AGI pipeline still outperforms the LUI pipeline. Once the accuracy of branch prediction drops down to around 80% the two types of machines perform equivalently. At lower levels of branch prediction accuracy the early branch resolution of the LUI pipeline allows it to run programs more quickly.

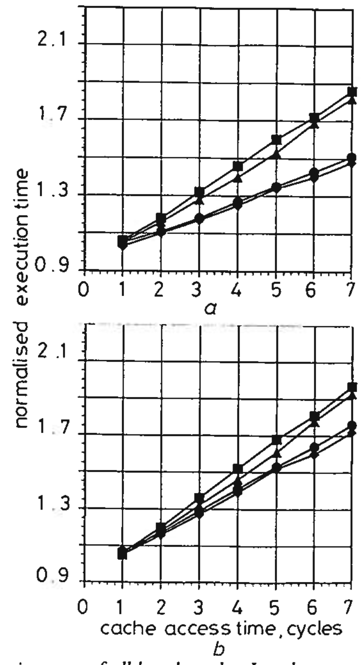

Fig. 5 *Harmonic mean of all benchmarks, I-cache access time = D-cache access time*
Results labelled MCC have been compiled by MIPS C compiler. Results labelled GCC have been compiled by Gnu C compiler. AGI pipeline still requires good branch prediction to outperform LUI pipeline
a Perfect branch prediction
b 90% branch prediction
—■— LUI-MCC
—●— AGI-MCC
—▲— LUI-GCC
—◆— AGI-GCC

5.2 Pipelines with multicycle instruction cache access time

The experiments so far assume that the instruction cache can be accessed in a single cycle; the pipelines described in Section 3 have a single IF stage. As the I-cache latency increases, the penalty for a mispredicted branch increases because more time is required to fetch the correct instruction from the memory system. In other words, the scope of a branch instruction grows.

In an LUI system with a multicycle I-cache access time, the branch penalty is no longer completely hidden by a single branch delay slot. As a consequence, the requirement that an AGI pipeline have accurate branch prediction to outperform an equivalent LUI pipeline may be eased. Figs. 5 and 6 show this is not the case. The I-cache access time has been set to equal the D-cache access time. The LUI pipeline experiences a branch penalty in this experiment but it is less affected by poor branch prediction than the AGI pipeline. Branch prediction still must be better than about 80% accurate for the AGI pipeline to have a performance advantage for machines with slow caches. On machines that have fast caches or poor branch prediction both pipelines have similar performance.

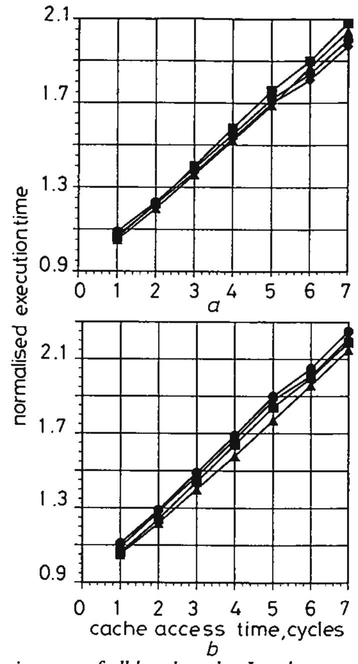

Fig. 6 *Harmonic mean of all benchmarks, I-cache access time = D-cache access time*
Results labelled MCC have been compiled by MIPS C compiler. Results labelled GCC have been compiled by Gnu C compiler. AGI pipeline still requires good branch prediction to outperform LUI pipeline
a 80% branch prediction
b 70% branch prediction
—■— LUI-MCC
—●— AGI-MCC
—▲— LUI-GCC
—◆— AGI-GCC

6 Conclusions

A number of processors have recently been announced that eliminate the load-use interlock by overlapping the execute stage of the pipeline with cache access rather than address generation. Some of these AGI machines are designed not only to execute code compiled specifically for them, but also to run codes compiled for older LUI implementations of similar architectures. When good branch prediction methodologies are available the rearranged pipeline provides improved performance for machines with moderate to large cache access times, even if existing binaries are used. When a branch-delay slot can hide instruction cache latency in an LUI pipeline, high branch prediction accuracy is required for the AGI pipeline to have a performance benefit. As the I-cache access time grows this trend remains the same.

Simply modifying the compiler's local scheduler shows only a small increase in the benefits of the AGI pipeline. Because basic blocks can be quite short in nonscientific programs the local scheduler does not have many instructions to work with. Global scheduling techniques may be able to further improve the performance of the AGI pipeline structure because these methods make more instructions available to be placed between the dependent instructions that cause the interlock.

Several questions remain unanswered. First, in the experiments described in this paper, perfect caches are assumed. In the presence of cache misses, the average time to fetch an instruction and operate on data memory will increase. Cache misses may be distributed such that the effect on these experiments is merely to increase the effective latency to the cache. However, they may be distributed such that pipeline behaviour changes noticeably as cache access time and miss rates change.

Secondly, we have simulated machines that have a single execution pipeline. In a processor with multiple pipelines, each stall cycle can delay the completion of many instructions rather than just one. This may also affect the performance difference between the two pipelines. We leave the study of these two issues as future work.

7 Acknowledgment

This work was supported by Defense Advanced Research Projects Agency under DARPA/ARO contract DAAL03-90-C-0028.

8 References

1 OLUKOTUN, O., MUDGE, T., and BROWN, R.: 'Performance optimisation of pipelined memory caches'. Proceedings of 19th annual international symposium on *Computer architecture*, Gold Coast, Australia, May 1992, IEEE Computer Society Press, pp. 181–190

2 'MIPS chip set implements full ECL CPU', *Microprocess. Rep.*, December 1989, **3**, (12). pp. 1, 14–19

3 KANE, G., and HEINRICH, J.: 'MIPS RISC architecture' (Prentice-Hall, Englewood Cliffs, NJ, 1992)

4 CASE, B.: 'Intel reveals Pentium implementation details', *Microprocess. Rep.*, 1993, **5**, (23), pp. 9–17

5 CRAWFORD, J.H.: 'The i486 CPU: executing instructions in one clock cycle', *IEEE Micro*, February 1990, **10**, (1), pp. 27–36

6 GWENNAP, L.: 'Cyrix describes Pentium competitor', *Microprocess. Rep.*, October 1993, **7**, (14), pp. 1, 6–10

7 GWENNAP, L.: 'Intel reveals Pentium implementation details', *Microprocess. Rep.*, March 1993, **7**, (4), pp. 9–17

8 HSU, P.Y.T.: 'Designing the TFP microprocessor', *IEEE Micro*, April 1994, **14**, (2), pp. 23–33

9 SECHREST, S., LEE, C.-C., and MUDGE, T.: 'Correlation and aliasing in dynamic branch prediction'. Proceedings of 23rd international symposium on *Computer architecture*, Philadelphia, USA, May 1996, IEEE Computer Society Press

10 GOLDEN, M., and MUDGE, T.: 'Hardware support for hiding cache latency'. Technical report CSE-TR-152-93, University of Michigan, Department of Electrical Engineering and Computer Science, Ann Arbor, MI, 48109-2122, USA, 1993

11 JOUPPI, N.P.: 'Cache write policies and performance'. Technical report, Digital Equipment Corp. Western Research Laboratory, 250 University Ave., Palo Alto, CA 94301, December 1991

12 JOHNSON, M.: 'Superscalar microprocessor design' (Prentice-Hall, Englewood Cliffs, NJ, 1991)

13 RAU, B.R., and FISHER, J.A.: 'Instruction-level parallel processing: history, overview, and perspective', *J. Supercomput.*, 1993, **7**, (1/2), pp. 9–50

14 KRISHNAMURTHY, S.M.: 'A brief survey of papers on scheduling for pipelined processors', *SIGPLAN Not.*, July 1990, **25**, (7), pp. 97–106

15 HWU, W.W., MAHLKE, S.A., CHEN, W.Y., CHANG, P.P., WARTER, N.J., BIRMINGHAM, R.A., OULLETTE, R.G., HANK, R.E., KIYOHARA, T., HAAB, G.E., HOLM, J.G., and LAVERY, D.M.: 'The superblock: An effective technique for VLIW and superscalar compilation', *J. Supercomput.*, 1993, **7**, (1/2), pp. 229–248

16 MAHLKE, S.A., HANK, R.E., McCORMICK, J.E., AUGUST, D.I., and HWU, W.W.: 'A comparison of full and partial predicated execution support for ILP processors'. Proceedings of 22nd annual international symposium on *Computer architecture*, Italy, June 1995, IEEE Computer Society Press

17 AUSTIN, T.M., PNEVMATIKATOS, D.N., and SOHI, G.S.: 'Streamlining data cache access with fast address calculation'. Proceedings of 22nd annual international symposium on *Computer architecture*, June 1995, (IEEE Computer Society Press)

18 ILIFFE, J.K.: 'A forward looking method of cache memory control', *Comput. Archit. News*, September 1987, **15**, (4), pp. 4–10

19 SOHI, G.S., and DAVIDSON, E.S.: 'Performance of the structured memory access SMA architecture'. Proceedings of 1984 international conference on *Parallel processing*, Bellaire, MI, August 1984, pp. 506–513

20 SMITH, J.E., and WEISS, S.: 'PowerPC 601 and Alpha 21064: a tale of two RISCs', *Comput.*, June 1994, **27**, (6), pp. 46–58

21 STALLMAN, R.M.: 'Using and porting GNU CC' (Free Software Foundation, Boston, NA, 1993, 2.4.5 edn.)

22 SMITH, M.D.: 'Tracing with pixie'. Center for integrated systems, Stanford University, Stanford, CA 94305-4070, 1.1 edition, April 1991

23 SITES, R.L.: 'Alpha architecture reference manual' (Digital Press, Maynard, MA, 1992

24 GOLDEN, M.: 'Reducing the penalty of branch and load hazards in pipelined microprocessors'. PhD thesis, Univeristy of Michigan, Ann Arbor, 1995