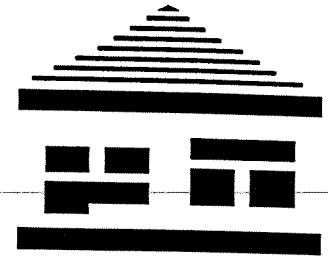
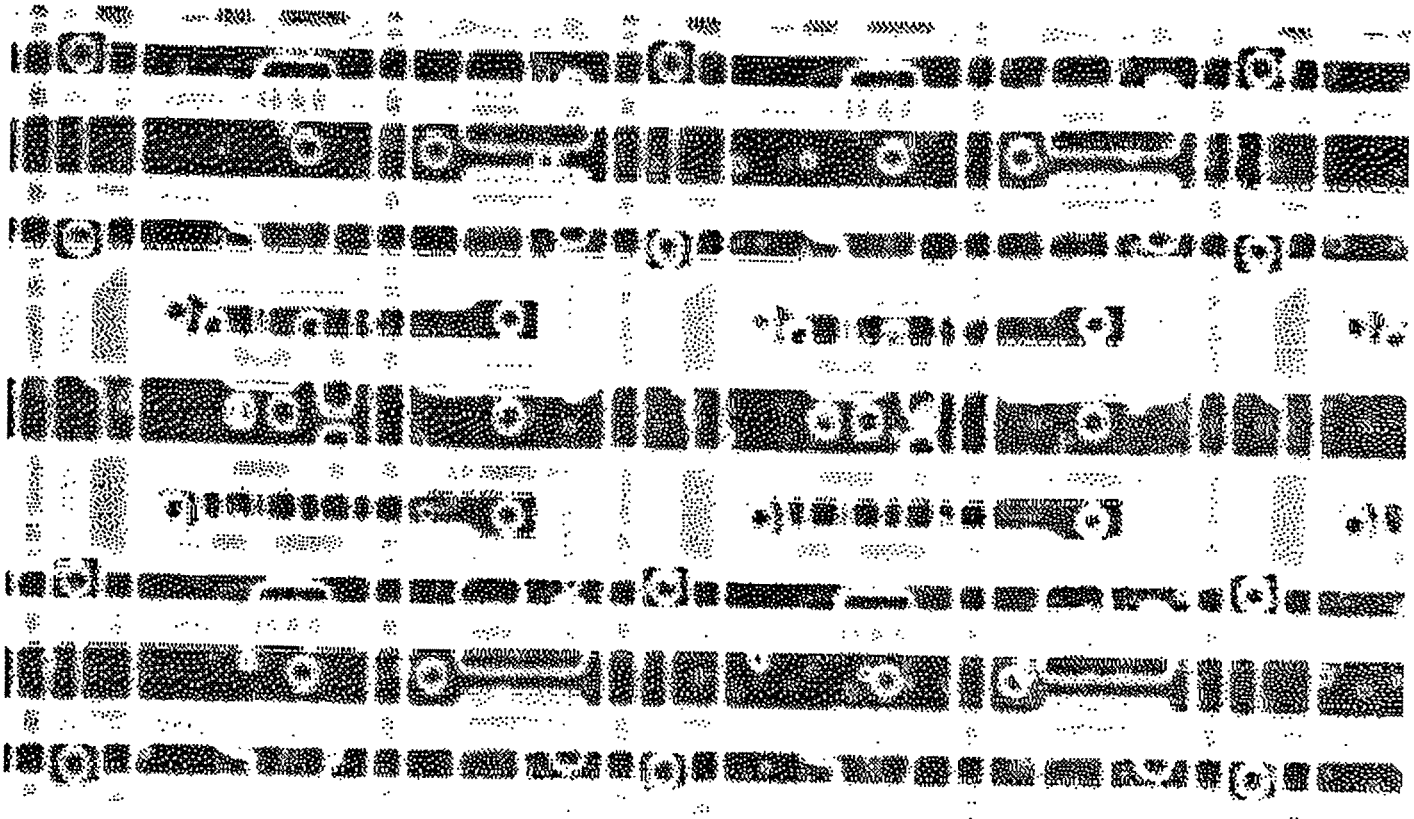


MICRO-27

Proceedings of the 27th Annual

International Symposium on Microarchitecture



Sponsored by

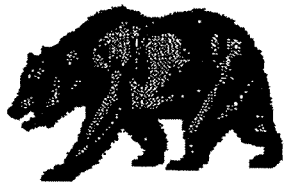
IEEE Computer Society, Technical Committee on Microprogramming and Microarchitecture (TC-MICRO)

Association for Computing Machinery - SIGMICRO

SIGMICRO Newsletter - a publication of the Association for Computing Machinery Special Interest Group on Microprogramming, Volume 25, December 1994

November 30-December 2, 1994

San Jose, California



A Comparison of Two Pipeline Organizations

Michael Golden and Trevor Mudge

Electrical Engineering and Computer Science Department

University of Michigan
1301 Beal Avenue
Ann Arbor, Michigan, 48109-2122
email: {mgolden,tnm}@eecs.umich.edu

Abstract

We examine two pipeline structures which are employed in commercial microprocessors. 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. We compare the performance of these two pipelines on existing binaries and on applications which have been recompiled with a local code scheduler that understands the difference in the pipeline structures. When branch prediction is more than 80% accurate and the data cache access time is greater than two cycles, the AGI pipeline performs significantly better than the LUI pipeline on existing binaries. Recompiling the benchmarks with a new local code scheduler provides little additional performance improvement.

Keywords: *cache memory, interlocks, memory system, pipelines, RISC*

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[14].

This paper will examine three types of "hazards" that can reduce the efficiency of a pipeline: branch, load, and address-generation hazards. In particular we will compare two pipeline organizations employed in several commercial machines that make different trade-offs between these three hazards. The first, which we shall refer to as the load-use interlock (LUI) pipeline, was introduced in the MIPS R6000—a short-lived ECL implementation of the MIPS instruction-set architecture (ISA) [13]. It is subject to branch hazards and load hazards, but not address-generation hazards. The second, which we shall refer to as the address-generation interlock (AGI) pipeline, was introduced in the TFP processor announced by Silicon Graphics Inc. The TFP also implements the MIPS ISA [4] [7]. It rearranges the LUI pipeline, moving the execute stage later in the pipeline, so that load hazards are eliminated. However, this change introduces address-generation hazards and increases the penalty for branch hazards. In this paper we will report on experiments to determine if these penalties are outweighed by the benefits of eliminating load hazards.

A precursor to the LUI pipeline is used in the MIPS R2000 and R3000. It does 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 and subsequently in the R4000, R4200, and R4400 [11]. The AGI pipeline is used in the Intel Pentium and the Cyrix M1[1], as well as the TFP [1][5] [6]. All three processors with AGI pipelines are also designed to preserve *binary compatibility* with earlier LUI microprocessors. A large body of software exists in the form of binaries optimized 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:

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

2. Does the AGI pipeline improve performance if the compiler performs local code scheduling specifically for this organization?

This paper is organized as follows. The next section discusses pipeline hazards in more detail. With this as background, Section 3 describes the LUI and AGI pipeline organizations. The compiler and simulation tools are described in Section 4. Experimental results are presented in Section 5 followed by some concluding remarks in Section 6.

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

MICRO 27- 11/94 San Jose CA USA

1.The M1 executes the Intel instruction set, but has one extra address calculation stage than the other pipelines.

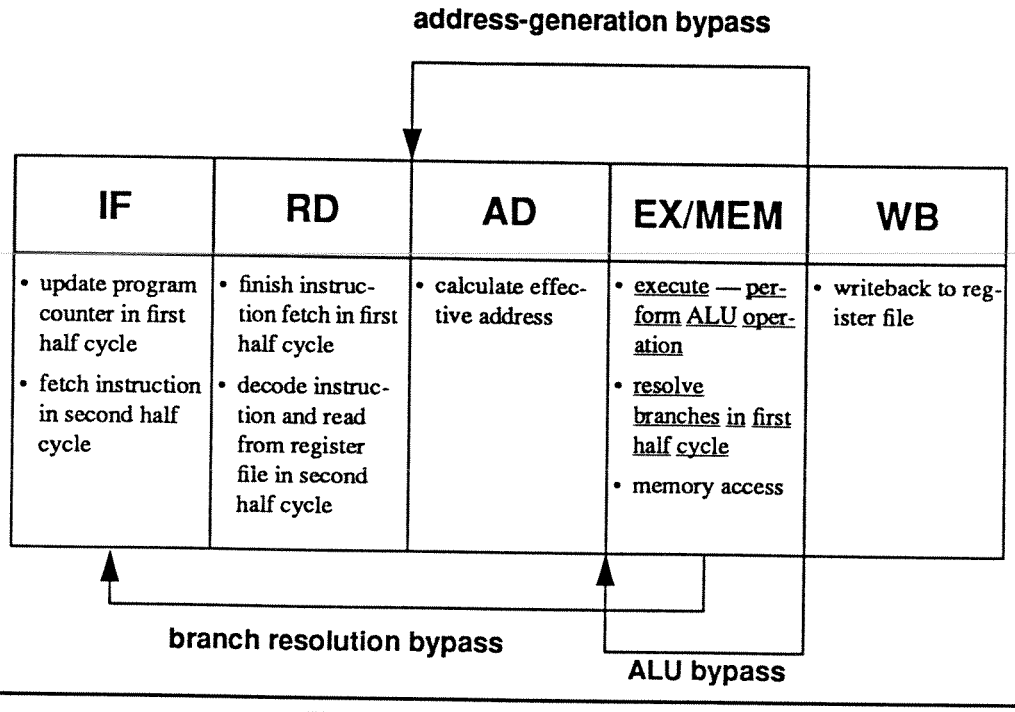


Figure 3: The AGI Pipeline

The five stages and bypass paths are shown. The actions in the EX/MEM stage that are underlined are moved from the EX stage in the LUI pipeline. See Figure 1.

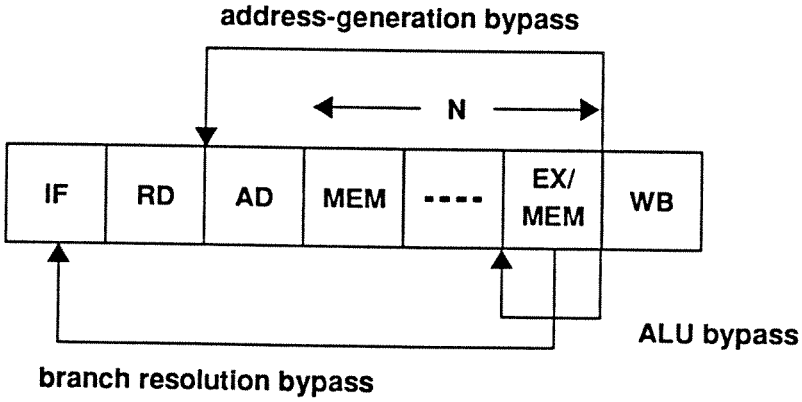


Figure 4: The AGI-N pipeline showing the bypass paths

The extra memory cycles and the corresponding increase in the address-generation bypass are shown.

instruction in the load-delay slot [11].

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 generalization 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 (see Figure 2). In an LUI-N pipeline, the scope of a load is N instructions and thus 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.

```
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)
```

Figure 5: A MIPS assembly language fragment

This code fragment illustrates a load-use hazard and an address-generation hazard.

3.2 The address-generation interlock pipeline

The AGI pipeline is shown in Figure 3. 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. 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. This address calculation is performed in the AD stage before the EX/MEM stage. By the time that an instruction which is dependent upon 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.

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. Second, 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 which takes N cycles to access the cache will require N-1 extra MEM stages. We refer to this as an AGI-N pipeline, as shown in Figure 4. In an AGI-N pipe, N instructions must be squashed every time a branch is mispredicted. Thus, address-generation interlocks can last from 1 through N cycles. If the first dependent load instruction is issued k cycles after the instruction which generates its base register, then the interlock will stall the pipeline for (N-k)+1 cycles.

The difference between the two pipeline organizations is further illustrated with the code fragment written in MIPS assembly language shown in Figure 5. NOPs in load-delay slots have been removed—load-use interlocks are modeled 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 is a branch-delay slot. Only the additional instructions in the branch scope for the AGI pipeline are speculatively executed.

4 The 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 [20]. 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 which optimize for the two different pipeline structures.

The experiments use the SPEC 92 integer benchmarks, summarized in Table 1. All of the benchmark programs were executed to completion using one of the "reference" input files provided by SPEC except `xlisp`, which used the "short" input file. The benchmarks are compiled three times. The MIPS C Compiler creates one version of each program. The MIPS C Compiler heavily optimizes the code and assumes a single load-delay slot. In effect, this provides a binary which is optimized for a single-cycle load-use hazard. GCC is used to create two versions of each benchmark. The versions differ in the cost function given to GCC's scheduling algorithm.

Benchmark Name	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	loada1	436 172 261	4.6
xlisp	short	1 171 528 797	3.0

Table 1: The SPEC92 integer benchmarks and their characteristics

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 which 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 prioritized, GCC attempts to schedule each instruction so that the pipeline will never interlock. For a discussion of scheduling techniques for pipelined processors, see [12].

To provide a binary which optimizes 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 optimized to reduce address-generation hazards, the scheduler is told that a two-cycle address-generation hazard is present. The study includes the MIPS C Compiler version because it is the standard compiler for systems using the MIPS processors. Comparing the code produced by GCC to the MIPS C Compiler's version, provides a check that the code which is produced by GCC for the AGI pipeline is equally well optimized.

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 M. Smith consumes the trace [19]. 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, because, as cache access times get longer, code expansion caused by NOPs in unfilled delay slots becomes a problem [17]. 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.

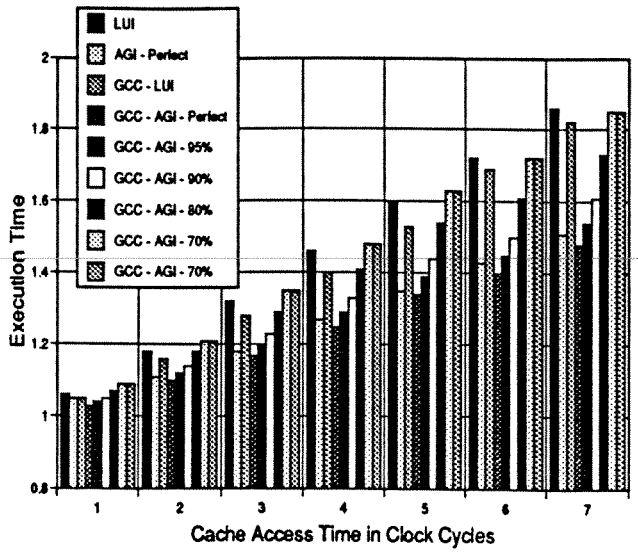


Figure 6: The harmonic mean of all benchmarks

5 Experimental results

The results of all experiments are summarized in Figures 6-12. In these figures, the x-axis shows the access time of the data cache in cycles. The y-axis shows an execution time which is normalized 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 presented in Figure 6. High numbers indicate poor performance. When the benchmarks `eqntott` and `xlisp` are simulated for large cache access times, their run times overflow the cycle-counting capabilities of the simulator. Because of this, some of the experiments are missing from Figure 8 and Figure 12 for cache access times of six and seven cycles. To make the comparison between pipelines fair, `xlisp` and `eqntott` are removed from the harmonic mean calculations for these two cache access times.

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 labeled "LUI" and "AGI - Perfect" in the Figures 6-12. For low cache access times, there is very little difference between the two pipeline organizations. As the access time increases beyond about 3 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.

The 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 Figures 6-12, these experiments are labeled "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 percent 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 3 in the case of `xlisp`—see Table 1—so modifying the code scheduling costs may not have a large effect. The limited improvement obtained from the compiler suggests that more aggressive optimization techniques, such as those in [8] may be needed. However, the performance of the Gnu C Compiler versus the MIPS C Compiler—compare LUI vs. 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 above. More sophisticated compiler techniques may provide further improvement.

The final set of results, labeled "GCC-AGI-X%" represent AGI pipelines with X% branch prediction over all branches, including unconditional jumps and calls. 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 was 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 is calculated with the following formula:

$$\text{Penalty} = N \times (1 - b) \times C_b$$

where N is the 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.

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.

Because significant performance improvement is seen in some of the benchmarks, even without sophisticated compiler support, one can examine the properties of the benchmark itself to see where the improvements occur. Programs which rely heavily on dynamic data structures see a particularly large benefit from the AGI pipeline. In the benchmark `sc`, the AGI pipeline outperforms the LUI pipeline even with poor branch prediction. `espresso` and `gcc` also realize significant performance benefits. In these programs, the program reads from records with many fields. A base register pointing to the beginning of the record needs to be set up, but only once. Once this register is initialized, the values in the fields can be loaded using constant offsets. Only the instruction which sets the base register can cause an address-generation interlock, while each load instruction that follows it has the potential of causing a load-use interlock. Using an AGI pipeline seems to be a good way to increase performance on these "pointer-chasing" benchmarks.

6 Conclusions

A number of processors have recently been announced which eliminate the load-use interlock by overlapping the execute stage of the pipeline with cache access rather than address generation. 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. If branch prediction is less than 80% accurate, the LUI pipeline provides faster run times.

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 such as superblock scheduling 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 which cause the interlock.

Two questions remain unanswered. First, in our experiments for the LUI-N and AGI-N pipelines, we assumed that the access time of the instruction cache remains at one cycle—the branch delay slot. In real systems it is unlikely that there would be significant difference in the access time of the instruction and data caches. For example, the DEC 21064 implementation of the Alpha architecture has a two-cycle access time for both caches [2] [16]. For each additional cycle of instruction cache access time, an IF stage must be added to the pipeline The scope of branches is increased by one instruction per additional IF stage. Once the extra IF stages are added, instructions must be speculatively fetched by both the LUI-N and the AGI-N pipelines after a branch is issued but before it is resolved. If the branch is mispredicted, these instructions must be squashed in both pipelines. This may affect the comparison between the two pipelines.

Second, we have simulated machines which 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.

Bibliography

[1] B. Case, "Intel reveals pentium implementation details," *Microprocessor Report*, vol. 5, no. 23, pp. 9–17, 1993.

[2] D. Dobberpuhl, R. Witek, R. Alimon, R. Anglin, S. Britton, L. Chao, R. Conrad, D. Dever, B. Gleseke, G. Hoeppner, J. Kowaleski, K. Kuchler, M. Ladd, M. Leary, L. Madden, E. McLellan, D. Meyer, J. Montanero, D. Priors, V. Rajagopalan, S. Samudraia, and S. Santhanam, "A 200 MHz 64b dual-issue CMOS microprocessor," in *Proc. '92 IEEE Int'l Solid-State Circuits Conf.*, pp. 106–107, February 1992.

[3] M. Golden and T. Mudge, "Hardware support for hiding cache latency," Technical Report CSE-TR-152-93, The University of Michigan, Department of Electrical Engineering and Computer Science, Ann Arbor, MI, 48109-2122, 1993.

[4] L. Gwennap, "TFP designed for tremendous floating point," *Microprocessor Report*, vol. 7, no. 11, pp. 9–13, August 1993.

[5] L. Gwennap, "Cyrix describes pentium competitor," *Microprocessor Report*, vol. 7, no. 14, pp. 1,6–10, October 1993.

[6] L. Gwennap, "Intel reveals Pentium implementation details," *Microprocessor Report*, vol. 7, no. 4, pp. 9–17, March 1993.

[7] P. Y. T. Hsu, "Designing the TFP microprocessor," *Micro*, vol. 14, no. 2, pp. 23–33, April 1994.

[8] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Birmingham, R. G. Oullette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The superblock: An effective technique for VLIW and superscalar compilation," *The Journal of Supercomputing*, vol. 7, no. 1/2, pp. 229–248, 1993.

[9] M. Johnson, *Superscalar Microprocessor Design*, Prentice Hall, Englewood Cliffs, N.J., 1991.

[10] N. P. Jouppi, "Cache write policies and performance," Technical report, Digital Equipment Corporation Western Research Laboratory, 250 University Ave., Palo Alto, CA, 94301, December 1991.

[11] G. Kane and J. Heinrich, *MIPS RISC Architecture*, Prentice-Hall, Englewood Cliffs, NJ, 1992.

[12] S. M. Krishnamurthy, "A brief survey of papers on scheduling for pipelined processors", *SIGPLAN Notices*, vol. 25, no. 7, pp. 97-106, July 1990.

[13] "MIPS chip set implements full ECL CPU," *Microprocessor Report*, vol. 3, no. 12, pp. 1,14–19, December 1989.

[14] K. Olukotun, T. Mudge, and R. Brown, "Performance optimization of pipelined memory caches," in *Proc. 19th Ann. Int'l Symp. Computer Architecture*, pp. 181–190, Gold Coast, Australia, May 1992, IEEE Computer Society Press.

[15] B. R. Rau and J. A. Fisher, "Instruction-level parallel processing: history, overview, and perspective," *The Journal of Supercomputing*, vol. 7, no. 1/2, pp. 9–50, 1993.

[16] J. Shortt. *Alpha AXP Architecture DECUS Presentation*, 1992.

[17] R. L. Sites, *Alpha Architecture Reference Manual*, Digital press, Maynard, MA, 1992.

[18] J. E. Smith and S. Weiss, "PowerPC 601 and Alpha 21064: a tale of two RISCs," *Computer*, vol. 27, no. 6, pp. 46–58, June 1994.

[19] M. D. Smith, "Tracing with pixie", Center for integrated systems, Stanford University, Stanford CA, 94305-4070, 1.1 edition, April 1991. Available by anonymous ftp from velox.stanford.edu.

[20] R. M. Stallman, *Using and Porting GNU CC*, Boston, MA: Free Software Foundation, Inc., 2.4.5 edition, 1993.

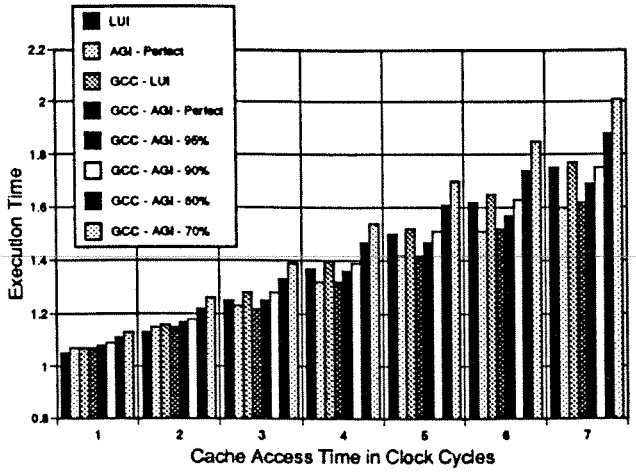


Figure 7: compress

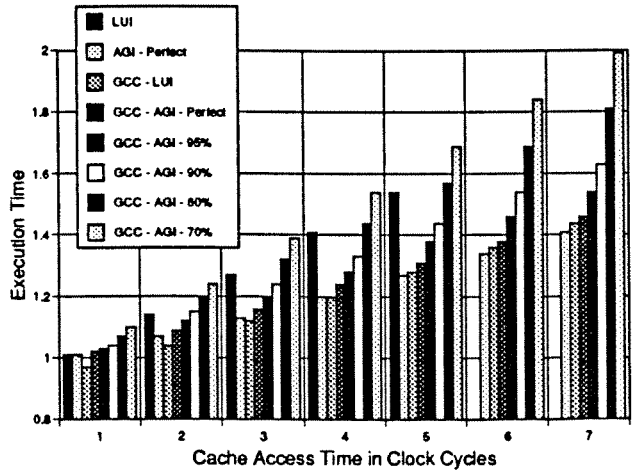


Figure 8: eqntott

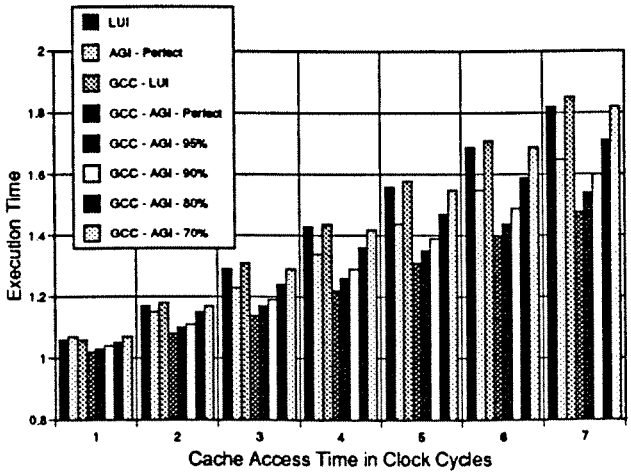


Figure 9: espresso

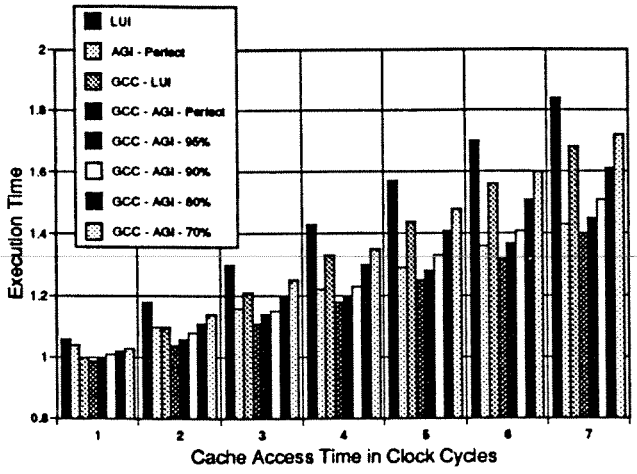


Figure 10: gcc

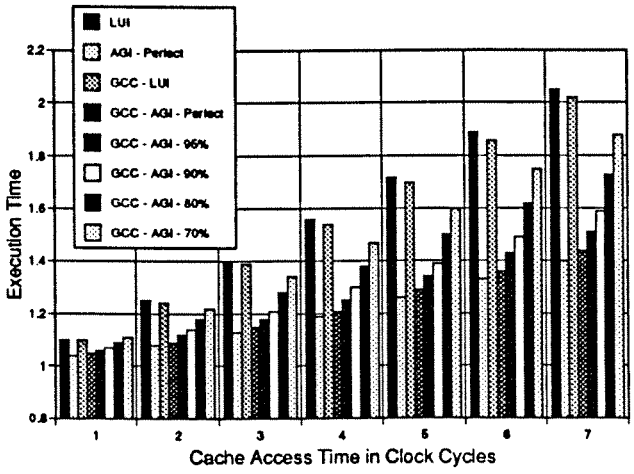


Figure 11: sc

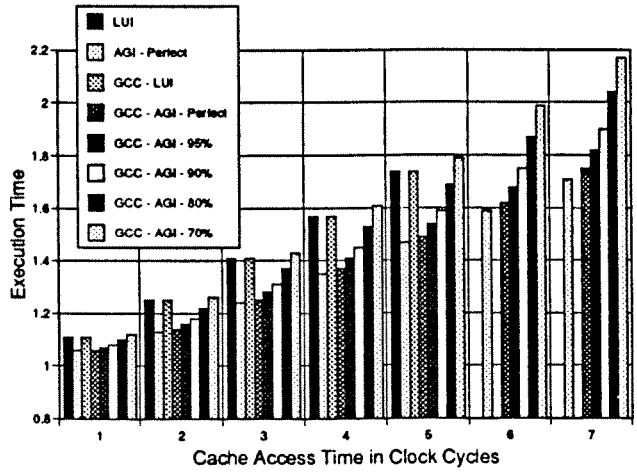


Figure 12: xlisp