RC 20857 (91318) 5/22/97
Computer Science/Mathematics 12 pages

Research Report

Run-Time Detection and Recovery from Incorrectly Reordered Memory Operations

Mayan Moudgill, Jaime H. Moreno

IBM Research Division
T.J. Watson Research Center
P.O. Box 218
Yorktown Heights, New York 10598

IBM Research Division
Almaden · T.J. Watson · Tokyo · Zurich · Austin

Run-time detection and recovery from incorrectly reordered memory operations

Mayan Moudgill (914-945-4348, mayan@watson.ibm.com)
Jaime H. Moreno (914-945-4340, jmoreno@watson.ibm.com)
IBM T.J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Abstract

We present a new approach for run-time detection of incorrectly reordered memory operations. This problem arises when a load operation is statically scheduled for execution earlier than sequentially preceding store operations, but the compiler cannot prove whether the memory locations accessed by the operations are disjoint. Previous techniques for dealing with this problem compare the memory addresses referenced by the reordered load and store instructions; if the addresses overlap, the load instruction is regarded as incorrectly reordered, and a recovery sequence is invoked to correct the error. Instead, the approach described here compares the *value* loaded out-of-order with the *value* in the corresponding memory location at the original position of the load instruction in the program; a recovery sequence is invoked whenever the two values are not identical. In contrast to previous techniques, this new approach is characterized by constant overhead independent of the type of interference among memory references (e.g., operands with different size, overlapping operands, multiple reordering) and handles special cases (e.g., asynchronous modifications, context switches).

We have incorporated this technique in CHAMELEON, our research compiler for instruction-level parallelism, and have applied it to the case of a VLIW processor. Simulation results have shown performance improvements up to 16% in some cases, and 4 to 7% in a wider sample, over scheduling without reordering ambiguous memory operations. The algorithm used selects load instructions for disambiguation in a way that causes negligible code expansion due to the recovery sequences, and very infrequent execution of such recovery sequences.

1 Introduction

Contemporary high-performance processors rely on instruction reordering for reducing the overall execution time of programs. In particular, the early (out-of-order) execution of memory load operations may hide delays in accessing memory, including potential cache misses, and may enable the use of idle resources on sequences of operations that depend on the data loaded. For example, in the second statement listed in Figure 1a, the early execution of the load instruction required to access the contents of the memory location pointed to by `*Y` may reduce the execution time of the statement.

One factor that limits the ability to reorder memory operations is *ambiguous memory references*; this is the case when a memory load operation appears after a memory store operation in a sequential instruction stream, and it is not possible to determine statically whether the memory locations accessed by the operations are disjoint. For example, the load operation implied by `*Y` in the second statement in Figure 1a appears after the store operation implied by `*X` in the first statement; if it is not possible to determine whether the locations accessed are disjoint, then performing the load operation earlier than the store operation might lead to incorrect results.

We say that a load operation and a sequentially preceding store operation *interfere* whenever the addresses of the memory locations accessed by the operations are ambiguous and actually overlap

```
*X = (a+b+2)<<4
r  = ((*Y)+c)^d
```

(a)

```
if ( X == Y )
  *X = (a+b+2)<<4
  r  = ((*Y)+c)^d
else
  r  = ((*Y)+c)^d
  *X = (a+b+2)<<4
endif
```

(b)

```
r  = ((*Y)+c)^d
*X = (a+b+2)<<4
if ( X == Y )
  r  = ((*Y)+c)^d
endif
```

(c)

Figure 1: Reordering memory operations: a) original sequence; b) safe reordering; c) recovery-based reordering

at run-time. In such a case, correct execution requires performing the load operation after the store operation has completed.

It is possible for a compiler to generate code that, at run-time, determines whether ambiguous load and store references actually interfere [1, 2]. If these references do not interfere, the sequence of instructions executed is one wherein the load instruction has been moved over the store instruction, as depicted in Figure 1b; on the other hand, if the references interfere, the instructions are executed in the original program order. We call this approach *safe reordering* because it checks for correctness before executing the reordered instructions.

Alternatively, in *recovery-based reordering* the load instruction is always executed before the store instruction, as depicted in Figure 1c [3, 4]. Additional code is inserted for checking whether there was interference among the reordered operations. If so, the load instruction and those operations depending on the value loaded are reexecuted; these extra instructions are called the *recovery sequence.*

Safe and recovery-based reordering as described above use an *interference test* which consists of comparing the *addresses* of the memory locations accessed by load and store operations; if such addresses are identical, the two operations are regarded as interfering.

In recovery-based reordering, the recovery sequecne is executed whenever there is interference, which implies performing the associated work twice. In spite of this fact, recovery-based reordering can yield better performance than safe reordering; in safe reordering, the out-of-order load instruction cannot be executed until the memory address referenced by the store instruction is known, thus delaying the early execution of the load instruction.

The research reported in this paper relates to software-based implementations (without hardware support) of recovery-based reordering. Existing realizations of this approach implement the interference test using a sequence of one or two instructions; the sequence compares equality among the memory addresses referenced by the ambiguous load and store operations, and performs a branch based on the result from the comparison. In the cases where the equality comparison is sufficient, the overhead introduced by the interference test is small. Unfortunately, testing for equality is not always sufficient; more complex interference tests, involving several instructions, have to be performed in some cases, as illustrated by the following examples:

Unaligned store/load operations. Store and load operations access several bytes (say 4), and addresses can be byte-aligned. As a result, the interference test must check for an overlap, such as `X-Y < 4 && Y-X < 4` or, preferably, `abs(X-Y) < 4`.

Objects of different size. Store and load operations access a different number of bytes (say a halfword and a word, 2 and 4 bytes respectively). In this case, the interference test must also check for overlap, but the check is more complex, such as `X-Y < 2 && Y - X < 4`.

Multiple store operations. Load operations are moved over several ambiguous store operations. As a result, the interference test must check for multiple addresses (detect interference with any of the store operations).

Asynchronous updates. Memory is modified by store operations from sources other than the running program, such as in a multitasking system, a system with asynchronous I/O, or a multiprocessor system. In these cases, the address-based interference test is not sufficient to detect that values have changed.

The limitations described above can be solved with dedicated hardware resources for monitoring addresses and check for interferences [5, 6]. However, such hardware resources are usually based on associative memories, so they are expensive and limited, and only a few memory locations can be monitored simultaneously. This restricts the degree of reordering possible; only a few out-of-order load operations can be "live" at any point in the program. Furthermore, address monitoring must be a continuous process; in particular, it is not possible to reliablely monitor a location across context switches. For correctness, interference must be reported for any reordered load operation which is live across a context switch.

In this paper, we propose a new approach for software-based run-time detection of incorrectly reordered ambiguous memory operations. In contrast to the earlier techniques, the approach described here compares the *value* loaded out-of-order with *the value in memory* at the same address at the in-order position of a load instruction in the program. If these values are the same, the out-of-order load operation has been performed correctly; otherwise, a recovery sequence must be executed. This approach is characterized by a low and constant overhead test (two or three instructions per load operation, independent from the number of ambiguous store operations and the size of the operands) which can be implemented on existing instruction-set architectures, without requiring any additional hardware. Moreover, the approach handles special cases such as asynchronous modifications to memory and context switches.

We have implemented this approach in a compiler targeted for a VLIW processor. The compiler reorders ambiguous memory operations, and generates the appropriate recovery sequences. Simulation results have shown performance improvements up to 16% in some cases, and 4 to 7% in a wider sample, over scheduling without reordering ambiguous memory operations. The reordering algorithm selects load instructions for disambiguation in a way that causes negligible code expansion due to the recovery sequences, and very infrequent execution of such recovery sequences.

The rest of this paper is organized as follows: in the next section, we provide more details on our scheme, its benefits, and its possible implementations using existing instruction sets. In Section 3, we introduce the environment in which we performed our experiments; this includes a description of the target processor, the compiler, and the simulation techniques. In Section 4, we describe the results of our experiments. Finally, we summarize our work in Section 5.

2 A coherence-based interference test

We propose an alternative approach towards implementing the test for the detection of interference among reordered memory operations. The proposed technique is built upon the following realization: if a load operation is scheduled for execution earlier than a preceding ambiguous store operation, correct program execution requires that the *value loaded out-of-order is the same as the value that would have been obtained if the load operation was executed in its original position.* Based on this realization, our interference test is as follows:

- re-execute the load operation at its in-order point;
- compare the newly loaded value with the value loaded out-of-order; and
- if the values are not identical, execute a recovery sequence.

For example, the code fragment depicted in Figure 1a is reordered and augmented with our interference test and recovery sequence as follows:

```
s = *Y
r = ((s)+c)^d
*X = (a+b+2)<<4
q = *Y
if( s != q )
  r = ((q)+c)^d
endif
```

We call this the *coherence test*, because it checks whether the value loaded out-of-order is coherent with (i.e., is the same as) the value in memory at the original position of the load operation. The proposed approach offers the following advantages:

- the test can be implemented in existing instruction-set architectures;
- the test uses three (or fewer) instructions per out-of-order load operation;
- the number of instructions required by the test is independent of the number of store operations over which the load instruction is moved;
- the test automatically handles partial modifications caused by overlapping store operations;
- the test automatically handles asynchronous updates to the memory location accessed by the reordered load operation[1];
- the test does not need any extra hardware.

As an example of its use with existing instruction set architectures, the coherence test can be implemented by using a sequence similar to the following:

```
load  r3, rX          ; reload the datum
cmp   cc0, r3, r4     ; r4 contains the out-of-order value
bne   cc0, recovery   ; branch if values not equal to recovery
```

On some architectures, such as Hewlett-Packard's PA-RISC [7], this may be a two instruction sequence, as follows:

```
load  r3, rX          ; reload
bne   r3, r4, fixup   ; compare the values and branch if not equal
```

An interesting variation is performing the transfer of control to the recovery sequence using a trap operation instead of a branch; the trap handler determines the address of the recovery sequence and branches to it. On the PowerPC [8] architecture, this can be implemented as:

```
load  r3, rX          ; reload
tne   r3, r4, handler; compare the values and trap if not equal
```

The requirement for an additional load operation might be considered as a drawback of the coherence test. However, the extra instruction-level parallelism exposed by dynamic disambiguation can be best exploited by wide-issue processors. Such processors have multiple functional units, including multiple load units. Consequently, we argue that the additional load operation is less of an issue for those machines; the performance gained by exploiting the extra instruction-level parallelism exposed by the coherence test compensates for the cost of the extra load operations.

[1] Assuming there is some support for multiprocessor memory coherency.

3 Application to a VLIW processor

We now describe the application of our interference test in the context of a VLIW processor. A general description of the VLIW architecture considered, its associated compiler, and the corresponding simulation environment are given in [9].

The compiler used in this work was designed to target very-long instruction word (VLIW) architectures that execute tree-instructions [10]. The primitive instructions composing the long instruction words are based on the PowerPC architecture [8]; departures from the PowerPC instruction set include:[2]

- larger register set than the 32 general-purpose, 32 floating-point, 8 condition registers[3] available in the PowerPC architecture;
- support for speculative non-trapping loads has been added;
- some complex operations have been deleted, including rotate-and-mask, update load/store, load/store multiple, and string operations.
- the "record" form of operations allows specifying any of the condition registers (instead of the implicit Condition Register 0);
- the displacement field in memory operations has been reduced from 16 to 12 bits;
- the encoding of some operations has been extended to 64 bits;
- some operations can support 32-bit immediate fields;
- some 3-input fixed-point operations, such as add&lshift, and&or have been added; these include all combinations of add and subtract with add, subtract, and shift, as well as all combinations of logic operations;
- some support for conditional execution has been added, in the form of conditional move and conditional store operations.

The compiler is extensively parameterized so that it can target VLIW processors with different features, such as issue width, number of functional units, instruction latencies, and register set. The preferred target implementation consists of up to sixteen operations (including eight load/store operations) per VLIW.

3.1 The CHAMELEON compiler

We have implemented reordering of ambiguous memory operations using the coherence test as an optimization pass in CHAMELEON, our research compiler (see Figure 2). CHAMELEON was designed to support research into instruction-level parallelism, and to evaluate the benefits of various architectural modifications when exploited through appropriate compiler optimizations.

The input to CHAMELEON is object code (*.o files) produced by a modified version of **xlc**, the standard RS/6000 C compiler [11], which is processed by an "object-code translator" that generates an assembly-like sequential representation (*.vinp files). The output from CHAMELEON is a VLIW program (tree-instructions) in an assembly language form (*.vasm files), which is either instrumented and translated into PowerPC assembly code (*.s files) that emulates the target VLIW processor, or is translated directly into PowerPC assembly code.

The modifications to **xlc** fall into two categories: first, we have turned off phases such as scheduling and loop transformations that would tend to obscure the original code sequence; second, we have added capabilities to convey information such as alias classes, registers live at function call and function return points, spill locations, etc. Among other benefits, this modified **xlc** allowed us to take advantage of an existing production front-end.

[2] The compiler can also be restricted to produce primitive instructions complying with the PowerPC architecture; such code is suitable for a wide-issue implementation of the PowerPC architecture.

[3] We treat the condition register fields in the PowerPC Condition Register as separate registers

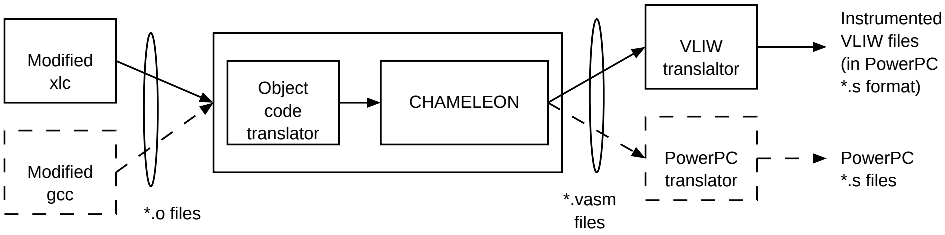

Figure 2: The CHAMELEON compiler environment

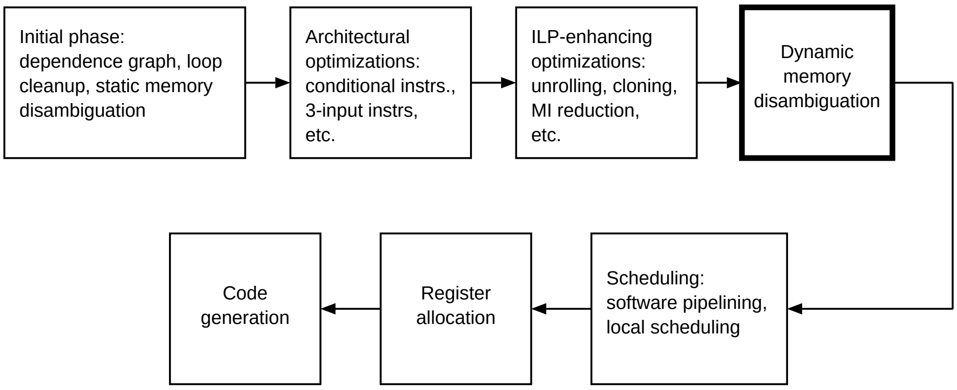

Figure 3: Phases in CHAMELEON

CHAMELEON has a fairly aggressive suite of optimizations which can be grouped into the following categories (see Figure 3):

Traditional. These include constant propagation, loop invariant code motion, dead-code elimination, and sub-expression elimination. These optimizations are applied throughout the compilation process. Moreover, since we use the output from **xlc**, we take advantage of **xlc**'s excellent set of traditional optimizations [12].

ILP-increasing. Optimizations designed for increasing the instruction-level parallelism, so that the scheduler can pack instructions more tightly. These include various loop transformations, such as unrolling, re-writing loops with commutative/associative operations (reductions), and re-writing cyclic dependences to reduce initiation intervals.

Architectural. Optimizations designed to exploit various architectural extensions. For instance, there is a phase that uses conditional move/store operations to convert if-then-else structures to straight-line code.

Optimizations not in CHAMELEON but found in other research compilers include profile-directed feedback and inter-procedural analysis. CHAMELEON has the ability to use profiling information but it normally uses synthetic branch probabilities produced by a variant of the Ball-Larus heuristics [13]. The results in this paper use such synthetic branch probabilities. We take no advantage of inter-procedural analysis; even the inter-procedural phase of **xlc** has been disabled.

The scheduler used by CHAMELEON is an enhanced version of selective scheduling [14]; as a result, all loops are subject to software pipelining, including those containing multiple paths and other loops. The original selective scheduling algorithm has been considerably modified; the changes include:

- examining all instructions in a loop for scheduling (instead of only instructions within a fixed "window");
- using heuristics for selecting the slot in which to schedule an instruction (instead of a "greedy" schedule); and
- sensitivity to register pressure (not scheduling an instruction if that might cause a register spill).

3.2 Memory disambiguation

Several techniques for static disambiguation of memory addresses are used in CHAMELEON. In particular:

- Aliasing information is passed from **xlc**, so most of the alias information available to a source-level compiler is used.
- Memory operations are further disambiguated based on symbolic address computation; symbolic addresses are computed for a whole procedure, and Banerjee and GCD tests are applied to such symbolic addresses.
- Inner loops containing memory dependencies which can be resolved by a simple loop-invariant test are cloned, creating two copies of the loop: the original (i.e., dependent) version, and a copy known to be free from memory dependencies.

Some of the store-load dependencies not resolved by the static disambiguation techniques summarized above are resolved by inserting the interference test proposed in this paper. We apply this disambiguation step as a separate phase, before scheduling, as opposed to doing it during scheduling. Currently, due to some implementation constraints in the compiler,[4] we are rather restrictive

[4] The new test reuses code from a partially completed optimization targeted for a hardware-assisted interference test; additional changes are required to be able to exploit the new test more extensively. We plan to make those changes in a future version of the compiler.

```
load   r3,rY       ; r3 = *Y
store  r7,r3       ; **Y = r7
load   r3,rY       ; r3 = *Y
load   r8,r3       ; r8 = **Y
```

Figure 4: "Almost always" dependence

Register class	Issue width 8	12	16
General-purpose	64	96	128
Floating-point	64	96	128
Condition	16	24	32

(a) Register set

Operation	Latency
Integer	1
Floating-point	3
Load	1
Integer divide	10
Integer multiply	3

(b) Instruction latencies

Figure 5: Size of register set and latencies of primitive operations

in using the proposed technique; the criteria used for selecting load operations subject to dynamic disambiguation are:

- the load operation must be in the most-probable path of a loop;
- there must be at least one store operation in the most-probable path of the loop upon which the load is ambiguously dependent;
- in sequences of the form shown in Figure 4 ("almost always" dependencies), the last load operation is assumed to be unambiguously dependent on the store operation;
- at most sixteen load operations are resolved by dynamic disambiguation per function.

Once a load operation is identified as a candidate for movement out-of-order, the interference test and the associated recovery sequence are introduced. Later, as operations dependent on the load are moved over the interference test (during scheduling and other phases), they are added to the recovery sequence.

The interference test has been implemented as a two instruction sequence: a load operation followed by a trap-if-not-equal operation. The results reported in Section 4 do not include the overhead arising from the execution of the trap handler, though the execution of the recovery sequences is measured. However, as can be inferred from the data gathered, the recovery sequence are invoked so infrequently that even a moderate overhead arising from the trap handler would not substantially change the resulting performance.

3.3 VLIW processor model

We consider three widths of a VLIW processor that uses all the extensions described earlier. The three widths are, respectively, capable of issuing any 8, 12 and 16 instructions per VLIW (including as many memory operations). The size of the register sets in each case are listed in Figure 5a.

The experiments were performed using the instruction latencies listed in Figure 5b, and assuming an infinite-size cache; a more detailed description of the simulation techniques used is given in [9].

Table 1: VLIW instruction ratio with respect to sequential code, with and without disambiguation

Benchmark	Issue-width 8 With	8 None	12 With	12 None	16 With	16 None
compress	4.14	3.82	5.14	4.50	5.79	4.83
eqntott	4.79	4.77	5.00	5.00	8.02	8.09
espresso	2.58	2.54	2.85	2.85	3.11	3.00
gcc	2.36	2.28	2.76	2.68	2.88	2.68
li	3.25	3.17	3.62	3.50	3.72	3.58

Table 2: Relative performance gain from coherence test

Benchmark	Issue-width 8	12	16
compress	7.78%	12.43%	16.52%
eqntott	0.37%	0.00%	-0.90%
espresso	1.75%	2.31%	3.40%
gcc	3.34%	2.87%	6.85%
li	2.50%	3.29%	3.70%

4 Simulation results

Table 1 reports the additional instruction-level parallelism (ILP) obtained in the VLIW code by adding dynamic disambiguation using the coherence test. This table indicates the ratio between the number of instructions executed by a RS/6000 processor running PowerPC code (i.e., with ILP=1) to the number of tree-instructions executed by the VLIW processor, for the different issue-widths. The PowerPC instruction counts used for these ratios are obtained from compiling the programs with **xlc** at optimization level -O2. The programs used are from the SPEC 92 and SPEC 95 suites, over the reference and training inputs, respectively.

Note that we compute instruction-level parallelism differently from many other results reported in the literature. Usually, the results reported are obtained by using the same compiler for both the parallel implementation and the sequential implementation. In contrast, the instruction counts for the sequential implementations are obtained using the best compiler available for the PowerPC architecture [12]. This is motivated by our original research goal, namely measuring the potential improvement in instruction-level parallelism in a PowerPC-based VLIW processor over existing PowerPC implementations. As can be inferred from Table 1, even under these stringent conditions, the improvement in ILP is comparable to that previously reported in the literature.

Table 2 quantifies the benefits of disambiguation using our proposed technique, by providing the relative performance gain obtained with reordering of ambiguous memory operations with respect to the case without such a reordering (i.e., the ratio among the values listed in Table 1). As it can be inferred from this table, the range in performance gain can be quite wide, from being detrimental as in the case of `eqntott` to almost 17% in `compress`. As expected, the gain is larger for wider-issue implementations.

Table 3: Behavior of reordering algorithm

Benchmark	Issue-width 8 Loads	8 Tests	8 Size	12 Loads	12 Tests	12 Size	16 Loads	16 Tests	16 Size
compress	14	47	602	14	51	806	14	53	1189
eqntott	53	183	666	71	270	965	89	315	1562
espresso	590	1369	4084	606	1573	6214	618	2193	8874
gcc	1549	2563	7657	1641	3027	11312	549	1371	6472
li	171	291	728	173	305	827	174	330	1049

Table 4: Dynamic behavior of reordered code

Benchmark	Issue-width 8 Tests	8 Failed	8 Opers	12 Tests	12 Failed	12 Opers	16 Tests	16 Failed	16 Opers
compress	2.8%	0	0	2.6%	0	0	2.5%	0	0
eqntott	0.3%	0	0	0.3%	0	0	0.3%	0	0
espresso	1.7%	101	202	1.6%	354	708	1.5%	354	708
gcc	1.3%	58	332	1.3%	294153	588516	1.2%	30	270
li	2.4%	0	0	2.4%	0	0	2.2%	0	0

Table 3 contains information regarding the behavior of the reordering algorithm and the compiler. This table includes the total number of out-of-order load operations, verification tests inserted, and operations in the recovery sequences. The number of tests inserted may differ from the number of out-of-order load operations because verification tests may be duplicated (e.g., during loop-unrolling). As stated earlier, the rather restrictive reordering algorithm currently implemented leads to scheduling only some ambiguous load operations out-of-order. Thus, the total number of reordered load instructions is rather small, and the maximum performance gain achieved in these experiments might be improved. We continue researching better criteria for scheduling such operations. The average number of operations per recovery sequence is rather small, in the range from about 3 to 5 operations, with the exception of benchmark compress which ranges from about 12 to 22 operations per sequence.

Table 4 contains information about the dynamic behavior of the reordered code with respect to disambiguation. This table indicates the fraction of the total operations executed that are a test (i.e., which represents the out-of-order loads), the number of test failures, and the total number of operations executed by the recovery sequences. Reordered load operations are executed rather infrequently, and fail even less frequently. In fact, apart from the anomalous situation with gcc in the 12-way configuration, the total number of test failures and recovery operations executed is insignificant. Even in that anomalous case, the number of recovery operations executed is less than 0.05% of the total number of operations in the program. The infrequent failure rate indicates that the corresponding store operations rarely interfere.

The data in the previous tables corresponds to the case of VLIW processors capable of issuing any 8, 12 or 16 instructions, including all load operations. In contrast, Table 5 illustrates the effects of restricting the number of load operations per VLIW for the case of a processor with issue-width

Table 5: Relative performance gain, issue-width 16

Benchmark	Load operations per VLIW 16	8	4
compress	16.52%	14.97%	16.55%
eqntott	-0.90%	-0.66%	-0.87%
espresso	3.40%	3.89%	2.06%
gcc	6.85%	4.39%	3.34%
li	3.70%	3.73%	1.96%

16. The entries represent the relative performance gain obtained with reordering with respect to the case without such a reordering, for each of the configurations in the table. (Thus, the first column in this table is the same as the last column in Table 2.) As can be inferred from this table, the coherence test allows exposing more instruction-level parallelism than without reordering the ambiguous memory operations, even in the case of fewer load operations per VLIW; as expected, in most cases the benefits decrease for fewer load operations per VLIW, though in some cases such as benchmark compress there is no real change.

5 Concluding remarks

We have presented a new approach for run-time detection of, and recovery from, incorrectly reordered memory operations. In contrast to earlier techniques which rely on run-time comparison of memory addresses, the approach described here compares a value loaded out-of-order with the value at the corresponding memory location at the in-order point of the load instruction. The approach is characterized by being applicable using existing instruction-set architectures, by introducing constant overhead independent from the complexity of the interference among memory operations, by not requiring special hardware resources, and by providing functionality not previously available such as dealing with asynchronous updates to memory.

We have described the application of the technique in static (compile-time) scheduling for a VLIW processor, wherein each statically generated ambiguous out-of-order load instruction is complemented with a test for run-time detection of incorrect execution, and a recovery sequence. Simulation results for VLIW processors of varying issue-width have indicated performance gains up to 16% with respect to scheduling without reordering the ambiguous memory operations, with negligible code expansion due to recovery code and very few invocations of such recovery code due to incorrect execution. These results were obtained using a compiler which performs aggressive optimizations towards extracting and exploiting instruction-level parallelism, in which reordering ambiguous memory operations is one of such optimizations.

The experiments reported use a rather restrictive algorithm for deciding which ambiguous memory operations can be reordered. We continue investigating alternative criteria for this decision; thus, we expect the results reported here can be improved further. In addition, we are exploring the suitability of the proposed interference test to the case of processors with narrower issue-width than those considered here.

Acknowledgements

We acknowledge and thank the contributions from Erik Altman for the VLIW-to-PowerPC instrumenting translator and associated simulation environment, Arkady Polyak for the object-code translator, and Rene Miranda for the VLIW-Native PowerPC translator. We also thank Kemal Ebcioglu, Brian Hall, Shy-Kwei Chen, Peter Oden, Balaram Sinharoy, Richard Goldberg and Norman Cohen, for their contributions to different parts of CHAMELEON.

References

[1] A. Nicolau, "Run-time disambiguation: coping with statically unpredictable dependencies," *IEEE Transactions on Computers*, vol. 38, May 1989.

[2] A. Huang, G. Slavenburg, and J. Shen, "Speculative disambiguation: a compilation technique for dynamic memory disambiguation," in *21st Intl. Symposium on Computer Architecture*, (Chicago, Illinois), pp. 200–210, 1994.

[3] G. Silberman and K. Ebcioglu, "An architectural framework for migration from CISC to higher-performance platforms," in *International Conference on Supercomputing*, pp. 198–215, 1992.

[4] F. Amerson, R. Gupta, V. Kathail, and M. Schlansker, "Memory processor that permits aggresive execution of load instructions." UK Patent Application GB 2265481A, Sept. 1993.

[5] M. Franklin and G. Sohi, "ARB: a hardware mechanism for dynamic reordering of memory references," *IEEE Transactions on Computers*, 199.

[6] F. McKeen, M. Adler, J. Emer, R. Nix, D. Sager, and P. Lowney, "Mechanism for enforcing the correct order of instruction execution." United States Patent 5,420,990, May 1995.

[7] G. Kane, *PA-RISC 2.0 architecture*. Prentice-Hall PTR, 1996.

[8] IBM, *PowerPC architecture*. IBM, 1993.

[9] J. Moreno, M. Moudgill, K. Ebcioglu, E. Altman, B. Hall, R. Miranda, S. Chen, and A. Polyak, "Architecture, compiler and simulation of a tree-based VLIW processor," Tech. Rep. RC 20495, IBM T.J. Watson Research Center, July 1996.

[10] K. Ebcioglu, "Some design ideas for a VLIW architecture for sequential natured software," in *Parallel Processing (Proceedings of IFIP WG 10.3 Working Conference on Parallel Processing)*, pp. 3–21, North Holland, 1988.

[11] IBM, *AIX XL C compiler*. IBM, 1993.

[12] M. Auslander and M. Hopkins, "An overview of the PL.8 compiler," in *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, pp. 22–31, June 1982.

[13] T. Ball and J. Laurus, "Branch prediction for free," in *Proceedings of the 1993 SIGPLAN Conference on Programming Language Design and Implementation*, pp. 300–313, June 1993.

[14] S. Moon and K. Ebcioglu, "An efficient resource-constrained global scheduling technique for superscalar and VLIW processors," in *25th International Conference on Microarchitecture (MICRO-25)*, pp. 55–71, IEEE Press, Dec. 1992.