A Scheme for Selective Squash and Re-issue for Single-Sided Branch Hammocks

Karthik Sankaranarayanan, Kevin Skadron
Dept. of Computer Science
University of Virginia
Charlottesville, VA 22904

Introduction

This abstract describes work to minimize re-execution of control independent instructions. This technique differs from prior work in its emphasis on compiler scheduling in order to minimize changes to the hardware of an out-of-order processor. Work so far has focused on single-sided branch hammocks.

A *branch hammock* [1] is an instruction sequence corresponding to an 'if' language construct. It is called *double-sided* when it corresponds to an 'if-then-else' construct and *single-sided* when it corresponds to an 'if-then' construct alone (*i.e.*, without the 'else' part). When the 'then' and the 'else' contexts contain only one basic block each, the branch hammock is called *simple*. In typical speculative processors, when a conditional branch corresponding to such a branch hammock is mispredicted, all the instructions fetched after the branch are *squashed*. However, the 'join' context is executed regardless of the direction of the branch. When a misprediction is detected, if the fetch engine of the processor went past the 'join' context, it fetched some potentially useful instructions too. If those instructions can be identified by some co-operative work between the compiler and the hardware, redundant re-fetching and re-execution can be eliminated. This work attempts to implement and evaluate such a co-operative mechanism in the particular case of *single-sided, simple branch hammocks.* An extended report of this work appears in [2].

Related Work

Rotenberg *et al.* [3] have analyzed control independence in superscalar processors. Their paper analyzes the bounds of potential performance improvement due to the exploitation of control independence and assesses the complexity of possible implementations. Sodani *et al.* [4] have done a detailed study on dynamic instruction reuse. The potential *branch hammock* reuse described above, called squash reuse, is a subset of such dynamic instruction reuse. However, the re-fetching of instructions is not eliminated in their technique, and reuse techniques typically require substantial, multi-ported lookup tables and other hardware support. Rychlik *et al.* [5], proposed the reduction of value misprediction penalties by re-issuing of value-mispredicted instructions to the functional units, thus eliminating their re-fetching and re-renaming. However, their work does not examine re-issue with respect to branch mis-speculation. Klauser *et al.* [6] proposed the dynamic predication technique for reducing misprediction penalties in case of simple branch hammocks. The instructions of a non-predicated instruction set are predicated dynamically using hardware augmentation. Stark *et al.* [7] proposed out-of-order fetch to reduce the impact of instruction cache misses. In their technique, instructions are inserted into the reservation stations out-of-order.

Overview of the Implementation

This work combines compiler scheduling with ideas from the above-mentioned works (control independence, re-use, re-issue) in the domain of optimizing *single-sided, simple branch hammocks.* In such hammocks, the join context is control independent of the branch and hence need not be re-fetched on a misprediction. The execution of the instructions in the 'join' context that are data independent of the 'then' context is not erroneous and hence these instructions can be safely re-used. However, the execution and the dependency information of the 'join' context instructions that *are* data dependent on the 'then' context is erroneous. Hence, they have to be re-issued after the misprediction recovery in order to receive the proper operand values. For the above branch hammock scenario, when there is co-operation from the compiler, minimal addition to the hardware can implement such *selective squashing* (the 'join' context is not squashed) and *selective re-issue* (the dependent instructions should be re-issued).

In our work so far, the scheduler identifies the *single-sided simple branch hammocks* in a program. It finds the instructions in the 'join' context that are data independent of the 'then' context and groups them together at the beginning of the 'join' context. The code motion takes place in such a way that no anti or output dependences are violated. It then annotates the branch instruction with the size of the hammock, the number of independent instructions and the offset of the join context. With such 'grouping support' from the scheduler, the hardware implementation of selective squash and re-issue can be done with minimal cost. Consider an out-of-order processor model with an in-order commit stage. The following figure (Fig. 1) shows a simple branch hammock in the instruction-reorder buffer as a mispredicted branch is about to be discovered.

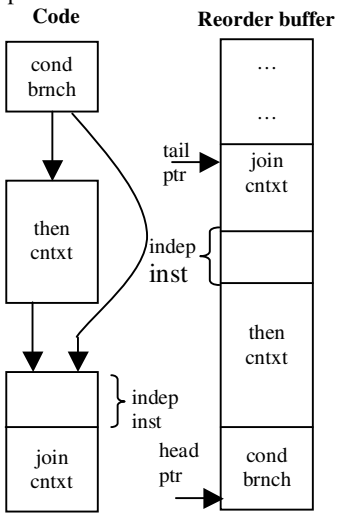

Fig. 1: State of the instruction-reorder buffer containing a hammock and a mispredicted branch.

Selective squashing of the 'then' context is easily implemented in the hardware by just bringing the head pointer to the beginning of the 'join' context. The effects of the 'then' context instructions on the register rename map is repaired by logical masking operations of the current map with the register

maps at the position of the branch and at the beginning of the join context. Also, the dependent instructions of the 'join' context should enter the rename stage once again. In order to achieve this as simply as possible, the implementation expands the instruction buffer between the fetch and decode stages and holds instructions there. The dependent instructions of a hammock are tagged in the instruction queue and are held there until the branch resolves, making this re-issue easy. On a misprediction, all instructions in the instruction queue except the tagged ones are squashed. This ensures that the dependent instructions get to re-issue to the functional units with the proper values. Also, when the mis-speculated branch was predicted taken, the processor remembers the start of the 'join' context in order to re-steer the fetch engine when necessary. Finally, on a successful branch resolution, all associated instructions are purged from the fetch queue. Together, these implementation features require minimal addition to hardware and should be feasible without affecting the clock rate. They also avoid large, dedicated, multi-ported re-use tables.

	compress	**gcc**	**go**	**ijpeg**	**li**	**m88ksim**	**perl**	**vortex**
Scheduled hammocks	5	984	341	68	13	85	40	13
Max. Size of hammocks	4	8	7	9	4	7	6	9
Max. Size of then cntxt	3	6	6	3	2	4	3	3
Max. No. of indep. Inst	2	6	4	6	1	3	4	1
Avg. Size of hammocks	4	4	3.3	4	3.4	3.9	4	5.2
Avg. Size of then context	2.4	2.2	2.2	2.3	2	2.4	2.1	2.2
Avg. No. of indep. Inst	1.4	1.4	1	1.5	1	1.2	1.7	1

Table 1: Static scheduling data

This work implements the features detailed above using the SimpleScalar v3.0 [8] simulator tool set and the PISA instruction set. Performance evaluation has been done by simulating SpecInt95 benchmarks on the modified simulator and then comparing with the unmodified version. For scheduling, the benchmarks were compiled to assembly using gcc 2.6.3 –O3, scheduled by the scheduler software, and then assembled into the binaries. These were the binaries run on the modified simulator. Data from the static scheduler is summarized in Table 1.

Initial Results and Future Work

Initial investigation of the results show that the candidates chosen by the scheduler are very small basic blocks of size 3-4 instructions, and the scheduler typically finds at most 1-2 independent instructions but often finds none. This result happens to be similar to success rates in filling branch delay slots [9], which makes sense because the scheduling task is essentially identical. The important difference here is that we are able to use scheduling to exploit control independence in wide-issue, out-of-order organizations and are not limited to a fixed delay-slot architecture.

Unfortunately, because single-sided hammocks expose so few independent instructions, the gains in IPC obtained with this technique are negligible, averaging less than 0.05%. Some benchmarks, like *compress*, have almost no suitable basic blocks (see Table 1). Based on the control independence studies in [3], we can see that our technique exploits very little of the possible control independence. This is mainly because of the restriction of our approach to *single-sided, simple branch hammocks.* Moreover, our technique involves only compile time decisions to expose independent instructions. For going beyond one basic block, dynamic techniques that make use of run-time information might be more useful. Dynamic predication [6], which uses such dynamic information, would provide better benefits in these scenarios. These results necessitate the extension of the scheduling technique to nested and double-sided hammocks also. An analysis of the effectiveness of predication in lieu of or in combination with such re-issue and re-use techniques should also be explored. A study of the contrasts and overlaps between the natures of control independence exploited by these techniques (predication, re-use/re-issue) is another interesting future direction. In addition to the immediate benefits from exploring control independence, it is also expected that this work will also guide efforts in using compiler analysis to directly improve branch-predictor performance and/or reduce predictor hardware requirements.

Acknowledgements

This material is based in part on work supported by the National Science Foundation under grant no. CCR-0082671.

References

[1] J. Ferrante, K. Ottenstein, and J. Warren. "The Program Dependence Graph and Its Use in Optimization". *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[2] K. Sankaranarayanan and K. Skadron. "A Scheme for Selective Squash and Re-issue for Single-Sided Branch Hammocks." Tech Report CS-2001-14, Univ. of Virginia Dept. of Computer Science, July 2001.

[3] E. Rotenberg, Q. Jacobson, J. Smith. "A Study of Control Independence in Superscalar Processors". In *Proc. of the 5th International Symposium on High Performance Computer Architecture*, January 1999.

[4] A. Sodani and G. S. Sohi. "Dynamic Instruction Reuse". In *Proc. of 24th Annual International Symposium on Computer Architecture*, July 1997.

[5] B. Rychlik, J. Faistl, B. Krug, and J. P. Shen. "Efficacy and Performance Impact of Value Prediction". In *Proc. of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, October 1998.

[6] A. Klauser, T. M. Austin, D. Grunwald, B. Calder. " Dynamic Hammock Predication for Non-predicated Instruction Set Architectures". In *Proc. of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, October 1998.

[7] J. Stark, P.B.Racunas, Y.N. Patt. "Reducing the Performance Impact of ICache Misses by Writing Instructions into the Reservation Stations Out-of-Order". In *Proc. of the 30th International Symposium on Microarchitecture,* November 1997.

[8] D. Burger, T. M. Austin, "The SimpleScalar Tool Set, Version 2.0", University of Wisconsin-Madison Computer Sciences Department Technical Report #1342, June 1997.

[9] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*, 2nd ed. Morgan Kaufman, San Francisco, 1996.