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

Technical Report – CS-2001-14

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

July 10, 2001

Abstract

This report describes work to minimize re-execution of control independent instructions in case of branch mispredictions. 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.

1. Introduction

A *branch hammock* [1] is a code fragment corresponding to an 'if' language construct. It is called as a *double-sided branch hammock* (fig 1a) when it corresponds to an 'if-then-else' construct and as a *single-sided branch hammock* (fig 1b) when it corresponds to an 'if- then' construct alone (i.e., without the 'else' part). When the 'then' and the 'else' contexts constitute only of one basic block each, the *branch hammock* is called *'simple'*. Otherwise, it is called a *'nested branch hammock'*.

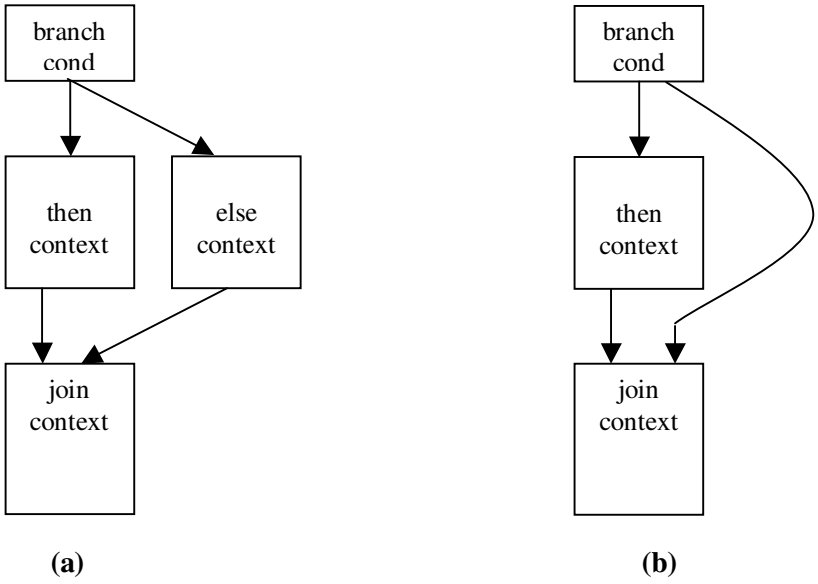

Fig. 1 – single and double-sided branch hammocks

In typical speculative processors, when there is a control misspeculation for a conditional branch corresponding to the *branch hammocks,* all the instructions fetched after the branch are squashed. However, the 'join' context is executed irrespective of whether the branch was taken or not. At the time of discovering the misspeculation, if the fetch engine of the processor had gone past the join context, it would have fetched some potentially useful instructions. Those instructions need not have been squashed and refetched. If those potentially useful instructions can be identified by some co-operative work between the compiler and the hardware, the misspeculation penalty can be reduced. Their squashing can be avoided and/or their refetching can be eliminated. This project attempts to implement such a co-operative mechanism for reducing misspeculation penalty in the particular case of *single-sided, simple branch hammocks.*

2. Related Work

Rotenberg *et al.* [2] have analyzed control independence in superscalar processors. Instructions that are executed irrespective of the direction of a branch are called control independent instructions. Their paper analyzes the bounds of potential performance improvement due to the exploitation of such control independence. It also assesses the complexity of various possible implementations. Sodani *et al*. [3] have done a detailed study on dynamic instruction reuse. The potential *branch hammock* reuse described above, called squash reuse, is a subset of the dynamic instructions reused in a program's execution. 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*. [4], 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. [5] 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.

3. Overview of the Project

This project combines the ideas of 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. If instructions can be found in the 'join'context that are data independent of the instructions in the 'then' context, their execution is not erroneous and hence, they need not be squashed. Such instructions are both control independent of the branch and data independent of the 'then' context. For simplicity sake, let us call these instructions as 'independent instructions'. Also, though the instructions in the 'join' context that are data dependent on the 'then' context have been executed with erroneous values, they need not be re-fetched if they are re-issued with the proper values. These instructions are control independent of the branch but are data dependent on the 'then' context. Again, for simplicity purposes, let us call them 'dependent instructions'. With the above observation, this project implements the 'selective squashing' (i.e., the 'join' context need not be squashed) and the 'selective re-issue' of instructions (i.e., the dependent instructions should be re-issued) with respect to the above *branch hammock* scenario.

If the instructions in the 'join' context that are independent of the 'then' context, are grouped together by the compiler and placed above the other dependent instructions in the 'join' context, and if the compiler can annotate the conditional branch with the number of independent instructions it could find, then the implementation of the selective re-issue and squashing can be done efficiently. The squashing can be implemented with just by switching the commit stage pointers. The re-issue can be done by re-renaming the dependent instructions (by retaining the dependent instructions in the instruction buffer). Implementing this way, the cost of the hardware is minimized and the control independence is also exploited. This project implements the above mentioned ideas and evaluates the implementation.

4. Implementation

The implementation can be categorized mainly into two parts viz. the scheduler and the processor implementation using SimpleScalar [6] simulator tool set. Each is explained in detail below

4.1 The Scheduler

The scheduler identifies the *single-sided simple branch hammocks* in a program. It also finds in the 'join' context, as many independent instructions as possible and groups them together at the beginning of the 'join' context. The scheduler has been written in C and takes Simplescalar assembly file as its input. It identifies the *single-sided simple branch hammocks* by walking through the assembly file in search of conditional forward branches devoid of any nesting.

Once it finds a basic block as a possible candidate, it analyses the dependences of each instruction in the 'join' context to determine which of them is independent of the 'then' context. Register dependences are easily identified based on the instruction format and the addressing mode. However, it is impossible to resolve memory dependences statically. So, the scheduler assumes that every load depends on all preceding stores in the basic block. As a result of this chain dependence analysis, the instructions in the 'join' context that are not truly dependent on the 'then' context are identified.

Now, the scheduler performs the 'code motion' of these independent instructions to the beginning of the 'join' context. In doing so, all correctness requirements are taken care of. No output or anti dependences are violated. Again, since it is not possible to resolve memory dependences statically, stores don't move past earlier loads or stores. Moreover, instructions also don't move beyond a label in the *hammock* because there could be entry to this label from some other part of the program.

After the grouping of the independent instructions, the scheduler now annotates the branch of this *hammock* with the size of the block, the number of independent instructions and the position of the join context. Instruction annotation is a facility provided by the simulator to allow a compile time tool to communicate data to the simulator hardware. Each instruction has a 16-bit annotation field that can be set by assembly directives and can be accessed in the hardware on demand. The scheduler modifies the assembly file to include the appropriate annotation directives at the position of the branch instruction. This output assembly file is compiled into a Simplescalar binary using an assembler with proper support for annotations [8]

4.2 The Hardware

The processor-level implementation of selective squashing and re-issue has been done using the SimpleScalar simulator tool set. The out-of-order (OOO) superscalar processor simulator of the tool set - *'sim-outorder'*, has been modified to incorporate selective squash and re-issue. *'sim-outorder'* unifies re-order buffer, reservation stations and physical registers into a common structure called as Register Update Unit (RUU).

Instructions are stored and retired in program order from this RUU. The following figure (fig. 2) shows a '*simple branch hammock*' in the RUU. Assume that the branch of the *hammock* was predicted not taken but actually taken. Also assume that this figure shows the time when the misprediction is detected.

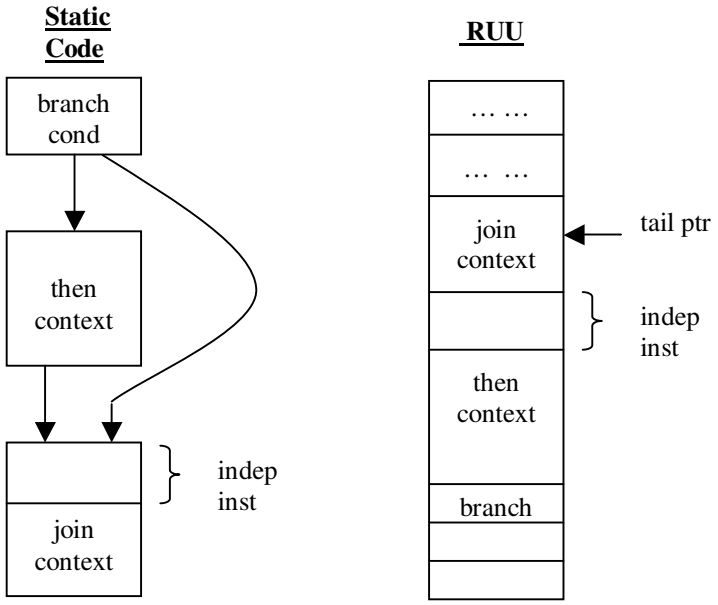

Fig. 2 – state of the RUU corresponding to a hammock

In the scenario of above mentioned 'grouping' support from the compiler, selective squashing can be easily implemented in the hardware. In case of normal squashing, head and tail pointers are just brought back together indicating that the RUU is emptied. For selective squashing, it is enough to bring the head pointer to the beginning of the 'join' context. This squashes only the then context. There is also a subtle issue in this way of implementing selective squashing – since the RUU is not emptied, and since some instructions remain in it, the register maps can't just be restored as they were at the branch position. However, in real processors, obtaining the new register map is just an easy masking operation of the 'branch-position' register map.

In SimpleScalar, register maps are implemented as copy-on-write buffers for easy squashing (create_vector, spec_create_vector). Hence, the implementation of 'register map recovery' in this project involves a long process of walking through the buffers and restoring them. However, in a real processor, this step won't be present and hence the selective squashing can be implemented with minimal hardware complexity.

In the above figure, at the time of the detection of misprediction, if the tail pointer is amidst the dependent instructions of the join context, then as before, only the 'then' context is squashed. However, the dependent instructions have not been executed with the right values. Moreover, their dependence information might also be wrong. So, they should enter the rename stage once again. In order to achieve this, the implementation holds them in the instruction queue that decouples the processor front end from the back

end. As they are inserted into the instruction queue, the dependent instructions are marked with a tag denoting the hammock branch. They are held in the instruction queue until the branch corresponding to the hammock resolves. On a misprediction, only all the untagged instructions in the instruction queue are squashed. The tagged ones are not squashed. This ensures that the dependent instructions get to re-issue to the functional units with the proper values. Finally, on a successful branch resolution, all associated instructions are purged from the fetch queue.

Both in selective squash and re-issue, another subtlety exists when the branch is predicted taken but is actually not taken. The processor should remember the PC of the start of the 'join' context and that of the tail of the RUU. After the misprediction is detected, the fetch is redirected to the fall through path of the branch and continues till the beginning of the 'join' context. After that, it is resteered to the point after the saved tail pointer. In a typical superscalar processor, multiple instruction fetch resteers means possibly many empty fetch slots too. Also, the 'remembering of PC' is actually a feedback datapath from the writeback/commit stage to the fetch stage and some extra registers. So considering the implementation complexity, this case could be omitted in a real processor. However, in this implementation, this is considered.

5. Performance Evaluation and Results

Performance evaluation of the techniques detailed above has been done by running the SpecInt95 benchmarks on the modified Simplescalar simulator. The benchmarks were first compiled into assembly and were scheduled by the instruction scheduler. The scheduled assembly files were compiled and linked into the binaries. These were the binaries run on the simulator. Table 1 shows the static scheduling data for the benchmarks.

	compress	gcc	go	ijpeg	li	m88ksim	perl	vortex
Scheduled hammocks	5	984	341	68	13	85	40	13
Max. hammock size	4	8	7	9	4	7	6	9
Max. 'then' context size	3	6	6	3	2	4	3	3
Max. no. of indep. Inst	2	6	4	6	1	3	4	1
Avg. hammock size	4	4	3.3	4	3.4	3.9	4	5.2
Avg. 'then' context size	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 – data from static scheduling

From the table, one might observe that typical schedulable hammock sizes are about 3-4 instructions. Also, the typical number of independent instructions found by the scheduler is typically 1 or 2 per hammock. In the perspective of potential performance improvement, this is on the lower side. However, one can expect these numbers. The task of the scheduler is to find control and data independent instructions. This task is the same as that of a compiler that tries to fill branch delay slots of a processor. Typical branch delay slot occupancy is 1-2 instructions. Branch delay slots are also filled with the

same kind of control and data independent instructions. So, the above numbers are realistic in spite of being on the lower side. Moreover, we can observe that the typical "then" context size is 2-3 instructions. This size is very conducive for predication. These instructions could very well be predicated too.

Actual measure of the performance is obtained by comparing the IPC of the benchmarks running on the baseline and modified SimpleScalar simulator. The simulation setup for this evaluation closely resembles the one in [9]. The benchmarks were each run for 100 million instructions after warm up periods mentioned in [9]. The inputs were standard reference inputs. The compiler optimizations used were '–O3' and ' –funroll-loops'. Since the simulations involve modeling of branch mis-speculation issues, a realistic good branch predictor configuration was chosen. A hybrid predictor with 4K entry selector that selects between a GAg predictor and a PAg predictor was chosen. The GAg predictor was with a 4K PHT and a 12-bit history register. The PAg had 1K BHT, 1K PHT and 10 bits of history. Also, a 32-entry return address stack and a 2K entry 2-way BTB were used in the predictor. Other configuration options were simulator defaults. From the simulations, it was found that the performance improvement for all the benchmarks other than *'gcc'* and *'go'* were very negligible. Table 2 summarizes the results of the simulations for *'gcc'* and *'go'*.

It can be seen that the performance improvement obtained from the benchmarks is very small and is of the order of one-hundredth of a percentage. Another interesting observation that can be made is that in static figures, *'gcc'* seemed to be a better candidate for performance improvement because the scheduler was able to find many hammocks and independent instructions. However, *'go'*s performance improvement is better than *'gcc'* s mainly because of its lower prediction accuracy. Lower prediction accuracy means more squashes and hence more savings due to selective squashing and re-issue.

	gcc	**go**
Branch prediction accuracy (address + direction) (in %)	89.6	82.3
No. of mispredicted branches	1.7M	2.1M
Number of hammocks seen on misprediction	6K	26 K
No. of useful hammocks	1376	3655
No. of re-used instructions	1383	3672
No. of re-issued instructions	236	51
IPC improvement (in %)	0.02	0.04

Table 2 – dynamic data from simulations

6. Conclusion

This project implements a scheme for selective squashing and re-issue of instructions for the specific case of '*single sided simple branch hammocks'* and evaluates the performance gains possible. From the results however, it can be seen that the performance gains obtained due to the technique are not substantial. In the scheduler, the unavailability of memory dependence information and high-level compiler information form the major causes. Since memory dependence can't be resolved at compile time, possibly independent instructions have to be considered dependent. Also, since the scheduler operates at the assembly language level, higher-level compiler information is not available to it. For e.g., instructions are not moved past empty labels. This restriction could be relaxed if higher-level information is available for the scheduler. These suggest that dynamic techniques like [5] could be more useful for exploiting control independence in hammocks.

As for the processor implementation, in spite of the simplicity of hardware implementation, instruction fetch resteers and useless hammocks may be some reasons for the low performance. When the branch of a hammock is predicted taken and is actually not taken, there is high amount of fetch resteer. This might lead to many empty slots in the fetch bandwidth. Also, even when an annotated branch is present in the RUU at the time of misprediction recovery, if the fetch engine had gone past the basic block, the instructions are fully squashed. This is due to the lack of information stored inside the processor. If state information can be stored for multiple branches, then this could be exploited. However, as a trade-off, the complexity of the processor now increases.

As an extension to this work, one might investigate the effectiveness of predication in lieu of such re-issue and re-use. The sizes of the basic blocks found by the scheduler strongly suggest this. The contrast between the natures of control independence exploited by these techniques (predication, re-use/re-issue) is an interesting aspect for investigation.

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 onProgramming Languages and Systems, 9(3): 319–349, July 1987.

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

[3] A. Sodani and G. S. Sohi. “Dynamic Instruction Reuse”. In Proc. of 24th Annual International Symposium on Computer Architecture, pages 194–205, July 1997.

[4] B. Rychlik, J. Faistl, B. Krug, and J. P. Shen. “Efficacy and Performance Impact of Value Prediction”, In Proc. of the international conference on Parallel Architectures and Compilation Techniques, Paris, October 1998.

[5] A. Klauser, T. M. Austin, D. Grunwald, B. Calder. “ Dynamic Hammock Predication for Non-predicated Instruction Set Architectures”. In Proc. of the international conference on Parallel Architectures and Compilation Techniques, Paris, October 1998.

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

[7] M. E. Benitez, J.W. Davidson, “Target-specific Global Code Improvement: Principles and Applications”, Department of Computer Science, University of Virginia, Technical Report, CS-94-42, November 4, 1994.

[8] M. Plakal, “SimpleScalar Fixes and Improvements” - “http:// www.cs.wisc.edu/ ~plakal/simplescalar/”

[9] K. Skadron, P.S. Ahuja, M. Martonosi, and D.W. Clark. "Branch Prediction, Instruction-Window Size, and Cache Size: Performance Tradeoffs and Simulation Techniques." IEEE Transactions on Computers, 48(11): 1260-81, Nov. 1999.