CS 752 Project Report

Dynamic Instruction Reuse – SPECfp92

December 4, 1997

Amit Marathe

Shilpa Sawale

Zak Smith

Abstract

This report further studies the phenomenon of dynamic instruction reuse as proposed by Sodani and Sohi [1]. We analyze the behavior of SPECfp92 benchmarks to complement the SPECint92 results in [1]. Our results show that, like integer programs, floating-point programs can gain significant speedup from a reuse buffer, but reuse of floating-point instructions does not contribute a significant amount to that speedup.

Introduction

Dynamic instruction reuse has recently been proposed by Sodani and Sohi [1] to exploit the phenomenon of the dynamic instruction stream containing multiple instructions with the same operands (and therefore producing the same values). They observed that such instructions do not have to be executed repeatedly - if the results of the first execution are stored in a reuse buffer (RB) then all later instructions can simply read the result from the RB without having to go through all the phases of execution. This reduces the latency and the contention for processor resources and allows dependent instructions to proceed earlier. Of course, speeding up a bottleneck resource often exposes another bottleneck and therefore does not lead to a matching speedup in the system. Thus the speedup obtained from dynamic instruction reuse is substantially lower than the percentage of instructions reused.

Oberman and Flynn [2] describe a special purpose RB, which only caches division and reciprocal instructions. Their main motivation was that although division and reciprocal instructions are relatively rare compared to other floating-point operations, their latency is high because hardware designers tend to focus attention on the more common instructions such as add and multiply. The high latency of divide and reciprocate can contribute to interlocks due to data and structural hazards which make it worthwhile to store the previous results in the RB. The general reuse strategy we simulated alleviates this problem for all floating-point instructions, which tend to have longer latency than most integer instructions. The general reuse strategy yields a large performance gain over the division and reciprocal only design because most reuse comes from program control structures that use integer instructions as our data confirmed and as was found in [1].

Three schemes were proposed in [1] for implementing the RB. In all schemes the RB operates as a cache for instruction results and is indexed by the low-order bits of the program counter (PC). Each RB slot can store information for as many instructions as the associativity of the RB. The schemes differ in the type and amount of additional information stored in each RB entry that enables identification of instruction reuse.

The first scheme, Sv, stores the operand values and the tag (the high-order bits of PC) along with the result in each RB entry. Whenever an instruction is decoded the PC is used to identify a RB slot and a search is carried out for the operand values of the instruction in all RB entries making up this slot. If a match is found the result can be used directly, bypassing the execution stage. This scheme differs from the others in that the RB entries for non-load/store operations are always valid and invalidations are not required to ensure correct behavior.

The scheme Sn stores the source register identifiers instead of the operand values in each RB entry. While this reduces the number of bits per RB entry it requires an invalidation mechanism to maintain integrity - whenever an instruction writes into a register, all RB entries having the same register as one of the source operands are marked invalid and do not contribute to further reuse.

Scheme Sn+d is an extension of scheme Sn - it also stores the dependence relationships between various instructions. This extra information allows multiple dependent instructions to be issued in a single cycle (provided they form a dependency chain and the first instruction in the chain can be reused) and thus allow the processor to exceed its data-flow limit.

Originally, we proposed a study to determine which program structures generally contribute to significant instruction reuse, but we decided that would have been too much for one semester. As a result, we modified our proposal to a study of floating-point reuse. To this end, we used the simulator developed by Avinash Sodani for his research, which is an extension of the SimpleScalar tool set [3].

Our Work

Of all the implementation schemes for dynamic instruction reuse, we decided to focus on the Sv scheme. The reason for this was twofold: first of all, because it is a very simple scheme it can be simulated very easily in software with very little time overhead. This was a significant factor influencing our decision given that we wanted to analyze the reuse behavior of many benchmarks in a limited duration. Sv is the best performing scheme for moderate to large RB size, and, more importantly, it would be straightforward to implement in hardware, a direct consequence of its simplicity. Thus we believe that not much was lost by our decision to run our simulations only with the Sv scheme.

We chose to use the SPECfp92 benchmarks to complement the SPECint92 benchmarks in [1]. We decided to perform our simulations with RB sizes of 256, 1024 and 4096 entries. On an out-of-order superscalar processor with a small RB, the number of instructions issued before a FP operation completes would be greater than the number of entries in the RB. FP instructions would thus tend to be flushed out of the RB before they had a chance to be reused. Therefore a 32 entry RB, while it has been studied for integer benchmarks [1], would be almost ineffective for FP programs. Also because simulation time increases with RB size the results have been restricted to 3 RB sizes which we felt were representative of the whole range from a 1-entry RB to an infinite RB. Moreover we decided to work with a fully associative RB, as it did not differ significantly from a 4-way RB in terms of reuse behavior.

The capacity of the register update unit and the load-store queue has been maintained at 32. We have just chosen reasonable values for these parameters -

small changes to these values are not going to change the results in a significant way. A 2-bit branch predictor has been used in all the runs. It turns out that this predictor is quite accurate for FP programs (consistently predicting about 95% of the branches correctly) and thereby drastically reducing squash reuse. Therefore, while the reuse behavior could be improved by working with a weaker predictor, that would not be a good idea from a performance viewpoint. The other parameters take on their default values.

Problems

Although our goal was to run every benchmark from SPECfp92, we were only able to obtain valid results for 10 of the programs. The main obstacle for those that did not complete was that after a large number of instructions had executed, but before the first checkpoint, the simulator would crash. Some of the benchmarks which we did obtain results did not run to completion, but did make it past one or more of the 100-million-instruction checkpoints, giving us reasonably accurate data. The results we obtained are given in the following graphs – values for runs that crashed are shown as zero.

Results

Discussion - tomcatv

The reuse in this program for the c.le.d instruction is probably due to common convergence behavior. The code for tomcatv generates a mesh by iteration, and must test the convergence of parameters to know when to terminate. In particular, one test for loop termination depends on two conditions that could be transformed into less-than-equal when compiled. It is likely that only one of these will change often, and the other will stay constant, allowing it to be reused.

Discussion – mdljdp2

This program has many conditions that depend on the absolute value of arguments. There are many such parameters used in the iterations, and not all of them change between iterations.

Discussion – alvinn

Alvinn is a program that trains a neural network using back propagation. It is a single precision floating-point benchmark, so it is not surprising that double precision FP instructions do not contribute much to the total reuse. Of all the FP instructions, mul_s is reused most often.

The branch prediction rate is very high (greater than 99% for all runs). Even then the speedup obtained is comparable to that for other benchmarks with less accurate branch prediction. From this we can conclude that general reuse in this benchmark is compensating for the lack of squash reuse.

Discussion – fpppp

Fpppp is a double precision floating-point benchmark from quantum chemistry. It does very little IO and is difficult to vectorize because of the presence of very large basic blocks. The 4096-entry RB run of this benchmark did not complete; it crashed after 400 million cycles.

Discussion – hydro2d

Hydro is a double precision floating-point benchmark from astrophysics, which solves the Napier-Stokes equations. As with most of the other programs the 4096-entry RB run did not complete. However the speedup obtained due to this RB size is spectacular - going above 40% for one run. That such a high speedup is achieved even with a 93% branch prediction rate suggests that most of the reused instructions are integer ones associated with the bookkeeping code.

Discussion – su2cor

Su2cor is a benchmark that calculates the mass of elementary particles. The reuse was not as focused on a specific one or two instruction types; it was spread evenly over 6 instruction types. This suggests that the algorithm does some redundant calculation.

Discussion – swm

Swm is a single-precision floating-point benchmark that solves the system of shallow water equations on a 256x256 grid. All runs of swm completed without any errors. The highest speedup obtained was 12.79%.

Discussion – ear

This is an inner ear model that filter and detects various sounds and generates speech signals. There was no significant floating-point instruction reuse.

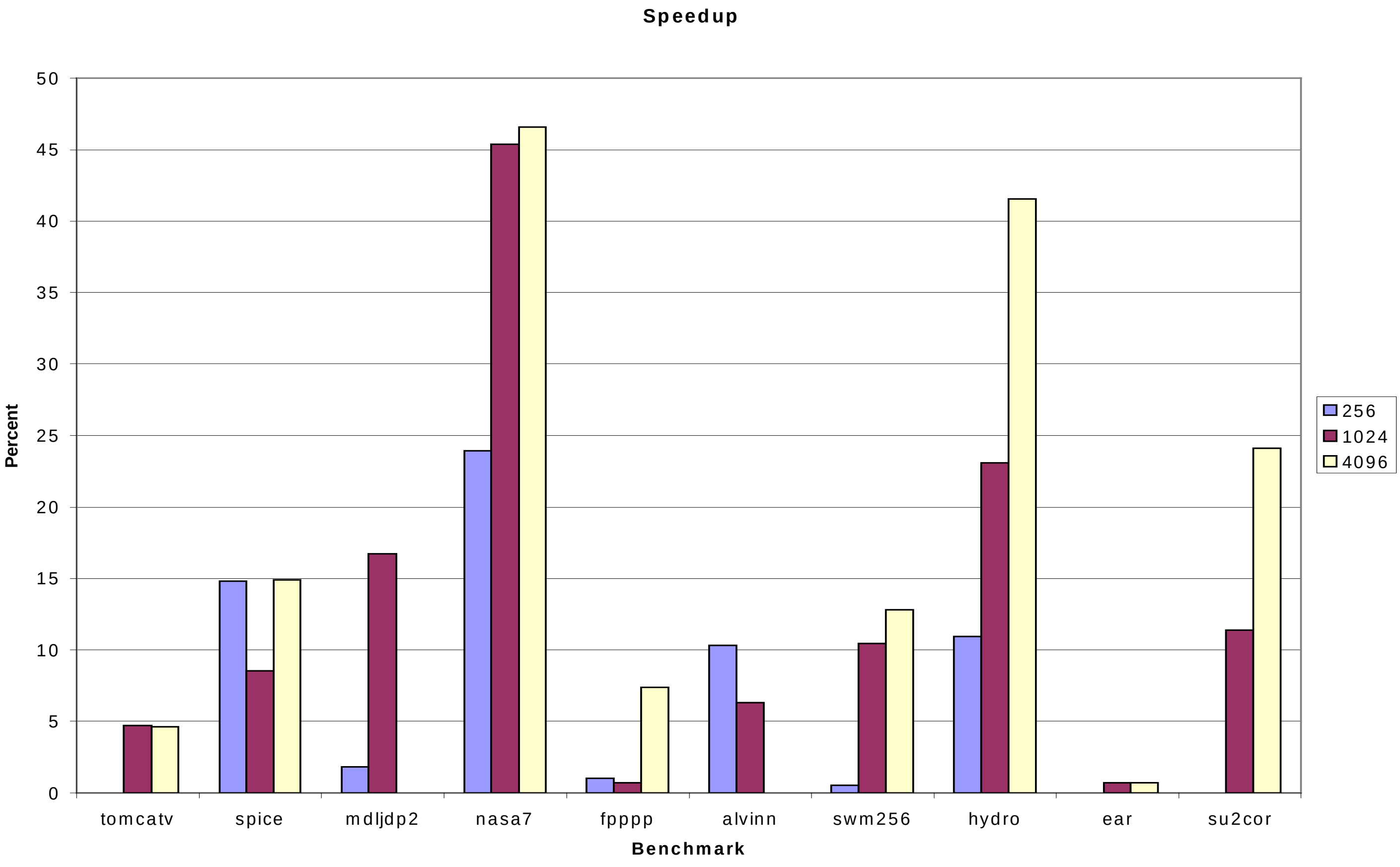
Discussion – spice

Spice simulates circuits built of primitive electrical components. Spice had practically no floating-point instruction reuse, and but gained a 40% speedup for the 1K RB from general reuse.

Discussion – nasa7

Nasa7 is a collection of seven floating-point kernels. Nasa7 enjoyed the highest speedup of all the benchmarks, in spite of zero floating-point instruction reuse. Nasa7 also had the highest percentage of instruction reuse.

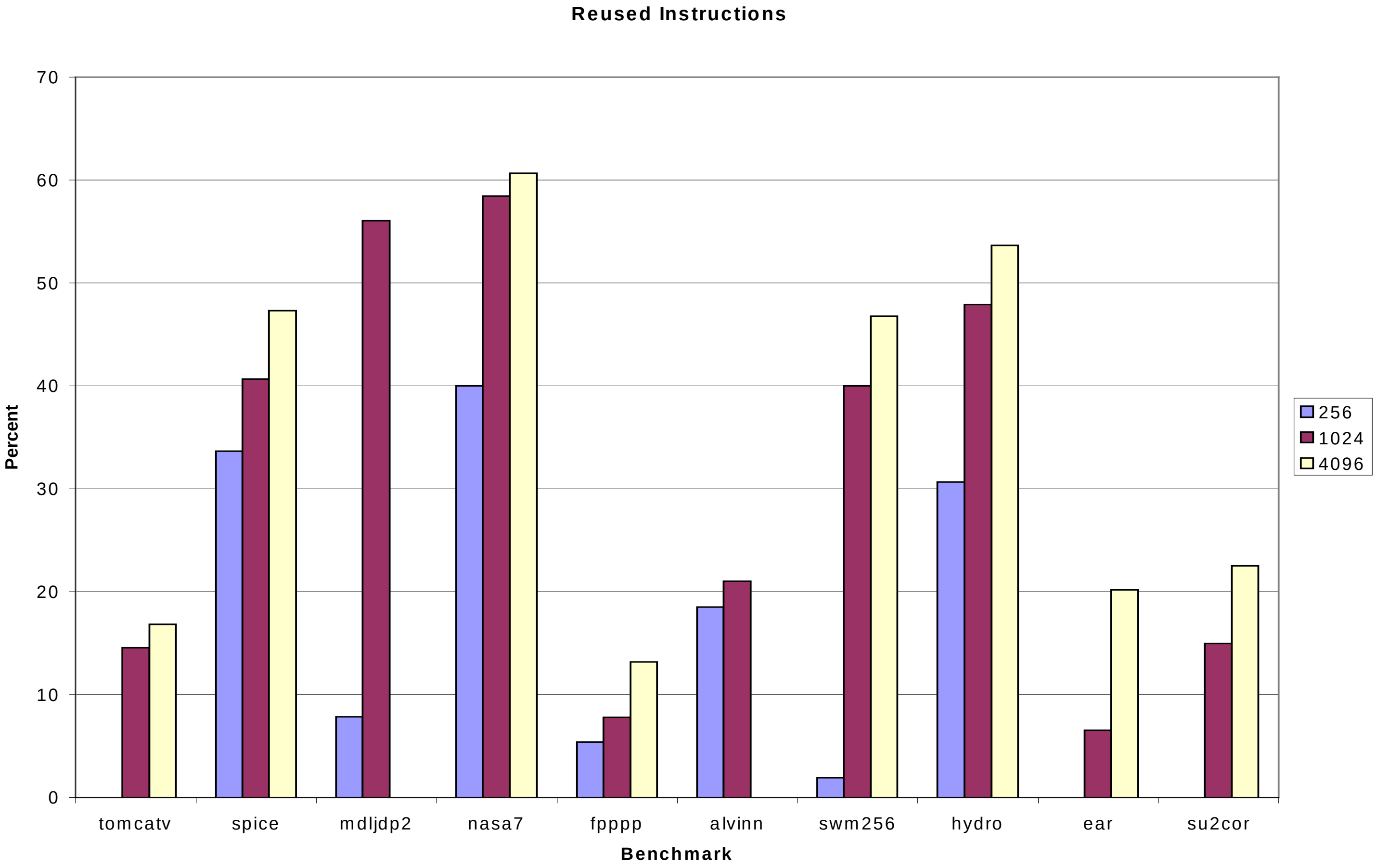
Speedup



The speedups are comparable to those found for integer programs in [1]. One would expect speedup to increase with RB size and that is true for most of the above results.

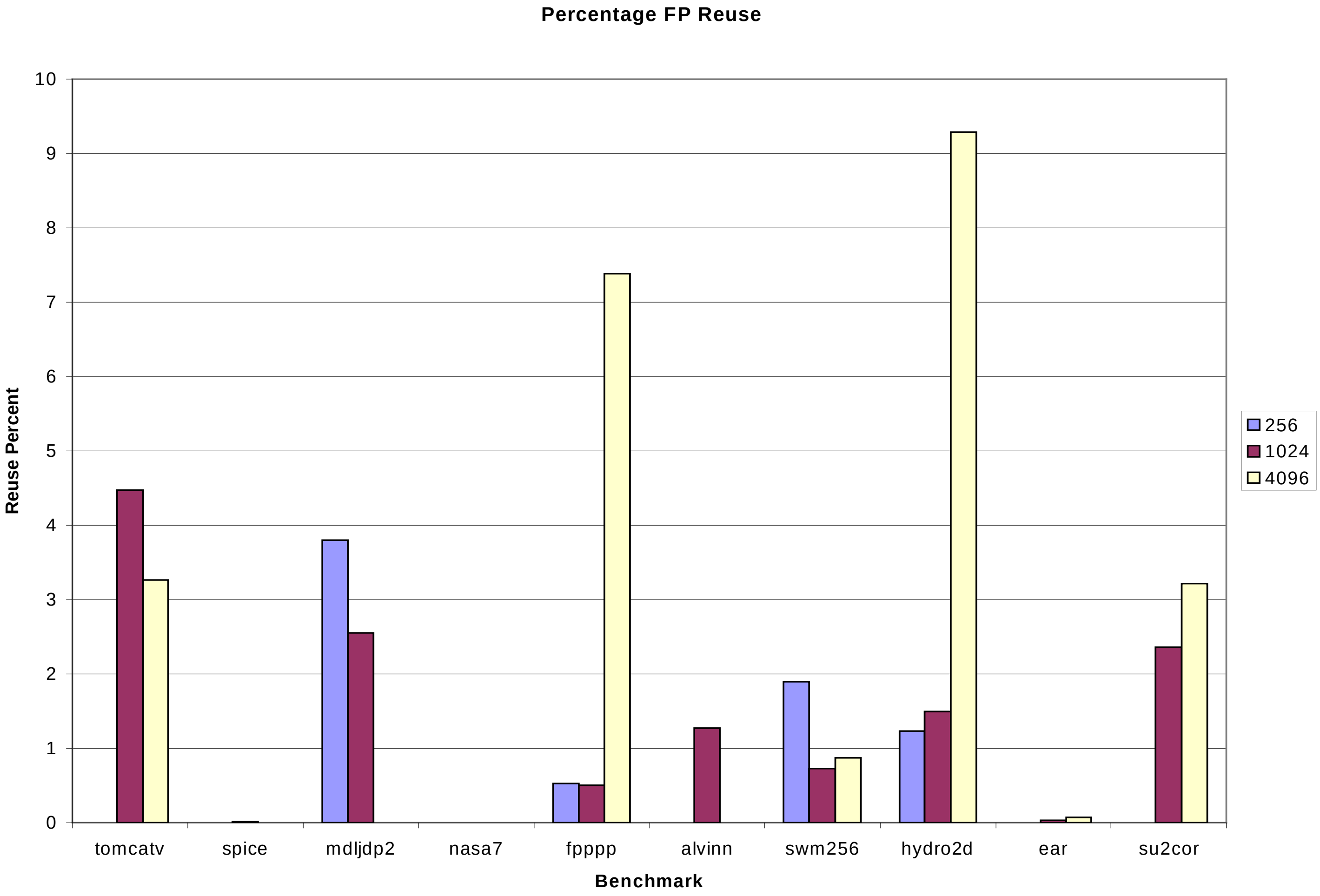
The few anomalies are probably due to the fact that not all the simulations ran to completion. Many benchmarks have different phases of execution that would have different reuse behavior, due to which reuse behavior of incomplete runs is not representative of the reuse behavior of the whole program.

Instruction Reuse



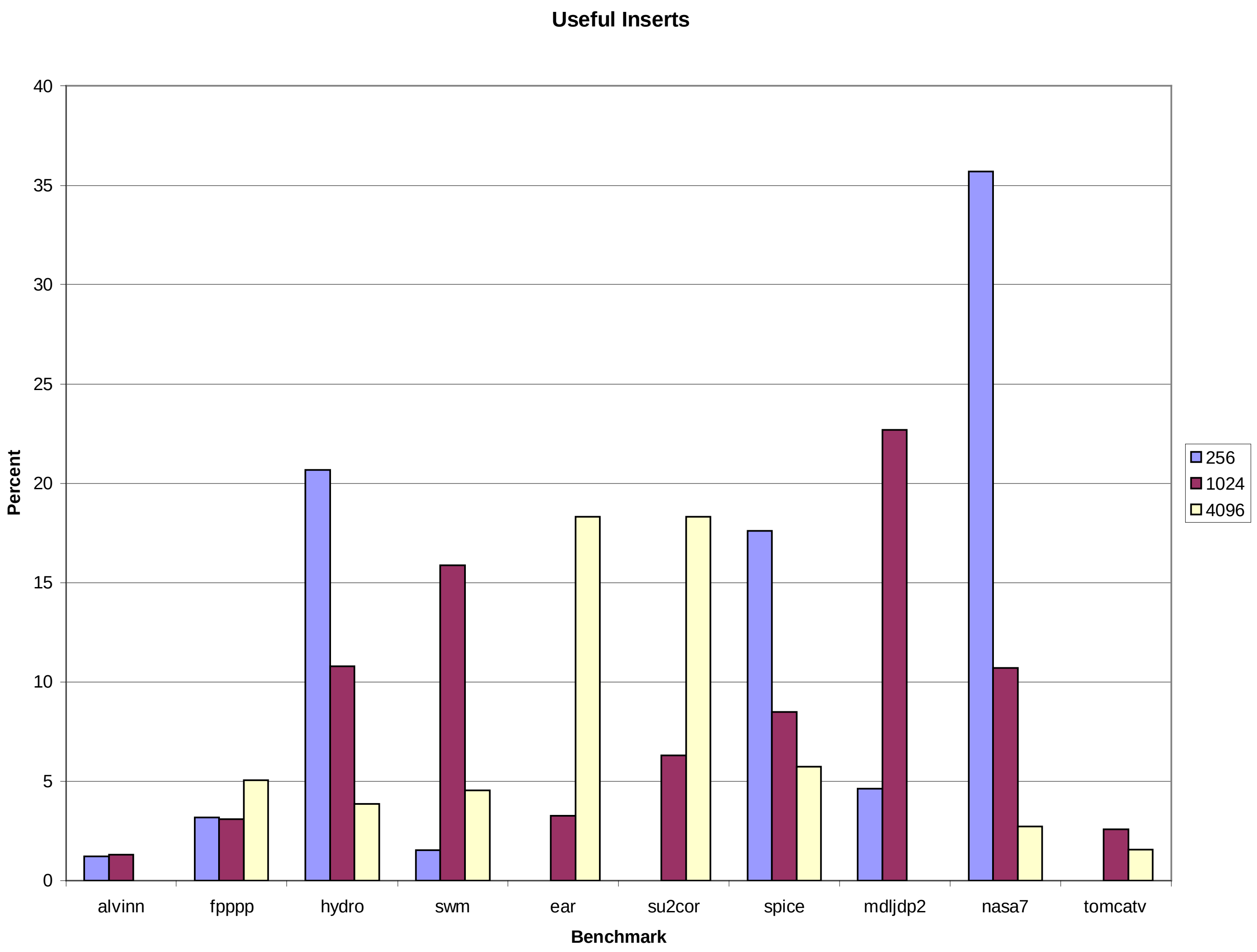
Reused instruction count is also comparable to the results published in [1]. This is not surprising since most of the reuse comes from integer instructions. The improvement in reuse percentage between the 1K and 4K cases is not as marked as that between the 256 and 1K cases. This is also true for speedup. A comparison of the above graphs also shows that the percentage reuse is much more than the percentage speedup.

Floating-Point Instruction Reuse



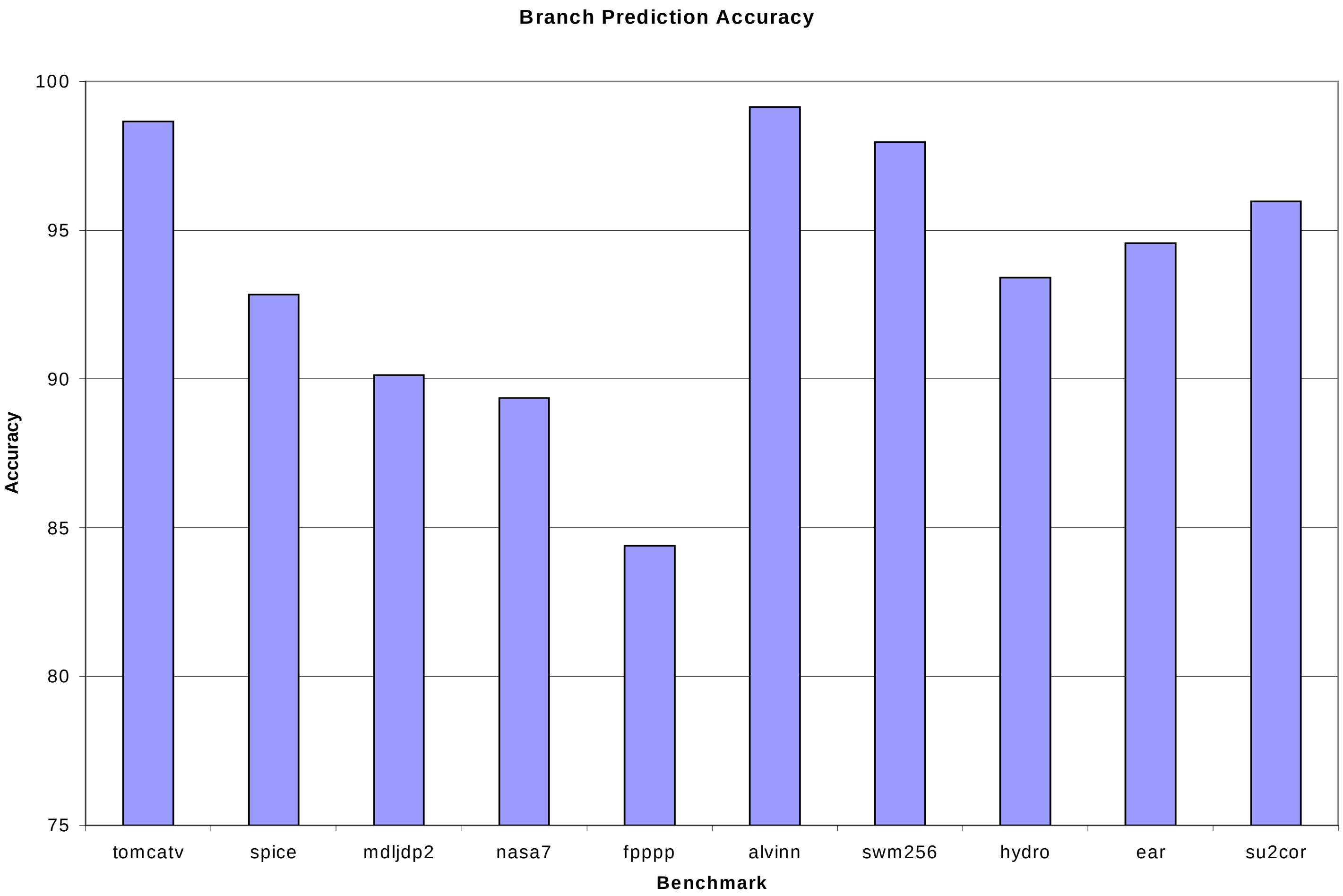
The above graph shows how much of the total instruction reuse in the program came from floating-point instructions. Since the latency of floating-point instructions is higher than the latency of integer instructions, it is important that the RB is large enough so a long-latency, floating-point instruction can stay in the RB until it can be reused. This is probably why the 4K RB cases of fpppp and hydro2d have much more reuse than their 1K runs.

Useful Inserts



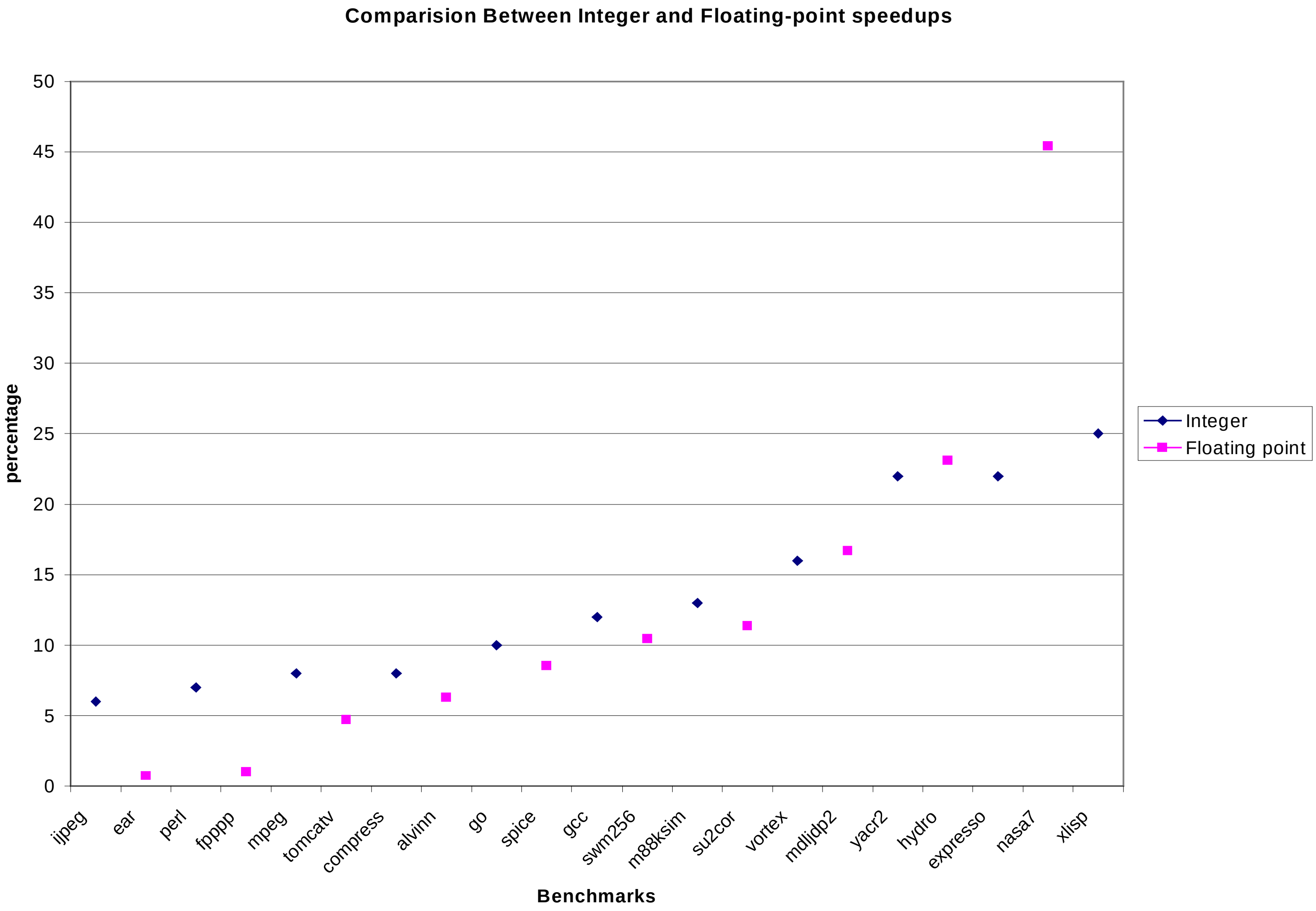
In some cases, as the RB size is increases, the fraction of useful RB insertions decreases. This suggests that the same performance could be attained using a smaller RB size, if the insertion policy could distinguish between instructions that are likely to be reused from those which are not likely to be reused.

Branch Prediction Accuracy



One would expect that RB size has no effect on branch prediction rates. This is nearly the case. We found that the prediction accuracy changed between RB sizes not more than a percent. These changes can be attributed to timing changes due to the shorted instructions, which could affect the history of the predictors. We show the no RB case in this graph, because of the above reason, and we mean to only show they are high. Since branch prediction rates are high, reuse of squashed execution is reduced.

Summary of Floating-Point and Integer Speedups



This graph shows the speedup obtained for integer and floating-point benchmarks using a 1K RB. The data for the integer benchmarks is taken from [1]. Except for nasa7, the speedups obtained are similar. The floating-point benchmarks generally have less speedup, however. This is due to less squash reuse in floating-point programs from higher branch-prediction accuracy, and floating-point instructions that are unlikely to be reused displacing re-usable bookkeeping instructions in the RB.

Conclusions

In general, reuse behavior and trends found in [1] also apply to the SPECfp92 programs, for the Sv scheme. However, nearly all of the performance gains of the floating-point benchmarks came from reuse of integer instructions. Even though SPECfp92 benchmarks are very compute-intensive, most branches are predicted correctly, thus squash reuse does not occur as often as in the integer programs.

No benchmark had more than 5% FP instruction reuse, which means FP reuse contributes very little to overall performance gain. Because FP instructions are rarely reused, it may be worthwhile to restrict insertion into the RB to only instructions that are commonly reused. This would allow more performance gain in a smaller area.

Performance gain measured by CPI increases with RB size until it plateaus. The difference in performance gain between the 4K and 1K-entry RB was not very significant. Moving to an 8K-entry RB would not improve performance for this type of program.

Future Work

For the sake of completeness, it would be good to run the SPECfp95 benchmarks to verify our conclusions against those programs.

In Avinash's paper, he found that certain operations are reused much more frequently than other operations. Our overwhelming result was that floating--point operations are rarely reused! Since we know reusing floating--point operations gives no performance gain, we can prevent insertion into the reuse buffer so that more slots are available for instructions which are more frequently reused.

A small percentage of instruction types contribute to the most reuse. To generalize the idea in the previous paragraph, a study could be performed to determine, on a range of workloads, which instructions yield little reuse, and what gains can be achieved by only inserting commonly-reused operations into the reuse buffer.

To better characterize the reuse buffer, these experiments could be run using an infinite RB. This would show which misses are compulsory and which are due to "conflict" misses.

For purposes of being complete, the experiments we performed should be redone using these last two methods to validate our claim that this behavior is common to both the integer and floating--point benchmarks.

There is an area tradeoff between the Sv and Sn schemes: because the Sv scheme saves the values of each operand instead of just the register name, it will take much more area than the Sn scheme. For example, for a machine with only 2-operand instructions, 32 32-bit registers, and a 1024 entry RB, the Sv scheme will use 32+32+22=86 bits per identifier. The Sn scheme will only take 5+5+22=32 bits per identifier. Thus, for the same number of entries, the Sv RB will take approximately 2.6 times more area.

Because the Sn scheme behaves differently than the Sv scheme, a study could be done to find the optimal RB configuration, based on the performance, area, and speed tradeoffs of these methods.

Acknowledgements

We would like to thank Avinash Sodani for helpful discussions related to the project and for assistance figuring out how to run the benchmarks.

Also, we thank Prof. Sohi for guiding our study and reminding us about the scope of the project.

References

[1] Avinash Sodani and Gurindar S. Sohi. Dynamic Instruction Reuse. Proceedings of the 24th International Symposium on Computer Architecture, June 1997.

[2] Stuart F. Oberman and Michael J. Flynn. On Division and Reciprocal Caches. Technical Report CSL-TR-95-666, Computer Systems Laboratory, Stanford University, April 1995.

[3] D. Burger, T.M. Austin, and S. Bennett. Evaluating Future Microprocessors: The SimpleScalar Tool Set. Technical Report CS-TR-961308, University of Wisconsin - Madison. July 1996.
(URL: http://www.cs.wisc.edu/~mscalar/simplescalar.html)