Putting the Fill Unit to Work: Dynamic Optimizations for Trace Cache Microprocessors

Daniel Holmes Friendly Sanjay Jeram Patel Yale N. Patt
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan 48109-2122
{ites, sanjayp, patt}@eecs.umich.edu

Abstract

The fill unit is the structure which collects blocks of instructions and combines them into multi-block segments for storage in a trace cache. In this paper, we expand the role of the fill unit to include four dynamic optimizations: (1) Register move instructions are explicitly marked, enabling them to be executed within the decode logic. (2) Immediate values of dependent instructions are combined, if possible, which removes a step in the dependency chain. (3) Dependent pairs of shift and add instructions are combined into scaled add instructions. (4) Instructions are arranged within the trace segment to minimize the impact of the latency through the operand bypass network. Together, these dynamic trace optimizations improve performance on the SPECint95 benchmarks by more than 17% and over all the benchmarks studied by slightly more than 18%.

1 Introduction

A microprocessor has three fundamental components: a means to supply instructions, a means to supply the data needed by these instructions, and a means to process these instructions. For high performance, the instructions and data must be effectively delivered at high bandwidth to a processing core capable of effectively consuming them.

The trace cache has been developed for high bandwidth instruction delivery. It has been demonstrated as an effective, low latency technique for delivering instructions to very wide issue machines [14, 4]. By placing logically contiguous instructions in physically contiguous storage, the trace cache is able to deliver multiple blocks of instructions in the same cycle without support from a compiler and without modifying the instruction set. Unlike other hardware techniques for delivering multiple blocks of instructions in a single cycle, the trace cache allows complexity to be moved out of the fetch-issue pipeline where additional latency impacts performance. The logic needed to prepare instructions for issue can be put in the trace cache fill pipeline. Our earlier work has shown that the latency of the fill pipeline has a negligible performance impact [4].

The techniques we present in this paper exploit the latency-tolerant nature of the fill pipeline by performing trace transformations within the major logic structure of the pipeline — the fill unit. Generally speaking, the transformations can perform a wide variety of tasks: dynamic retargeting of an ISA [9], pre-analysis of the dependencies within a trace [19], and dynamic predication of hard-to-predict short forward branches are some examples. Additionally the fill unit provides a strong framework for the dynamic tuning of code sequences. It is this class of transformations we will examine in this paper.

The fill unit is uniquely qualified to perform these optimizations for several reasons. Since it is not on the critical path, the fill unit can perform multi-cycle operations without adversely affecting performance. Because it combines multiple blocks of instructions from a single path of execution, it can easily perform cross-block optimizations. And since it is not architecturally visible, the fill unit can tailor its optimizations to the characteristics of the microarchitectural implementation without alterations to the ISA. The fill unit and its relation to the trace cache is shown in the trace cache data path in figure 1.

We present four types of dynamic code tuning techniques. The first is a technique that marks instructions which move a value from one register to another register as explicit move instructions. These can be completely handled by the register renaming logic. Many ISAs, such as the MIPS [13] and Alpha [16] architectures, do not support an explicit register-to-register move instruction so instructions which pass an input operand unchanged to the destination,

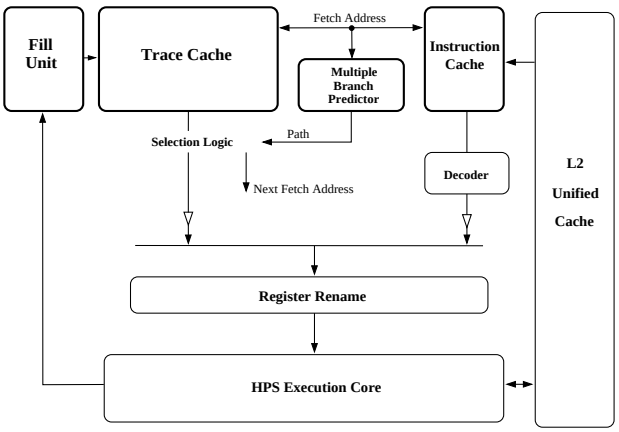

Figure 1. The trace cache datapath.

ADD Rx ← Ry + 0 for example, are used by compilers to perform the move. Instead of using execution resources to accomplish the move, we propose the simple technique of renaming the output register, *Rx* in this case, to have either the same physical register or the same operand tag as the input. This renaming completes the execution of the move instruction.

The second technique we present is called reassociation. Here, the fill unit combines the immediate values of instructions which are known to execute together, thereby removing a value dependency between the two. The transformation is illustrated in the following code sequence.

ADDI Rx ← Ry + 4

ADDI Rz ← Rx + 4

can become

ADDI Rx ← Ry + 4

ADDI Rz ← Ry + 8

This trace optimization is similar to one that a simple compiler performs, but within the fill unit it is applied across basic blocks. The dynamic nature of trace construction allows this to be applied across branches which are problematic for multi-block compilers. Often compilers of this sort only join blocks if the code can be predicated or if the joining branches are determined to be very strongly biased through code profiling. As the fill unit allows the creation of segments across function calls, reassociation can also be applied across procedure boundaries.

The third technique creates scaled add instructions. It is an application of dependence collapsing [15] using the fill unit as the dynamic mechanism to perform the collapsing. Many add instructions are directly dependent on shift instructions where the shift is a short distance immediate shift operation. Such shift-add combinations are commonly used when accessing arrays of data items. Here, we convert such an add into an operation where the shift and the add can be performed in one cycle.

The fourth technique is a simple heuristic to deal with the communication delays associated with bypassing operand values from one functional unit to another. As the dependencies between instructions within a trace cache line are stored explicitly, the order of the instructions no longer conveys dependency information. The fill unit has the freedom to place instructions in any order it wishes. It can base the ordering of instructions upon specifics of the execution microarchitecture and thereby reduce the impact of communication delays.

There are a number of advantages to performing optimizations in the fill unit as opposed to either in the decode logic or by the compiler. Placing time consuming logic in the issue path has a severe performance impact. Placing the logic in the fill unit, where the latency has a negligible performance impact, avoids this penalty. Furthermore, the fill unit has advantages over a compiler in certain respects. Fill unit optimizations improve performance of existing executables. Also, the fill unit may perform optimizations without the difficulties of increased register pressure or fix-up code that affect compiler transformations. The optimizations performed by the fill unit rely on the fact that later instructions will only be executed if earlier instructions are executed.

Our model for achieving high performance requires a strong compiler to take advantage of information available at compile time, coupled with aggressive hardware to exploit information only available during execution. Like dynamic branch prediction and out-of-order execution, dyanmic trace optimizations performed by the fill unit are a tool the hardware can use to improve the performance of static code.

2 Previous Work

The foundation of this work is the initial research performed on the trace cache by several groups. Its initial incarnations were developed by Melvin and Patt [8], Peleg and Weiser [12], and Johnson [7]. The concept was demonstrated by Rotenberg et al. [14] to be a low latency fetch device and developed by Patel et al. [11, 10] to be a very high bandwidth device.

Franklin and Smotherman [3] as well as Nair and Hopkins [9] have explored the run-time manipulation of the code stream by the fill unit. In both cases the fill unit is used to dynamically retarget a scalar instruction stream into

pre-scheduled instruction groups for a statically-scheduled execution engine.

The creation of the scaled add instructions is an instance of a concept called instruction collapsing explored by Vassiliadis et al [20]. Sazeides et al [15] looked into the performance potential of generalized *instruction collapsing* and the frequency of occurrence of certain instruction groups.

3 Experimental model

A pipeline simulator that allows the modeling of wrong path effects was used for this study. The simulator was implemented using the SimpleScalar 2.0 tool suite [1]. The SimpleScalar instruction set is a superset of the MIPS-IV ISA [13], modified such that architected delay slots have been removed and that indexed (register plus register) memory operations have been added. In the execution model, all instructions undergo four stages of processing before retirement: fetch, issue, schedule, execute. All stages take at least one cycle.

The fetch engine, capable of supplying up to 16 instructions per cycle, includes a large 2K entry, 4-way set associative trace cache. The trace cache requires approximately 156KB storage: 128KB for the 4-byte instructions and 28KB for 7 bits of pre-decode information associated with each instruction. Each trace cache line contains up to 16 instructions, with at most three conditional branches. (Unconditional branches are not considered to terminate blocks within trace segments.) Returns, indirect branches and serializing instructions force the trace segment to terminate. Subroutine calls do not. In addition there is a 4KB, 4-way set associative supporting instruction cache. A 1MB unified second level cache provides instructions and data with a latency of six cycles in the case of first level cache misses. Misses in the second level cache take 50 cycles to be fetched from memory if there is no bus contention.

The baseline configuration uses inactive issue [4]. With inactive issue, all blocks within a trace cache line are issued into the processor whether or not they match the predicted path. The blocks that do not match the prediction are issued inactively. When the branch that ended the last active block resolves, if the prediction was correct, the inactive instructions are discarded. If the prediction was incorrect, the processor has already fetched, issued and possibly executed some instructions along the correct path.

Furthermore, the trace cache implements both branch promotion and trace packing [10]. Branch promotion dynamically identifies conditional branches which are strongly biased. These branches are then promoted to receive a static prediction. The bias threshold was set to 64 consecutive occurrences. Trace packing allows the fill unit to pack as many instructions as will fit into a segment, without regard to naturally occurring block boundaries.

The branch predictor modeled, designed for use with branch promotion, is composed of three separate pattern history tables, each table consisting of an array of saturating 2-bit counters. The first table provides the prediction for the first conditional branch in the segment, while the second table provides the prediction for the second conditional branch and the third table predicting the third branch. As branch promotion reduces the number of times multiple branch predictions are needed, the PHT tables are skewed in terms of the number of entries each contain. The tables contain 64K entries, 16K entries and 8K entries respectively. With an 8KB bias table (needed for promotion), the storage cost of the branch predictor is roughly 32KB.

The execution engine is composed of 16 functional units, each unit capable of all operations. The functional units are grouped into four symmetric clusters of four functional units each. Instructions may forward their results for back-to-back execution to other functional units within the cluster, but an additional cycle of latency is required to forward a result to another cluster. Instructions are dispatched for execution from a 32-entry reservation station associated with each functional unit. A 64KB, 4-way set associative, L1 data cache is used for data supply. It has a load latency of one cycle after the address generation is complete. The model uses checkpoint repair [5] to recover from branch mispredictions and exceptions. The execution engine is capable of creating up to three checkpoints each cycle, one for each block supplied. The memory scheduler waits for addresses to be generated before scheduling memory operations. No memory operation can bypass a store with an unknown address.

All experiments were performed on the SPECint95 benchmarks and on several common UNIX applications [18]. The benchmark executables were compiled using gcc version 2.6.3 with -O3 optimizations. Table 1 lists the number of instructions simulated and the input set, if the input was derived from a standard input set[1]. All simulations were run until completion with the exceptions of li and ijpeg.

4 Dynamic Trace Optimizations

In this section, we provide the details for the specific trace optimizations we have proposed. We describe the

[1] Vortex and go were simulated with abbreviated versions of the SPECint95 test input set. Compress was simulated on a modified version of the test input with an initial list of 30000 elements.

Benchmark	Inst Count	Input Set
compress	95M	test.in
gcc	157M	jump.i
go	151M	2stone9.in
ijpeg	500M	penguin.ppm
li	500M	train.lsp
m88ksim	493M	dhry.test
perl	41M	scrabbl.pl
vortex	214M	vortex.in
gnuchess (ch)	119M	
ghostscript (gs)	180M	
pgp	322M	
gnuplot (plot)	284M	
python	220M	
sim-outorder (ss)	100M	
tex	164M	

Table 1. Benchmarks

hardware modifications required and provide the performance improvements from these optimizations.

4.1 Baseline Fill Unit

As instructions are retired by the machine, they are collected by the fill unit and combined into traces. As mentioned in the previous section, each trace can contain up to 16 instructions and include 3 conditional branches. The baseline fill unit also performs branch promotion, as described in [10]. Furthermore, the fill unit explicitly marks dependency information within the traces it constructs. Doing so simplifies the dependency checking required when the trace is later fetched from the trace cache.

Schemes for explicitly recording dependency information within a group of instructions have been proposed before [17, 19]. In our scheme, we record dependencies using an extra 7 bits per instruction. 3 bits are added to an instruction's destination to identify whether the destination is live-out of its checkpoint. Because we create 3 checkpoints per cycle, we require extra bits to identify situations where the destination is overwritten within another checkpoint issued that same cycle. We require 2 bits (1 bit per source operand) to identify whether the sources are defined internally or are live-in to the trace. If the value is internal, the register identifier is modified to indicate the instruction producing the value, which simplifies tag generation. Finally, 2 bits are required to identify an instruction's block number within a trace. These bits simplify the backup process on exceptions and mispredictions.

4.2 Register Moves

The proposed optimization is not one of altering the instructions of the executable, but rather one of how to handle a particular sub-class of instructions — register-to-register move instructions. These instructions are unique in that they do not perform any calculation, but simply pass a previously calculated value from one architected register to another. Such move operations are generated by compilers when performing common subexpression elimination, for register-based argument passing, and for initializing values to zero.

In dynamically scheduled processors, register renaming logic eliminates the artificial stalls due to false dependencies between instructions. As instructions are decoded, the renaming logic maps the architectural destination registers onto the larger physical register space. Similarly, the renaming logic must determine the correct mapping for each source register. When an instruction executes, its associated physical register is updated with the computed result.

Even though they appear as compute instructions, register-to-register move instructions do not need to be handled through the normal means of instruction execution. Instead, the register renaming logic can be modified to create a mapping for the destination of the move operation using the information stored in the mapping of the source. The instruction does not need to be sent to a reservation station nor to a functional unit before completion. The instruction is complete once the mapping for the destination has been made.

The modifications required to the renaming logic depend on the state maintenance mechanism being used. For schemes similar to checkpoint repair, the register alias table (RAT) entry for the source of the move operation is copied into the allocated RAT entry for the destination. If the source value is not yet ready, the source and destination physical registers share the same tag and both will be updated when the pending instruction executes. If the value is ready, the source physical register is copied to form the result of the move instruction.

If the state maintenance mechanism is a reorder buffer, then the physical register file must be separated from the reorder buffer queue. Each reorder buffer entry contains a pointer indicating the physical register allocated for that instruction. The reorder buffer entry for a move operation points to the same physical register as its source. Associated with each physical register is a count of the number of reorder buffer entries which refer to it. As instructions retire, the associated count is decremented. When the count reaches zero, the physical register is returned to the free list. A similar mechanism, which allows the x86 instruc-

tion FXCH to execute in zero cycles, has been patented by Intel [2].

Figure 2 presents an example of the execution of register move instructions by the rename logic. A partial snapshot of the speculative state after each instruction is renamed is shown for both checkpoint repair and a reorder buffer. In order to simplify the diagram, only the relevant fields of each RAT and reorder buffer entry are shown.

For checkpoint repair, the process of renaming the first instruction involves creating a RAT entry for its destination, R3. The second instruction is a move and creating its entry requires a lookup of the RAT entry for R3. Both the tag and value for R3 are duplicated for R4. This way, all instructions which require R3 or R4 as a source will receive either the same value (if it's ready) or the same tag (to identify the value when it's distributed).

For the reorder buffer, renaming the first instruction involves adding an entry for it at the tail of the buffer. Also a physical register is allocated from the free list to hold the value of R3. The reorder buffer entry is modified to point to the newly allocated physical register. In renaming the move instruction, an entry for the instruction is added to the tail of the reorder buffer. However, no new physical register is allocated. Instead the reorder buffer entry points to the same physical register already assigned to R3. Subsequent instructions sourcing R3 or R4 will receive either the same value (if it's ready) or the same index into the physical register file (to identify the value when it's distributed).

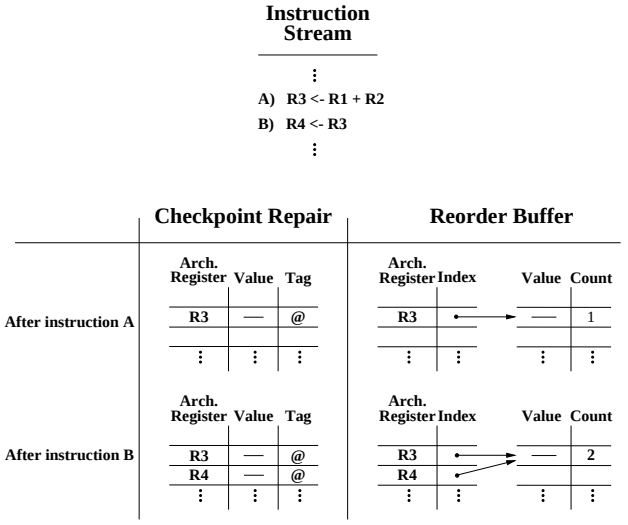

Figure 2. Executing move instructions in the rename hardware.

It should be clear that adaptations to both of the renaming mechanisms require a read of the move source mapping before the move destination mapping can be made. While this can be pipelined over two cycles without incurring latency on processing the move, instructions within the trace which source the result of the move must be modified to avoid a cycle of delay before being added to the instruction window. The fill unit handles this by modifying instructions within the trace cache line which are dependent upon the move operation to be dependent upon the source of the move instead.

The SimpleScalar architecture, like many of the prevalent commercial ISAs, does not have a specific instruction that copies a value from one register to another. Rather there are a number of ALU instructions which can be used to implement a register move instruction. The simplest is the add immediate instruction, *ADDI Rx ← Ry + 0*. When the immediate value is a zero, the value of register *Ry* is copied into register *Rx*. As the SimpleScalar semantics require that *R0* always maintain the value zero, there are a large number of instructions, both immediate and register based, which effectively carry out a register move instruction.

We propose that the fill unit detect and mark with a single bit such move instructions. By placing the detection logic in the processing path of the fill unit, the decode and rename logic can execute move instructions without having to pay for the latency of detecting them.

Figure 3 shows the performance improvement in instructions retired per cycle (IPC) of handling register move instructions in this manner. The performance increases an average of 5% across all the benchmarks. These move instructions accounted for 6% of the dynamic instruction stream. Results of an investigation of the SPECint95 compiled using the Digital C compiler (version 3.11 optimization level "-02 -Olimit 3000") also show this number to be 6%.

A secondary, but important, effect of early execution of register moves is that these instructions require no execution resources and thus incur no delays due to artifacts such as functional unit arbitration or latency through the operand bypass network.

4.3 Reassociation

Reassociation is an optimization which takes dependent pairs of immediate instructions, recomputes the immediate of the latter instruction, and modifies it to use the source register — instead of the destination register — of the earlier instruction. For example with the code sequence

ADDI Rx ← Ry + 4

ADDI Rz ← Rx + 4

the second instruction can be altered to *ADDI Rz ← Ry +*

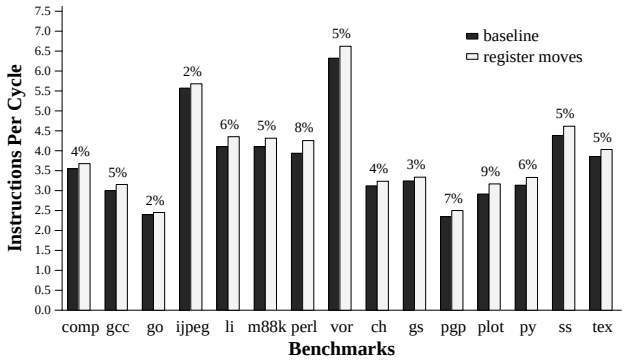

Figure 3. IPC improvement of register move instructions.

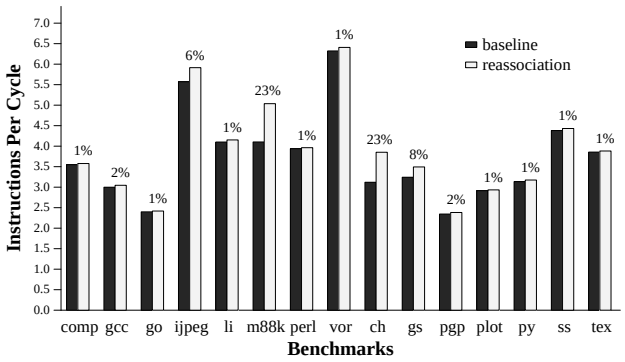

Figure 4. IPC improvement of fill unit reassociation.

8 . This optimization shortens the height of the dependency chain and has the potential to reduce the path length through critical portions of the code.

Implementing this optimization requires that the fill unit be able to detect such code sequences and create the appropriate immediate for the dependent instruction. As the dependency information is already being calculated, logic added to the fill unit examines the opcodes of dependent instructions and, if they are pairable immediate instructions, computes the new immediate field using a 16-bit ALU. Reassociation does not change the basic format of an instruction stored in the trace cache as we do not reassociate instructions for which the resulting immediate would be larger than the existing 16-bit immediate field.

Although the gcc compiler does reassociation as one of its standard optimizations, there are still a significant number of additional code sequences detected by the fill unit where reassociation can be applied. We have inhibited the application of reassociation by the fill unit, allowing it to only reassociate instructions which cross a control flow boundary. We do this in order to limit the impact of the compiler on our results. We have run simulations in which this restriction is not enforced and have seen no significant performance increase, indicating that the compiler is doing a good job of reassociation within the basic blocks.

The results of adding reassociation logic to the fill unit are plotted in figure 4. The results show a wide range of improvements. For ten of the fifteen benchmarks reassociation provides little improvement, 1% to 2%. However for both m88ksim and chess, reassociation produces significant improvement, boosting IPC by 23%. The benchmarks ijpeg and gs fall somewhere in between, improving by 6% and 8% respectively.

4.4 Scaled Adds

Much like the reassociation optimization, the creation of scaled add instructions reduces the length of the dependency chain through the program. Modifying the functional unit to perform a scaled add instruction allows a pair of dependent instructions such as

SHIFTI Rw ← Rx << 1

ADD Ry ← Rw + Rz

to be transformed into

SHIFTI Rw ← Rx << 1

SCALED ADD Ry ← (Rx << 1) + Rz.

These short immediate shifts and dependent adds (or the additions associated with an address computation) occur frequently in integer code as they are commonly used to calculate the memory location of an element within an array. For instance, to calculate the memory address of an entry in an array of 32-bit integers on a byte addressable machine, it suffices to shift the integer array index by 2 bits and add it to the base address of the array. In their work, Sazeides et al [15] presented results showing that approximately 5% of the instruction stream was composed of this type of pairing for the SPARC v.8 architecture. Our results with the SimpleScalar architecture are very similar.

Although reassociation requires no changes to the execution hardware, the scaled add instruction does. The ALU must be able to shift an input operand by a variable amount before the addition begins. As some current architectures support scaled adds and scaled loads [16, 6] and current implementations of them handle the operation in a single cycle, we expect that this additional logic will not affect the critical path of the processor. To ensure this, we have limited the size of the shift to be no more than 3 bits. This

limits the additional path length through the ALU to approximately 2 gate delays. Moreover, by limiting the size of the shift in this manner, the trace cache needs to store only two additional bits per instruction to maintain the scaled add information.

The logic required by the fill unit to create scaled add instructions scans the set of dependent operations for shift and add pairs and moves the 2-bit shift amount to the shift field of the add instruction. As we only allow one operand to be shifted, the fill unit may also have to interchange the order of the source operands of the add instruction.

We have applied this technique to our fill unit model, allowing small immediate shifts to combine with both dependent add and dependent load/store instructions. Figure 5 shows the results of this experiment. The improvements range from a low of 1% for li, vortex, pgp and plot to a high of 8% for go and tex. The average improvement for all the benchmarks is 3.7%.

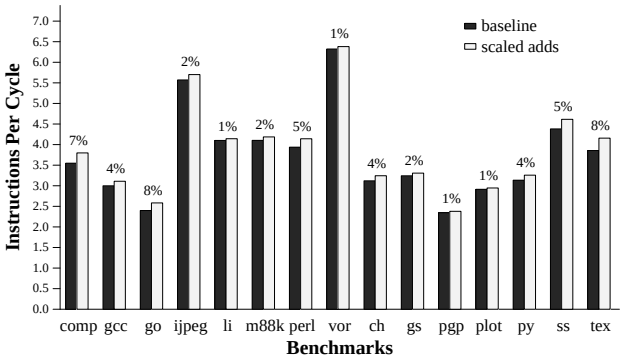

Figure 5. IPC improvement of implementing scaled add instructions.

4.5 Instruction Placement

Another way in which the fill unit has a unique opportunity to improve the performance of the code is by reducing the penalties associated with the latencies of the bypass network. With a clustered backend, many instructions are often unable to execute the cycle after their source operands are produced as they must wait for the value to be forwarded from the producing functional unit to the awaiting instruction's functional unit.

The fill unit can help address this problem by steering the instructions to specific functional units in order to reduce the amount of cross cluster communication. This can be accomplished either by adding a field to each instruction which indicates the functional unit the instruction should use. Or, as the true register dependencies are explicitly marked, the fill unit can physically reorder the instructions within the trace cache line. However, as the memory dependencies still need to be conveyed, the trace cache must maintain information about the original order of the instructions. Either of these techniques requires a 4-bit field to be added to each instruction.

Allowing the fill unit to physically reorder the instructions has the advantage of removing the routing crossbar from the issue path of the trace cache to the node table. It does not remove it from the processor altogether but rather moves it into the fill unit logic where the latency of the operation is less critical.

We have implemented a simple code placement hueristic in the fill unit to mitigate the effects of bypass latency. For each issue slot the fill unit looks for an instruction that is dependent upon an instruction already placed in that cluster. If no dependent instruction is found, the first unplaced instruction is put in that issue slot.

Figure 6 compares the results of a configuration where the fill unit implements this instruction placement with the baseline. The average improvement in IPC is 5%. Ijpeg posts the largest improvement of 11%, while tex benefits by only 1%.

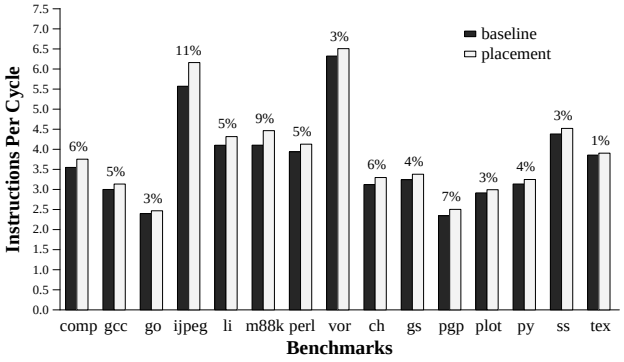

Figure 6. IPC improvement of fill unit instruction placement.

The percentage of on-path instructions which incurred cross-cluster communication delays for both the baseline and instruction placing configurations is shown in figure 7. For this graph we only count instructions whose last arriving source value was delayed by the bypass network. The graph shows that this simple placement algorithm does reduce the number of times the bypass latency is a factor. On avarage the placement algorithm reduces the absolute number of occurances from 35% to 29%.

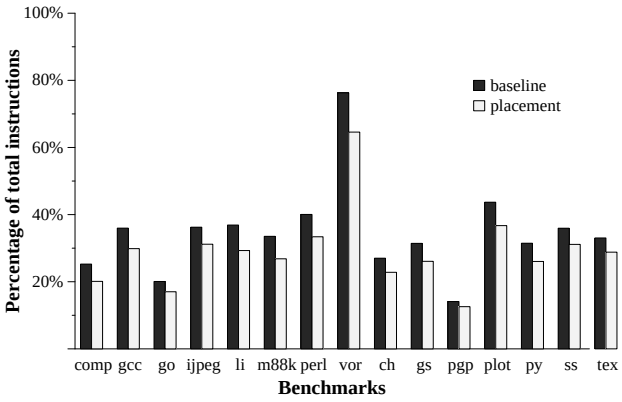

Figure 7. Reduction of instructions whose last arriving value was delayed by the bypass network.

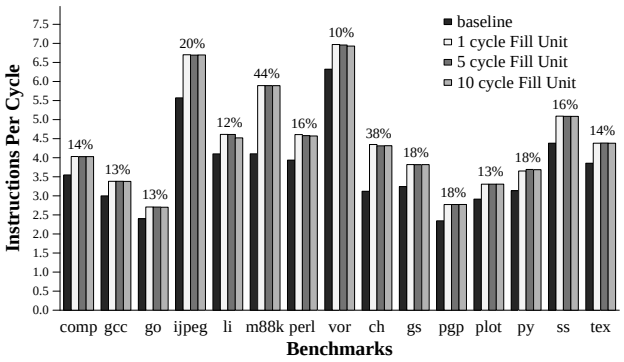

Figure 8. IPC improvement of the combined optimizations.

4.6 Combined Results

Figure 8 compares the IPC results of the baseline configuration with the results of the model in which all of the above dynamic optimizations are implemented: register moves are flagged by the fill unit and handled in decode, the fill unit reassociates source operands of immediate instructions, scaled add instructions are created by the fill unit, and the fill unit performs code placement. Each instruction in the trace cache is enlarged by 7 bits to enable these optimizations — 1 bit for register moves, 2 bits for scaled adds, and 4 bits for instruction placement. To account for the additional logic placed within the fill unit, we have varied the latency through the structure, setting it to 1 cycle, 5 cycles and 10 cycles. The percent improvments for the 5 cycle fill unit are shown and average slightly better than 18% across the benchmarks. Compress, gcc, go and plot show the least improvement, gaining only 13% or 14%, while m88ksim improves by 44% and chess by 38%. As can be seen the fill unit latency has a negligable impact on the performance.

Table 2 presents the percentage of correct path instructions that are optimized by the fill unit for each benchmark. On average slightly more than 13% of the instructions had some form of transformation applied to them. The benchmarks m88ksim and chess had more than a fifth of their instructions optimized.

5 Conclusion

We have demonstrated that the fill unit can be useful for performing dynamic trace optimizations.

The fill unit is uniquely qualified to perform some optimizations for several reasons. Since it is not on the critical path, the fill unit can perform multi-cycle operations without affecting performance. By combining multiple blocks of instructions from a single path of execution, it can easily perform cross-block optimizations. And the fill unit can tailor its optimizations to the characteristics of the microarchitectural implementation without modifying to the ISA.

We have presented four dynamic trace optimizations. Combined they improve performance on the SPECint95 benchmarks by more than 17% and over all the benchmarks studied by slightly more than 18%. These optimizations require an additional 7 bits to be added to each instruction within the trace cache line.

The full extent of the abilities of the fill unit have yet to be determined. The implementation of more aggressive optimizations, such as common subexpression elimination, may yield further improvements. It may also be advantageous to perform optimizations that require that the full trace cache line be treated as an atomic block. Dead code elimination, for example, could be used if the proper recovery mechanisms were in place to handle the cases in which the correct path of execution only follows a portion of the trace cache line. Furthermore, it may be possible to allow the fill unit to perform dynamic trace optimizations across multiple trace cache lines. Again the proper safeguards would need to be enforced, but enlarging the scope of the optimization window could increase the opportunities the fill unit encounters.

6 Acknowledgments

We would like to acknowledge the members of the HPS research group, both past and present, Jared Stark in particular, for their help. Thanks also to Stamatis Vassiliadis for

Benchmark	Register Moves	Reassociation	Scaled Adds	Total
compress	3.0%	1.5%	3.8%	8.3%
gcc	6.4%	2.2%	3.1%	11.7%
go	2.5%	0.7%	9.6%	12.8%
ijpeg	4.6%	2.1%	5.9%	12.6%
li	8.0%	2.1%	1.3%	11.4%
m88ksim	8.2%	12.9%	1.2%	22.3%
perl	6.3%	1.1%	3.3%	10.7%
vortex	9.4%	3.9%	1.9%	15.2%
gnuchess	3.4%	10.4%	5.7%	19.5%
ghostscript	4.6%	7.9%	1.9%	14.4%
pgp	7.9%	4.0%	1.0%	12.9%
gnuplot	11.3%	1.4%	2.3%	15.0%
python	6.3%	2.8%	2.8%	11.9%
sim-outorder	4.9%	1.1%	3.1%	9.1%
tex	3.1%	0.6%	5.2%	8.9%

Table 2. Percentage of instructions to which tranformations were applied.

his insights and for hosting us in Delft while we worked on the paper. We would also like to thank our corporate sponsors — Intel, HAL, IBM, and AMD — whose generous support is greatly appreciated.

References

[1] D. Burger, T. Austin, and S. Bennett. Evaluating future microprocessors: The SimpleScalar tool set. Technical Report 1308, University of Wisconsin - Madison Technical Report, July 1996.

[2] D. W. Clift, J. M. Arnold, R. P. Colwell, and A. F. Glew. Floating point register alias table FXCH and retirement floating point register array. U.S. Patent Number 5,466,352, 1996.

[3] M. Franklin and M. Smotherman. A fill-unit approach to multiple instruction issue. In *Proceedings of the 27th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 162–171, 1994.

[4] D. H. Friendly, S. J. Patel, and Y. N. Patt. Alternative fetch and issue techniques from the trace cache fetch mechanism. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, 1997.

[5] W. W. Hwu and Y. N. Patt. Checkpoint repair for out-of-order execution machines. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 18–26, 1987.

[6] Intel Corporation. *Pentium Processor User's Manual Volume 1: Pentium Processor Data Book*, 1993.

[7] J. D. Johnson. Expansion caches for superscalar microprocessors. Technical Report CSL-TR-94-630, Stanford University, Palo Alto CA, June 1994.

[8] S. W. Melvin and Y. N. Patt. Performance benefits of large execution atomic units in dynamically scheduled machines. In *Proceedings of Supercomputing '89*, pages 427–432, 1989.

[9] R. Nair and M. E. Hopkins. Exploiting instruction level parallelism in processors by caching scheduled groups. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 13–25, 1997.

[10] S. J. Patel, M. Evers, and Y. N. Patt. Improving trace cache effectiveness with branch promotion and trace packing. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, 1998.

[11] S. J. Patel, D. H. Friendly, and Y. N. Patt. Critical issues regarding the trace cache fetch mechanism. Technical Report CSE-TR-335-97, University of Michigan Technical Report, May 1997.

[12] A. Peleg and U. Weiser. Dynamic flow instruction cache memory organized around trace segments independant of virtual address line. U.S. Patent Number 5,381,533, 1994.

[13] C. Price. *MIPS IV Instruction Set, revision 3.1*. MIPS Technologies, Inc., Mountain View, CA, 1995.

[14] E. Rotenberg, S. Bennett, and J. E. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, 1996.

[15] Y. Sazeides, S. Vassiliadis, and J. E. Smith. The performance potential of data dependence speculation and collapsing. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, 1996.

[16] R. L. Sites. *Alpha Architecture Reference Manual*. Digital Press, Burlington, MA, 1992.

[17] E. Sprangle and Y. Patt. Facilitating superscalar processing via a combined static/dynamic register renaming scheme. In *Proceedings of the 27th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 143–147, 1994.

[18] J. Stark, P. Racunas, and Y. N. Patt. Reducing the performance impact of instruction cache misses by writing instructions into the reservation stations out-of-order. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 34 – 43, 1997.

[19] S. Vajapeyam and T. Mitra. Improving superscalar instruction dispatch and issue by exploiting dynamic code sequences. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 1–12, 1997.

[20] S. Vassiliadis, B. Blaner, and R. J. Eickemeyer. Scism:a scalable compound instruction set machine. *IBM Journal of Research and Development*, 38:59–78, 1994.