Compiler-Based Frame Formation for Static Optimization

Feng Shi, Sobeeh Almukhaizim[*], Pey-Chang Lin and Yiorgos Makris
Electrical Engineering Dept.
Yale University
New Haven, CT 06511, USA

Abstract

We selectively generate and optimize the frames constructed by the rePLay architecture statically. Since static analysis provides a global view of the interaction between the basic blocks and a bigger aggressive optimization space, we propose a method to construct the frames using profiling and static analysis. Frame selection and optimization are analyzed in the criteria to produce well-optimized, frequently executed frames with minimum recovery penalty. In addition, hardware support is reduced to only perform mis-speculation recovery. Empirical results show frame-optimized code outperforming baseline code on the SPEC integer benchmarks.

1 Introduction

Current microarchitectures are moving to deeper pipelines, wider issue widths and larger number of functional units. For high performance processors, extracting sufficient Instruction-Level Parallelism (ILP) is an increasing challenge. Unfortunately, control dependencies prevent exploiting these hardware resources efficiently [8, 20]. In an effort to boost ILP, hardware-based (dynamic) [5, 9, 14, 18] and compiler-based (static) [7, 11, 12, 13, 17] techniques have been proposed to reduce control dependencies. The *goal* is clear: identify and optimize large sequences of frequently executed code by folding the branches between the basic blocks. As the speculation when folding branches is not always correct, some form of recovery is provided to tolerate mis-speculated execution.

Dynamic techniques adaptively generate and optimize the frequently executed code at run time, while static techniques analyze the program code at compile time. Unfortunately, dynamic methods may incur a prohibitive implementation cost [4] and the lack of dynamic information, manifested in a low branch prediction accuracy, reduces the benefits of static methods. We present an aggressive compiler-based technique that takes advantage of both dynamic and static techniques to enhance the performance of superscalar architectures. The proposed method utilizes profiling information to construct frames [18] at compile time. Profiling information aids the static analysis of the program flow to select among the frames. The extended code length and atomic nature of frames enables more aggressive optimization algorithms at compile time. Consequently, the proposed technique requires minimal hardware support; only to recover from mis-speculated frames.

The remainder of this paper is organized as follows. In section 2, we discuss previous hardware and compiler-based techniques. Section 3 presents the proposed frame construction and optimization method. We present experimental results evaluating the proposed method in section 4 and conclude in section 5.

[*]The author is supported through a scholarship from Kuwait University.

2 Related Work

ILP is limited by the number of instructions in a basic block. In order to achieve higher levels of ILP, processors fetch and execute instructions from multiple basic blocks in each cycle. Several compiler and hardware techniques have been proposed. Most of these schemes group sequential basic blocks into larger entities, thereby reducing control dependencies, increasing the fetch rate and allowing more opportunities for optimization. Previous approaches include superblock formation [13], predicated execution using hyperblocks [17], VLIW treegion scheduling [12], block-structured ISA [11] and frame scheduling [6].

Superblocks comprise a block of instructions with single entry and multiple exits [10, 13]. A superblock is formed based on static branch analysis and/or profiling information. Profiling identifies blocks that frequently execute in sequence and places them in consecutive locations. All side entrances to a superblock are removed by tail duplication. Therefore, when a superblock is entered, it is likely that it will execute completely.

Control dependencies in hyperblocks [17] are converted into data dependencies through the process of *if-conversion*. Similar to superblocks, successive sets of instructions are combined and optimized to form a single hyperblock, however, hyperblocks may include multiple paths if the bias cannot be determined. A disadvantage with predication is the increase in the critical path length. The processor must wait for the data dependency to resolve rather than speculate the control dependency at the fetch stage.

VLIW treegions [12] use multiple execution paths to improve the compiler chances to speculate operations. Treegion formation combines trees of basic blocks with a rooted Control Flow Graph (CFG) into a single VLIW instruction. The scheduling of instructions can use or ignore the profiling information. A disadvantage is the increased pressure on the compiler to find enough independent instructions to fill each VLIW instruction and to schedule the conditional operations early enough for multi-way jumps.

Block-Structured ISA [11] enlarges basic blocks and considers the newly generated blocks as the architectural atomic unit. Since the blocks in the enlarged blocks are chosen using static information, a limit may be reached in the quality of the enlarged blocks generated. Hardware support is required through the branch predictor for multiple branch predictions.

Mesocode [22] is a code format designed to improve the fetch bandwidth efficiency of Itanium® processors. A trace-driven post-pass compiler identifies frequently executed *streams*, or contiguous instructions executed in between two taken branches, and encodes them into mesocode regions which are attached as an appendix to the original code. On average, the size of a stream is about twice the size of a basic block. The machine supporting mesocode can predict, fetch, and execute these streams efficiently.

The rePLay microarchitecture [18] is a hardware mechanism for frame creation and optimization. A frame is an atomic region combining multiple basic blocks by converting branches into assertions. The average frame length is usually much longer than the length of a block or a trace. The hardware support requires a frame constructor, a programmable engine for frame optimization, a frame cache, a region sequencer and a recovery mechanism for incompletely executed frames. As the program executes, the frame constructor collects instructions and converts highly biased branches into assertions. The candidate frame is sent to the optimization engine to perform classical compiler optimizations. After the optimized frames are stored in the frame cache, the sequencer is updated with the new set of available frames. The sequencer is responsible for dispatching either basic blocks or frames based on the confidence that a frame will execute completely. If an assertion fires, the architectural state is recovered by the recovery mechanism.

The design of the rePLay microarchitecture is a significant challenge as frames are constructed dynamically in hardware at an extremely fast rate. Moreover, the set of optimizations that may be performed on a frame is limited by the affordable hardware cost and the time the optimizer can work on the frame. Nonetheless, the frames that can be constructed if the hardware constructor is capable of generating and optimizing these frames result in a large speedup. As will be discussed in the next section, we propose moving the frame constructor and optimizing engine to the compiler, hence, the hardware cost can be kept low without sacrificing the speedup.

3 Proposed Method

The proposed method aims at reducing the gap between static and dynamic techniques. We propose modifying the compiling process to utilize profiling information while constructing frames. The frame constructor in the rePLay architecture provides the largest stream of instructions when compared to other dynamic and static techniques. We propose performing static analysis to *selectively* include frames in the executable binary. The frames are either constructed by the rePLay architecture or statically by imitating the behavior of the rePLay frame constructor. The binary can run on any architecture that has speculation supported. The branch predictor of the targeted architecture must update the branch history only when a frame is successfully executed. When an assertion fires, the machine state is recovered and the execution is redirected to the corresponding original basic block. In order to reduce the penalty associated with firing assertions, we follow the same observation as in [6] and push the assertions as high as possible within the frame.

Similar to block-structured ISAs and treegions, the proposed method significantly reduces the hardware cost. However, the proposed method can effectively control the number and size of included frames; a goal that the other static schemes may not control. Moreover, the scalability in performance of block-structured ISAs and treegions is highly questionable as they must generate treegions and enlarged atomic blocks that match the processor width. The advantages of the proposed method as compared to dynamic techniques are more evident. First, optimizations using time-consuming data flow analysis can be easily performed at compile time. Second, most of the dynamic techniques use a separate cache to hold the dynamically generated frames or traces. In contrast, the proposed method is able to utilize the complete instruction cache to store the frames. Third, hazardous conditions to dynamically constructed frames, such as subroutine calls, are easily identified and tolerated using static analysis. Finally, the dynamic construction of frames must be done on-the-fly to sustain the optimizer throughput [6]. The arrival of frames in the rePLay architecture is one frame every 110 clock cycles; the design of an optimizing engine that operates at that rate remains a significant design challenge.

3.1 Hardware Support for Frames

We have modified the HPL-PD [15] ISA to incorporate the speculative nature of the proposed method. Three additional instructions were added: FSTART, AST and LAST. A frame start instruction (FSTART) is inserted in the beginning of each frame to notify the processor of a speculative state of execution. Like a NOOP instruction, FSTART uses only one issue slot. All conditional branches in a frame are transformed into assertions (AST) and all unconditional branches are nullified. The last AST in a frame is converted to a last assertion (LAST); if all ASTs and the LAST do not fire then we have a frame hit. All operations between FSTART and LAST are simply viewed as speculative operations and their results should not be committed until the LAST is correctly executed. Similar to a reorder buffer in superscalar architectures, a frame buffer stores the temporary results for registers and memory. If an AST fires, the results in the frame buffer are simply discarded and the program is redirected to the corresponding original code.

Instructions may generate exceptions during the execution of frames. All the operations between FSTART and LAST in a frame are considered speculative; any raised exception is not handled immediately and the exception bit of the destination register is set [2]. If the exception bit of a source register is set, then the exception is propagated and the exception bit of the destination register is also set. ASTs and LAST are implicit check points where if any prior exception occurred then the state is recovered and the execution is redirected to the original code. Genuine exceptions will be raised again and handled during the execution of the original code.

3.2 Frame Analysis

The program trace is analyzed to find *good* sequences of basic blocks to form frames. Several factors determine the quality of a frame. First, the size of the frame should not be very large. Indeed, generating a large frame may lead to undesirable side-effects such as code expansion or increase in the instruction cache miss rate. Second, the execution frequency of the frames should be high. The speedup from a certain frame is limited by the number of times the frame is executed. The third consideration is the optimization potential of a frame. Sequences of basic blocks with strong data dependencies limit code optimization and, therefore, are poor candidates for constructing frames and should be excluded. The fourth consideration is the penalty of a fired assertion in a frame. As in the case when a branch is mispredicted, an assertion that fires because the frame will not execute completely causes the processor to nullify these executions. The fifth consideration comes from the hardware/software recovery mechanism for supporting frames. As previously stated, the speculative state for frames is limited by the microarchitectural support on the number of speculative instructions that it can keep track of. Naturally, the frame constructor must not generate frames that require more bookkeeping than the microarchitecture can offer.

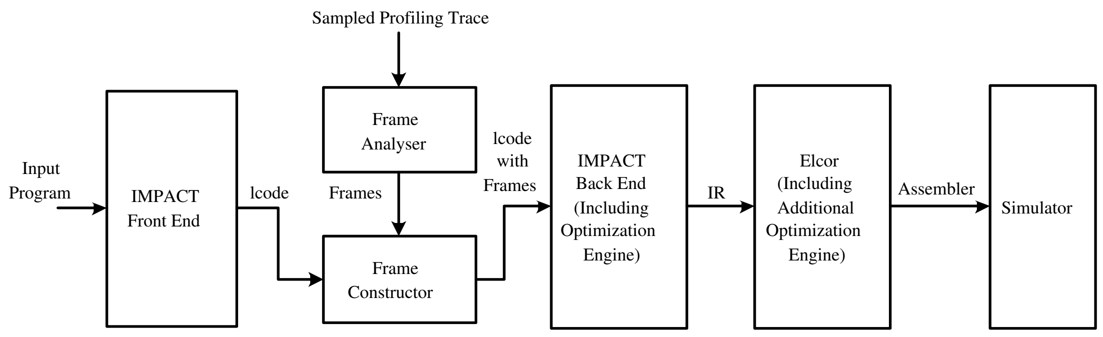

Figure 1. Frame Formation and Optimization Tool Based on Trimaran Tool Set

In order to find candidate basic blocks for frame formation, frame analysis takes place in two steps. In the first step, a frame parsing program detects hot spots in the trace by counting the repetition for each pattern of certain lengths. The patterns with an execution frequency beyond a predefined threshold are selected. The threshold can be an absolute value defined for each trace or a percentage of the trace length. In the second step, the starting block and the ending block are determined for each selected pattern. Although trace patterns with high frequencies are selected in the previous step, the range for each pattern is not clear yet. For example, the trace parsing program detects a frequently executed pattern of length 7 that is a subset of a frequently executed pattern of length 10; one frame suffices in most of these cases.

Let L be the number of execution cycles it takes to completely execute a frame. We introduce a quantitative criterion called the effective frame length (L_{eff}) to assess the quality of a pattern. The effective length of a frame is the *average* number of execution cycles each time a frame is completely executed:

$$L_{eff} = \frac{C_f}{N_{Hit}} = L + \frac{R_{Miss}}{R_{Hit}} \times P \tag{1}$$

where C_f is the total number of execution cycles the program spends in this frame, N_{Hit} is the total number of times that the frame is completely executed, P is the average penalty if an assertion fires and R_{Hit} and R_{Miss} are the percentages of times that the frame is completely and incompletely executed, respectively. R_{Hit} and R_{Miss} are related to the conditions under which the frame is executed. In this work, frames are statically linked into the program; a frame is executed every time the first basic block of the frame is referenced. The average speedup (S_f) of a frame is:

$$S_f = \frac{L_{original}}{L_{eff}} \tag{2}$$

where $L_{original}$ is the number of execution cycles of the original set of basic blocks. Substituting L_{eff} using (1), we obtain:

$$S_f = \frac{L_{original} \times R_{Hit}}{L \times R_{Hit} + P \times R_{Miss}} \tag{3}$$

S_f depends on L and P for every candidate frame. Since L and P are attributes of a frame, they can only be accurately computed after the frame is formed and optimized. The quality of frames can be enhanced by integrating the pattern selection along with the frame formation and optimization procedure. Heuristic methods for estimating L and P may be utilized in order to reduce the computation complexity, however, the quality of frames will be reduced.

3.3 Implementation

We utilize a basic heuristic to select frames based on the frame size, the execution frequency, the *entrance rate*, and the *exit rate*. The *entrance rate* is the percentage of times a candidate frame is completely executed and the *exit rate* is the ratio between the entrance rate of a candidate frame, extended by one more block, to the entrance rate of the original candidate frame. A high entrance rate indicates the likelihood that a frame will completely execute if the first basic block is executed. A low exit rate indicates that adding the block in consideration will greatly reduce the probability of completely executing the frame.

We consider frame candidates with an entrance rate higher than 80% for the first three basic blocks. The size of frames is set between 3 to 30 basic blocks; a frame smaller than 3 blocks is assumed too small to optimize and a frame bigger than 30 blocks is too large for the hardware recovery mechanism to maintain. The frame selection procedure selects between frame candidates and decides which basic block begins or ends a frame. A frame is terminated if the exit rate is smaller than 0.95 or if the frame size is bigger than the upper bound on the number of blocks. Only those frames whose executions are not covered by other longer frames are selected.

The frame formation and optimization stage is performed after the frames are selected. A frame formation and optimization tool was developed based on the Trimaran tool set [21]. As illustrated in figure 1, the tool generates frames based on the *lcode* representation of the program and the frame information gathered in the frame selection phase. All the operations in the basic blocks of a frame are duplicated; branches are replaced by assertions and operations are packed into a single block. The frame is inserted into the original program and all original branch operations that branch to the first basic block of the frame are retargeted to the head of the frame. The new program performs the same operations of the original program but can be further optimized with Trimaran. The set of optimizations applied by IMPACT [3] is: dead code removal, reverse copy propagation, constant propagation, copy propagation, memory copy propagation, common subexpression elimination, redundant load/store elimination, constant combination, constant folding, strength reduction, code motion, operation folding, operation cancellation, sign extension removal and register renaming. During the generation of machine-dependent intermediate representations, additional optimizations such as forward copy propagation, dead code elimination and common subexpression elimination are applied by Elcor [16]. At this point, the assembly code is generated along with the schedule length of all the basic blocks and frames.

Benchmarks		
Name	*Input Set 1*	*Input Set 2*
bzip2	input.compressed	NA
gcc	amptjp.i	c-dec1-s.i
go	2stone9.in	5stone21.in
gzip	input.combined	input.program
ijpeg	vigo.ppm	penguin.ppm
li	training	testing
mcf	training	inp.in
twolf	training	NA
vortex	persons.250	persons.1k
vpr	training 1	testing 1

HPL-PD Architecture			
Register File Type	*Size*	*Operation Type*	*Latency*
General purpose: static	64	Integer	1
General purpose: rotating	64	Integer Mul/Div	3/8
Floating-point: static	64	Floating-Point	3
Floating-point: rotating	64	FP Mul/Div	3/8
Predicate: static	256	Load Lev1/Lev2/Lev3	2/7/35
Predicate: rotating	64	Store	1
Branch-target	16	Branch	1
Control	64	AST/LAST	1

Table 1. Architecture and Benchmark Characteristics

Benchmark	*bzip2*	*gcc*	*go*	*gzip*	*ijpeg*	*li*	*mcf*	*twolf*	*vortex*	*vpr*	*Average*
Number of Frames	9	639	147	27	41	30	30	51	57	50	108.1
Code Expansion (%)	10.7	0.7	4.8	9.4	5.3	6.7	2.0	3.5	1.5	4.6	4.9
Avg. # of Basic Blocks	11.5	4.89	3.7	4.6	5.7	4.9	5.0	4.1	9.8	5.0	5.9
Avg. # of Operations	75.9	30.0	21.9	29.0	29.3	27.3	26.4	27.1	44.4	50.4	36.2
Avg. Length (L)	20.5	11.0	10.3	11.3	10.0	10.8	9.8	12.4	16.5	19.7	13.2
Avg. Frame Hit Rate	0.99	0.91	0.94	0.99	0.90	0.91	0.93	0.89	0.98	0.85	0.93
Avg. L_{eff}	20.6	11.7	10.8	11.4	10.6	11.5	10.4	13.1	16.9	21.1	13.8
Avg. Reduction (%)	10.1	7.5	-0.4	-0.6	3.4	27	-0.1	-0.7	22.7	6.5	7.5
Avg. Speedup/Frame	2.79	2.09	1.63	1.67	2.19	2.28	2.01	1.75	3.09	1.98	2.15

Table 2. Frame Characteristics

4 Experimental Results

In order to analyze the performance of the proposed method, six SPEC00 and four SPEC95 integer benchmarks were simulated to completion. The SPEC95 benchmarks were chosen due to compilation errors by Trimaran on the SPEC00 version of these benchmarks. All the benchmarks were compiled using the default optimization flags of Trimaran 3.0.b [21] distribution with region formation set to basic blocks and the executable generated for the HPL-PD [15] architecture. The architecture was configured based on the default settings of an 8-wide HPL-PD architecture: four integer units, two floating units, two memory units, one branch unit and an ideal instruction cache. The ISA was modified to include the FSTART, AST and LAST instructions [1]. The benchmarks used and the architecture characteristics are listed in table 1. We compared the frame-optimized code with the baseline code generated by Trimaran using path profiling with the same optimization flags.

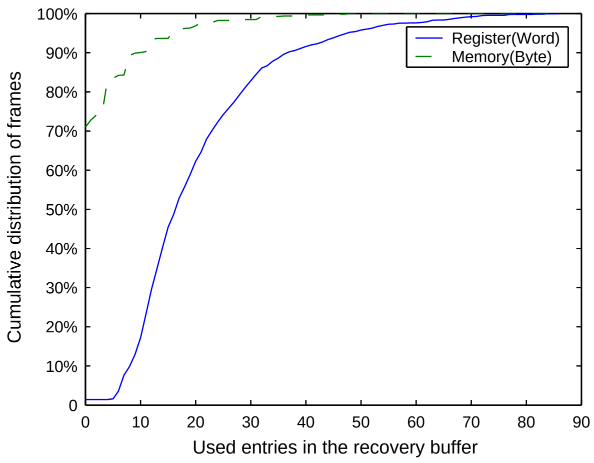

Figure 2. % of Frames vs. Buffer Size

4.1 Frame Selection

The first input set of table 1 is used as a profiling input to construct frames. Frames are identified using a trace sampling method. First, frequently executed basic blocks in each benchmark are identified. The execution time of the frequent blocks is usually above 90% of the total program execution time. The trace is randomly sampled for 100M clock cycles to find frames containing these frequent blocks according to the criteria described in section 3.3.

The characteristics of the frames generated for the simulated benchmarks are illustrated in table 2. The number of frames is reported in the second row of the table. It can be seen that an average of 108 frames were generated across all the benchmarks. Including all the generated frames resulted in an average code expansion of only 5%. The average number of basic blocks and operations in frames for all the benchmarks is illustrated in the forth and fifth row, respectively. The execution time of the frames (L) is 14 cycles on average. With a frame hit rate of 93%, L_{eff} is almost the same as L, since the penalty effect of a mis-speculated frame is marginal. When the frames are optimized, we obtain an average reduction of 7.5% in the number of operations, as compared to the original basic blocks. In some instances, additional operations, such as register spill/read operations, added by the compiler during the optimization process, increase the number of operations. Finally, the average speedup of frames is in excess of 100% as the compiler is able to find more operations to execute in parallel.

The size of the recovery frame buffer was not restricted during the analysis. Figure 2 illustrates the cumulative frame distribution according to the size of the register/memory buffer. The required size of the frame buffer ranges between 0 – 86 registers and 0 – 48 memory bytes. A buffer of 64 registers and 32 memory bytes would handle more than 98% of the frames across all the benchmarks.

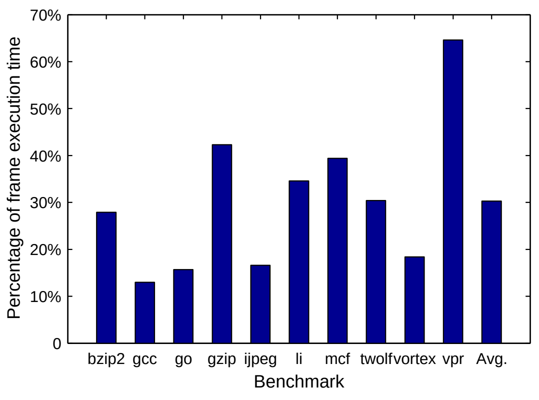

Figure 3. % of the Execution Time of Frames

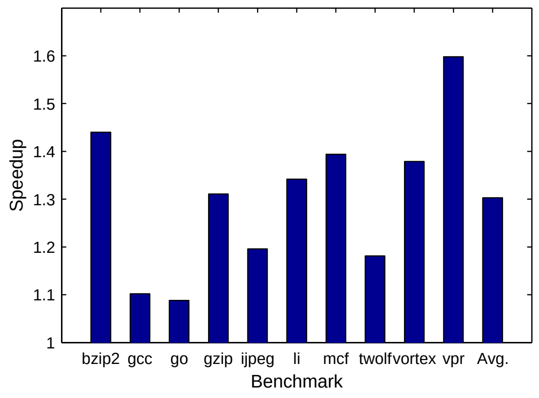

Figure 5. Speedup (Perfect Profiling)

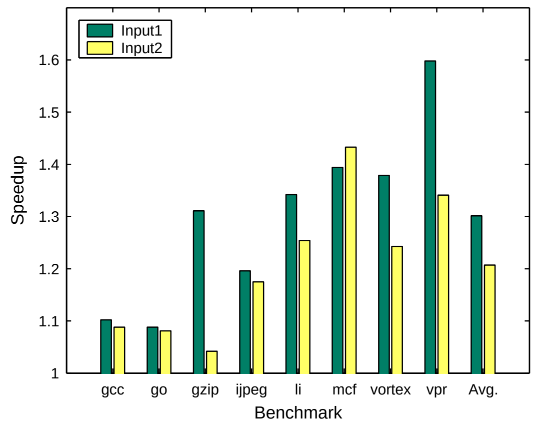

Figure 4. Performance Across Input Sets

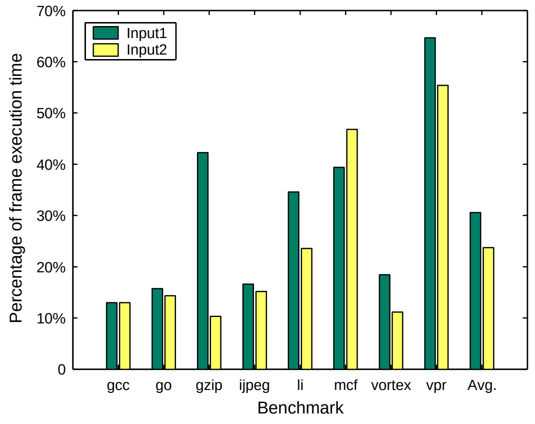

Figure 6. % of the Execution Time of Frames

4.2 Performance (Perfect Profiling)

The speedup of frames depends on two factors: the average speedup and the execution frequency for each frame. These two factors were measured on the frames generated for the first input set. The average of 108 frames per benchmark, constituting less than 5% of the code size, contributes a considerable portion of the program total execution time, as illustrated in figure 3. Combining these frequent frames with the average speedup per frame yields a speedup of 31% on the benchmarks, as illustrated in figure 5.

4.3 Performance Across Input Sets

We simulated eight of the optimized benchmarks for a second input set, as indicated in table 1. Figure 4 compares the speedup of each benchmark on the second input to the perfect profiling case. Despite the enhanced performance of *mcf* and the sustainable performance of *gcc*, *go*, and *ijpeg*, we do observe tremendous performance degradation in benchmarks like *gzip*. On average, the speedup dropped to 21%. The performance degradation is attributed to insufficient profiling and/or frame corruption. Insufficient profiling fails to provide a full-scale picture about the potential hot spots in the program, hence the execution time of the generated frames does not dominate the execution time of the program at runtime. For example, the hot spots of *gzip* under the first input set and under the second input set do not fully overlap. In addition, some functions are only executed under one of the two input sets. The lack of accurate profiling information is further worsened by the trace sampling method, as the complete trace is not analyzed in order to reduce the analysis complexity. Frame corruption degrades the performance if the hit rate of a frame decreases when the program switches to another input set. Since the frame corruption is caused by the dynamic behavior of the program, a purely static scheme can do very little about it.

We analyzed the frame execution time and hit rate across input sets. Figure 6 illustrates the percentage of frame execution time for each benchmark on both input sets if the frames are generated based on perfect profiling for the first input set. The benchmarks with degraded performance in figure 4 have almost the same drop in frame execution time in figure 6. We also observed corrupted frames in the benchmarks with the second input set, as illustrated in table 3. The table lists the frame distribution in percentage according to the change in the frame hit rate across different input sets. The frame hit rate of the majority of the frames is reduced by less than 10%, while the frame hit rate of a few frames increases slightly. Despite the overall unaffected frame hit rate, the real problem remains the frequency with which these frames are executed.

We reevaluate the constructed frames of *gzip* to account for the program behavior on different input sets. We constructed the new frames based on profiling across input sets . In this case 55 frames were generated based on the second input set profiling, and 5 more frames were appended according to the first input set profiling. The generated frames resulted in a speedup of 38% for the first input set

Benchmark	*> 0%*	*-(0-10%)*	*-(10-20%)*	*-(20-30%)*	*-(30-40%)*	*-(40-50%)*	*-(≥ 50%)*
gcc	0.16%	99.84%	0	0	0	0	0
gzip	36%	44%	4%	0	4%	0	12%
go	29.93%	67.35%	0.68%	2.04%	0	0	0
ijpeg	56.41%	43.59%	0	0	0	0	0
li	30%	33.33%	23.33%	10%	3.33%	0	0
mcf	58.06%	41.94%	0	0	0	0	0
vortex	7.02%	92.98%	0	0	0	0	0
vpr	12%	36%	20%	24%	6%	2%	0
average	28.70%	57.38%	6.00%	4.51%	1.67%	0.25%	1.5%

Table 3. Frame Hit Rate Degradation

Benchmark	Superblock			Proposed Method		
	Code Exp.	Speedup1	Speedup2	Code Exp.	Speedup1	Speedup2
gzip	342.3%	-14.5%	20.0%	9.4%	31.1%	4.2%
gzip(across inputs)	342.3%	-14.5%	20.0%	22.9%	37.9%	44.4%
li	62.8%	18.1%	6.2%	6.7%	34.2%	25.4%
mcf	675.8%	33.9%	29.9%	2.0%	39.4%	43.3%

Table 4. Speedup in Comparison to Superblocks

and 44% for the second input set. The program spent 58% and 63% of the execution time in executing the frames, respectively.

The proposed method is compared to the superblock technique in table 4. Superblocks are generated by Trimaran with the default configuration using path profiling, while frames are constructed by profiling input set 1 for each benchmark except in the second case of *gzip* where both input sets are profiled. The proposed method incurs far less code expansion than superblocks as illustrated in the second and fifth columns. The speedup of superblocks for input set 1 and 2 is listed in the 3rd and 4th column respectively, and for proposed method in the 6th and 7th column. In most cases the proposed method performs better than the superblock technique, except in the insufficient profiling case of *gzip*.

5 Conclusions

We propose a technique to construct and optimize frames in the compilation process by incorporating profiling and static analysis. The proposed method remedies the shortcomings of dynamic/hardware-based methods, such as extensive optimizations and global view of the program. Moreover, minimal hardware support is required to only perform mis-speculation recovery. Our experiments on the SPEC integer benchmarks indicate an average speedup in execution time of 31% in case of perfect profiling and 21% across input sets.

Acknowledgements

The authors would like to thank Daniel Friendly (Yale University) for his contributions to this work. We also would like to acknowledge the helpful discussions with Hao Wang, Haiyong Xie (Yale University) and Rodric M. Rabbah (Georgia Institute of Technology) on this project.

References

[1] S. Aditya, V. Kathail, and B. Rau, “Elcor's machine description system: Version 3.0,” *Technical Report HPL-1998-128, HP Laboratories*, 1998.

[2] D.I. August, D.A. Connors, S.A. Mahlke, J.W. Sias, K.M. Crozier, B. Cheng, P.R.Eaton, Q.B. Olaniran, and W. Hwu “Intergrated predicated and speculative execution in the IMPACT EPIC architecture,” in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pp. 227–237, 1998.

[3] P.P. Chang, S.A. Mahlke, W.Y. Chen, N.J. Water, and W.W. Hwu, “IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors,” in *Proceedings of the 18th Annual Int'l Symposium on Computer Architecture*, pp. 266–275, May 1991.

[4] Y. Chou, P. Pillai, H. Schmit, and J. Shen, “PipeRench implementation of the Instruction Path Coprocessor,” in *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 147–158, 2000.

[5] Y. Chou, and J.P. Shen, “Instruction path coprocessors,” in *Proceedings of the 27th International Symposium on Computer Architecture*, pp. 270–281, June 2000.

[6] B. Fahs, S. Bose, M. Crum, B. Slechta, F. Spadini, T. Tung, S. Patel, and S. Lumetta, “Performance characterization of a hardware mechanism for dynamic optimization,” in *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pp. 16–27, 2001.

[7] J.A. Fisher, “Trace scheduling: A technique for global microcode compaction,” *IEEE Trans. Comput.*, vol. C-30, pp. 478–490, 1981.

[8] C.C. Foster, and E.M. Riseman, “Percolation of code to enhance parallel dispatching and execution,” *IEEE Trans. Comput.*, vol. C-21, pp. 1411–1415, 1972.

[9] D.H. Friendly, S.J. Patel, and Y.N. Patt, “Putting the fill unit to work: dynamic optimizations for trace cache microprocessors,” in *Proceedings of the 31st Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 173–181, 1998.

[10] R. Hank, S. Mahlke, R. Bringmann, J. Gyllenhaal, and W. Hwu, “Superblock formation using static program analysis,” in *Proceedings of the 26th annual international symposium on Microarchitecture*, pp. 247–255, 1993.

[11] E. Hao, P. Chang, M. Evers, and Y. Patt, “Increasing the instruction fetch rate via block-structured instruction set architectures,” in *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pp. 191–200, 1996.

[12] W. Havanki, S. Banerjia, and T. Conte, “Treegion scheduling for wide issue processors,” in *Proceedings of 4th International Symposium On High-Performance Computer Architecture*, pp. 266–276, Feb. 1998.

[13] W.W. Hwu, S.A. Mahlke, W.Y. Chen, P.P. Chang, N.J. Warter, R.A. Bringmann, R.G.Ouellette, R.E. Hank, T. Kiyohara, G.E. Haab, J.G.Holm, and D.M.Lavery, “The superblock: An effective structure for vliw and superscalar compilation,” *The Journal of Supercomputing*, vol. 7, no. 1, pp. 229–248, January 1993.

[14] Q. Jacobson, J.E. Smith “Instruction pre-processing in trace processors,” in *Proceedings of 5th International Symposium On High-Performance Computer Architecture*, pp. 125–129, Jan. 1999.

[15] V. Kathail, M. Schlansker, and B. Rau, “HPL-PD architecture specification: Version 1.1,” *Technical Report HPL-93-80 (R.1). HP Laboratories*, 1994.

[16] V. Kathail, M.S. Schlansker, and B.R. Rau, “Compiling for EPIC architectures,” in *Proceedings of the IEEE*, vol. 89, pp. 1676–1693, Nov. 2001.

[17] S. Mahlke, D. Lin, W. Chen, R. Hank, and R. Bringmann, “Effective compiler support for predicated execution using the hyperblock,” in *25th Annual International Symposium on Microarchitecture*, 1992.

[18] S. Patel and S. Lumetta, “rePLay: A hardware framework for dynamic optimization,” *IEEE Trans. Comput.*, vol. 50, no. 6, pp. 590–608, 2001.

[19] M.S. Schlansker, and B.R.Rau, “EPIC: Explicitly Parallel Instruction Computing,” *Computer*, vol. 33, no. 2, pp. 37–45, Feb. 2000

[20] G.S. Tjaden, and M.J. Flynn, “Detection and parallel excution of independent instructions,” in *IEEE Trans. Comput.*, vol. C-19, pp. 889–895, Oct. 1970.

[21] Trimaran toolset, available at *http://www.trimaran.org*.

[22] M. Eng, H. Wang, P. Wang, A. Ramirez, J. Fung, and J. Shen, “Mesocode: Optimizations for Improving Fetch Bandwidth of Itanium Processors,” in *Workshop on Complexity-Effective Design*, May 2002.