Superblock Formation Using Static Program Analysis

Richard E. Hank Scott A. Mahlke Roger A. Bringmann John C. Gyllenhaal Wen-mei W. Hwu

Center for Reliable and High-Performance Computing
University of Illinois
Urbana, IL 61801

Abstract

Compile-time code transformations which expose instruction-level parallelism (ILP) typically take into account the constraints imposed by all execution scenarios in the program. However, there are additional opportunities to increase ILP along some execution sequences if the constraints from alternative execution sequences can be ignored. Traditionally, profile information has been used to identify important execution sequences for aggressive compiler optimization and scheduling. This paper presents a set of static program analysis heuristics used in the IMPACT compiler to identify execution sequences for aggressive optimization. We show that the static program analysis heuristics identify execution sequences without hazardous conditions that tend to prohibit compiler optimizations. As a result, the static program analysis approach often achieves optimization results comparable to profile information in spite of its inferior branch prediction accuracies. This observation makes a strong case for using static program analysis with or without profile information to facilitate aggressive compiler optimization and scheduling.

Index terms: superblock, superscalar, VLIW, optimization, code scheduling, static program analysis

1 Introduction

Compilers for superscalar and VLIW processors utilize information about the direction of conditional branches for advanced optimization and scheduling techniques. Examples of these techniques include trace scheduling [1], superblock optimization, and superblock scheduling [2]. The fundamental principle behind these techniques is to identify frequent execution sequences. Optimization and scheduling are then applied to the frequent execution sequences ignoring constraints associated with the alternative execution sequences. In this manner, more efficient schedules are realized for those frequently executed code sequences. Techniques using conditional branch direction information have been shown effective at increasing ILP, which is necessary to fully utilize superscalar and VLIW processor resources [1] [2].

Profiling is an effective means to provide the compiler with branch direction information. Profiling is the process of selecting a set of inputs for a program, executing the program with these inputs, and recording its run-time behavior. This information is used by the compiler during subsequent recompilation of the program to determine the most frequent direction of each branch. The benefit of profile information is its accuracy in predicting branch direction [3] [4]. However, there are several disadvantages to profiling. First the process of compilation, profiling, and recompilation is extremely time consuming. Second, profiling may not be applicable in all environments, such as embedded systems where gathering the profile data is usually not feasible. Finally, branch prediction data is limited to sections of the program exercised by the chosen input sets.

Through the use of profile information, it has been determined that branches typically go in one direction most of the time [4]. If that direction can be determined with reasonable accuracy without profiling, the compiler techniques utilizing this information can still provide the same benefit. An alternate approach to profile-based branch prediction is to use static analysis of the program code structure [5] [6] [7] [8]. The compiler applies heuristics to determine the most likely path of each branch. Using static program analysis for branch prediction has several advantages over profile-based branch prediction. The first is speed; there is no need to profile the program and then recompile. Also, static analysis is feasible in all environments because run-time information is not required. Finally,

static analysis is independent of the input sets, allowing 100% coverage of all branches. The primary disadvantage of static analysis based branch prediction is that it is less accurate than profile-based branch prediction.

The goal of the paper is twofold. First, we show that hazard-less paths selected by static analysis tend to be frequently executed. Thus, the benefits of aggressive optimization based on static analysis tend to approximate the benefits achieved with profile information. This will be demonstrated via superblock formation and superblock optimizations. Second, this paper points out that branch prediction accuracy is not necessarily the most meaningful metric for code optimizations that use branch direction information. The heuristics used in static program analysis tend to avoid hazardous conditions such as subroutine calls in the paths identified for aggressive optimization. As a result, the static program analysis may actually enable comparable or better optimization results in spite of its inferior prediction accuracy. This observation not only sheds new light on the value of static program analysis in the absence of profile information, it also argues for using the static program analysis in the presence of profile information. [1]

The next section motivates superblock formation using static branch analysis. Section 3 describes the static heuristics used for path selection. Section 4 describes the static analysis based superblock formation algorithm. Section 5 presents the performance results of static analysis. Concluding remarks are given in Section 6.

2 Motivating Superblock Formation Using Static Branch Analysis

The purpose of code optimization and scheduling is to minimize execution time while preserving program semantics. When this is done globally, some optimization and scheduling decisions may decrease the execution time for one control path while increasing the time for another path. By making these decisions in favor of the more profitable path, an overall performance improvement can be achieved. The superblock is the means by which the IMPACT compiler makes optimization and scheduling decisions in this manner [2].

[1] Due to space limitations, the use of static program analysis in the presence of profile information is beyond the scope of this paper. A comprehensive treatment of the subject merits a separate paper of its own.

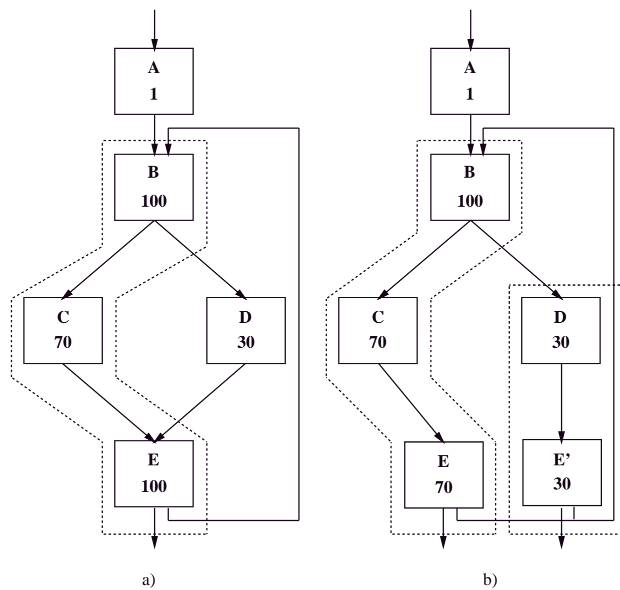

Figure 1: Superblock formation: a) trace selection, b) tail duplication.

A superblock is a block of instructions in which control may only enter at the top but may leave at one or more exit points. When the execution stream enters a superblock, it is likely that all basic blocks in that superblock will be executed. Superblock formation takes place in two steps. Traces, or sets of basic blocks which tend to execute in sequence [1], are first identified in a program using execution profile information [9]. Tail duplication is then performed to eliminate any side entrances to the trace [10].

The example shown in Figure 1a) illustrates the formation of superblocks using profile information. The execution frequencies of each block are as shown in the figure. Each conditional branch is predicted to take the path of highest execution frequency. For example, the conditional branch terminating block **B** is predicted to branch to block **C**. Thus the a trace is formed from blocks **B**, **C**, and **D**. Now, to complete the superblock formation process, tail duplication is performed on block **E** to eliminate the side entrance into the trace. The blocks **D** and **E'** may now also be combined to form a superblock; see Figure 1b).

The formation of superblocks allows optimization along one execution path while ignoring the constraints from alternative execution paths. Consider the example in Figure 2a). A load of a global variable has been added to block **B** and a hazard [2] has

[2] A hazard is an instruction or set of instructions that inhibit optimization and scheduling. A more rigorous definition will be

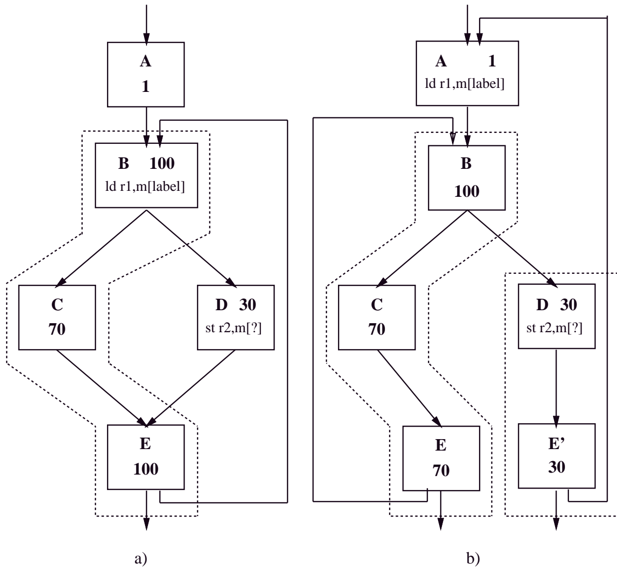

Figure 2: Profile information selects hazard free path a) superblock formation, b) optimization.

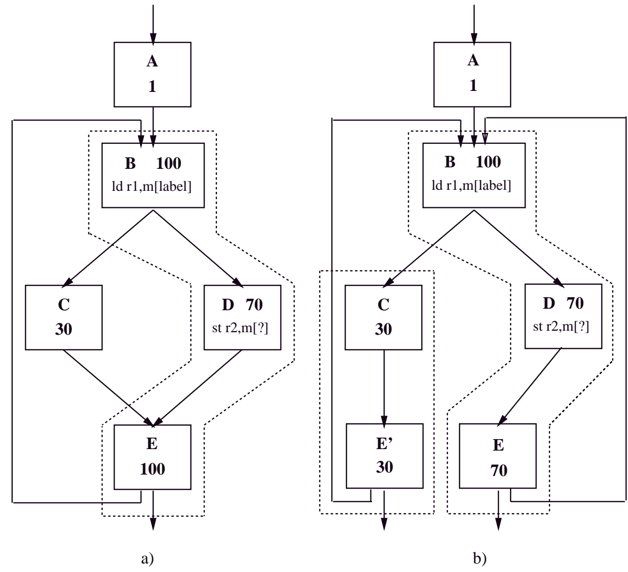

Figure 3: Profile information selects path with hazard: a) superblock formation, b) optimization.

been placed in block **D**. In this case, the hazard is an ambiguous store, i.e. a memory store whose address is not known at compile time. Consider a loop containing a load instruction and an ambiguous store, the load cannot be optimized out of the loop since it is unclear whether the load and the store access the same memory location. The ambiguous store poses a similar problem for scheduling, since the load instruction may not be speculated above the store in an attempt to improve the schedule.

Using the same superblocks formed in Figure 1b) and applying superblock loop invariant code removal [11], the load may be removed from the superblock loop **BCE** and placed in the loop preheader **A**, see Figure 2b). In this example, the superblock generated (**BCE**) with profile information avoids the hazard and allows further optimizations to be profitable. Thus, the profile information corresponds to the most profitable path for optimization.

Now, lets assume the weights for blocks **C** and **D** are interchanged as shown in Figure 3a). The profile information results in the formation of the superblock **BDE**. Unfortunately, since this superblock includes the ambiguous store, the load instruction cannot be optimized out of the loop; see Figure 3b). The use of profile information has led to the formation of a superblock that does not permit the loop invariant optimization allowed in the previous example. As this example shows, accurate branch prediction does not always lead to better optimization and scheduling opportunities.

given in Section 3

This leads us to consider a set of static branch analysis heuristics that avoid including hazardous instructions within superblocks while at the same time eliminating the overhead of profiling from the compilation process. Figure 4a) shows the superblock **BCE** formed using static analysis. The static heuristics prefer the path from **B** to **C** since it avoids the ambiguous store. As a result, the more profitable path is able to be optimized which results in more efficient code 30% of the time. This loop will execute more efficiently than the loop in Figure 3b) despite the less accurate branch prediction. Figures 3 and 4 demonstrate that by excluding these undesirable instructions from a superblock, the improved opportunities for optimization and scheduling may overcome the apparent lack of accuracy in the static heuristics.

3 Static Analysis Heuristics

Traditionally, a static branch analyzer attempts to determine the most likely direction of conditional branches. The merit of the analyzer is measured by computing the resulting branch prediction accuracy. With compiler optimization and scheduling, the primary goal of the static branch analyzer is to maximize optimization and scheduling freedom. Therefore,

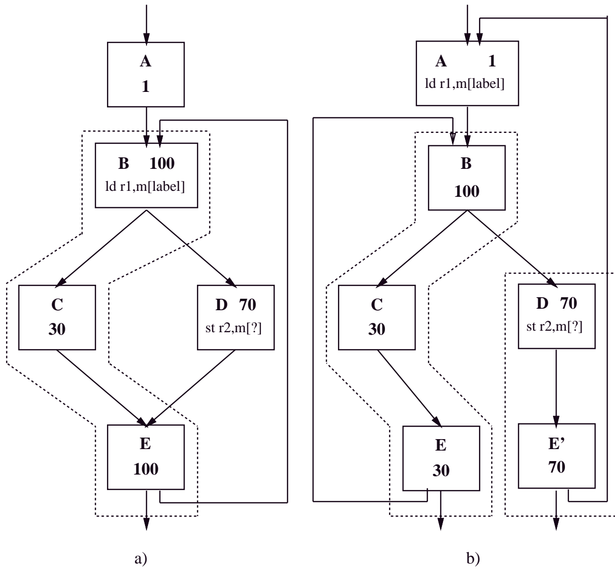

Figure 4: Static heuristics select less frequent hazard free path: a) superblock formation, b) optimization.

the heuristics and the effectiveness measures of conventional static branch analysis must be modified for compiler optimization and scheduling.

Conditional branches may be classified in two major categories, loop branches and non-loop branches. A branch is a loop branch if either of its edges is a loop back edge or a loop exit [12]. Prediction of loop branches is straight forward; loop back edges are likely and predicted taken, whereas loop exits are unlikely and are predicted not taken. This corresponds with the typical assumption of compilers that loops tend to iterate many times. Few techniques attempt to go beyond loop back-edges to find additional optimization and scheduling opportunities. Therefore, the prediction of loop branches does not significantly affect compiler optimization and scheduling.

Prediction of non-loop branches, on the other hand, is a fundamental part of many global optimization and scheduling techniques. For superblock techniques, the predictions are utilized to form superblocks. Superblock optimization and scheduling are restricted when hazardous instructions are included within a superblock. Therefore, the primary step of the static branch analyzer is to select branch directions so as to avoid hazards. The secondary step is to predict the most likely direction of the remaining branches using a set of path selection heuristics. In the remainder of this section, the specific heuristics for hazard avoidance and path selection are discussed.

3.1 Hazard Avoidance

A *hazard* is an instruction or group of instructions whose side effects may not be completely determined at compile time. Hazards force a compiler to make conservative optimization and scheduling decisions in order to ensure program correctness is maintained. As a result, suboptimal code is often produced by the compiler. Six classes of hazardous instructions have been identified:

1. I/O instructions
2. subroutine calls
3. synchronization instructions
4. ambiguous stores
5. subroutine returns
6. jumps with indirect target addresses

Classes one through four are instructions which modify part of the program state that may not be precisely determined at compile time. These hazards typically act as a barrier when included in a superblock since few instructions may be optimized or scheduled across them. For example, a memory access cannot be moved above or below a subroutine call without detailed interprocedural analysis. Similarly, a store whose address is not known at compile time (ambiguous store) prevents all optimizations of memory instructions in which the ambiguous store lies in between.

Classes 5 and 6 represent a different type of hazard which must be avoided for effective compiler optimization and scheduling. These hazards are instructions for which likely succeeding instructions may not be identified. For subroutine returns, the return point may be any of the static call sites of the function. Also, optimizing across subroutine returns requires interprocedural optimization and scheduling. Therefore, a superblock is terminated when a subroutine return is encountered. Similarly, a jump with indirect address can go to any location whose label has been used as data, so it is extremely difficult to determine a likely target with static analysis. Again, superblocks are terminated when such a jump is encountered.

The static branch analyzer predicts conditional branches so as to avoid optimization and scheduling hazards. The heuristic utilized is stated as follows: If a successor block contains a hazardous instruction or unconditionally passes control to a block containing a hazardous instruction and the successor block does not post-dominate the branch, select the other path.

3.2 Path Selection

After all branches which may be predicted using hazard avoidance heuristics are performed, path selection heuristics are used to predict the remaining branches which are not predicted. Path selection heuristics predict the direction of a conditional branch using the opcode of the branch, its operands, and/or the contents of the successor blocks. The heuristics are presented from highest to lowest priority in determining the prediction of the branch.

Pointer Heuristic. If the branch contains one or more operands which are pointers, the following heuristics are used: a pointer is not likely to be NULL and two pointers are not likely to be equal [8]. The branch direction satisfying these conditions is selected as the likely target. This heuristic is made more effective by annotating conditional branch instructions in the intermediate language of the compiler to indicate pointer operands.

Loop Heuristic. A conditional branch which either enters or avoids a loop is predicted to enter the loop [8]. Since programs tend to spend a great deal of time in loops, the intuitive prediction for a branch of this type is to enter the loop.

Opcode Heuristic. Branch prediction using the branch opcode has been frequently utilized by researchers and designers [5] [6] [7]. By performing an analysis of a set of benchmark programs for an architecture, one can determine whether a particular branch opcode is usually taken or not taken. All branches with the same opcode are then predicted using the direction indicated by the study. The major problem with opcode based prediction is that the results tend to vary greatly between benchmarks.

A more limited heuristic is used in this study to reduce the variance in prediction accuracy among benchmarks [8]. The following two opcode heuristics are applied by our branch analyzer:

1. Negative numbers are unlikely.
2. Floating point comparisons are unlikely to be equal.

The first heuristic predicts a branch target which results when all branch operands are non-negative. For example, a branch on condition $a < 0$ is predicted as not taken since "a" must be negative for the branch to be taken. This heuristic is applied for both integer and floating point comparisons. The second heuristic extends predictability of floating point branches by predicting all equal comparisons as unlikely and all not equal comparisons as likely.

```
1) bne r1,r2,label1 - Select fall thru (Guard Heuristic)
     :
2) beq r1,r2,label2 - Select taken (Store Heuristic)
     :
3) bne r1,r2,label3 - Select taken (Pointer Heuristic)
```

Figure 5: Set of branches with related operands before correction.

```
1) bne r1,r2,label1 - taken
     :
2) beq r1,r2,label2 - fall thru
     :
3) bne r1,r2,label3 - taken
```

Figure 6: Set of branches with related operands after correction.

Guard Heuristic. A branch which guards a use of one of its source operands is predicted in the direction which leads to the use [8]. The underlying assumption behind this heuristic is that the guard is detecting exception conditions and the desired path will be the one where the operand is used before being redefined. Unfortunately on many of the cases where this heuristic performs well, it is subsumed by the more accurate pointer heuristic.

Branch Direction Heuristic. The final heuristic applied by our branch analyzer selects the branch path based on the direction of the taken path. Backward branches are predicted taken and forward branches are predicted not taken. The idea is that if the taken path is backwards, then the branch is likely to be the back edge of a loop and therefore likely taken. On the other hand, if the taken path of the branch is forward the branch is likely to fall through. This heuristic is also used by several current architectures to perform dynamic branch prediction, namely the HP PA-RISC™ [13] and the DEC Alpha™ [14]. The usefulness of this heuristic for static prediction is questionable. On one hand, if the branch is backwards it is probably a loop back edge and will be predicted taken if this is the case. On the other hand, predicting forward branches as not taken, in our experience, performs poorly on integer benchmarks, thus this heuristic is used only if none of the previous heuristics apply.

Related Branches. After all path selection heuristics are applied, the predicted direction of each branch is made consistent with the predicted directions of any related branches. Branches are related if they have the same operands. Given a set of related branches, all branches are made consistent with the strongest individual prediction.

```
trace_formation()
{
        perform loop detection
        sort by loop nesting level
        for each loop {
                create breadth first list of loop blocks
                for each unvisited block {
                        grow_trace( block )
                }
        }
        create breadth first list of function blocks
        for each unvisited block
                grow_trace( block )
}
```

Figure 7: Trace formation algorithm.

```
grow_trace( seed_block )
{
        trace = { seed_block }
        current_block = seed_block
        while ( 1 ) {
                mark current_block visited
                if current_block contains indirect jump
                        break;
                if current_block contains subroutine return
                        break;
                likely_block = predicted target
                if likely_block visited
                        break;
                /* loop back-edge */
                if likely_block dominates current_block
                        break;
                trace = trace ∪ likely_block
                current_block = likely_block
        }
}
```

Figure 8: Trace growth algorithm.

The heuristic is illustrated with the example in Figure 5. The individual heuristic prediction of each of these branches is as shown in the figure. Branch **3)** has the strongest individual prediction, pointer heuristic and the analyzer makes the assumption that **r1** and **r2** are very unlikely to be equal. Thus the predictions branches **1)** and **2)** are likely to be incorrect and are corrected. The final predicted direction of each branch is shown in Figure 6. The performance of this heuristic is heavily dependent on the strongest individual heuristic prediction in the group of related branches.

4 Static Analysis Based Superblock Formation

Traditionally superblocks were formed by using profile information to select a trace in both the forward and backward direction from a particular basic block [9]. Without profile information the compiler no longer has detailed information regarding the execution characteristics of the function. The previous section discussed the static heuristic used to determine branch paths. However, these heuristics only provide information in the forward direction.

After the branch paths have been identified, superblock formation is performed. As discussed in Section 2, superblock formation takes place in two steps: trace formation and tail duplication. The trace formation algorithm, shown in Figure 7, begins by performing loop analysis to generate static instruction execution frequencies. The loops are processed starting from the innermost nesting level, the idea being that we want to form traces in the most frequently executed portions of the code first. The header of the loop is chosen as a seed and a trace is grown using the algorithm shown in Figure 8. The trace growing algorithm adds the most desirable destination block of the branch as indicated by the static analysis heuristics. Blocks are added to the trace until one of the next four situations arise:

- The current block contains an indirect jump.
- The current block contains a subroutine return.
- The next block has already been visited.
- The next block dominates the current block and therefore is a loop back-edge.

When growth of a trace is terminated, another seed block is chosen from a breadth first list of the loop blocks. The process continues until each basic block within the loop belongs to a trace. After traces have been formed within each loop the process continues with the remaining unvisited blocks in the function in breadth first order. Once the traces within the program have been identified, IMPACT's standard tail duplication algorithm is used to form the superblocks.

Benchmark	Benchmark Description
cccp	GNU C preprocessor
cmp	compare files
compress	compress files
eqn	format math formulas for troff
eqntott	boolean equation minimization
espresso	truth table minimization
grep	string search
lex	lexical analyzer generator
li	lisp interpreter
qsort	quick sort
tbl	format tables for troff
sc	spreadsheet
wc	word count
yacc	parser generator

Table 1: Benchmarks.

5 Experimental Evaluation

In this section the effectiveness of superblock formation using static analysis methods is evaluated. Two important issues are shown. First, hazardless paths selected by static analysis tend to be frequently executed. Second, branch prediction accuracy does not necessarily correspond to optimization quality and that it is possible for static analysis based superblock formation to achieve comparable performance to profile-based support block formation, even with the presence of less accurate branch prediction.

5.1 Methodology

The branch analyzer has been implemented within the IMPACT-I C compiler. The IMPACT-I compiler is a prototype optimizing compiler designed to generate efficient code for VLIW and superscalar processors [15]. The benchmarks used in this study consist of the 14 non-numeric programs in Table 1. The programs consist of 5 non-numeric programs from the SPECint92 suite and 9 other commonly used non-numeric programs.

The processor model used in this study is an in-order issue superscalar with register interlocking. The processor is assumed to have uniform function units, 64 integer and 64 floating-point registers, and 1 branch delay slot. The instruction set is based on the instruction set of the HP PA-RISC processor, and the instruction latencies assumed are those of the HP PA-RISC 7100 (see Table 2). For each machine configuration, the program execution times, assuming an ideal cache, are derived from execution driven simulations of the benchmarks in Table 1. For the experiments, the issue rate of the processor is varied from 1 to 8 and the base

Function	Latency	Function	Latency
Int ALU	1	FP ALU	2
memory load	2	FP multiply	2
memory store	1	FP divide(SGL)	8
branch	1 / 1 slot	FP divide(DBL)	15

Table 2: Instruction latencies.

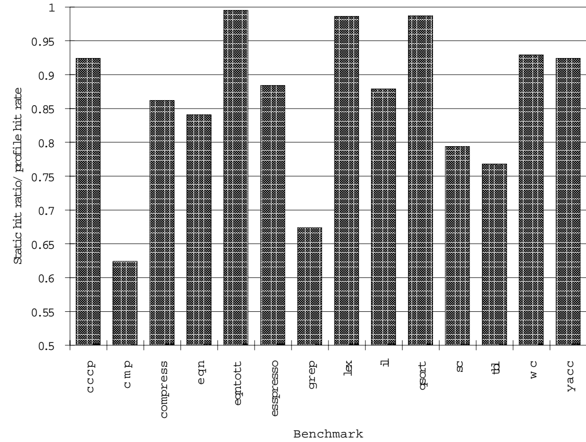

Figure 9: Branch Prediction Accuracy

for all speedup calculations is the result of basic block scheduling on a single issue processor.

5.2 Results

5.2.1 Prediction Accuracy

The performance of the static analysis based branch heuristics discussed in Section 3 are shown in Figure 9. The results in Figure 9 are shown as a percentage of profile-based prediction hit rate for all conditional branches, both loop and non-loop branches. This is calculated by summing the number of times each branch takes the path predicted by our analyzer divided by the sum of the number of times each branch takes the path predicted by the profile information. A ratio of one does not imply perfect branch prediction, it indicates that our branch analyzer selected the same direction that the profile-based predictor selected for every branch. Overall, our branch analyzer agrees with profile-based branch predication approximately 86% of the time. This illustrates that hazard free paths tend to correspond to the paths selected by profile information.

5.2.2 Superblock Performance

The performance of static analysis based and profile-based superblock formation and optimizations for a

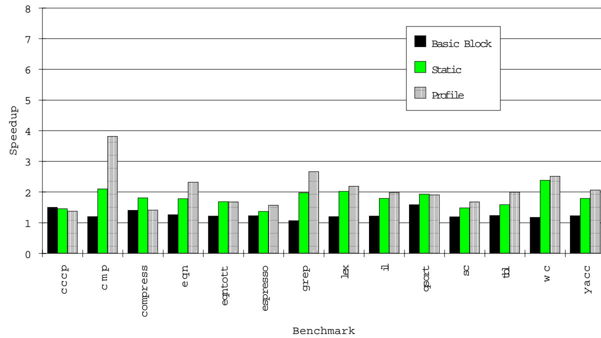

Figure 10: 2-issue speedup for basic block scheduling, static analysis based superblock formation, and profile based superblock formation

2-issue processor is compared in Figure 10. Static analysis based superblock formation and optimizations achieve comparable performance to profile-based methods for most benchmarks. This confirms the conclusion made in Section 5.2.1 that hazard free paths tend to be frequently executed.

The benchmarks **compress**, **eqntott**, and **qsort** actually performed better than their profile-based counterparts. Choosing hazard free paths to form superblocks exposes more optimization and scheduling opportunities than simply selecting the most likely paths. The behavior is illustrated in Figures 3 and 4 in Section 2. The benefits of hazard avoidance are clearly visible for these benchmarks due to the high effectiveness of the path selection heuristics (94% accuracy compared with profile). For other benchmarks, the benefits of chosing hazard free paths were overshadowed by the poor performance of the path selection heuristics compared with the profile-based method. This is clearly visible in the benchmark **cmp** due to the two frequently executed branches that were mispredicted by the path selection heuristics. These mispredicted branches negated any possible benefits of the hazard avoidance heuristics. This motivates the use of hazard avoidance heuristics in concert with profile information for path selection.

Figure 11 compares the performance of a 4-issue processor. As in the 2-issue case, static program analysis has comparable performance to the profile based method. Notice, that **eqntott** and **compress** still out perform profile-based methods. Also, the performance difference for **cmp** between static analysis and profile-based is widening. One final note, several of the benchmarks achieve super-linear speedup over the base processor in the 2-issue and 4-issue case. This occurs because the base processor only executes traditionally optimized code.

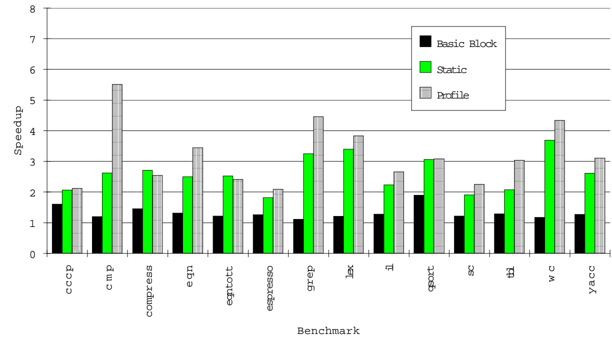

Figure 11: 4-issue speedup for basic block scheduling, static analysis based superblock formation, and profile based superblock formation

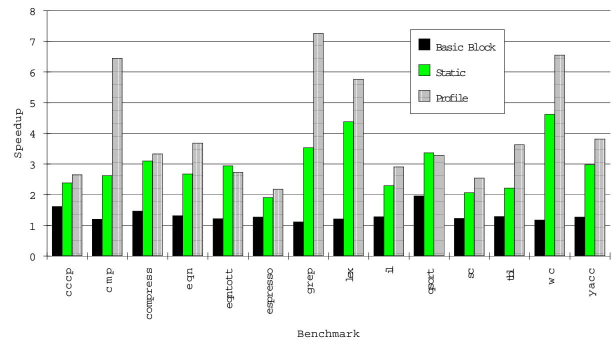

Figure 12: 8-issue speedup for basic block scheduling, static analysis based superblock formation, and profile based superblock formation

The performance comparison for 8-issue is shown in Figure 12. Profile-based superblock formation now has a much larger advantage over the static analysis based approach for **cmp**, **grep**, **lex**, and **wc**. This indicates that either better static analysis based branch prediction heuristics are needed or that profile information is necessary to adequately take advantage of the large number of available resources in an 8-issue machine.

6 Concluding Remarks

We have implemented a set of static program analysis heuristics within the IMPACT compiler to determine important execution sequences in the absence of profile information. These heuristics try to avoid hazardous conditions such as subroutine calls in the paths identified for aggressive optimizations. We have shown that static program analysis heuristics can facilitate

optimizations and scheduling to achieve results comparable to profiling. Also, we point out that branch prediction accuracy is not necessarily the most meaningful metric of branch handling for the compiler optimizer.

Currently, we are conducting detailed performance studies to further understand and improve the static analysis heuristics. In the future, we would like to investigate tradeoffs involved in combining profile information with static program analysis. Furthermore, we would like to study the effectiveness of static program heuristics in the context of predicated compilation where many of the hard to predict branches can be eliminated through predicated execution.

Acknowledgements

The authors would like to thank all members of the IMPACT research group for their comments and suggestions. This research has been supported by the National Science Foundation (NSF) under grant MIP-9308013, Joint Services Engineering Programs (JSEP) under Contract N00014-90-J-1270, Intel Corporation, the AMD 29K Advanced Processor Development Division, Hewlett-Packard, SUN Microsystems, NCR and the National Aeronautics and Space Administration (NASA) under Contract NASA NAG 1-613 in cooperation with the Illinois Computer laboratory for Aerospace Systems and Software (ICLASS).

References

[1] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Transactions on Computers*, vol. c-30, pp. 478–490, July 1981.

[2] W. W. Hwu *et al.*, "The Superblock: An effective structure for VLIW and superscalar compilation," *Journal of Supercomputing*, pp. 229–248, July 1993.

[3] W. W. Hwu, T. M. Conte, and P. P. Chang, "Comparing software and hardware schemes for reducing the cost of branches," in *Proceedings of the 16th International Symposium on Computer Architecture*, pp. 224–233, May 1989.

[4] J. A. Fisher and S. M. Freudenberger, "Predicting conditional branch directions from previous runs of a program," in *Proceedings of the 5rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 85–95, October 1992.

[5] J. E. Smith, "A study of branch prediction strategies," in *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pp. 135–148, May 1981.

[6] J. K. F. Lee and A. J. Smith, "Branch prediction strategies and branch target buffer design," *IEEE Computer*, January 1984.

[7] S. Bandyopadhyay, V. S. Begwani, and R. B. Murray, "Compiling for the CRISP microprocessor," in *Proceedings of IEEE COMPCON 1987*, pp. 96–105, February 1987.

[8] T. Ball and J. R. Larus, "Branch prediction for free," in *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pp. 300–313, June 1993.

[9] P. P. Chang and W. W. Hwu, "Trace selection for compiling large C application programs to microcode," in *Proceedings of the 21st International Workshop on Microprogramming and Microarchitecture*, pp. 188–198, November 1988.

[10] P. P. Chang, S. A. Mahlke, and W. W. Hwu, "Using profile information to assist classic code optimizations," *Software Practice and Experience*, vol. 21, pp. 1301–1321, December 1991.

[11] S. A. Mahlke, W. Y. Chen, J. C. Gyllenhaal, W. W. Hwu, P. P. Chang, and T. Kiyohara, "Compiler code transformations for superscalar-based high-performance systems," in *Proceedings of Supercomputing '92*, pp. 808–817, November 1992.

[12] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools.* Reading, MA: Addison-Wesley, 1988.

[13] Hewlett-Packard Co., *PA-RISC 1.1 Architecture and Instruction Set Reference Manual.* Cupertino, CA: Hewlett-Packard Co., 1990.

[14] Digital Equipment Corporation, *Alpha Architecture Handbook.* Maynard, MA: Digital Equipment Corporation, 1992.

[15] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu, "IMPACT: An architectural framework for multiple-instruction-issue processors," in *Proceedings of the 18th International Symposium on Computer Architecture*, pp. 266–275, May 1991.