University of Alberta

Library Release Form

Name of Author: Paul Normand James Berube

Title of Thesis: *Aestimo*: A Feedback-Directed Optimization Evaluation Tool

Degree: Master of Science

Year this Degree Granted: 2005

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as herein before provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

Paul Normand James Berube
10821 33A Avenue NW
Edmonton, AB
Canada, T6J 3C2

Date: ____________________

University of Alberta

Aestimo: A FEEDBACK-DIRECTED OPTIMIZATION EVALUATION TOOL

by

Paul Normand James Berube

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Fall 2005

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled *Aestimo*: **A Feedback-Directed Optimization Evaluation Tool** submitted by Paul Normand James Berube in partial fulfillment of the requirements for the degree of **Master of Science**.

Dr. José Nelson Amaral
Supervisor

Dr. Robert Holte

Dr. Bruce Cockburn
External Examiner

Date: ____________________

Abstract

Feedback-directed optimization (FDO) is a compiler technique that enhances the ability of a compiler to make good optimization decisions. A training run provides the compiler with a profile that summarizes the run-time behavior of the program. Most studies that use FDO techniques use either a single input for both training and performance evaluation, or a single input for training and a single input for evaluation. However, the run-time behavior of a program is influenced by the data it is processing. Benchmark creators and compiler designers rely on the assumption that selecting a "representative" training input will result in effective FDO.

This exploratory study addresses an important open question: How important is the selection of training data for FDO? Likely, the answer to this question is not constant across all optimizations that use profile information. How sensitive are individual compiler transformations to the selection of training data used with FDO? Does training on different inputs result in different optimization decisions at compile time? Furthermore, do these different decisions result in changes in program performance?

This thesis introduces *Aestimo*, a tool developed to quantify the differences between FDO logs for inlining and `if` conversion from the Open Research Compiler (ORC) for SPEC CINT2000 benchmark programs trained on a large number of inputs. *Aestimo* also compares the performance of programs trained on different inputs, and the performance of programs compiled with and without FDO.

Training on different inputs does lead to different optimization decisions and different levels of program performance in most cases. Training on different inputs results in as much as a 5% difference in performance with `if` conversion, and in as much as a 6% difference in performance with inlining, on a workload of inputs. Also, evaluating FDO performance on different inputs can lead to substantially different performance results. *Aestimo* finds differences in best-case FDO performance on different inputs for the same program larger than 13% for `if` conversion, and larger than 20% for inlining. Finally, *Aestimo* reveals that the current `if`-conversion heuristics in the ORC always results in performance degradation for the Itanium 2 processor when FDO is used.

Acknowledgements

I would like to thank my supervisor, Dr. Amaral, for his guidance and assistance during my studies. Also, several people provided assistance and additional inputs for benchmark programs: Martin Schoenert, Steve Linton, and Alexander Hulpke for `GAP`, Robert M. Hyatt for `crafty`, and David Temperley for `parser`. Thanks to Reinhold Weicker and Kaivalya Dixit from SPEC for our conversations about the procedures and guidelines for selecting the ref and train inputs for SPEC benchmarks. Also, I am indebted to George Hicks and Henryk Modzelewski for providing me with access to the Itanium machines in the Monster cluster at the University of British Columbia.

I would also like to thank the members of my examining committee for their insightful questions regarding my work, and for their suggestions for improving this thesis.

Finally, I want to thank my Mom and Dad for supporting me and putting up with me over the last two years, and for the last months in particular. I couldn't have done it without you.

This work was supported by a Post-Graduate Scholarship from the Natural Sciences and Engineering Research Council of Canada (NSERC).

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Traditionally, programs are compiled *statically*, that is, without any information beyond what the compiler can extract from the source code. When static optimization is used, the compiler must use heuristics to guess which are the important, frequently executed sections of the code and which are infrequently or never-executed sections of the code, such as initialization routines and error handlers. This situation is problematic, as many optimizations attempt to make the frequent case fast, often at the expense of less-frequently-executed sections of code. Therefore, static optimization must be conservative in cases where the runtime behavior of the program cannot be confidently predicted at compile time.

Feedback-directed optimization (FDO), also known as profile-guided optimization, is traditionally a compiler technique that enhances the ability of a compiler to make good optimization decisions [12]. In a very general sense, FDO can be considered to be a spectrum of performance-enhancing techniques that rely on measurements of run-time program behavior [36]. This spectrum includes a large variety of methods to enhance program performance, including: a developer manually tweaking program code, hardware mechanisms such as branch predictors, and run-time program optimizations such as just-in-time compilation of Java bytecode to native assembly code. However, this thesis uses a much narrower, traditional definition of FDO.

When traditional FDO is used, several additional steps are required during the compilation process. First, the program is compiled with additional instrumentation code to record statistics about run-time program behavior to a file. Then, this *instrumented binary* is run on a *training input* to generate a file containing run-time program statistics, which is called a *profile*. Finally, the program is recompiled. The compiler reads the profile file and replaces its static estimates of program behavior with the values recorded in the profile. Usually an internal compiler variable is set to tell optimizations that profile information has replaced the static estimates.

Because FDO requires multiple compilations of the same program, it is important to distinguish between a *program*, which is the algorithm encoded in the source code, and a *binary*, which is a particular compiled version of a program. When any of the inputs to the compilation process are changed (*e.g.*, compiler, command-line parameters, profile information, target architecture, or

```
data = getDataBlock(blockNum);
icrc = integrityCheck(data);
if(icrc == DATA_OK) {
  // ...  some preparation code ...
  processData(data, blockNum);
  // ...  some finalization/cleanup code ...
} else {
  // ...  log the error ...
  // ...  initialize recovery ...
  reTry(blockNum);
}
```

Figure 1.1: A motivating example for FDO

source code), a different binary is produced. Thus, compiling a program using FDO and training on one input will result in one binary, but training on a different input will result in a different binary.

Consider the code fragment in Figure 1.1. Statically, a compiler might consider both branches of the `if` equally likely. In that case, the true branch will probably not be optimized if it would reduce performance on the false branch. Should inlining of `processData(data, blockNum)` or `reTry(blockNum)`, or both, be performed? To limit code growth, only a frequently executed function call should be inlined, but which branch is more frequently executed?

Alternately, some compilers perform additional branch analysis [6]. Since error codes are conventionally represented by negative integers, the test against `DATA_OK` (which is presumably a non-negative constant) could be correctly identified as checking for an error condition. In this case, the compiler assumes that an error is an infrequent exception, and optimizes the true path. If this code is acquiring data from a reliable source, such as a hard drive or a wired network connection, error rates would be very low and the false branch would almost never be executed. On the other hand, if the data comes from an unreliable source, such as a noisy wireless connection, then the false branch may execute very frequently. By recording statistics during the execution of the program running on real data, FDO provides more accurate information to the compiler to allow for better code generation in such cases.

Most studies that use FDO techniques use either a single input for both training and performance evaluation, or a single input for training and a single input for evaluation [11, 17, 18, 37, 28, 25, 14, 33, 16, 40]. This is not a wise practice because the run-time behavior of a program is influenced by the data it is processing. Few studies have investigated the impact of the training input used in FDO on the performance of the resulting binary, either on an individual input or on a workload of inputs. Instead, both benchmark creators and compiler designers rely on the assumption that selecting a single "representative" training input will result in effective FDO. The tasks of defining what representative means and of selecting some input that meets this definition are typically left to the benchmark creator, who is usually familiar with the program.

There are several problems with this approach. First, most compiler users will likely be less successful than a benchmark designer at selecting a representative training input when they use FDO on a non-benchmark program. Second, there are several possible definitions of a representative input. Is a representative input representative of a typical workload of inputs to the program, or is it representative of the input that will be used for performance evaluation? In the latter case, should the training input be distinct from the evaluation input? Should it be a subset of the evaluation input? Or, should it be a mix of those two options?

While it may seem that one solution is obviously correct, there are competing schools of thought on the issue [38]. On one side of the issue are those who believe that including any portion of the evaluation input in the training input represents an unrealistic scenario. A program would rarely be run on the same data twice, since the results of the first computation could be stored and reused directly. Including evaluation data in the training input thus provides the compiler with more accurate data than would be available in a production environment, and may exaggerate the performance benefits of FDO.

On the other hand, some benchmark designers point out that including a portion of the evaluation data in the training data is an easy way to ensure that the training data is representative of a real workload. They argue that since a large portion of the evaluation data is not used for training, the characteristics of that portion of the data could vary substantially from the data used for training. This would counteract any possible impact of providing the compiler with artificially accurate profile information. Furthermore, they argue that there are several classes of programs where it is perfectly reasonable to select a subset of the actual data as the training set in a production environment. Data is frequently organized as records, which are processed independently. Selecting a sample of records from the full data set is a natural and easy method to create a representative training data set.

At this time, there are no regulations for the SPEC benchmarks [19] to specify whether training data should or should not include data from the reference input set. In fact, there are examples of both situations in the benchmarks used in this study.

Therefore, an important question remains open: How important is the selection of training data for FDO? It is likely that the answer to this question is not constant across all optimizations that use profile information. Therefore, a more appropriate question is: How sensitive are individual compiler transformations to the selection of training data used with FDO?

This large question should be decomposed into more manageable parts. First, does the selection of training data change the optimization decisions that are made during compilation? For example, does the selection of a different training input change which callsites are inlined in a program? If the answer to this question is "no," then the task is complete: Input selection is irrelevant for feedback-directed optimization. More likely, however, different optimizations applied to different programs exhibit varied measures of input selection sensitivity.

Even if different optimization decisions are made, these differences might not be significant. Thus, an important second question is: Do the differences in optimizations decisions result in different levels of performance? If training on different inputs results in significantly different levels of performance, then input selection for FDO is an important issue.

These questions will not be easily answered. Furthermore, the answers will likely vary depending on the selection of compiler and architecture investigated. This thesis reports the results of an initial exploratory investigation that provides the following contributions:

- Defines two metrics to quantify differences in optimization decisions.
- Introduces an experimental methodology to investigate the impact of input selection on a single optimization.
- Performs an extensive experimental study using the SPEC CINT2000 benchmarks with a large number of additional program inputs to investigate the feedback-directed `if` conversion and inlining optimizations in the Open Research Compiler (ORC) for the IA-64 family of processors.
- Determines that training input selection does impact the optimization decisions made during FDO compilation.
- Observes that training input selection often has a significant impact on program performance, both on a workload of inputs and on individual inputs.
- Confirms that FDO has the potential to significantly improve program performance, and determines that this is usually the case with inlining.
- Demonstrates that feedback-directed `if` conversion in the ORC usually reduces program performance.
- Confirms that using the same input for both training and evaluation usually leads to the best performance results.

Chapter 2 provides additional background information about FDO and the ORC infrastructure. Chapter 3 describes the experimental setup, and defines the metrics used to measure profile differences. The results of an experimental study are presented in Chapter 4. Related work is discussed in Chapter 5. Chapter 6 identifies future work and concludes.

Chapter 2

Background

2.1 Profiling and Feedback-Directed Optimization

Feedback-directed optimization uses a program execution profile to determine which portions of the code are frequently executed and how control flows through the program at run time. This information is useful to optimize code that contains control flow such as `if` statements. On the other hand, control flow due to loops does not benefit from profile information because loop behavior is easily predicted at compile time. Moreover, optimizing loop code is virtually always beneficial. In fact, the Open Research Compiler (ORC), used in this study, includes loop frequency counts in its profile information but ignores this information when performing loop optimizations.

Ball and Larus show how to place counters to capture the frequency of each branch in a program with a minimum number or counters [7]. They also show that simply counting branch frequencies is insufficient to correctly identify the most frequently taken path through a section of code. They then present an efficient instrumentation technique to capture the frequency of each execution path through a function [8].

Despite the existence of these profiling techniques, the ORC inserts counters to record the frequencies of every branch in a program. The ORC does not implement path profiling.

2.2 Compiler Infrastructure

The Open Research Compiler (ORC) is an open-source compiler [1]. The principal contributors to the development of the ORC are Intel and the Chinese Academy of Sciences. The ORC is based on the code base of SGI's Pro64 compiler [5], which was released as the open-source Open64 compiler [2] in 2001. The ORC focuses on producing high-performance code, and is frequently used for compiler research. To support this aim, the ORC has a rich profiler to support its FDO infrastructure that provides, among other things:

- Dynamic instruction counts for each function
- Invocation count of each function

- Taken and not-taken frequency counts for each branch
- Loop statistics
- `Switch-case` case frequencies
- `Call` and `return` frequencies for each callsite
- Stride profiles
- Value profiles

The IA-64 processor family is the only target for the ORC. Consequently, the ORC combines a mature code base with state-of-the-art compiler technology tuned for Itanium processors. When a 3-stage FDO compilation process is used, the performance of the ORC 2.1 on the SPEC CINT2000 benchmarks is within 5% of Intel's ECC 7.0 compiler, and exceeds the performance of GCC 3.1 [4, 3]. This study uses the latest release of the ORC, version 2.1.

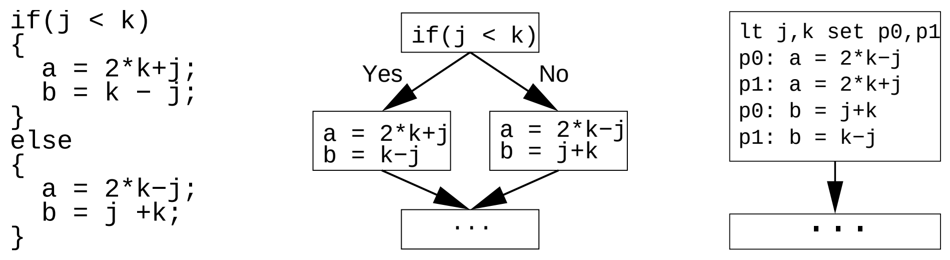
This thesis investigates two optimizations that make use of the frequency information provided by profiling: `if` conversion and function inlining. The code base of the ORC is roughly 130MB, spread across nearly 8500 files and 267 directories. Thus, locating, understanding, and correctly instrumenting an optimization has the potential to be a very involved task. This task is made more involved by the scarcity of detailed documentation for the compiler. `If` conversion was selected because (1) it was moderately easily located in the source code, (2) it is contained in a small number of source files, and (3) it is easily instrumented to output and use the optimization logs required for the study. Inlining was selected because it is an optimization known to have a significant impact on performance. Furthermore, inlining provides a natural starting point for the investigation because the facilities to output and use the inlining log were pre-existing in the ORC.

2.2.1 `If` conversion

`If` conversion is a program transformation that attempts to reduce branch misprediction penalties and hazards that arise in code with control flow. Furthermore, as a side effect of eliminating branches, `if` conversion can increase the amount of Instruction Level Parallelism (ILP) in program code and allow greater flexibility for instruction scheduling. Both these properties are important for EPIC architectures such as the Itanium[1] and the Itanium 2, as discussed in Section 3.3. In addition, `if` conversion can enhance the performance improvements gained by software pipelining loops.

In order to execute `if`-converted code, an architecture must support predicated instructions. A predicate is a special-purpose single-bit register, *p0, p1, etc.*. Predicates can be set or cleared by the results of comparisons, or can be calculated from other predicate values. A predicated instruction is a normal machine instruction, prefaced by a reference to a predicate register. If the bit in that

[1]Itanium and Itanium 2 are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.



```
if(j < k)
{
  a = 2*k+j;
  b = k - j;
}
else
{
  a = 2*k-j;
  b = j +k;
}
```

```
lt j,k set p0,p1
p0: a = 2*k-j
p1: a = 2*k+j
p0: b = j+k
p1: b = k-j
```

(a) Original code (b) Original CFG (c) `If converted` CFG

Figure 2.1: High-level example of `if` conversion

predicate register is on, or the predicate is true, then the results of the instruction are committed; otherwise the result of any computation is discarded and does not change any state in the machine.

Figure 2.1(a) shows a simple branch. Figure 2.1(b) shows the same branch as a control flow graph (CFG). A CFG is composed of *basic blocks* (BBs). A basic block is a single-entry single-exit sequence of instructions where execution can only start with the first instruction in the sequence. Moreover, if the first instruction is executed, then every instruction in the BB must be executed in order. Consequently, the first instruction in a BB must be either the first instruction in a function, or the target of a branch instruction. Either the last instruction of a BB is a branch or a return instruction, or the next instruction after the BB is the target of a branch. Every branch is the last instruction of some BB.

Since `if` conversion changes branches into predicate calculations, `if` conversion allows BBs to be merged together. In Figure 2.1(c), the `p1` predicate is set to 1 and the `p0` predicate is set to 0 if `j` is less than `k`. Otherwise, when the result of the test is false, the values assigned to the predicates are reversed. The instructions on the `Yes` path are guarded by the `p1` predicate, and the instructions on the `No` path are guarded by the `p0` predicate. Then, the instructions from both branches can be merged into the BB that contained the test before `if` conversion. The instructions from the two paths can be intermingled arbitrarily, and can be scheduled anywhere in the new BB after the instruction that computes the predicate values.

When the code is not `if`-converted, the direction of the branch determines whether the instructions on the `if` or `else` path should be executed. Either set of instructions may enter the execution pipeline, but not both. If the processor mispredicts the branch, then the wrong instructions will be fetched and put into the pipeline. Subsequently, the pipeline will be flushed, and execution will restart with instructions from the correct path. Many branches are easily predicted. For example, the exit test at the beginning of a loop is only taken once in each loop execution, but is not taken for every iteration of the loop. However, some branches are inherently difficult to predict [6], [23] (pp. 313-314), and thus benefit most from `if` conversion.

On the other hand, if the code is `if converted`, then all the instructions from both sides of the branch are fetched and enter the execution pipeline. All of these instructions do consume processing resources, though the expectation is that the processor would otherwise have idle functional units. In exchange, there is no danger of a branch misprediction since the branch has been eliminated. Predicates are computed in time to determine which instructions should be committed and which should be discarded without delaying execution.

When making an `if`-conversion decision, the branch is first checked to ensure that `if` conversion is legal. Then, the execution times for both the predicated and non-predicated versions of the code are estimated to determine the profitability of the transformation. These estimates are based on the following factors:

1. **Taken vs. Not-Taken Time:** If the code for one side of the branch is much longer than the other, if-conversion will delay the execution of the shorter path. The execution time for each path is estimated statically.

2. **Resource Use:** If `if` conversion would lead to stalls due to insufficient processor resources, it may not be beneficial. Resource use is estimated statically.

3. **Branch Probability:** The probability that the branch is taken is used to estimate the branch misprediction cost and to weigh the above characteristics when estimating execution times. The branch probability is taken from profile information if available, or estimated based on the type of branch otherwise.

If the average estimated execution time is reduced by `if` conversion, the transformation is performed. The transformed region may be part of a path from another branch, and may become part of a larger predicated region if additional `if` conversion is performed.

Hyperblocks are single-entry multiple-exit scheduling regions that rely on `if` conversion to remove control flow within a region. Hyperblocks were introduced by Mahlke [31]. He found, through simulation, that they could provide on average a 3-fold speedup for a collection of programs on a hypothetical EPIC processor capable of issuing 4 instructions per cycle and implementing full support for predicated execution. These simulations provided incentive for the design of hardware implementations of similar processors, such as the Itanium. However, later studies using the Itanium revealed that the performance benefits of `if` conversion on this architecture are fairly small. In particular, Choi *et al.* concluded that the performance benefits of `if` conversion due to reduced branch misprediction for the SPEC CINT2000 benchmarks on the Itanium are upper-bounded at about 2-3% [13].

The ORC contains algorithms to produce predicated code using either hyperblocks (path-based predication) or `if` conversion (individual branch-based predication). `If` conversion is used by default unless hyperblocks are explicitly selected by a command line option. We expect that the compiler designers had good reasons to prefer `if` conversion over hyperblocks, and therefore perform our study using `if` conversion.

2.2.2 Inlining

Function inlining, or simply inlining, is a common optimization that results in significant performance gains. Inlining replaces a function call with the code of the function that would ordinarily be called. The function call is referred to as the *callsite*, the function that contains the callsite as the *caller*, and the function that would be called as the *callee*. There may be multiple callsites for the same callee, and each is treated individually.

Inlining improves performance in several ways. Most obviously, inlining a function removes the function call and the associated overhead of pushing arguments onto the stack and saving and restoring registers. Also, since the function code is included in the body of the caller, locality in the instruction stream can be improved. Most importantly, other optimizations have the potential to be more effective. For example, optimizations such as dead code elimination, constant propagation, and redundant subexpression elimination can propagate changes into the callee code without the requirement to maintain the generality of the original function. Consider a function that does additional or alternate processing if a parameter is true or false. If the compiler can prove that the parameter is always set to true (*e.g.*, it is hard coded to true) at a particular callsite, then the test on that parameter and the non-taken branch can be eliminated from the inlined code.

However, excessive inlining can degrade performance. Inlining increases code size, and can reduce instruction cache performance. Furthermore, larger functions require more time to optimize. This is particularly problematic since several static optimizations have super-linear compile times. To prevent excessive compile times, many optimizations are guarded by timers that abort optimization after an extended period of time. Therefore, excessive code growth can limit the effectiveness of more expensive optimizations. Finally, the inlined code introduces more variables and temporary values that increase register pressure. If these values do not fit in the register file, then additional register spills and restores are needed.

There are many factors that determine if inlining is performed at a callsite. The main intuition for the majority of the filters that control inlining is that the callsite should be frequently executed to maximize the benefits of inlining, and both the caller and the callee should be small to avoid the negative effects of code size expansion. In the ORC, the compiled size of a function is estimated from higher-level representations according to the formula:

$$size = StatementCount + CallCount + 5 * BasicBlockCount$$

Each statement results in one or more machine instruction. Each function call requires code to push arguments on the stack and a call instruction. A basic block is a section of code with a single entry point at the first instruction, and no exits until the last instruction, which may be a branch or a return instruction. Since many basic blocks are small, five instructions per basic block is a reasonable rough estimate.

Inlining in the ORC uses a *temperature* heuristic, which is augmented by Zhao's *adaptive in-*

lining and *cycle_density* heuristics [40]. Temperature measures the expected benefit of inlining, and can roughly be explained as the ratio of the contribution of a callsite to the execution time of the program compared to the proportion of the size of the callee to the size of the entire program. A *hot* callsite is one that accounts for a large amount of program execution time from a small callee. Therefore, the hotter a callsite, the more benefit is expected from inlining that callsite.

Inlining is performed if the temperature of a callsite exceeds a threshold. Adaptive inlining allows the threshold to vary depending on the program size. Small programs benefit from a lower threshold and more aggressive inlining, while larger programs require a higher threshold to prevent excessive inlining. Applications are categorized as Large, Median, or Small, and the temperature threshold is adjusted accordingly.

Callsites may account for a large proportion of execution time due to frequent execution or due to high trip-count loops inside the callee. The temperature heuristic is not effective at distinguishing these two cases. However, inlining will only be effective at enhancing performance in the case where the call is made frequently. A high trip-count loop can be optimized effectively without inlining, but inlining it will likely produce the negative effects described above. The cycle_density ratio identifies these *heavy functions* by comparing the amount of execution time spent in the function to the number of times the function was called. Only those callees with a low cycle_density should be inlined.

Zhao shows that the addition of adaptive inlining improves performance on the SPEC CINT2000 benchmarks by more than 5% compared to temperature alone. Also, while cycle_density has little impact on performance, it reduces code bloat by as much as 27% by preventing the inlining of a small number of infrequently called functions. These experiments were performed on an Itanium processor, with FDO training on the SPEC training inputs, and evaluated on the SPEC reference inputs.

As discussed above, the ORC's inlining heuristics rely heavily on the frequency of execution of each callsite, the execution frequency of each function, and the number of cycles spent in each function. While these measures can be estimated statically, they can be much more accurately determined by collecting profile information. Loop optimization classically assumes that each loop iterates 10 times. There are standard expectations of branch probabilities for various classes of branches. These estimates can be used to generate estimates for the quantities used by the inlining heuristics. However, profile information is very valuable for calculating heuristic values. In particular, profile information can provide much better measures of loop trip counts for use with the cycle_density heuristic, and will result in much more accurate temperature values. Therefore, inlining should be more effective when FDO is used.

Chapter 3

Experimental Setup

3.1 Metrics

This thesis addresses two primary questions: (1) does profiling on different training inputs result in different optimization decisions in the compiler? and (2) do these modified decisions significantly affect program performance? The latter question can be answered by experimentation, and will be dealt with in Chapter 4. This section addresses the first question. It develops methods to quantitatively measure the differences between sets of optimization decisions. These metrics provide a concrete measure of the extent to which the selection of training data influences the way that a program is optimized by a compiler.

During the compilation process, selected compiler decisions are written to a log file. For clarity, a particular instance where a decision is made is referred to as a *choice*, and the selected outcome of the choice is a *decision*. For example, at a callsite *foo* in a program, the compiler has a *choice* about inlining *foo*, which results in a *yes* or *no decision*.

Figure 3.1 shows the callsites of a simple program that will serve as a running example. Assume that there is additional code, which is omitted for brevity and clarity, in each of the functions. Three possible inlining logs are presented in Figure 3.2. The notation `caller.callee` is used to name callsites.

Log files record the compiler's choices and decisions for an optimization during a single com-

```
void foo() {}

void bar() {
  foo();
}

int main(int argc, char* argv[]) {
  foo();
  bar();
}
```

Figure 3.1: Callsites in a simple program

callsite	**log 1**	**log 2**	**log 3**	**log 4**
`bar.foo`	`inline`	`call`	`inline`	`call`
`main.foo`	`call`	`call`	`call`	`inline`
`main.bar`	`call`	`inline`	`inline`	`inline`
`main.bar.foo`		`inline`	`inline`	`inline`

Figure 3.2: Some possible inlining logs

callsite	$\vec{v}_1$	$\vec{v}_2$	$\vec{v}_3$	$\vec{v}_4$
`bar.foo`	`1`	`0`	`1`	`0`
`main.foo`	`0`	`0`	`0`	`1`
`main.bar`	`0`	`1`	`1`	`1`
`main.bar.foo`	`0`	`1`	`1`	`1`

Figure 3.3: Log files converted to vectors

pilation. All the logs for a given benchmark and optimization are processed together. Each log is converted into a vector. Each vector is the same length, with one entry for every unique choice recorded in the set of logs. By convention, a 0 is recorded in the vector for a negative decision (choosing not to perform an optimization), while a positive non-zero value is recorded for a positive decision (choosing to perform the optimization). In the case where a choice is not present in one or more logs, a default value of 0 is recorded. This situation may arise any time the existence of one decision depends on a previous positive decision. By making a negative decision for one choice, the compiler implicitly makes negative decisions for all choices that depend on a positive decision for that first choice. For example, the *main.bar.foo* callsite does not exist in log 1 in Figure 3.2, so it is assigned the default value of 0 in the vectors in Figure 3.3.

Once each of the n logs has been converted into a vector $\vec{v}_i$, the Difference and Coverage metrics can be calculated. The terms log and vector are used interchangeably to refer to vectors $\vec{v}_i$.

3.1.1 Difference

The difference metric quantifies the difference between two logs. It is defined as the squared length of the difference vector between two log vectors $\vec{v}_i$ and $\vec{v}_j$:

$$\delta(\vec{v}_i, \vec{v}_j) = |\vec{v}_i - \vec{v}_j|^2$$

In the case where binary decisions are recorded in the vectors as 0s and 1s, $\delta(\vec{v}_i, \vec{v}_j)$ is simply the

	$\vec{v}_1$	$\vec{v}_2$	$\vec{v}_3$	$\vec{v}_4$
$\vec{v}_1$	0	3	2	4
$\vec{v}_2$		0	1	1
$\vec{v}_3$			0	2
$\vec{v}_4$				0

Table 3.1: Values for the difference metric

Hamming distance between the vectors[1]. Difference values for the example are given in Table 3.1.

δ grows as the number of choices that resulted in different decisions in the two logs increases. Therefore, this metric gives a direct indication of the extent to which a different selection of training input can result in different optimization decisions during compilation. However, δ has no concept of the relative importance of the decisions. Two logs that differ only regarding insignificant decisions may have the same δ value as two logs that only differ with respect to a few key decisions. Therefore, there may be no correspondence between the difference score and performance.

3.1.2 Alignment

The common implicit assumption of most work that uses FDO is that as long as the training dataset is "representative" of usual program behavior, the particular dataset used for training is inconsequential. If this is the case, then the optimization logs based on profiles from different training inputs should not vary significantly. The difference metric can identify differences between a pair of logs, but does not answer the question of how much the logs agree with each other across the entire set of logs. The *alignment* metric quantifies the level of agreement between one optimization log and the collective choices made across the logs from all the inputs for a program.

To calculate an alignment score for a log, first calculate the *combined total* vector:

$$\vec{T} = \sum_i \vec{v}_i$$

$\vec{T}$ can be seen as a measure of agreement between all the logs. A choice that frequently results in a positive decision will have a high value recorded at its index in $\vec{T}$, while a decision that is usually decided negatively will have a low value in $\vec{T}$. In the example, $\vec{T} = [2\ \ 1\ \ 3\ \ 3]^T$.

The alignment of a log $\vec{v}_i$ is defined as:

$$\alpha_i = \frac{\vec{T} \cdot \vec{v}_i}{\sum_j \vec{T}[j]}$$

α is most usefully reported as a percentage, where the sum of the elements of $\vec{T}$ is used as the denominator. Recall that the dot product of two vectors, $\vec{x} \cdot \vec{y} = |\vec{x}||\vec{y}|cos(\theta)$, where θ is the angle between the vectors. Therefore, α is related to the angle between a log and $\vec{T}$. Since α_i is the accumulation of the element-wise products of $\vec{T}$ and $\vec{v}_i$, α will be large only if $\vec{v}_i$ has positive values (*i.e.*, positive decisions) at the same indexes as many other logs. If a log has no positive decisions, α will be 0. On the other hand, if a log has a positive decision for every choice for which any log records a positive decisions, α will be 100%. In the example, $\alpha_1 = \frac{2}{9} = 22\%$, $\alpha_2 = \frac{6}{9} = 67\%$, $\alpha_3 = \frac{8}{9} = 89\%$, and $\alpha_4 = \frac{7}{9} = 78\%$.

[1]The Hamming distance is the number of bits that are different between two equal-length binary vectors

3.1.3 Differences Between Logs

Logs may differ in two primary ways. First, the positive decisions in one log may be a superset (or subset) of the positive decisions in another. Alternately, two logs may make different decisions, such that the intersection of the two sets of positive decisions is small. Practically, the differences between two logs will fall somewhere on the continuum between these extremes, but will generally tend toward one or the other. It would be useful to distinguish between these two cases, since the first case represents more aggressive application of an optimization, while the second case represents a divergence of optimization strategies. Intuitively, the second case shows a more fundamental change in the behavior of the compiler than the first, and consequently a more significant difference between the training inputs that generated the logs in question.

The difference metric cannot distinguish between the two cases, since it merely counts the differences between the sets of positive decisions, without regard for whether one log is performing more optimization or different optimization than the other. On the other hand, the alignment metric does not directly measure the relationship between any pair of logs. However, when alignment and difference are considered together, they provide insight into the relationships between logs.

Let us consider first the cases where difference scores are low. In this case, the low difference scores are sufficient to identify the logs as very similar. Since there are few differences between the logs, alignment values are expected to be very high.

However, if differences between logs are larger, and one log has a higher alignment score than the other, it is likely that one log is roughly a superset of the other. Conversely, if the logs differ but have very similar alignment, then the difference is likely due to different optimization strategies rather than a difference in how frequently an optimization was performed. A low alignment value reinforces this conclusion, since it indicates that a larger proportion of choices were different between the logs.

3.2 Benchmarks and Inputs

Feedback-directed optimization involves a multi-step compilation process. First, an instrumented version of the program is compiled. This instrumented binary is run on a training input, and emits a profile that describes the run-time behavior of the program during that run. Finally, the program is recompiled. During this compilation, the compiler uses information from the profile file to guide code transformations. This study uses a workload of inputs for each program. Training is done once for each input in the workload. The evaluation of each of the resulting binaries is measured by running it on all the inputs in the workload.

In order to study the impact of various training datasets on the performance of feedback-directed optimizations, this study uses the standard SPEC CINT2000 benchmarks and their corresponding datasets. SPEC provides three sets of inputs for each program for use during performance evaluation.

The test input set is a very small input that is provided to allow easy verification that the system is configured properly for the compilation and execution of the benchmark program. The train input set consists of a small or medium-sized input for use during the training run of FDO. The ref (reference) input set is the input set used for performance evaluation. The reference inputs are large, and usually run for several minutes. SPEC provides one test and one train input for each program in the suite. The reference input set often contains a single ref input, but occasionally consists of several inputs that are processed in consecutive runs of the program.

The inputs provided by SPEC are insufficient for this study. The test inputs are very small, and thus might not be adequate for use during the training run for FDO. Both the test and train inputs are reduced in size compared to the ref inputs, and thus may be unsuitable for use during performance evaluation. Even in the best cases, there are only a small number of inputs in the SPEC reference workload. Therefore, all the SPEC inputs are included in the workloads for our programs, and are supplemented with additional inputs. These additional inputs are chosen to be representative of a larger range of inputs to the benchmark programs. Where possible, the benchmark authors have been consulted during the input selection process so that their expert knowledge of the program can provide insight and intuition to select inputs.

Some SPEC benchmark programs were omitted due to problems compiling them with the ORC. All benchmarks were used unmodified from the source code provided by SPEC. In some cases, newer versions of the programs were available that may have alleviated some experimental difficulties. Nonetheless, this study uses the original benchmark code in order to preserve consistency with other works.

Following are brief descriptions of the benchmark programs and the workloads used. Tables summarize the workload for each program, and provide additional details about each input. The average time for a statically optimized binary to run on each input on the Itanium 2 is presented as a quantitative measure of each input's size and complexity.

`Bzip2` is a popular compression utility that uses the Burrows-Wheeler block sorting text compression algorithm and Huffman coding. The additional inputs for `bzip2` are a collection of files in common formats. Files in these formats are often distributed over the Internet, or archived by users, and compression is usually employed in both of these scenarios. The `bzip2` workload is given in Table 3.2. `Bzip2` was not run on the log and combined inputs.

`Gzip` is another popular compression utility that uses Lempel-Ziv coding (LZ77). `Gzip` uses the same workload as `bzip2` (Table 3.2), with the addition of the log and combined inputs. SPEC does not provide details about the combined input, but judging by its name and the fact that it is `gzip`'s train input, it is reasonable to speculate that combined is a collection of parts taken from the `gzip` reference inputs.

`MCF` is a multi-commodity flow solver that uses the network simplex algorithm. The workload for `MCF` consists of the SPEC inputs along with several randomly generated problem instances using

Input	**Description**	**Size (MB)**	**Runtime (s)**	
			bzip2	**gzip**
mp3	An audio file encoded as an MPEG1 layer 3 audio stream using 128 Kbps constant bit rate encoding.	34	163.32	42.78
jpeg	A large image compressed using the JPEG image format, using a high quality setting.	15	147.42	40.99
xml	An exported iTunes [24] music library in XML format. The library contains approximately 2800 songs.	4.2	93.83	15.01
docs	A collection of Word, WordPerfect and RTF formated text documents, Excel and Quattro Pro spreadsheets, and PowerPoint presentations.	4.8	521.11	31.64
pdf	A collection of developer manuals for digital signal processors, as PDF documents.	16	117.85	36.89
mpeg	A video encoded as an MPEG-1 video stream.	2.9	157.21	42.86
compressed	The SPEC train input for `bzip2`, and the SPEC test input for `gzip`.	1.0	26.75	1.29
reuters	ASCII text from the Reuters collection [30].	4.4	55.35	44.37
gap	The 254.gap SPEC CINT2000 benchmark program binary compiled with optimization and without feedback by the ORC 2.1 compiler.	3.4	86.53	72.75
graphic	A SPEC reference input for both `bzip2` and `gzip`. A large TIFF image.	6.3	73.67	41.23
program	A SPEC reference input for both `bzip2` and `gzip`. A program binary.	3.3	73.15	67.62
random	A SPEC test input `bzip2`, and a SPEC reference input for `gzip`. Random data.	8.0	5.79	33.85
source	A SPEC reference input both for `bzip2` and `gzip`. A tarball of source code.	9.1	53.00	37.44
log	A SPEC reference input for `gzip`. A webserver log.	4.2	71.37	17.98
combined	The SPEC train input for `gzip`.	3.0	131.70	22.45

Table 3.2: Workload for `bzip2` and `gzip`

Input	Trips		Runtime (s)
	Time-Tabled	Dead-Head	
ref	16555	194581	530.34
test	646	2789	0.21
train	5985	84449	31.42
synth-0	10000	200000	596.99
synth-1	11000	200000	814.63
synth-2	12000	200000	1168.17
synth-3	13000	200000	1566.10
synth-4	14000	200000	1999.12
synth-5	5000	200000	56.78
synth-6	6000	200000	100.56
synth-7	7000	200000	162.47
synth-8	8000	200000	269.21
synth-9	9000	200000	386.11

Table 3.3: Workload for `MCF`

Input	Board Positions	Search Depth Limit	Runtime (s)
ref	5	11 - 12	133.60
test	4	7 - 8	2.99
train	4	8 - 10	18.61
wac-001	10	12	113.46
wac-051	10	12	165.31
wac-151	10	12	347.76
wac-251	10	12	275.01

Table 3.4: Workload for `crafty`

varied parameters. Each problem instance is composed of timetabled trips and dead-head trips, which are used to create the problem graph. Our testing showed that the difficulty of a problem instance is related to the ratio between the number of the two trip types. Unfortunately, efforts to contact the benchmark author to verify this result or gather additional insight into the problem failed. Therefore, we selected a number of deadhead trips similar to the SPEC reference input, and varied the number of timetabled trips. Table 3.3 provides additional details. Notice that the run times for the synthetic inputs span a range from about 10% to almost 400% the runtime of the SPEC reference input.

`Crafty` is a high-performance chess-playing program. The SPEC inputs used in the workload are each collections of chess positions to solve (determine if the current player will win or loose). The additional inputs are small collections of board positions arbitrarily selected from a large set provided by the program's author. Additional details can be found in Table 3.4. The additional inputs for `crafty` also show variation in program difficulty, based on program runtime.

`Parser` is a natural language parser that attempts to label words in English sentences with their correct part of speech. The version of parser in the SPEC CINT2000 suite is version 2, while the current version is version 4.0. The newer version can parse sentences faster, and can handle sentences that cause the SPEC version to abort in mid-run. Manual checking of inputs was required

Input	**Description**	**Sentences**	**Runtime (s)**
ref	The SPEC ref input.	7759	292.54
test	The SPEC test input.	848	1.87
train	The SPEC train input.	343	6.66
alice	Text from “Alice’s Adventures in Wonderland” by Lewis Carroll. Digital text is from the Project Gutenberg repository [10].	773	609.36
pa	Text from the news posts from December 29, 2004 through May 6, 2005 at Penny-Arcade [27], a popular video-game news and webcomic website.	2227	432.20
relativity	Text of “Relativity: The Special and General Theory” by Albert Einstein. Digital text is from Project Gutenberg [22]. Some manual processing was performed to fix sentences with equations and figure references.	590	534.52
worlds	Text from “The War of the Worlds” by H. G. Wells. Digital text is from Project Gutenberg [39].	2456	592.83
02-05words	Those sentences with only 2 - 5 words, inclusive, from the pa, alice, relativity and worlds inputs.	452	0.33
06-10words	Sentences with 6 - 10 words from the Project Gutenberg inputs.	1181	2.33
11-15words	Sentences with 11 - 15 words from the Project Gutenberg inputs.	1271	8.95
16-20words	Sentences with 16 - 20 words from the Project Gutenberg inputs.	1220	37.30
21-25words	Sentences with 21 - 25 words from the Project Gutenberg inputs.	1083	141.66

Table 3.5: Workload for `parser`

Input	Parameter	Runtime (s)
ref	N/A	173.29
test	N/A	0.87
train	N/A	6.67
snf200-300	200 and 300	0.73
snf525	525	4.86
snf750	750	18.99
snf900	900	35.99
snf1025	1025	59.82
snf1150	1150	84.34
snf1260	1260	114.47

Table 3.6: Workload for GAP

to prune our additional input of such offending sentences. Descriptions of our additional inputs are given in Table 3.5. The pa input was selected to exercise `parser`'s code that handles words not found in its dictionary, and is an example of informal writing. Alice was selected as an example of unusual word use and sentence structure. Relativity provides an example of more formal technical writing, while worlds provides more common word use and sentence structure, as well as dialog. The inputs for `parser` are varied in both the number of sentences and the resulting runtime, though the two measures are not strongly correlated.

GAP (Groups, Algorithms and Programming) is an interpreter for a mathematical language oriented for computations on groups. The version in the SPEC benchmark is V3R4P3, modified for the benchmarks to run on 64-bit architectures. However, the 64-bit porting was not complete, and only ensured that the test, training, and reference inputs, supplied to SPEC when the benchmark was submitted, ran correctly [35]. Therefore, there are limitations on the variety of input programs that can be selected for GAP. Several additional inputs were tried, but most caused incorrect behavior (*e.g.*, infinite loops). Consequently, there must be sections of code in the benchmark that none of our inputs exercise, namely those sections responsible for incorrect program behaviors. Our additional inputs are a single program, with a varied input parameter. The goal of varying the parameter is two-fold: first, as the parameter grows, the numbers used in calculations will grow and the interpreter will shift from machine integer arithmetic to long integer arithmetic. Second, as the parameter increases, the performance bottleneck should shift from the CPU to the memory hierarchy. While the SPEC test and train inputs are distinct, both overlap the computations specified in the SPEC ref input (*i.e.* some of the calculations performed by the test and train inputs are also exactly performed in the ref input) [35]. The inputs used in the GAP workload are listed in Table 3.6. The table does not indicate if the desired changes in program behavior are realized, but the run times for the additional inputs are quite varied.

VPR (Versatile Place and Route) is a tool to place and route circuits for Field-Programmable Gate Arrays (FPGAs). This benchmark has been split in two, with one copy for each of the main program tasks. In this way, training on placement inputs is prevented from creating binaries that perform

Input	Size (Logic Blocks)	Runtime (s)	
		Place	Route
ref (clma)	8383	87.63	82.52
test	N/A	1.05	0.44
train	N/A	9.90	9.23
alu4	1522	7.34	6.32
apex2	1878	10.77	7.97
apex4	1262	6.06	5.53
bigkey	1707	9.73	8.54
des	1591	9.11	16.81
diffeq	1497	8.03	4.79
dsip	1370	7.12	5.60
elliptic	3604	26.75	21.37
ex1010	4598	38.17	23.98
ex5p	1064	5.15	6.23
frisc	3556	27.14	23.82
misex3	1397	6.89	5.79
pdc	4575	39.09	127.63
s298	1931	9.55	4.16
s38417	6406	59.96	28.75
s38584.1	6447	60.63	30.80
seq	1750	9.59	7.52
spla	3690	28.59	30.80
tseng	1407	5.17	2.41

Table 3.7: Workload for VPR

poorly on routing inputs (and vice versa), as these effects would exaggerate the differences between training inputs. Both the placement and routing versions of the experiments use the same set of input circuits, but perform only the appropriate task. The additional inputs for the VPR workloads are the circuits from the FPGA Place-and-Route Challenge [9]. The SPEC ref input is the clma input from the FPGA challenge, thus this input is only included once. Table 3.7 lists the inputs in the VPR workloads. While there is variation in the run times for the inputs, for both placement and routing, 13 of the 22 inputs have run times of less than 10s.

Of the remaining SPEC CINT2000 benchmark programs, perlbmk, vortex, and twolf caused the ORC to crash during compilation when flags to emit the inlining log were used. GCC and eon are known problems with the ORC 2.1 when optimization is used in conjunction with feedback. Since these benchmarks could not be compiled, they were thus omitted from this study.

3.3 Architectures

Both the Itanium and the Itanium 2 implement the 64-bit IA-64 Explicitly Parallel Instruction Computing (EPIC) Instruction Set Architecture (ISA) [32]. EPIC uses in-order issue of bundles of instructions. Each bundle contains 3 instructions that can be executed in parallel, and must conform to one of the 10 patterns of instruction types specified by the ISA (such as Memory-Integer-Branch

(MIB) or Memory-Integer-Integer (MII)). The EPIC ISA relies heavily on compiler technology. The compiler is responsible for exposing Instruction Level Parallelism (ILP) and effectively scheduling instruction to ensure (1) that bundles contain few null operations, and (2) that stop instructions (a barrier to parallel execution) are avoided. Furthermore, the compiler must make effective use of the advanced features of the architecture such as hardware-supported control and data speculation, instruction predication, the Register Stack Engine (hardware spill and restore), cache hints and data cache prefetch instructions.

While both the Itanium and the Itanium 2 are theoretically capable of fetching, issuing, executing and retiring two bundles in each cycle, the Itanium does not have sufficient execution resources to frequently achieve this level of performance in practice. The Itanium 2 increases the number of integer units from 4 to 6, the number of multimedia units from 4 to 6, and the number of load/store ports from 2 to 4. Of the 100 possible sequences of two bundle types (*e.g.*, an MIB bundle followed by a MII bundle), only 28 can be fully issued on the Itanium. The additional execution resources of the Itanium 2 allow an additional 47 sequences to be fully issued. Of course, the performance gained by this additional capability depends on the actual sequences of bundle types generated by the compiler for a particular program.

Additionally, the Itanium has a 10-stage pipeline, while the Itanium 2 has an 8-stage pipeline. Due to this shorter pipeline, the negative impact of branch misprediction is expected to be reduced, since a pipeline flush results in less lost work. Consequently, the benefit of `if` conversion should be less on the Itanium 2since the performance gain from `if` conversion is partially due to a reduction in branch mispredictions.

This study does not directly compare performance on the two platforms since the Itanium 2 has a distinct advantage in terms of both computational resources and clock frequency. Furthermore, we are not interested in raw system performance, but rather on the effect of compiler decisions during feedback-directed optimization on system performance.

Our experiments on the Itanium were performed on two 4-processor 733-MHz machines with 6 GB of RAM. Files are located in an NFS-mounted directory, though file-system performance should have a negligible performance impact since the SPEC benchmarks are specifically modified to minimize disk access. Our Itanium 2 machine has a 1.3-GHz processor and 1 GB of RAM. Files are located on the local disk. All the machines run RedHat Linux 7.2 with version 2.4 SMP kernels.

Chapter 4

Results

In order to evaluate FDO in the ORC, we created *Aestimo*[1]. *Aestimo* is a performance evaluation tool that automates the process of compiling, executing, and evaluating the input programs on their workloads. Figure 4.1 provides an overview of *Aestimo*.

The experiments performed by *Aestimo* required the creation of a large number of binaries. A flow diagram for *Aestimo*'s compilation process is presented in Figure 4.2. The bold boxes indicate "final products" that are subsequently used by *Aestimo*. Each benchmark program is compiled statically once for each optimization being studied to create the "static" binary, and to create the static optimization logs. The compiler flags used for the static compilation are the same as for the profiled case, except for the omission of flags that refer to the profile file. Only one instrumented binary is created for each program. However, the remaining steps in the flow diagram are performed for each optimization/input pair.

Aestimo produces binaries that only use profile-guided decisions for the optimization under investigation for each of the inputs in the workload. First, a training run executes the instrumented binary on the input. Then, the benchmark is compiled using the generated profile data, and an optimization log is emitted for the optimization in question. The binary produced at this point is discarded. Finally, *Aestimo* recompiles the benchmark statically. However, the optimization log is used to instruct the compiler to make the same decisions for that optimization as it did during the full profile-guided compilation. In this way, optimization decisions based on profile information (rather than static estimates) are used only for the optimization in question. The binaries produced by this final compilation are referred to as FDO binaries.

During the final compilation, the compiler may not be able to perform every optimization listed in the log. For example, if the log is for `if` conversion, there may be a function that is not inlined without profile guidance. In that case, any `if` conversion listed in the log for the inlined code will be ignored. On the other hand, any additional optimizations that become profitable due to a forced decision will still be available to the compiler. For example, if the log forces a callsite to be inlined, any static optimizations applicable to the inlined code will still be applied. Therefore, our technique

[1]*Aestimo* is a Latin verb whose meaning is similar to that of the English verb *evaluate*

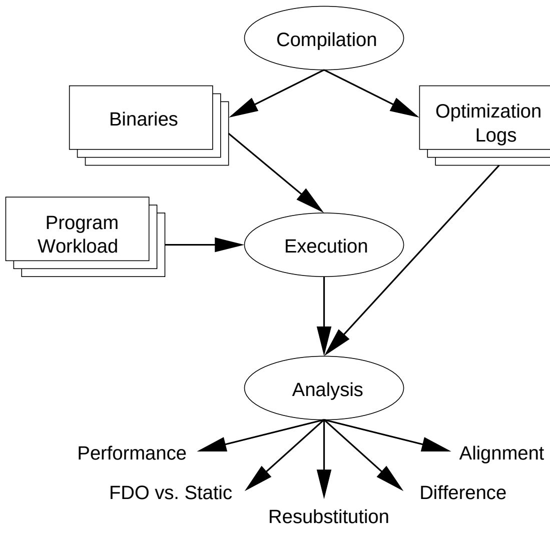


Figure 4.1: Overview of *Aestimo*

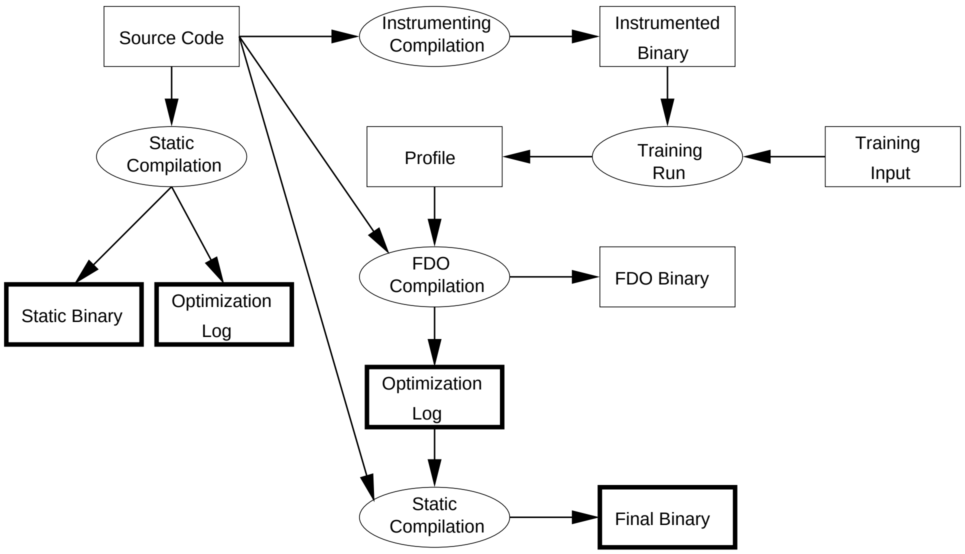


Figure 4.2: Compilation process

Benchmark	**Processor Time (hr:min:sec)**	
	Itanium	**Itanium2**
bzip2	433:56:13	79:13:22
crafty	67:06:19	22:30:59
gap	44:47:15	14:46:52
gzip	61:12:16	24:05:20
parser	246:14:46	94:23:45
mcf	822:05:34	306:02:20
vpr.place	84:53:30	30:12:08
vpr.route	104:46:11	31:19:45
Total	1865:02:06	602:34:34
Total	**2467:36:40 (102.8 days)**	

Table 4.1: Total processor time of experiments

ensures that any opportunity to apply the optimization in question will result in the same decision as in the full feedback-directed case, while not ignoring cascading effects due to the interrelatedness of optimizations.

After the compilation process, *Aestimo* executes each of the FDO binaries on each of the inputs in the program workload five times. The combined run times of the experiments performed by *Aestimo* are presented in Table 4.1. These figures include only the time required to perform the five trial executions of each FDO `if` conversion or inlining binary on each input in the workload. The time to compile each of the 976 binaries (8 instrumented binaries, 16 static binaries, 232 full-FDO binaries, 116 FDO `if` conversion binaries, and 116 FDO inlining binaries, for each of the two processors) is not included. Furthermore, the time to perform the 464 training runs on the instrumented binaries (which can run an order of magnitude slower than the optimized binary) to generate profiles for FDO are omitted from these figures. Nonetheless, the experiments represent more than 102 machine days worth of processing.

Once execution is complete, *Aestimo* analyses the program run times and the optimization logs, and reports the results. The optimization logs are used to calculate the difference and alignment metric scores (Section 4.1). The run times of the static and FDO binaries are compared to evaluate performance on the workload (Section 4.2) and the effectiveness of FDO compared to static optimization (Section 4.4). FDO run times are also used to investigate the usefulness more accurate profile information by comparing resubstitution with the performance of other FDO binaries (Section 4.3).

4.1 Profile Differences

Let's return to the first question: Does training on different inputs result in different compile-time decisions? *Aestimo* calculates scores for the difference and alignment metrics defined in Chapter 3 for each of the benchmarks. These scores are summarized in tables similar to Table 4.2. Each pairing of logs results in a difference score. The second and third columns of the table report the mean and standard deviation of the difference scores, defined in Section 3.1, for the FDO log listed in the first column paired with all the other FDO logs. The Max column reports the maximum difference between a log and any other FDO log. The Static column reports the difference metric when a log is compared to the static log. The final column of the table reports the alignment score for the log. The static log is included in the combined total vector when calculating alignment scores.

Other relevant information is recorded in the last four rows of each table. The number of distinct positive decisions encountered in all `if` conversion logs, or the number of callsites listed in the inlining logs, indicates the length of the vectors used to calculate the metrics. Choices with `Yes` or `No` consensus are those where the same decision is made in every log. Full consensus is achieved when every log is in agreement about the decision. FDO consensus ignores the static log, and checks for consensus among the FDO logs only. The number of choices without consensus indicates the maximum possible number of choices where two logs could disagree. For example, in Table 4.2 there are 87 branches that are `if converted` in at least one log. All the FDO logs agree that 15 branches should be `if converted`, and that 31 of them should not be. Therefore, 41 branches remain where different FDO logs make different decisions.

References to logs in this section refer only to the FDO logs, and omit the static optimization log. When relevant, the static log will be identified explicitly.

Graphs of the raw difference and alignment scores can be found in Appendix A.

4.1.1 `If` conversion

Emitting the logs of if-conversion decisions required a small change to the ORC. We inserted a small segment of code to output the source file name, function name, and area and basic block lists for each region that is if-converted. Therefore, only positive choices are recorded in the log file.

An excerpt from the static `if` conversion log for `bzip2` is provided in Figure 4.4. The transformations indicated by this excerpt are illustrated in Figure 4.3. The four `if` conversion transformations occur in the `sendMFTValues` function, and result in the creation of a large predicated region from five basic blocks. An area is a data structure used by the ORC to represent a single-entry region of code. Before `if` conversion, each BB in a program is an area. However, as `if` conversion removes branches and merges BBs, areas grow to include multiple BBs. In Figure 4.3, each rectangle represents an area, and each number represents a BB. A dashed box represents a BB that has been merged into a larger area. The edges between areas represent control flow transitions. Initially, each area consists of a single basic block and is named for the BB that it contains. For example, an area containing BB 42 is named A42.

The first two lines in Figure 4.4 are for `if` statements that do not have an `else` path, as shown if Figure 4.3(a). The first line indicates a positive `if` conversion decision for the branch at the end of A97. Area A99 is the branch target when the `if` at the end of A97 evaluates to false. Thus, A99 is not predicated. Instead, the branch in the A97 is converted to a predicate calculation, and A98 is

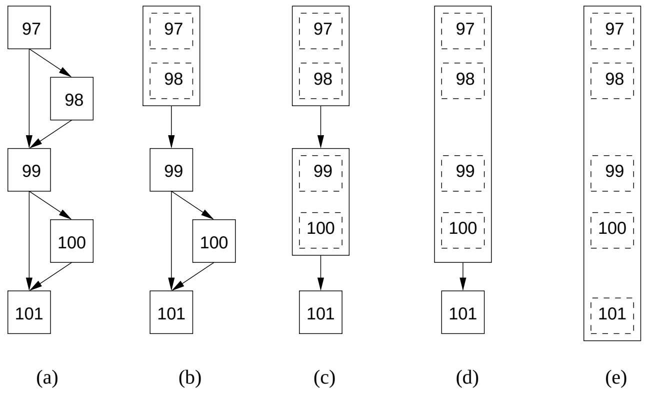


Figure 4.3: `If` conversion performed in the log excerpt

File	Function	Areas	Area List
`bzip2.c`	`sendMTFValues`	3	`A97{ 97} A98{ 98} A99{ 99}`
`bzip2.c`	`sendMTFValues`	3	`A99{ 99} A100{ 100} A101{ 101}`
`bzip2.c`	`sendMTFValues`	2	`A97{ 97 98} A99{ 99 100}`
`bzip2.c`	`sendMTFValues`	2	`A97{ 97 98 99 100} A101{ 101}`

Figure 4.4: `If` conversion log excerpt

predicated. The contents of A98 (BB 98) are then appended to A97 (Figure 4.3(b)). The second line records that the branch at the end of A99 should also be `if`-converted. This decision causes A100 to be predicated and appended to A99 (Figure 4.3(c)).

The last two lines in Figure 4.4 record decisions to eliminate unnecessary control flow. Line three of the log records a decision to append A99 to A97 (Figure 4.3(d)). Line four is similar for A97 and A101 (Figure 4.3(e)). The final area is larger than any of the five original BBs, and contains no control flow. Therefore, it provides more opportunities for optimizations such as common subexpression elimination and instruction scheduling than the same region of code before `if` conversion.

As explained in Section 3.1, when *Aestimo* processes the logs, any choices that are missing from a log are (correctly) assumed to be a negative decision (not `if-converted`). Neither the difference nor the alignment metric are affected by recording only positive choices. A choice that is negative in all logs will not appear in the vectors. Thus, it cannot contribute to the difference. Moreover, negative choices never contribute to the alignment score. A consequence of recording only positive decisions is that there can never be a choice with No consensus: such a choice would not appear in any log, and thus *Aestimo* does not know about it. However, there are a very large number of regions in every program that are evaluated for `if` conversion, but are never `if`-converted. Recording only positive decisions also means that the number of choices that have a No FDO consensus is exactly the number of choices where static performed `if` conversion but no FDO log did.

In most cases, the largest differences between logs are between the static log and FDO logs. Therefore, profiling does, in general, result in significantly different optimization decisions that static optimization. Nonetheless, the selection of training input can also result in significant differ-

Input	**Mean**	**Std Dev**	**Max**	**Static**	**Alignment (%)**
combined	13.64	10.32	27	43	84.21
compressed	14.64	10.38	27	45	82.81
docs	14.43	10.37	28	42	84.74
gap	19.57	12.17	33	62	48.60
graphic	11.93	9.50	25	41	86.49
jpeg	16.21	12.04	28	61	51.40
log	15.43	11.18	30	38	89.47
mp3	16.21	12.04	28	61	51.40
mpeg	16.29	11.82	27	60	52.81
pdf	13.50	11.17	29	39	90.35
program	16.29	11.82	27	60	52.81
random	19.00	12.62	31	64	46.32
reuters	17.07	12.01	33	39	91.58
source	13.36	11.34	29	39	90.53
xml	13.29	11.09	28	40	89.12
Distinct Positive Decisions					87
Choices with `Yes` Consensus					14 Full, 15 FDO
Choices with `No` Consensus					0 Full, 31 FDO
Choices without Consensus					73 Full, 41 FDO

Table 4.2: `If` conversion metric scores for `bzip2` on the Itanium

ences in the optimizations decisions made by the compiler.

`Bzip2` presents some interesting alignment values. In Tables 4.2 and 4.3, 6 of the 15 inputs result in alignment scores less than 55%, while the remaining 9 have alignment scores greater than 80%. There is no similar pattern in the difference scores. *Aestimo* can perform a *cut* operation, where the inputs in a workload are split into two groups according to their alignment score. If an input has an alignment score greater than the *cut value*, it is assigned to the *high cut group*, but if it has an alignment score lower than the cut value, it is assigned to the *low cut group*. The static log is included in both groups. After the cut is made, the metric scores are recalculated for each group separately. Tables 4.4 and 4.5 show the results of cutting the `bzip2` workload on the Itanium at 55%.

Differences between logs after the cut are small in both groups. This indicates that training on different inputs results in two distinct `if` conversion optimization strategies for the Itanium. The consensus values for the cut groups show that training on inputs that result in larger alignment scores results in more `if` conversion than training on the inputs with lower alignment scores. The low alignment scores after the cut for inputs in the low cut set are due to their large differences with static. Unfortunately, there do not appear to be significant differences between the decisions made when training uses members of the same cut group.

On the Itanium 2, the results of the cut are similar to the Itanium. However, in the high cut group, the combined input still results in a mean difference score larger than 41, while the other inputs have difference scores less than 12. Therefore, training on the combined input results in significantly different `if` conversion decisions than training on any other input in the workload.

Input	**Mean**	**Std Dev**	**Max**	**Static**	**Alignment (%)**
combined	52.93	20.24	69	5	90.20
compressed	19.36	15.96	44	45	81.87
docs	20.93	15.95	44	45	82.60
gap	26.00	20.21	69	70	41.67
graphic	18.07	15.69	42	43	85.67
jpeg	23.29	20.58	69	70	44.44
log	18.64	16.47	40	41	87.57
mp3	24.57	20.54	67	72	40.64
mpeg	23.29	20.58	69	70	44.44
pdf	18.07	15.69	42	43	85.67
program	23.29	20.58	69	70	44.44
random	24.57	20.54	67	72	40.64
reuters	21.43	16.24	39	40	88.30
source	18.64	16.47	40	41	87.57
xml	18.64	16.47	40	41	87.57
Distinct Positive Decisions				94	
Choices with `Yes` Consensus				14 Full, 14 FDO	
Choices with `No` Consensus				0 Full, 3 FDO	
Choices without Consensus				80 Full, 77 FDO	

Table 4.3: `If` conversion metric scores for `bzip2` on the Itanium 2

Input	**Mean**	**Std Dev**	**Max**	**Static**	**Alignment (%)**
gap	6.00	3.24	8	62	62.37
jpeg	2.00	2.24	5	61	65.98
mp3	2.00	2.24	5	61	65.98
mpeg	2.40	2.78	6	60	67.53
program	2.40	2.78	6	60	67.53
random	4.40	3.02	8	64	60.31
Distinct Positive Decisions				81	
Choices with `Yes` Consensus				14 Full, 15 FDO	
Choices with `No` Consensus				0 Full, 57 FDO	
Choices without Consensus				67 Full, 9 FDO	

Table 4.4: `If` conversion metric scores for `bzip2` low cut group (cut = 55%) on the Itanium

Input	**Mean**	**Std Dev**	**Max**	**Static**	**Alignment (%)**
combined	5.88	3.65	10	43	83.04
compressed	6.88	3.78	12	45	81.94
docs	6.75	4.05	11	42	83.48
graphic	4.62	2.31	8	41	84.36
log	7.00	3.61	10	38	88.11
pdf	4.88	2.53	7	39	88.77
reuters	8.12	4.22	12	39	90.31
source	4.62	2.96	8	39	88.99
xml	4.75	3.03	8	40	87.67
Distinct Positive Decisions				86	
Choices with `Yes` Consensus				36 Full, 37 FDO	
Choices with `No` Consensus				0 Full, 32 FDO	
Choices without Consensus				50 Full, 17 FDO	

Table 4.5: `If` conversion metric scores for `bzip2` high cut group (cut = 55%) on the Itanium

Input	Mean	Std Dev	Max	Static	Alignment (%)
ref	225.83	128.88	384	492	53.29
test	247.33	120.64	341	405	54.28
train	230.83	130.40	390	494	46.75
wac-001	239.50	147.55	436	516	52.04
wac-051	239.83	147.54	442	516	50.28
wac-151	244.17	144.14	434	524	51.96
wac-251	404.50	185.23	442	194	60.73
Distinct Positive Decisions				920	
Choices with `Yes` Consensus				75 Full, 78 FDO	
Choices with `No` Consensus				0 Full, 104 FDO	
Choices without Consensus				845 Full, 738 FDO	

Table 4.6: `If` conversion metric scores for `crafty` on the Itanium

Input	Mean	Std Dev	Max	Static	Alignment (%)
ref	156.17	71.47	173	511	59.27
test	176.50	81.90	219	491	57.71
train	193.33	91.67	234	520	54.63
wac-001	188.33	90.17	234	560	64.04
wac-051	179.33	84.16	222	550	61.19
wac-151	180.67	82.67	202	544	62.21
wac-251	157.00	71.57	166	518	60.96
Distinct Positive Decisions				935	
Choices with `Yes` Consensus				79 Full, 117 FDO	
Choices with `No` Consensus				0 Full, 338 FDO	
Choices without Consensus				856 Full, 480 FDO	

Table 4.7: `If` conversion metric scores for `crafty` on the Itanium 2

Input	Mean	Std Dev	Max	Static	Alignment (%)
ref	419.56	174.81	488	335	83.93
snf1025	173.33	250.54	517	206	95.70
snf1150	173.33	250.54	517	206	95.70
snf1260	189.78	239.81	506	233	94.74
snf200-300	177.78	244.86	517	198	95.59
snf525	175.56	245.35	512	203	95.68
snf750	173.33	251.19	518	207	95.66
snf900	173.33	250.54	517	206	95.70
test	444.89	215.04	518	577	77.49
train	430.22	215.63	510	517	79.75
Distinct Positive Decisions				1723	
Choices with Yes Consensus				1021 Full, 1024 FDO	
Choices with No Consensus				0 Full, 32 FDO	
Choices without Consensus				702 Full, 667 FDO	

Table 4.8: `If` conversion metric scores for GAP on the Itanium

Input	Mean	Std Dev	Max	Static	Alignment (%)
ref	260.00	264.41	294	335	91.62
test	200.00	240.15	294	577	85.31
train	166.00	186.43	226	517	88.23
Distinct Positive Decisions				1700	
Choices with Yes Consensus				1079 Full, 1104 FDO	
Choices with No Consensus				0 Full, 283 FDO	
Choices without Consensus				621 Full, 313 FDO	

Table 4.9: `If` conversion metric scores for GAP SPEC inputs on the Itanium

Alignment scores for all inputs on `crafty` are low, in the 50-60% range, compared to the other benchmarks where alignment is usually greater than 80%. Furthermore, difference scores are quite large compared to the number of choices without consensus. Therefore, there is significant disagreement between the logs, and no dominant optimization strategy. These results indicate that the inputs selected for crafty are significantly varied, in terms of the `if` conversion decisions they produce. Consequently, any performance variations between these FDO binaries can be more confidently linked to the selection of training input than in cases such as `bzip2`, where the selection of training input has a limited impact on optimization decisions.

Recall from Section 3.2 the difficulty of selecting additional inputs for GAP. Table 4.8 and 4.11 indicates that varying the parameter in the additional input, snf, may not have induced the changes in memory behavior and large-number processing methods that we desired. Alternatively, these changes did occur, but did not result in different `if` conversion decisions. The differences scores for the snf inputs are less than half those of the SPEC ref, test, and train inputs. Furthermore, the high alignment scores for the snf inputs suggests that they tend to agree with each other. On the other hand, the high difference scores and lower alignment scores of the SPEC inputs suggest that training on these inputs results in substantially different decisions than training on the snf inputs. Further investigation reveals that the maximum differences occur between snf and SPEC inputs.

Input	**Mean**	**Std Dev**	**Max**	**Static**	**Alignment (%)**
snf1025	11.83	14.72	35	206	98.20
snf1150	11.83	14.72	35	206	98.20
snf1260	36.33	16.68	43	233	97.08
snf200-300	20.50	14.60	43	198	98.01
snf525	17.67	12.83	38	203	98.10
snf750	11.33	12.47	32	207	98.18
snf900	11.83	14.72	35	206	98.20
Distinct Positive Decisions				1635	
Choices with `Yes` Consensus				1394 Full, 1413 FDO	
Choices with `No` Consensus				0 Full, 172 FDO	
Choices without Consensus				241 Full, 50 FDO	

Table 4.10: `If` conversion metric scores for `GAP` `snf` inputs on the Itanium

Input	**Mean**	**Std Dev**	**Max**	**Static**	**Alignment (%)**
ref	461.78	238.91	554	626	76.29
snf1025	185.78	275.76	553	219	96.10
snf1150	186.44	276.06	554	220	96.12
snf1260	193.78	258.98	537	239	95.18
snf200-300	189.33	271.33	550	208	96.14
snf525	186.44	270.99	547	213	96.17
snf750	185.78	275.76	553	219	96.10
snf900	185.78	275.76	553	219	96.10
test	438.89	235.50	532	600	78.58
train	446.00	245.51	545	613	77.95
Distinct Positive Decisions				1782	
Choices with `Yes` Consensus				1002 Full, 1012 FDO	
Choices with `No` Consensus				0 Full, 119 FDO	
Choices without Consensus				780 Full, 651 FDO	

Table 4.11: `If` conversion metric scores for `GAP` on the Itanium 2

Input	Mean	Std Dev	Max	Static	Alignment (%)
combined	10.57	11.04	43	62	87.24
compressed	44.36	12.63	50	25	83.37
docs	14.14	10.85	47	66	76.88
gap	15.07	9.87	40	59	82.92
graphic	12.36	9.96	40	65	88.38
jpeg	10.57	11.19	43	62	87.24
log	19.50	11.59	50	55	75.97
mp3	12.14	11.62	45	64	85.88
mpeg	10.57	10.52	43	60	85.54
pdf	12.00	11.50	45	62	84.28
program	12.36	9.93	42	59	83.26
random	15.50	11.42	48	59	80.75
reuters	14.36	10.18	44	59	81.66
source	16.36	10.48	46	63	76.20
xml	15.00	10.64	45	60	78.36
Distinct Positive Decisions					108
Choices with Yes Consensus					28 Full, 28 FDO
Choices with No Consensus					0 Full, 12 FDO
Choices without Consensus					80 Full, 68 FDO

Table 4.12: If conversion metric scores for gzip on the Itanium

Additionally, the differences between the snf and SPEC inputs are the only differences in this study where comparison to another FDO if conversion log results in a larger difference than comparison to the static log.

Cutting the workload at 85% separates the snf inputs from the SPEC inputs. Table 4.9 shows metric scores for the SPEC inputs on the Itanium. While the mean difference scores for these inputs are lower than when calculated for the entire workload, they are still quite large. Therefore, there are significant differences in the if conversion decisions made depending which SPEC input is used for training. On the other hand, Table 4.10 shows the scores for the snf inputs. In this case, all the inputs are very similar. The consensus results indicate that the inputs make different decisions for no more than 50 of the 1635 if conversion choices recorded in the logs. On the Itanium 2, the results of this cut are very similar. Therefore, it is less likely that selecting different training inputs from among the snf inputs will result in significant performance differences.

In Table 4.12 difference and alignment scores are fairly uniform across all logs for gzip on the Itanium. However, compressed has a much larger difference score than the other logs. On average, compressed disagrees with other logs for more than 2/3 of the choices without consensus. This large difference score appears to have no impact on compressed's alignment score. Examination of the log files reveals that training on compressed leads to more if conversion than training on other inputs. While training on other inputs results in 47-54 positive if conversion decisions, training on compressed results in 77 positive if conversion decisions. On the Itanium 2, compressed's metric scores do not distinguish it from the other inputs. For this processor, FDO if conversion results in between 49 and 59 positive if conversion decisions, and training on compressed results

Input	Mean	Std Dev	Max	Static	Alignment (%)
combined	12.36	5.91	21	74	87.46
compressed	15.29	8.52	26	67	77.67
docs	21.29	7.87	30	77	77.78
gap	17.71	8.34	30	71	82.62
graphic	17.43	7.25	26	71	84.60
jpeg	12.00	6.56	22	71	87.02
log	22.43	9.08	30	67	79.21
mp3	15.57	8.93	28	71	78.99
mpeg	11.64	5.15	19	70	84.71
pdf	14.64	7.27	25	74	82.18
program	17.43	7.64	26	73	84.49
random	16.00	8.77	26	65	78.99
reuters	16.71	7.50	28	71	83.39
source	19.36	7.96	27	76	74.92
xml	16.71	6.70	24	69	78.22
Distinct Positive Decisions					125
Choices with `Yes` Consensus					27 Full, 30 FDO
Choices with `No` Consensus					0 Full, 43 FDO
Choices without Consensus					98 Full, 52 FDO

Table 4.13: `If` conversion metric scores for `gzip` on the Itanium 2

in only 49.

As shown in Tables 4.14 and 4.15, there are virtually no differences between the FDO logs for `MCF`. In fact, on the Itanium 2, the FDO logs have no more than four different decisions between them. synth-5 results in the most distinct `if` conversion decisions on the Itanium, with a difference scores of 8-10 when compared to the other FDO logs. However, unless some of these few decisions are critical to performance, it is unlikely that there will be any significant variation in performance between the FDO binaries for `MCF`.

Tables 4.16 and 4.17 suggests a negative correlation between the difference and alignment scores for `parser`. One might suspect that higher alignment scores correspond to more `if` conversion. However, this is not the case. The FDO `if` conversion logs for the both processors have between 103 and 131 positive decision recorded in them. The median, 121, corresponds to alice in both cases, while the log for ref records 123 positive `if` conversion decisions. Despite this result, alice has the lowest alignment score, while ref has the largest alignment score. Therefore, the differences between inputs to `parser` represent substantially different `if` conversion decision. Comparing the results on the Itanium and Itanium 2, it appears that FDO results in similar decisions on both processors.

However, static optimization performs significantly more `if` conversion on the Itanium 2 than on the Itanium: the static log contains 58 decisions on the Itanium, but 204 on the Itanium 2. This result is in contrast to the other programs, where results were similar across processors. Furthermore, the larger number of functional units in the Itanium 2 should make `if` conversion profitable in more cases than on the Itanium. Therefore, intuition suggests that `if` conversion should be performed

Input	Mean	Std Dev	Max	Static	Alignment (%)
ref	1.25	2.33	8	32	92.43
synth-0	2.58	2.62	10	34	93.46
synth-1	2.58	2.62	10	34	93.46
synth-2	1.83	2.39	9	33	93.05
synth-3	1.25	2.33	8	32	92.43
synth-4	1.25	2.33	8	32	92.43
synth-5	8.58	2.74	10	40	79.75
synth-6	1.25	2.33	8	32	92.43
synth-7	1.25	2.33	8	32	92.43
synth-8	1.25	2.33	8	32	92.43
synth-9	1.25	2.33	8	32	92.43
test	3.08	2.49	10	34	92.84
train	1.25	2.33	8	32	92.43
Distinct Positive Decisions					67
Choices with `Yes` Consensus					23 Full, 28 FDO
Choices with `No` Consensus					0 Full, 27 FDO
Choices without Consensus					44 Full, 12 FDO

Table 4.14: `If` conversion metric scores for `MCF` on the Itanium

Input	Mean	Std Dev	Max	Static	Alignment (%)
ref	1.08	1.05	2	31	92.14
synth-0	2.42	1.50	4	33	93.15
synth-1	2.42	1.50	4	33	93.15
synth-2	1.67	1.11	3	32	92.74
synth-3	1.08	1.05	2	31	92.14
synth-4	1.08	1.05	2	31	92.14
synth-5	1.92	1.50	4	33	93.75
synth-6	1.92	1.50	4	33	93.75
synth-7	1.92	1.50	4	33	93.75
synth-8	1.08	1.05	2	31	92.14
synth-9	1.08	1.05	2	31	92.14
test	1.92	1.50	4	33	93.75
train	1.08	1.05	2	31	92.14
Distinct Positive Decisions					63
Choices with `Yes` Consensus					28 Full, 33 FDO
Choices with `No` Consensus					0 Full, 26 FDO
Choices without Consensus					35 Full, 4 FDO

Table 4.15: `If` conversion metric scores for `MCF` on the Itanium 2

Input	Mean	Std Dev	Max	Static	Alignment (%)
02-05words	65.73	22.20	75	135	66.20
06-10words	34.82	16.76	55	142	82.22
11-15words	31.36	18.44	62	145	84.71
16-20words	26.45	22.63	71	158	90.91
21-25words	26.45	23.64	71	158	90.91
alice	33.55	19.78	70	155	85.32
pa	25.91	23.15	71	158	90.24
ref	38.09	18.27	64	155	84.51
relativity	32.27	23.35	75	154	86.26
test	45.00	18.65	53	139	74.61
train	40.45	18.76	59	150	80.74
worlds	27.00	23.24	73	156	89.02
Distinct Positive Decisions				200	
Choices with `Yes` Consensus				5 Full, 62 FDO	
Choices with `No` Consensus				0 Full, 35 FDO	
Choices without Consensus				195 Full, 103 FDO	

Table 4.16: `If` conversion metric scores for `parser` on the Itanium

Input	Mean	Std Dev	Max	Static	Alignment (%)
02-05words	64.00	21.29	72	171	65.58
06-10words	35.45	17.54	55	198	78.56
11-15words	34.18	19.04	61	206	79.53
16-20words	28.18	21.70	67	202	84.78
21-25words	29.27	22.72	70	205	85.51
alice	34.55	19.54	66	199	80.01
pa	28.73	22.33	68	203	84.18
ref	40.55	18.65	62	199	78.80
relativity	34.55	23.26	72	197	81.64
test	48.00	20.02	58	195	70.95
train	42.91	19.50	57	196	75.36
worlds	30.91	23.24	72	199	83.57
Distinct Positive Decisions				280	
Choices with `Yes` Consensus				33 Full, 66 FDO	
Choices with `No` Consensus				0 Full, 111 FDO	
Choices without Consensus				247 Full, 103 FDO	

Table 4.17: `If` conversion metric scores for `parser` on the Itanium 2

Input	Mean	Std Dev	Max	Static	Alignment (%)
alu4	3.90	3.54	12	45	96.49
apex2	3.90	3.54	12	45	96.49
apex4	3.90	3.54	12	45	96.49
bigkey	3.90	3.54	12	45	96.49
des	6.76	3.96	16	45	95.39
diffeq	7.33	4.15	14	41	98.61
dsip	6.76	3.96	16	45	95.39
elliptic	10.38	4.29	16	43	95.72
ex1010	5.81	3.75	14	47	94.88
ex5p	3.90	3.54	12	45	96.49
frisc	6.19	3.98	12	39	98.24
misex3	3.90	3.54	12	45	96.49
pdc	8.10	4.18	14	49	93.96
ref	6.19	3.98	12	39	98.24
s298	6.57	3.55	12	47	93.74
s38417	6.19	3.98	12	39	98.24
s38584.1	6.19	3.98	12	39	98.24
seq	3.90	3.54	12	45	96.49
spla	8.10	4.18	14	49	93.96
test	6.95	3.72	16	49	95.24
train	3.90	3.54	12	45	96.49
tseng	7.33	4.15	14	41	98.61
Distinct Positive Decisions					158
Choices with `Yes` Consensus					102 Full, 106 FDO
Choices with `No` Consensus					0 Full, 32 FDO
Choices without Consensus					56 Full, 20 FDO

Table 4.18: `If` conversion metric scores for VPR (place) on the Itanium

more frequently on the Itanium 2than on the Itanium, not less.

`VPR` metric scores are similar to those of `MCF`. Mean difference scores are low, and alignment scores usually exceed 90%. Therefore, it is unlikely that there will be performance differences between FDO binaries. On the other hand, static differences are higher, particularly for the routing component of `VPR`, and performance differences between the FDO and static binaries are more likely.

4.1.2 Inlining

Inlining logs are generated using existing ORC compiler flags. In particular, the flag combination: `-Wj,-tt19:0x40000 -Wj,-tt19:0x80000` emits the inlining decision to the file `orc_script.log`. This file contains a section for each function that is compiled, and lists each callsite as either a `CALL` or `INLINE` decision. An example from `bzip2` is given in Figure 4.5. The entry is for the function `sortIt`, which has callsites for `panic` on line 2268, `qSort3` on line 2235, and `simpleSort` on line 2146. All three called functions, as well as `sortIt`, are found in the `bzip2.o` object file. Of the three calls, only the call to `qSort3` is inlined. Some optimizations can change the number of entries in a log file. Thus, each callsite encountered in a log is inserted

Input	**Mean**	**Std Dev**	**Max**	**Static**	**Alignment (%)**
alu4	4.29	3.61	12	50	96.16
apex2	4.29	3.61	12	50	96.16
apex4	4.29	3.61	12	50	96.16
bigkey	5.43	3.51	14	48	96.58
des	8.48	4.51	18	48	95.46
diffeq	6.57	4.41	12	44	97.83
dsip	8.48	4.51	18	48	95.46
elliptic	10.76	4.86	18	48	95.42
ex1010	6.19	3.83	14	52	94.62
ex5p	4.29	3.61	12	50	96.16
frisc	6.57	4.41	12	44	97.83
misex3	4.29	3.61	12	50	96.16
pdc	8.48	4.42	16	54	93.75
ref	6.57	4.41	12	44	97.83
s298	6.95	3.78	14	52	93.54
s38417	6.57	4.41	12	44	97.83
s38584.1	6.57	4.41	12	44	97.83
seq	4.29	3.61	12	50	96.16
spla	8.48	4.42	16	54	93.75
test	8.67	4.23	18	52	95.32
train	4.29	3.61	12	50	96.16
tseng	9.05	4.19	16	44	98.53
Distinct Positive Decisions				169	
Choices with `Yes` Consensus				108 Full, 111 FDO	
Choices with `No` Consensus				0 Full, 36 FDO	
Choices without Consensus				61 Full, 22 FDO	

Table 4.19: `If` conversion metric scores for VPR (place) on the Itanium 2

Input	Mean	Std Dev	Max	Static	Alignment (%)
alu4	6.29	5.62	15	87	92.29
apex2	6.29	5.62	15	87	92.29
apex4	8.00	4.69	19	84	94.08
bigkey	8.29	4.41	22	87	93.38
des	8.19	4.79	19	84	93.99
diffeq	9.62	5.91	25	86	94.30
dsip	7.33	4.77	18	91	91.73
elliptic	8.86	5.95	23	87	93.12
ex1010	6.29	5.62	15	87	92.29
ex5p	7.24	4.64	18	83	93.95
frisc	8.86	5.95	23	87	93.12
misex3	6.29	5.62	15	87	92.29
pdc	19.62	6.88	29	73	86.50
ref	8.29	5.14	21	85	94.43
s298	6.95	5.11	18	90	91.42
s38417	12.19	5.70	28	83	96.69
s38584.1	10.95	6.48	28	85	95.21
seq	6.29	5.62	15	87	92.29
spla	6.29	5.62	15	87	92.29
test	17.05	5.78	29	78	93.08
train	6.29	5.62	15	87	92.29
tseng	10.19	6.23	27	84	95.08
Distinct Positive Decisions				170	
Choices with `Yes` Consensus				73 Full, 83 FDO	
Choices with `No` Consensus				0 Full, 47 FDO	
Choices without Consensus				97 Full, 40 FDO	

Table 4.20: `If` conversion metric scores for `VPR` (route) on the Itanium

Input	Mean	Std Dev	Max	Static	Alignment (%)
alu4	5.90	5.28	15	86	93.05
apex2	5.90	5.28	15	86	93.05
apex4	7.62	4.47	19	83	94.65
bigkey	8.57	4.10	20	84	93.79
des	7.81	4.54	19	83	94.57
diffeq	8.57	4.93	21	81	96.10
dsip	7.62	4.72	17	88	92.31
elliptic	7.81	5.17	21	82	95.04
ex1010	5.90	5.28	15	86	93.05
ex5p	6.86	4.39	18	82	94.53
frisc	9.14	5.45	23	84	93.56
misex3	5.90	5.28	15	86	93.05
pdc	18.29	6.01	29	72	89.14
ref	7.81	5.17	21	82	95.04
s298	7.24	5.14	18	87	92.03
s38417	11.71	5.84	26	80	97.07
s38584.1	9.90	5.62	24	80	96.92
seq	5.90	5.28	15	86	93.05
spla	5.90	5.28	15	86	93.05
test	16.67	5.57	29	77	93.75
train	5.90	5.28	15	86	93.05
tseng	9.14	5.31	23	79	96.80
Distinct Positive Decisions				180	
Choices with `Yes` Consensus				85 Full, 95 FDO	
Choices with `No` Consensus				0 Full, 46 FDO	
Choices without Consensus				95 Full, 39 FDO	

Table 4.21: `If` conversion metric scores for VPR (route) on the Itanium 2

```
COMPILE ("bzip2.o",sortIt,NOREG) {
  CALL (2268,0,"bzip2.o",panic,NOREG)
  INLINE (2235,0,"bzip2.o",qSort3,NOREG) {
  }
  CALL (2146,0,"bzip2.o",simpleSort,NOREG)
}
```

Figure 4.5: Inlining log excerpt

into a table with its caller, its callee, and its line number, to ensure that all vectors are of the same length, and that each choice has a unique index that is the same in every vector.

Unfortunately, the information written to the log file is not sufficient to uniquely identify every callsite. If multiple calls to the same function occur on the same line of source code, they will have identical entries in the log file and will collide in the table. Furthermore, multiple calls in long statements (such as an `if` statement with many tests) that span multiple lines are considered to occur on the same line. In these cases, the choice is recorded as a `1` in the vector if any of the callsites are inlined. However, this aliasing problem is minor: in total, there are 128 callsites for the Itanium and 126 callsites for the Itanium 2 where aliasing occurs. For both processors, there are only 8 aliased callsites where the same decision is not made for all of the indistinguishable log entries: 3 (of 246) for `gzip`, 4 (of 4366) for `GAP`, and 1 (of 1464) for `bzip2`.

The results of metric calculations for inlining are similar to those presented for `if` conversion in Section 4.1.1. The results from the two processors are very similar. Static inlining results in the largest differences compared to other logs, while the differences between the profile-guided logs are much smaller. The tendency for alignment scores to be high suggest that either there is insufficient variety between the inputs in the workloads, or that inlining in the ORC is not very sensitive to inputs selection.

The consensus values for the `bzip2` indicate that the FDO inlining logs are not very similar. While there are a large number of callsites where there is consensus to not perform inlining, there are no callsites that are universally inlined for either processor. This fact is related to the observation that the FDO logs either have difference scores larger than 140 and alignment scores less than 7%, or difference scores less than 90 and alignment scores greater than 45%. The logs with lower difference scores also have much lower differences compared to static. Tables 4.23 and 4.24 show the results of cutting the inputs for the Itanium into two groups. The inputs that resulted in alignment scores greater than 45% are quite similar. Inputs in this group have difference scores and high alignment values when they are cut from the rest of the inputs. In fact, there are only 15 callsites where training on different inputs from this group results in different inlining decisions.

On the other hand, cutting the inputs with low alignment scores from the rest of the workload reveals that there are significant differences between the inputs in this group. Difference values are still very high, and alignment scores are only slightly larger than when calculated using the entire workload. Furthermore, there is very little consensus between the logs in this groups, and there is

Input	**Mean**	**Std Dev**	**Max**	**Static**	**Alignment (%)**
combined	82.21	82.79	162	69	53.22
compressed	81.00	80.91	159	74	51.51
docs	155.50	43.23	158	203	5.89
gap	81.93	83.05	162	71	52.81
graphic	80.93	81.97	160	75	52.53
jpeg	159.21	44.25	162	207	6.23
log	80.21	78.64	156	77	50.14
mp3	157.36	43.74	160	205	6.10
mpeg	159.21	44.25	162	207	6.23
pdf	156.43	43.48	159	204	6.03
program	82.36	82.66	162	73	53.01
random	80.00	79.83	157	76	51.30
reuters	156.43	43.48	159	204	6.03
source	81.00	82.90	161	72	53.15
xml	149.93	41.63	152	197	5.48
Callsites (Vector Length)					1464
Choices with `Yes` Consensus					0 Full, 0 FDO
Choices with `No` Consensus					779 Full, 835 FDO
Choices without Consensus					685 Full, 629 FDO

Table 4.22: Inlining metric scores for `bzip2` on the Itanium

Input	**Mean**	**Std Dev**	**Max**	**Static**	**Alignment (%)**
docs	155.00	69.42	158	203	11.45
jpeg	158.33	70.89	162	207	12.12
mp3	156.67	70.16	160	205	11.85
mpeg	158.33	70.89	162	207	12.12
pdf	155.83	69.79	159	204	11.72
reuters	155.83	69.79	159	204	11.72
xml	150.00	67.10	152	197	10.65
Callsites (Vector Length)					1464
Choices with `Yes` Consensus					0 Full, 0 FDO
Choices with `No` Consensus					793 Full, 919 FDO
Choices without Consensus					671 Full, 545 FDO

Table 4.23: Inlining metric scores for `bzip2` low cut group on the Itanium

Input	**Mean**	**Std Dev**	**Max**	**Static**	**Alignment (%)**
combined	5.43	3.63	10	69	91.74
compressed	6.29	3.03	9	74	88.78
gap	4.57	3.23	9	70	92.56
graphic	4.86	2.66	7	75	90.55
log	7.71	3.44	10	77	86.42
program	5.71	3.46	9	73	91.38
random	6.00	2.71	7	76	88.43
source	4.00	2.45	7	72	91.62
Callsites (Vector Length)					183
Choices with `Yes` Consensus					58 Full, 69 FDO
Choices with `No` Consensus					43 Full, 99 FDO
Choices without Consensus					82 Full, 15 FDO

Table 4.24: Inlining metric scores for `bzip2` high cut group on the Itanium

Input	Mean	Std Dev	Max	Static	Alignment (%)
combined	79.50	23.08	88	121	3.99
compressed	86.14	78.69	159	74	48.70
docs	150.50	46.01	158	203	6.23
gap	87.50	80.60	162	71	49.86
graphic	86.07	79.85	160	75	49.71
jpeg	154.21	46.91	162	207	6.59
log	84.79	76.77	156	77	47.54
mp3	154.21	46.91	162	207	6.59
mpeg	154.21	46.91	162	207	6.59
pdf	153.29	46.69	161	206	6.52
program	87.64	80.46	162	73	50.14
random	84.86	77.79	157	76	48.55
reuters	151.43	46.24	159	204	6.38
source	86.43	80.59	161	72	50.22
xml	144.93	44.62	152	197	5.80
Callsites (Vector Length)				1464	
Choices with `Yes` Consensus				0 Full, 0 FDO	
Choices with `No` Consensus				774 Full, 830 FDO	
Choices without Consensus				690 Full, 634 FDO	

Table 4.25: Inlining metric scores for `bzip2` on the Itanium 2

still no callsite that all logs agree should be inlined. The low cut group logs contain an order of magnitude more callsites than the logs of the high cut group. Nonetheless, all FDO logs contain between 82 and 93 positive inlining decisions. Therefore, training on inputs in the low cut group must result in the repeated inlining of callsites in inlined code. Each callsite in an inlined callee creates a new callsite in the logs. In order to increase the number of callsites in the logs from 183 to 1464, this situation must have occurred very frequently. Since the logs in the low cut group do not agree on which callsites should be inlined, they must represent decisions to inline different call chains. Consequently, training on different input in this group must result in different hot sections of code. Thus, training on different inputs from the low cut group results in significantly different inlining decisions, and are thus well suited to our study.

The results of cutting the workload for the Itanium 2 generates very similar results to those discussed above for the Itanium. However, the combined input results in significantly different results. First, on the Itanium combined is in the high cut group, but is in the low cut group on the Itanium 2. Furthermore, training on combined results in inlining only 9 callsites. Consequently, combined's mean difference score of 85.43 is approximately the mean number of inlined callsites in the other logs, while its alignment score of 1.71% reflect nearly complete disagreement with the other inlining logs.

In Tables 4.26 and 4.27 the consensus information indicates that FDO reduces the amount of inlining performed for `crafty`. The number of choices without consensus indicates that there are about 200 callsites where static optimization makes a different decision than all the FDO logs. These differences are explained by the number of callsites with `No` consensus: the FDO logs agree to not

Input	**Mean**	**Std Dev**	**Max**	**Static**	**Alignment (%)**
ref	17.00	11.97	26	230	86.96
test	33.33	16.63	44	244	81.39
train	16.67	12.14	26	228	87.42
wac-001	22.67	13.28	40	230	86.91
wac-051	27.33	15.06	44	234	86.17
wac-151	22.50	11.85	33	239	86.17
wac-251	18.50	11.54	31	229	86.96
Callsites (Vector Length)				891	
Choices with `Yes` Consensus				198 Full, 204 FDO	
Choices with `No` Consensus				429 Full, 629 FDO	
Choices without Consensus				264 Full, 58 FDO	

Table 4.26: Inlining metric scores for `crafty` on the Itanium

Input	**Mean**	**Std Dev**	**Max**	**Static**	**Alignment (%)**
ref	20.33	10.72	26	230	86.60
test	34.67	16.79	44	244	81.29
train	25.00	12.30	34	234	85.11
wac-001	23.33	13.44	40	230	86.92
wac-051	27.00	15.00	44	234	86.32
wac-151	24.17	12.27	33	239	86.04
wac-251	20.17	11.17	31	229	86.83
Callsites (Vector Length)				891	
Choices with `Yes` Consensus				196 Full, 202 FDO	
Choices with `No` Consensus				428 Full, 626 FDO	
Choices without Consensus				267 Full, 63 FDO	

Table 4.27: Inlining metric scores for `crafty` on the Itanium 2

Input	Mean	Std Dev	Max	Static	Alignment (%)
ref	178.33	155.40	554	960	60.38
snf1025	120.56	166.11	499	1019	59.71
snf1150	120.56	165.96	499	1017	59.34
snf1260	120.56	165.96	499	1017	59.34
snf200-300	143.67	165.31	541	1049	58.06
snf525	125.67	167.40	517	1033	60.86
snf750	122.56	167.68	504	1024	59.59
snf900	535.00	193.81	602	536	81.03
test	210.33	170.22	602	1014	62.19
train	198.56	171.63	600	1014	60.86
Callsites (Vector Length)				4366	
Choices with Yes Consensus				223 Full, 225 FDO	
Choices with No Consensus				2986 Full, 3445 FDO	
Choices without Consensus				1157 Full, 696 FDO	

Table 4.28: Inlining metric scores for GAP on the Itanium

Input	Mean	Std Dev	Max	Static	Alignment (%)
ref	131.00	49.75	167	960	66.71
snf1025	65.22	82.14	189	1019	66.78
snf1150	65.44	81.67	189	1017	66.34
snf1260	65.44	81.67	189	1017	66.34
snf200-300	90.11	60.82	167	1049	64.76
snf525	70.56	74.04	179	1033	68.03
snf750	67.00	83.05	192	1024	66.67
snf900	65.44	82.60	190	1020	66.86
test	164.56	73.13	192	1014	68.60
train	151.89	69.72	182	1014	67.19
Callsites (Vector Length)				4366	
Choices with Yes Consensus				223 Full, 225 FDO	
Choices with No Consensus				2993 Full, 3894 FDO	
Choices without Consensus				1150 Full, 247 FDO	

Table 4.29: Inlining metric scores for GAP on the Itanium 2

inline about 200 callsites that are inlined by static optimization. While the maximum difference between FDO logs is moderate, the mean difference scores are large compared to the maximum, indicating that training on different inputs results in different optimization strategies.

With GAP, there are a large number of callsites without consensus. Furthermore, mean differences are quite large, and the maximum differences between FDO logs approach the total number of choices without consensus. On the Itanium, the snf900 log has a much larger difference and alignment values than the other logs. Training on snf900 results in more inlining than training on the other inputs: Other logs inline between 297 and 405 callsites, but the snf900 log inlines 820 callsites. Unlike the results of if conversion, training on different snf inputs does cause different inlining decisions to be made.

There are a small number of differences between most of the FDO logs for gzip. However, on the Itanium, the log for docs has a mean difference score more than four times larger than any other

Input	Mean	Std Dev	Max	Static	Alignment (%)
combined	14.79	24.07	96	108	78.38
compressed	15.00	24.38	97	109	77.25
docs	91.79	25.92	102	80	83.73
gap	12.21	22.08	86	104	69.74
graphic	13.36	23.27	92	108	72.24
jpeg	15.00	24.38	97	109	77.25
log	12.50	22.65	88	104	67.69
mp3	13.57	23.43	93	109	71.10
mpeg	12.50	22.71	90	106	71.22
pdf	12.50	22.65	88	104	67.69
program	12.29	22.41	87	103	68.83
random	21.36	24.10	102	108	66.21
reuters	12.50	22.65	88	104	67.69
source	14.64	23.25	94	106	74.97
xml	12.29	22.41	87	103	68.83
Callsites (Vector Length)					246
Choices with `Yes` Consensus					18 Full, 31 FDO
Choices with `No` Consensus					80 Full, 109 FDO
Choices without Consensus					148 Full, 106 FDO

Table 4.30: Inlining metric scores for `gzip` on the Itanium

Input	Mean	Std Dev	Max	Static	Alignment (%)
combined	7.86	5.05	14	108	87.97
compressed	8.07	5.36	13	109	86.72
docs	6.79	5.83	17	103	76.19
gap	6.71	5.49	18	104	77.19
graphic	7.14	4.43	18	108	80.58
jpeg	8.07	5.36	13	109	86.72
log	7.14	5.81	16	104	74.81
mp3	7.50	4.05	17	109	79.20
mpeg	6.57	3.15	14	106	79.20
pdf	7.86	5.05	14	108	87.97
program	6.79	5.83	17	103	76.19
random	15.14	5.00	18	108	73.93
reuters	7.14	5.81	16	104	74.81
source	8.43	2.92	12	106	83.58
xml	6.79	5.83	17	103	76.19
Callsites (Vector Length)					246
Choices with `Yes` Consensus					18 Full, 33 FDO
Choices with `No` Consensus					110 Full, 192 FDO
Choices without Consensus					118 Full, 21 FDO

Table 4.31: Inlining metric scores for `gzip` on the Itanium 2

Input	Mean	Std Dev	Max	Static	Alignment (%)
ref	0.58	1.45	5	9	95.22
synth-0	0.58	1.45	5	9	95.22
synth-1	0.58	1.45	5	9	95.22
synth-2	0.58	1.45	5	9	95.22
synth-3	0.58	1.45	5	9	95.22
synth-4	1.50	1.52	6	10	89.57
synth-5	0.58	1.45	5	9	95.22
synth-6	0.58	1.45	5	9	95.22
synth-7	0.58	1.45	5	9	95.22
synth-8	0.58	1.45	5	9	95.22
synth-9	0.58	1.45	5	9	95.22
test	5.00	1.57	6	12	92.17
train	1.33	0.97	4	10	96.09
Callsites (Vector Length)			32		
Choices with Yes Consensus			11 Full, 14 FDO		
Choices with No Consensus			7 Full, 12 FDO		
Choices without Consensus			14 Full, 6 FDO		

Table 4.32: Inlining metric scores for MCF on the Itanium

input, as well as an elevated alignment score. Training on docs results in inlining 136 callsites on the Itanium, while training on the other inputs results in only 42-55 inlined callsites. Conversely, random also has larger than average difference scores on both processors. However, random also has the lowest alignment score in both cases. Therefore, while training on random results in about the same quantity of inlining as training on other inputs, the inlining decisions that are made are significantly different than when other training inputs are used. This result is not surprising: random data has no structure and, in general, cannot be compressed. Thus, it is unlikely that training on random data will exercise any of the paths in the code that perform compression.

As with `if` conversion, there are virtually no differences between the inlining logs for MCF. Tables 4.32 and 4.33 show that the FDO logs had different decisions for no more than 6 callsites, while the average difference between logs is less than 1 different decision. Therefore, unless inlining this single callsite is a key factor for performance, MCF will likely achieve the same levels of performance regardless of which training input is used.

With `parser`, mean difference scores are large, and there are a significant number of callsites without consensus. Therefore, training on different inputs does result in different inlining decisions for `parser`. Alice on the Itanium has a larger difference score than the other logs, and a much lower alignment score. Training on alice likely results in about half as much inlining as training on other inputs for the Itanium.

There are virtually no differences between the FDO inlining logs for the placement task of VPR. Of the 877 callsites in the program, training on different inputs results in different decisions for at most 4 callsites. However, the differences between the FDO logs and the static log are large. The consensus data shows that static optimization inlines 457 callsites that not inlined in any of the FDO

Input	Mean	Std Dev	Max	Static	Alignment (%)
ref	0.58	1.45	5	9	95.22
synth-0	0.58	1.45	5	9	95.22
synth-1	0.58	1.45	5	9	95.22
synth-2	0.58	1.45	5	9	95.22
synth-3	0.58	1.45	5	9	95.22
synth-4	1.50	1.52	6	10	89.57
synth-5	0.58	1.45	5	9	95.22
synth-6	0.58	1.45	5	9	95.22
synth-7	0.58	1.45	5	9	95.22
synth-8	0.58	1.45	5	9	95.22
synth-9	0.58	1.45	5	9	95.22
test	5.00	1.57	6	12	92.17
train	1.33	0.97	4	10	96.09
Callsites (Vector Length)					32
Choices with `Yes` Consensus					11 Full, 14 FDO
Choices with `No` Consensus					7 Full, 12 FDO
Choices without Consensus					14 Full, 6 FDO

Table 4.33: Inlining metric scores for `MCF` on the Itanium 2

Input	Mean	Std Dev	Max	Static	Alignment (%)
02-05words	164.09	68.25	228	385	52.53
06-10words	137.36	59.10	202	383	58.28
11-15words	126.45	60.16	213	350	74.70
16-20words	119.00	59.36	197	356	75.05
21-25words	133.91	53.36	186	355	71.85
alice	191.36	68.20	260	427	38.69
pa	159.36	67.12	228	367	75.23
ref	120.64	56.88	198	369	77.60
relativity	128.82	51.92	179	386	68.29
test	161.55	64.08	260	357	76.02
train	125.18	60.51	213	372	72.64
worlds	122.27	56.11	189	376	74.66
Callsites (Vector Length)					1186
Choices with `Yes` Consensus					62 Full, 82 FDO
Choices with `No` Consensus					542 Full, 714 FDO
Choices without Consensus					582 Full, 390 FDO

Table 4.34: Inlining metric scores for `parser` on the Itanium

Input	Mean	Std Dev	Max	Static	Alignment (%)
02-05words	178.73	72.70	245	414	67.12
06-10words	129.45	56.35	196	399	72.19
11-15words	114.55	51.74	180	353	81.77
16-20words	109.64	55.04	189	361	81.06
21-25words	118.55	61.58	229	355	82.80
alice	129.09	61.84	198	395	68.46
pa	148.55	69.67	245	367	75.42
ref	110.91	52.25	195	369	79.83
relativity	117.64	54.58	182	386	70.28
test	148.73	56.12	198	357	78.71
train	113.64	52.35	185	372	76.40
worlds	111.09	58.47	206	376	76.38
Callsites (Vector Length)				1186	
Choices with `Yes` Consensus				111 Full, 152 FDO	
Choices with `No` Consensus				519 Full, 688 FDO	
Choices without Consensus				556 Full, 346 FDO	

Table 4.35: Inlining metric scores for `parser` on the Itanium 2

Input	Mean	Std Dev	Max	Static	Alignment (%)
alu4	1.14	1.30	3	472	71.77
apex2	1.14	1.30	3	472	71.77
apex4	1.14	1.30	3	472	71.77
bigkey	1.14	1.30	3	472	71.77
des	1.14	1.30	3	472	71.77
diffeq	1.14	1.30	3	472	71.77
dsip	1.81	1.56	4	471	72.08
elliptic	1.43	0.68	2	471	70.93
ex1010	2.38	1.56	4	469	68.99
ex5p	1.14	1.30	3	472	71.77
frisc	1.43	0.68	2	471	70.93
misex3	1.14	1.30	3	472	71.77
pdc	2.38	1.56	4	469	68.99
ref	2.38	1.56	4	469	68.99
s298	1.81	1.56	4	471	72.08
s38417	2.38	1.56	4	469	68.99
s38584.1	2.38	1.56	4	469	68.99
seq	1.14	1.30	3	472	71.77
spla	2.38	1.56	4	469	68.99
test	1.81	1.56	4	471	72.08
train	1.14	1.30	3	472	71.77
tseng	1.81	1.56	4	471	72.08
Callsites (Vector Length)				877	
Choices with `Yes` Consensus				39 Full, 50 FDO	
Choices with `No` Consensus				366 Full, 823 FDO	
Choices without Consensus				472 Full, 4 FDO	

Table 4.36: Inlining metric scores for VPR (place) on the Itanium

Input	Mean	Std Dev	Max	Static	Alignment (%)
alu4	1.14	1.30	3	472	71.77
apex2	1.14	1.30	3	472	71.77
apex4	1.14	1.30	3	472	71.77
bigkey	1.14	1.30	3	472	71.77
des	1.14	1.30	3	472	71.77
diffeq	1.14	1.30	3	472	71.77
dsip	1.81	1.56	4	471	72.08
elliptic	1.43	0.68	2	471	70.93
ex1010	2.38	1.56	4	469	68.99
ex5p	1.14	1.30	3	472	71.77
frisc	1.43	0.68	2	471	70.93
misex3	1.14	1.30	3	472	71.77
pdc	2.38	1.56	4	469	68.99
ref	2.38	1.56	4	469	68.99
s298	1.81	1.56	4	471	72.08
s38417	2.38	1.56	4	469	68.99
s38584.1	2.38	1.56	4	469	68.99
seq	1.14	1.30	3	472	71.77
spla	2.38	1.56	4	469	68.99
test	1.81	1.56	4	471	72.08
train	1.14	1.30	3	472	71.77
tseng	1.81	1.56	4	471	72.08
Callsites (Vector Length)					877
Choices with Yes Consensus					39 Full, 50 FDO
Choices with No Consensus					366 Full, 823 FDO
Choices without Consensus					472 Full, 4 FDO

Table 4.37: Inlining metric scores for VPR (place) on the Itanium 2

Input	Mean	Std Dev	Max	Static	Alignment (%)
alu4	9.86	12.54	36	456	80.28
apex2	9.86	12.54	36	456	80.28
apex4	9.76	12.87	37	457	79.90
bigkey	15.57	6.77	28	458	82.16
des	12.05	12.95	41	455	77.51
diffeq	12.52	9.79	31	459	81.25
dsip	30.33	10.75	41	452	84.06
elliptic	9.76	12.87	37	457	79.90
ex1010	9.86	12.54	36	456	80.28
ex5p	12.43	12.33	39	451	76.65
frisc	11.29	11.90	37	453	77.82
misex3	9.76	12.87	37	457	79.90
pdc	26.62	10.01	45	447	69.96
ref	14.14	12.56	41	451	76.02
s298	33.48	12.51	45	450	84.55
s38417	9.86	12.54	36	456	80.28
s38584.1	9.86	12.54	36	456	80.28
seq	9.76	12.87	37	457	79.90
spla	13.86	12.39	41	451	75.33
test	32.33	11.80	42	451	83.75
train	9.76	12.87	37	457	79.90
tseng	23.38	7.15	37	458	83.30
Callsites (Vector Length)				877	
Choices with Yes Consensus				66 Full, 87 FDO	
Choices with No Consensus				327 Full, 732 FDO	
Choices without Consensus				484 Full, 58 FDO	

Table 4.38: Inlining metric scores for VPR (route) on the Itanium

Input	Mean	Std Dev	Max	Static	Alignment (%)
alu4	9.86	12.54	36	456	80.28
apex2	9.86	12.54	36	456	80.28
apex4	9.76	12.87	37	457	79.90
bigkey	15.57	6.77	28	458	82.16
des	12.05	12.95	41	455	77.51
diffeq	12.52	9.79	31	459	81.25
dsip	30.33	10.75	41	452	84.06
elliptic	9.76	12.87	37	457	79.90
ex1010	9.86	12.54	36	456	80.28
ex5p	12.43	12.33	39	451	76.65
frisc	11.29	11.90	37	453	77.82
misex3	9.76	12.87	37	457	79.90
pdc	26.62	10.01	45	447	69.96
ref	14.14	12.56	41	451	76.02
s298	33.48	12.51	45	450	84.55
s38417	9.86	12.54	36	456	80.28
s38584.1	9.86	12.54	36	456	80.28
seq	9.76	12.87	37	457	79.90
spla	13.86	12.39	41	451	75.33
test	32.33	11.80	42	451	83.75
train	9.76	12.87	37	457	79.90
tseng	23.38	7.15	37	458	83.30
Callsites (Vector Length)					877
Choices with Yes Consensus					66 Full, 87 FDO
Choices with No Consensus					327 Full, 732 FDO
Choices without Consensus					484 Full, 58 FDO

Table 4.39: Inlining metric scores for VPR (route) on the Itanium 2

logs. On the other hand, FDO results in the inlining of at most 54 callsites. VPR's routing task results in larger variations between the FDO logs. However, the differences between static optimization and FDO inlining are still very large compared to the differences between FDO logs. Therefore, profiling makes a significant difference in the compiler's ability to identify hot callsites in VPR, and, as will be presented in Section 4.2.2, consequently improves program performance.

4.1.3 Conclusions

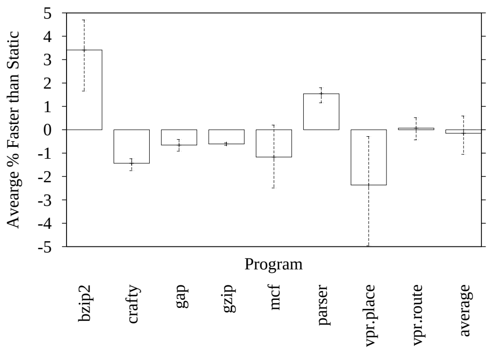
Overall, different training inputs result in different optimization decisions for both `if` conversion and inlining. In almost every case, there are much more significant differences between the static logs and the profile-guided logs than between any pair of FDO logs. Furthermore, the consensus data shows that FDO usually results in the same decisions for the majority of choices, and that in most cases the differences between FDO logs are confined to a fairly small proportion of the choices made for a program. In the extreme, several inputs for MCF and for the placement task of VPRresult in identical optimization decision. Therefore, using or not using FDO appears to have a more significant impact on the optimization decisions made by a compiler than the selection of the training input used for FDO. However, since the performance of a program is often most significantly impacted by a small number of important optimization decisions, these results do not imply that the selection of training input is not important.

4.2 Run-Time Performance

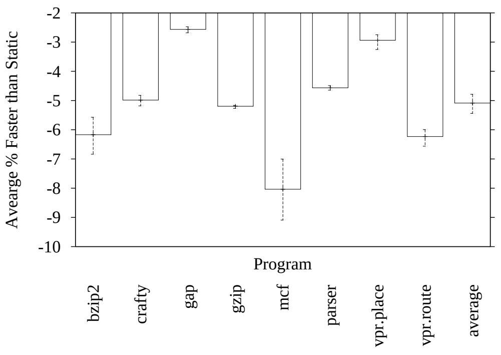
Aestimo measures the run times for the binaries produced with the FDO optimization log from training on each of the inputs in the workload, as well as for the statically optimized binary. Each binary is run on each of the inputs in the workload five times. Unfortunately, a larger number of trials would have taken a prohibitive amount of computing time. Therefore, along with the average performance, *Aestimo* reports results based on the minimum and maximum times from among the five trials. The main bar in the graphs shows the average of the five trials, while the error bars show the minimum and maximum values obtained. Values for the error-bars are determined using identical calculations to those used with the average time. All run times are measured as the user component reported by the UNIX `time` command. All the graphs presenting performance results in this chapter will use these conventions.

Reported performance results may vary depending on the method used to summarize the raw run-time data [23] (pp. 24-39). If the total run times for a workload are compared, a few long-running inputs could dominate the run time and the comparison would effectively ignore shorter-running inputs. Alternatively, the run time for each element of the workload can be normalized. In this case, each input is equally important in the comparison, but this is not always a desirable characteristic.

Therefore, *Aestimo* provides performance results calculated using two methodologies: an arithmetic sum of run times and a geometric sum of run times. The arithmetic sum aggregates the raw



(a) Itanium



(b) Itanium 2

Figure 4.6: Average performance of FDO `if` conversion

run times on each of the inputs in the workload for a given binary. The sum is reported as a percent faster than the same measure for the statically compiled binary. The geometric sum is similar, but it normalizes the run times against the (arithmetic average) static time before aggregating. Precisely, the geometric sum is defined as:

$$G_I = \sum_{j \in W} \frac{time_I(j)}{time_{static}(j)}$$

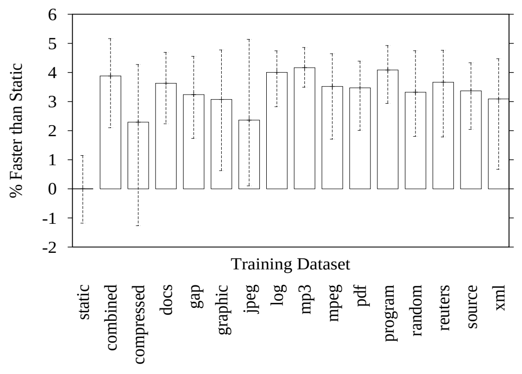
where W is the workload, $I \in W$ is the training input used to create the binary, and $time_I(j), j \in W$, is the time for the binary trained in input I to run on the input i. The results labeled static are included in the graphs to display the variance between the minimum and maximum times for execution with the statically optimized binary.

The metrics referred to in this section are the difference and alignment metrics defined in Section 3.1 and reported in Section 4.1.

4.2.1 `If` conversion

`If` conversion is known to provide modest performance improvements at best [13]. On the Itanium, profile-guided `if` conversion has mixed effects on performance. On the Itanium 2, *Aestimo* finds that profile-guided `if` conversion invariably results in an (often substantial) performance reduction. Apparent performance may vary between the two evaluation metrics, but the conclusions of the performance evaluation are not affected by the run time aggregation metric chosen.

Figure 4.6 presents the average arithmetic performance of the FDO binaries for each program. On the Itanium, FDO `if` conversion results are mixed, but on average FDO `if` conversion has little effect on performance. FDO `if` conversion makes little difference to the `gzip` and VPR routing. `Bzip2` and `parser` are consistently slightly improved by FDO `if` conversion, while it reduces performance on `crafty`, `gzip`, and VPR's placement task. GAP and MCF show small performance


(a) Arithmetic

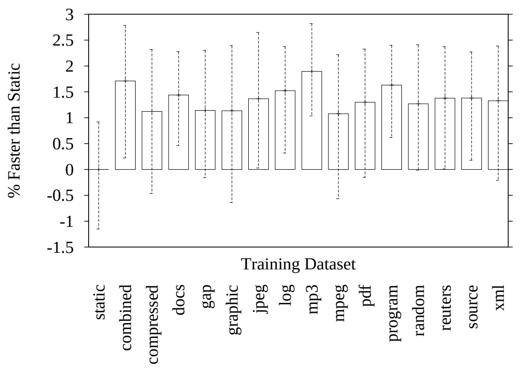
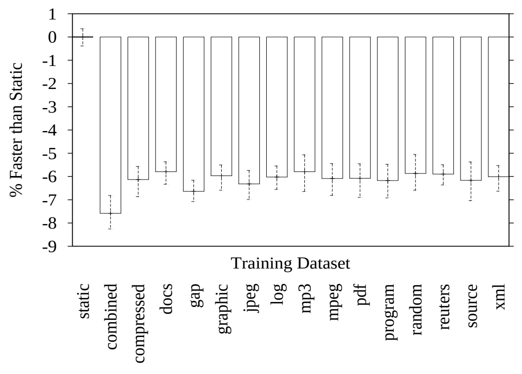

(b) Geometric

Figure 4.7: Performance of `bzip2` with `if` conversion on the Itanium


(a) Arithmetic

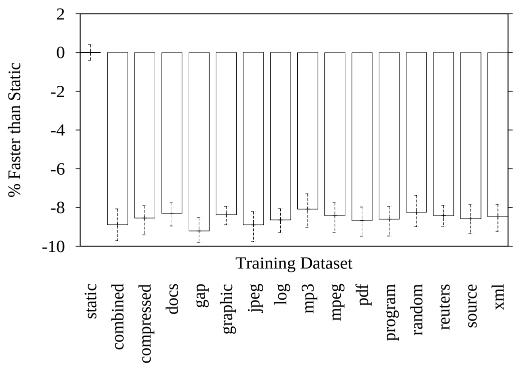
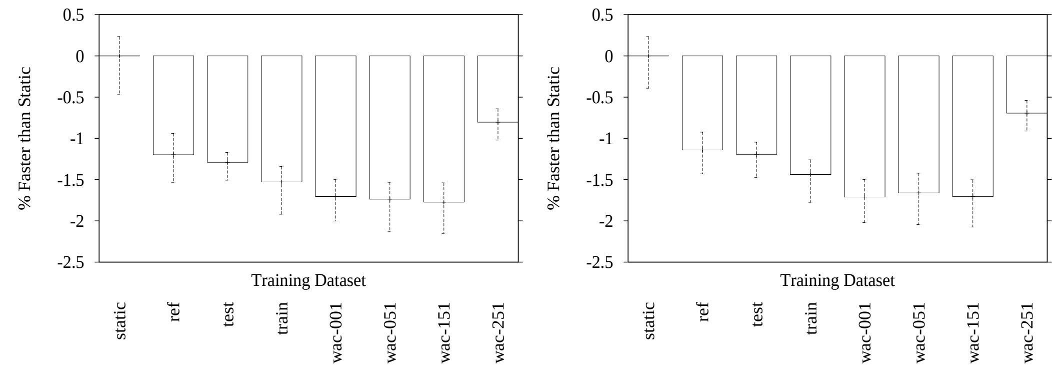

(b) Geometric

Figure 4.8: Performance of `bzip2` with `if` conversion on the Itanium 2

reduction when trained on most inputs, but also have small performance gains when trained on two or three of the inputs in the workload.

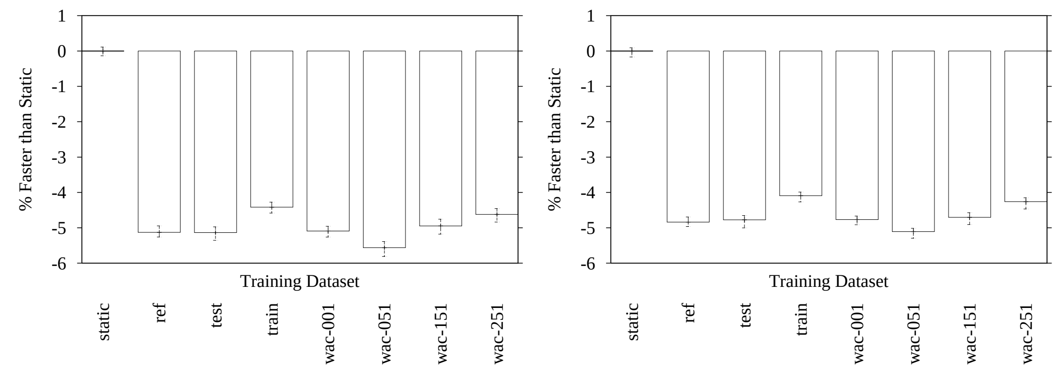
Results on the Itanium 2 are disappointing. FDO is intended to improve performance, but this is clearly not the case with FDO `if` conversion. For nearly every benchmark, performance is reduced uniformly regardless of which training input is used. Furthermore, the average difference score appears to have no correlation with performance. For example, with `parser`, average difference scores range from 28 to 64, but all FDO binaries exhibit identical performance. Perhaps the most important observation of FDO `if` conversion is that performance is nearly always reduced by more than 3%, by 5% on average, and by as much as 8% for `MCF`.

Run time variances for FDO `if` conversion binaries are frequently large, making it impossible to distinguish between the performance of binaries trained on different inputs. Recall that the `if` conversion alignment scores for `bzip2` are split into two groups: inputs with alignment scores less than 55%, and inputs with alignment scores greater than 80%. Despite these differences, Figures 4.7



(a) Arithmetic (b) Geometric

Figure 4.9: Performance of `crafty` with `if` conversion on the Itanium



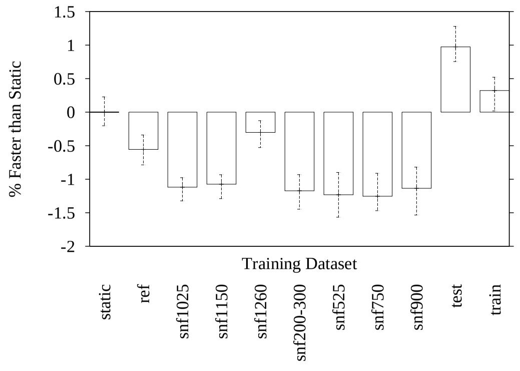
(a) Arithmetic (b) Geometric

Figure 4.10: Performance of `crafty` with `if` conversion on the Itanium 2

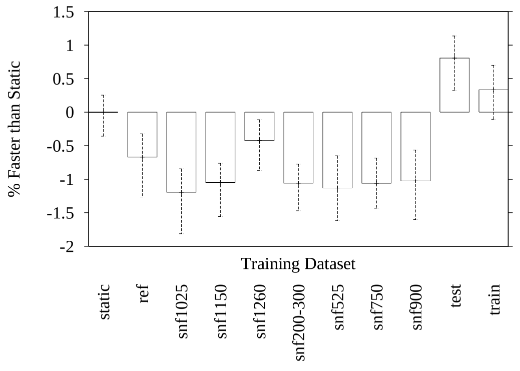
and 4.8 show no significant performance differences between the FDO binaries.

There are distinguishable differences in the performance of several `crafty` binaries on both processors, though the difference between the best and worst performance seen in Figures 4.9 and 4.10 is less than 1%. The average difference metric does not vary substantially between the binaries, expect for wac-251 on the Itanium, which has a score of 405 (see Table 4.6). The other inputs result in scores less than 248. Training on wac-251 also results in the best performance on `crafty`, which suggests that the different profile information provided by this input results in different optimization decisions that do impact performance.

Similarly, there is some correspondence between larger average difference scores and greater performance for GAP on the Itanium. Most inputs result in difference scores less than 200 (see Table 4.8), and performance penalties of about 1%. The SPEC ref, test, and train inputs have average difference scores exceeding 400. Figure 4.11 shows that they also result in the best performance on this benchmark. However, training on the snf1260 input results in better performance than training

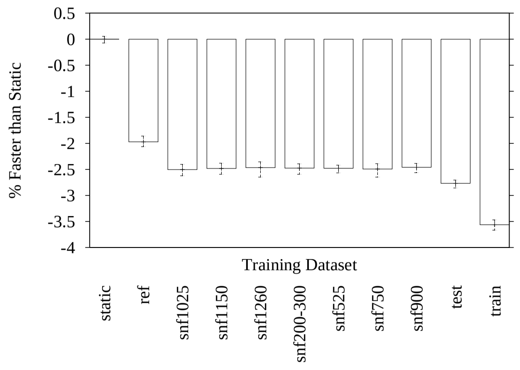


(a) Arithmetic

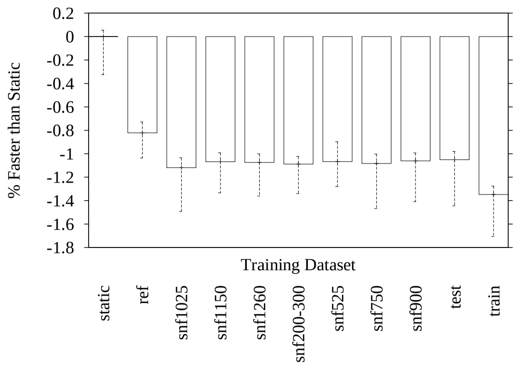


(b) Geometric

Figure 4.11: Performance of `GAP` with `if` conversion on the Itanium

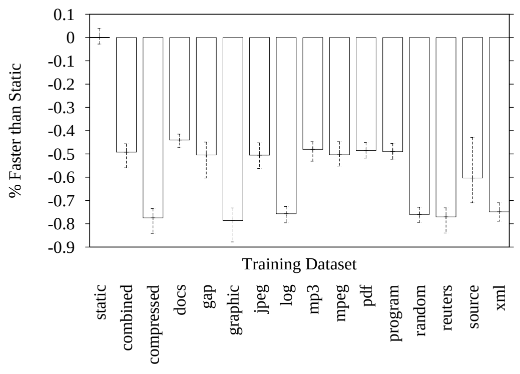


(a) Arithmetic



(b) Geometric

Figure 4.12: Performance of `GAP` with `if` conversion on the Itanium 2


(a) Arithmetic

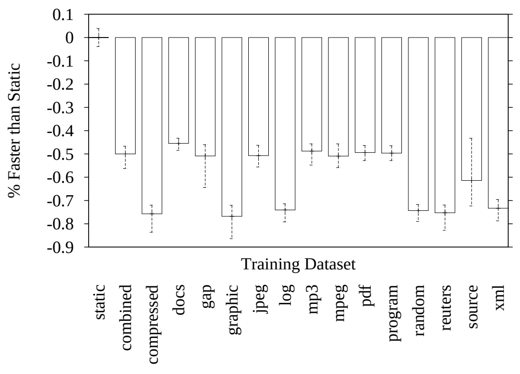
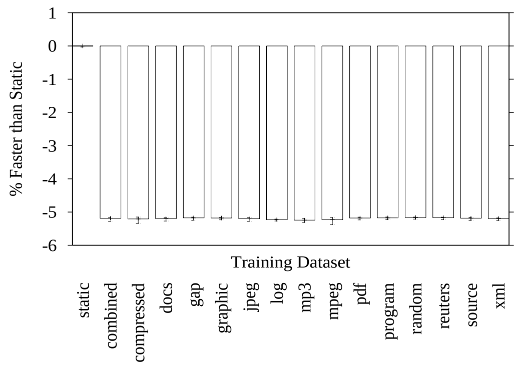

(b) Geometric

Figure 4.13: Performance of `gzip` with `if` conversion on the Itanium


(a) Arithmetic

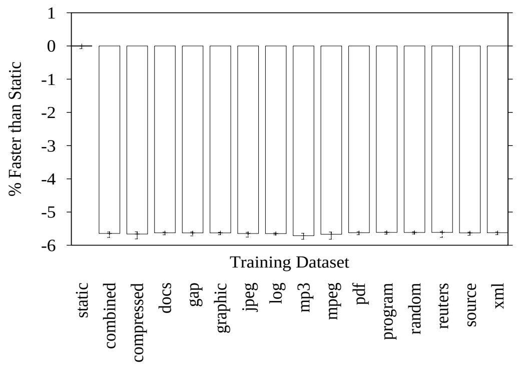
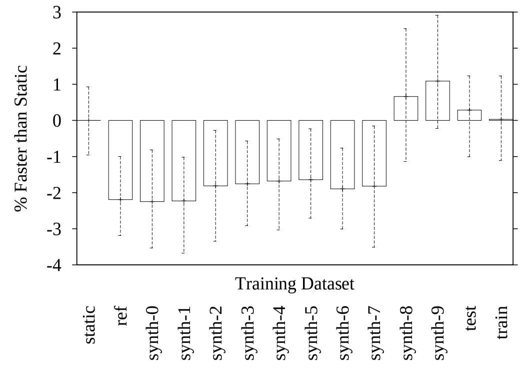

(b) Geometric

Figure 4.14: Performance of `gzip` with `if` conversion on the Itanium 2

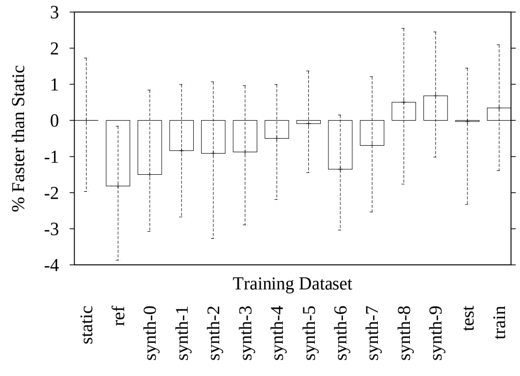
on ref, but results in an average difference score of only 190.

On the other hand, GAP on the Itanium 2 also displays distinguishable levels of performance, as seen in Figure 4.12. In this case, most inputs have an average difference score of about 185, while the SPEC ref, test, and train inputs have average difference scores larger than 400. However, while training on ref results in the best performance among the FDO binaries, training on test and train results in the two worst performing binaries.

`Gzip` and `MCF` on the Itanium are also cases where changes in metric scores do not correlate with changes in performance. With `gzip`, there are two statistically distinct levels of performance in Figure 4.13: 0.5% slower and 0.8% slower than static. Unfortunately, these differences are too small to be practically significant. There are no trends in the metric scores to suggest these two levels of performance. Both the log and docs logs have lower than average alignment scores, but display different levels of performance. The mean difference scores for the binaries that are 0.8% slower than static appear to be higher than average, but many binaries at the -0.5% performance level have

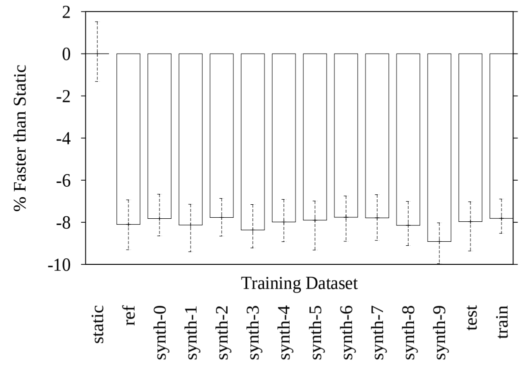


(a) Arithmetic

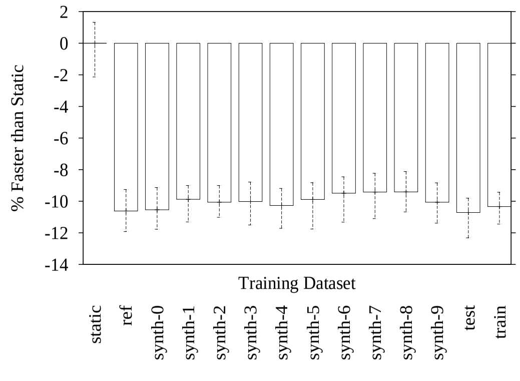


(b) Geometric

Figure 4.15: Performance of `MCF` with `if` conversion on the Itanium

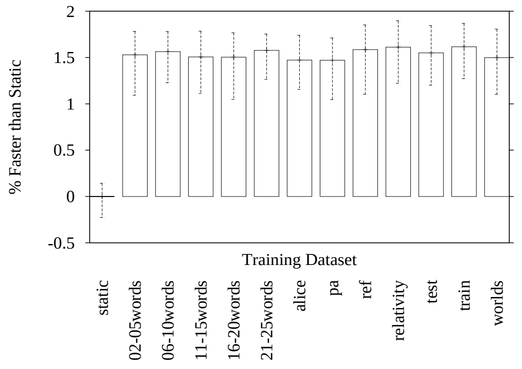


(a) Arithmetic

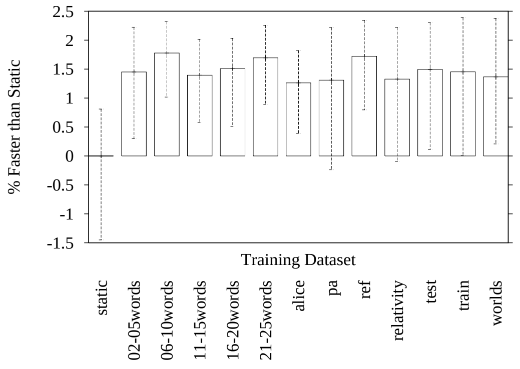


(b) Geometric

Figure 4.16: Performance of `MCF` with `if` conversion on the Itanium 2

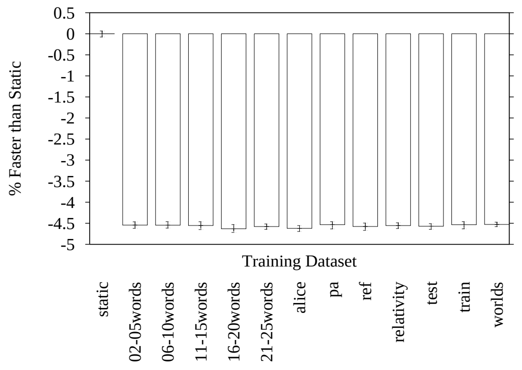


(a) Arithmetic

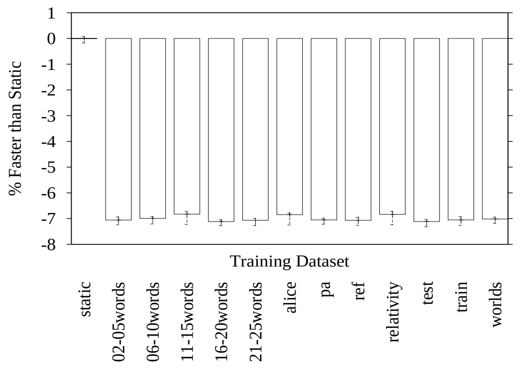


(b) Geometric

Figure 4.17: Performance of `parser` with `if` conversion on the Itanium



(a) Arithmetic



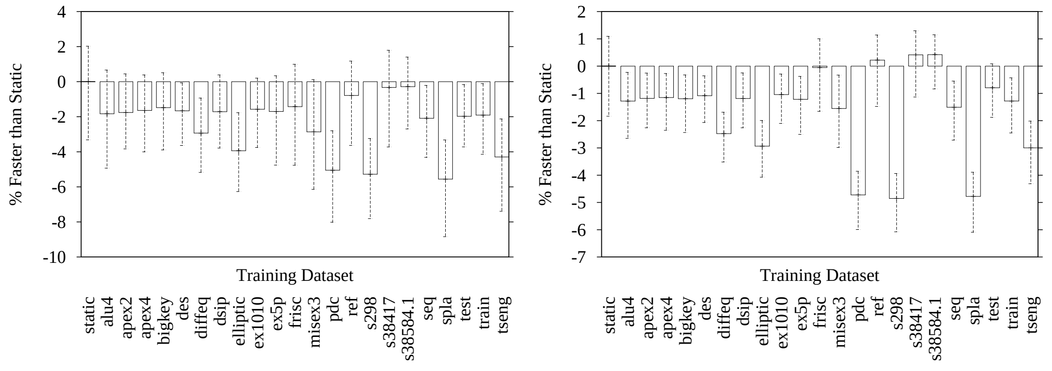
(b) Geometric

Figure 4.18: Performance of `parser` with `if` conversion on the Itanium 2

differences larger than graphic. Furthermore, while all other inputs have mean difference scores less than 20, compressed results in a mean difference score of 44. However, the performance of the binary trained on compressed is indistinguishable from 5 other binaries.

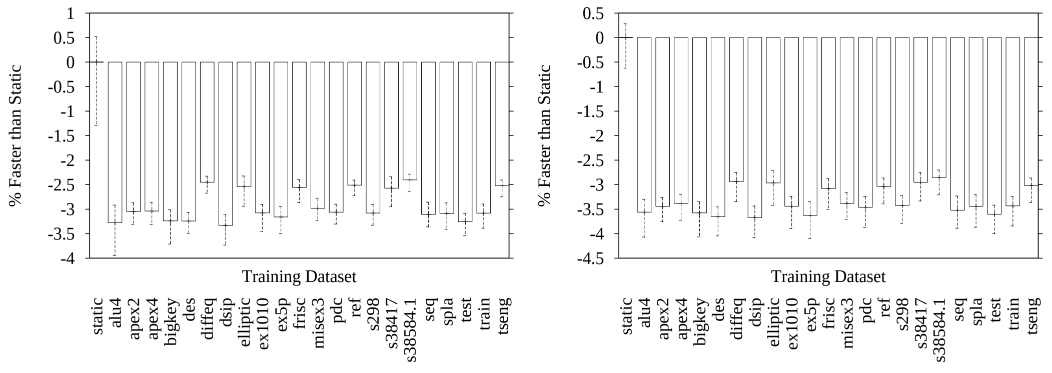
Similarly, synth-5 for `MCF` results in a significantly larger mean difference score and a lower alignment score than the other inputs, but does not display any significant differences in performance. Conversely, training on synth-8, synth-9, test, and train results in improved performance, but they have nearly the same metric scores as the other input. Therefore, there is no clear connection between the difference and alignment scores and performance.

Despite the moderate mean difference scores for `parser`, differences in `if` conversion decisions do not appear to result in any variations in performance among the FDO binaries. Recall that the FDO logs had consensus for about half of the `if` conversion choices for `parser`. Therefore, it is likely that the choices that do not have consensus among the FDO logs are those that are not important for the performance of the program, while those choices with consensus are a superset



(a) Arithmetic (b) Geometric

Figure 4.19: Performance of VPR (place) with `if` conversion on the Itanium

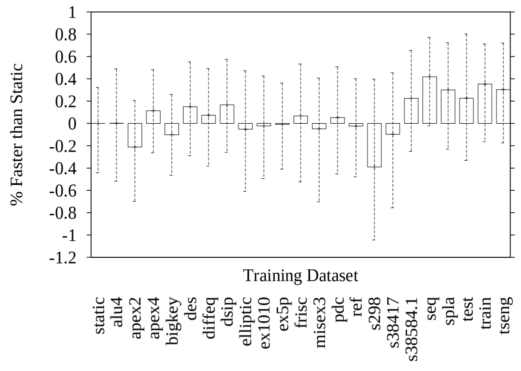


(a) Arithmetic (b) Geometric

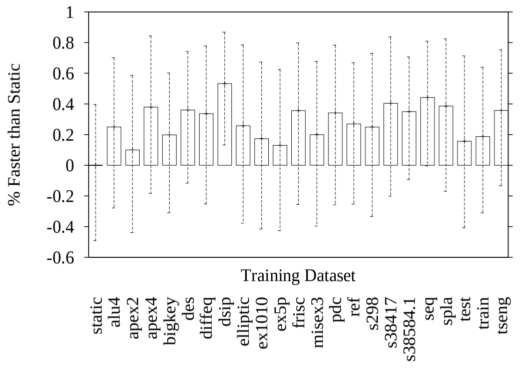
Figure 4.20: Performance of VPR (place) with `if` conversion on the Itanium 2

of the most important `if` conversion choices. The large reduction in performance on the Itanium 2 is likely due to some of the 111 `if` conversion choices that the FDO logs had consensus not to `if` convert, but that static optimization chooses to `if` convert.

The pdc, s298, and spla inputs for the placement task of VPR have the worst performance for this program on the Itanium (see Figure 4.19). These inputs also have the lowest alignment scores. Therefore, there may be a correlation between alignment score and performance in this case. However, on the Itanium 2, there are no inputs with distinguishing metric scores, nor any that result in very significantly different performance. There are no distinguishing features for the inputs for the routing task of VPR, either (see Figures 4.21 and 4.22). The consensus numbers indicate that the main difference between static and feedback-directed optimization is about 50 `if` conversion choices where static performs `if` conversion but FDO does not. This result suggests that not enough `if` conversion is being performed on the Itanium 2 when FDO is used.

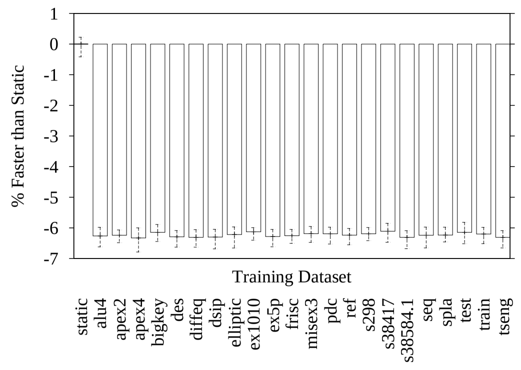


(a) Arithmetic

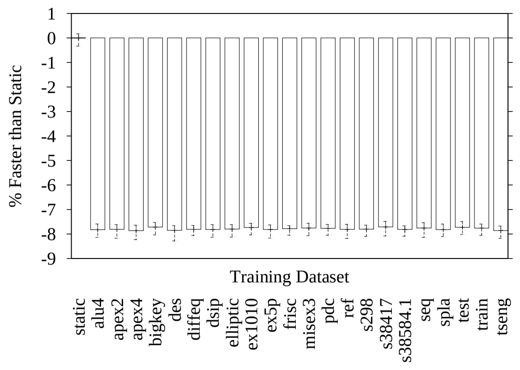


(b) Geometric

Figure 4.21: Performance of VPR (route) with `if` conversion on the Itanium

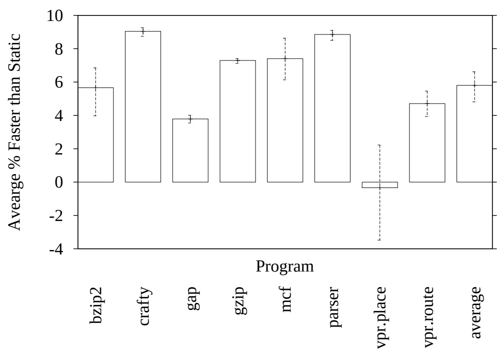


(a) Arithmetic

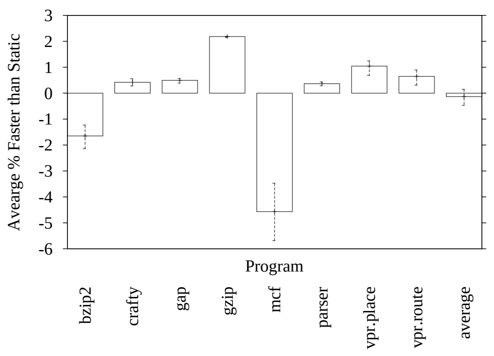


(b) Geometric

Figure 4.22: Performance of VPR (route) with `if` conversion on the Itanium 2



(a) Itanium



(b) Itanium 2

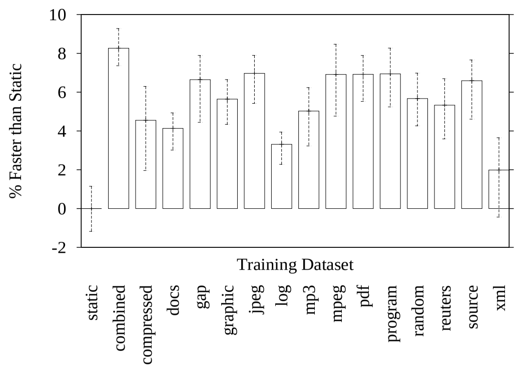
Figure 4.23: Average performance of FDO inlining

4.2.2 Inlining

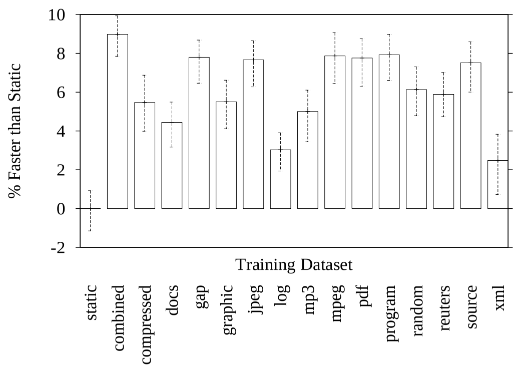
Inlining is an important optimization that can benefit greatly by knowing the hot functions and frequent callsites in a program. Therefore, we expect significant performance gains from profile-guided inlining. However, if different inputs exercise different parts of the code, or otherwise result in different relative frequencies for important functions and callsites, overall performance on the workload may vary.

The experimental results show that there are significant performance impacts from feedback-directed inlining. Furthermore, there are several cases where the selection of training input has a significant and substantial impact on performance. Figure 4.23 shows the average arithmetic performance of FDO inlining on each program. FDO inlining improves performance by 6% on average on the Itanium. However, while FDO inlining has little impact on performance for the Itanium 2 on average, the largest average performance gain is slightly more than 2%, but the largest performance reduction is almost 5%. For individual binaries, FDO inlining improves performance by as much as 12% or reduces it by up to 6%, while the performance difference between training on two inputs can approach 7%.

Figure 4.24 shows performance gains for `bzip2` from profile-guided inlining on the Itanium. Despite the large variances, training on the combined input results in performance gains of about 8%, while training on xml improves performance by only 2%. Figure 4.25 indicates that training input selection is also important for `bzip2` on the Itanium 2. Training on some inputs, such as log or docs, has a negligible impact on performance. However, training on combined reduces performance by over 5%. Recall from Section 4.1.2 that inputs resulted in either difference scores of about 150 and alignment scores of about 6%, or difference scores of about 80 and alignment scores of about 50% for both processors. However, there is no obvious correlation between these scores and performance. For example, training on either mpeg or program results in similar performance

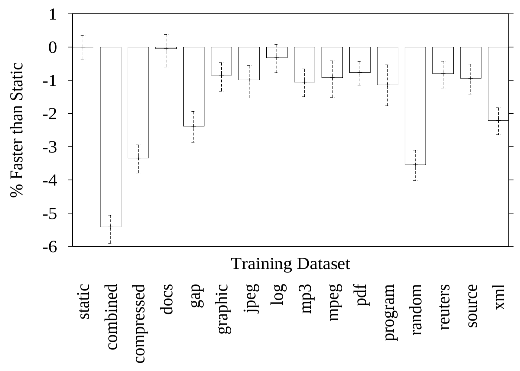


(a) Arithmetic

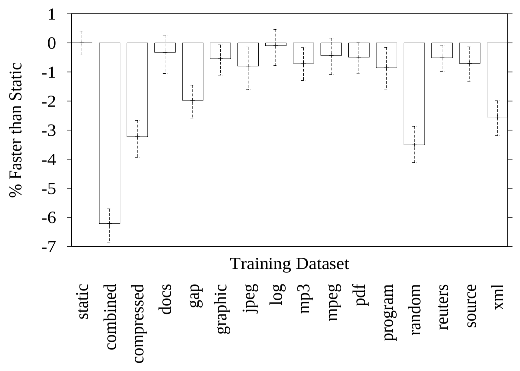


(b) Geometric

Figure 4.24: Performance of `bzip2` with inlining on the Itanium

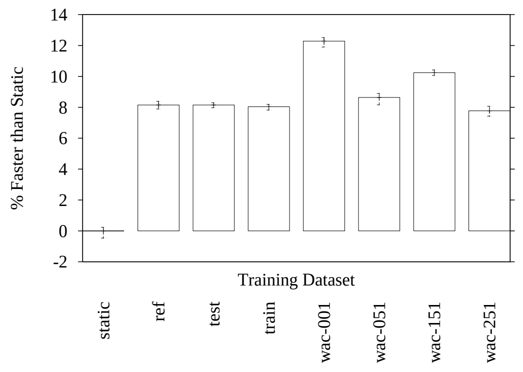


(a) Arithmetic

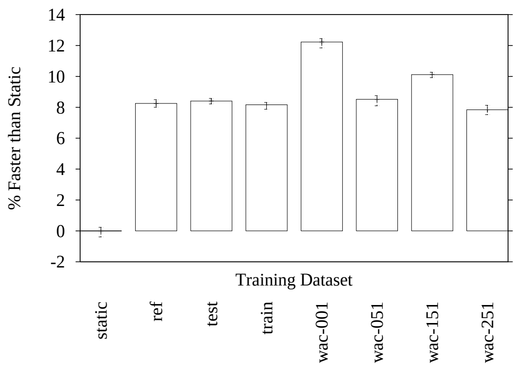


(b) Geometric

Figure 4.25: Performance of `bzip2` with inlining on the Itanium 2

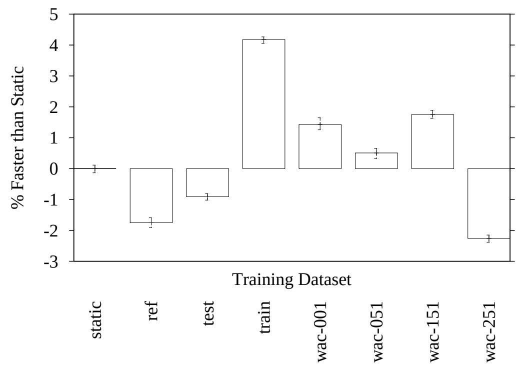


(a) Arithmetic

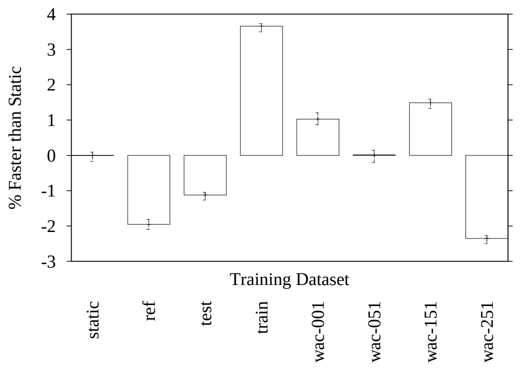


(b) Geometric

Figure 4.26: Performance of `crafty` with inlining on the Itanium



(a) Arithmetic



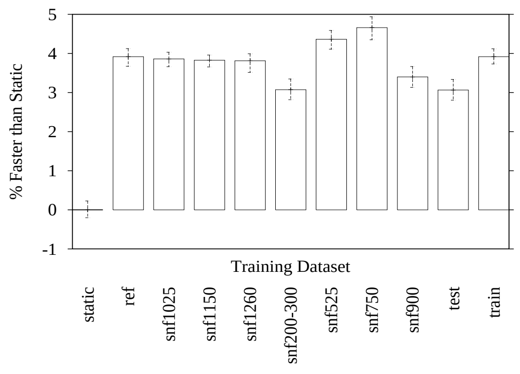
(b) Geometric

Figure 4.27: Performance of `crafty` with inlining on the Itanium 2

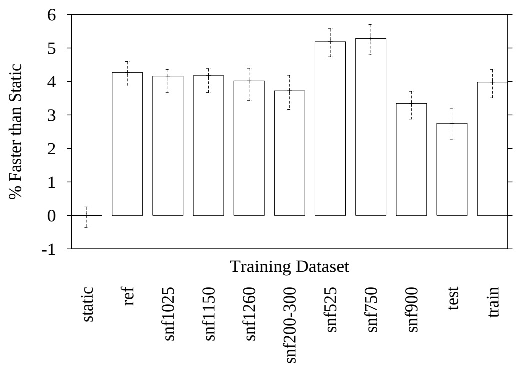
despite mpeg having an alignment score of about 6% and program having an alignment score of about 50%.

`Crafty` on the Itanium achieves the largest performance improvement from profile-guided inlining observed in our study. As shown in Figure 4.26, training on most inputs results in performance improvement of 8%. However, 10% and 12% gains can be achieved by training on wac-151 and wac-001 respectively. However, the metric scores do not distinguish these inputs in any way. There are inputs with both higher and lower mean difference and alignment scores. In particular, test has the largest mean difference score and lowest alignment score, while train has the smallest mean difference score and largest alignment score. Nonetheless, the performance of the binaries trained on these two inputs is identical.

The performance impact of FDO inlining for `crafty` on the Itanium 2 varies significantly with the training input chosen, as seen in Figure 4.27. However, these variations in performance seem uncorrelated to the difference or alignment metrics. Ref and wac-251 have the lowest mean

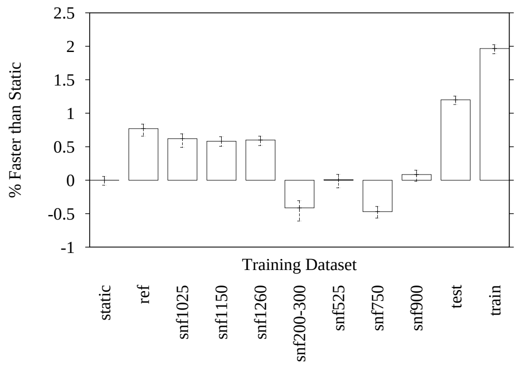


(a) Arithmetic

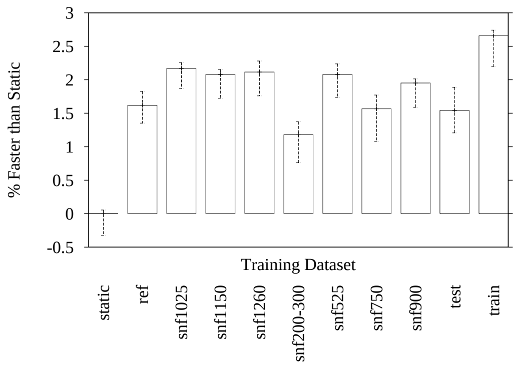


(b) Geometric

Figure 4.28: Performance of GAP with inlining on the Itanium



(a) Arithmetic

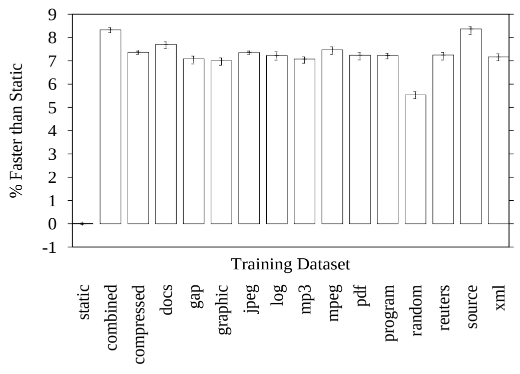


(b) Geometric

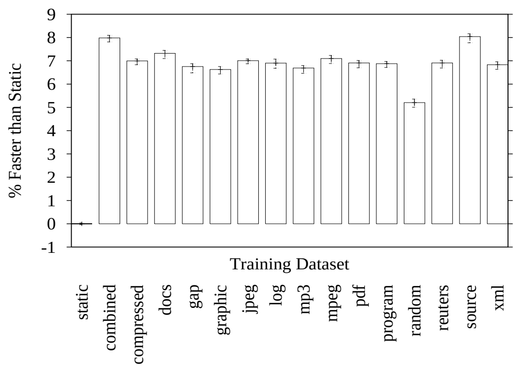
Figure 4.29: Performance of GAP with inlining on the Itanium 2

differences, and also the worst performance. However, test has the largest mean difference, but also results in reduced performance. On the other hand, train improves performance by 4%, but has mean difference and alignment scores that are on neither extreme when compared to the scores of other inputs.

GAP on the Itanium displays small performance variations across the workload. Figure 4.28(a) shows that the least performance improvement is about 3% for test, and the greatest is about 4.5% for snf750. snf750 and snf200-300 have very similar metric scores, but dissimilar performance. The SPEC ref, test, and train inputs all have mean difference scores approximately 50% larger than the other inputs, but have similar levels of performance. The performance impact of inlining for GAP on the Itanium 2 is small. However, for both the snf200-300 and snf750 inputs, the arithmetic measure results in slightly worse performance than static on the workload, while the geometric measure shows a small performance improvement. Training on these two inputs results in a longer total time to process the entire workload, despite an average improvement in per-input

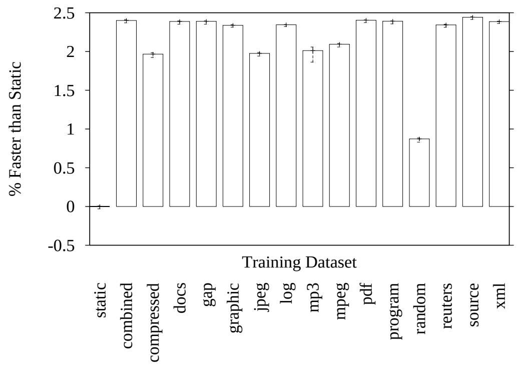


(a) Arithmetic

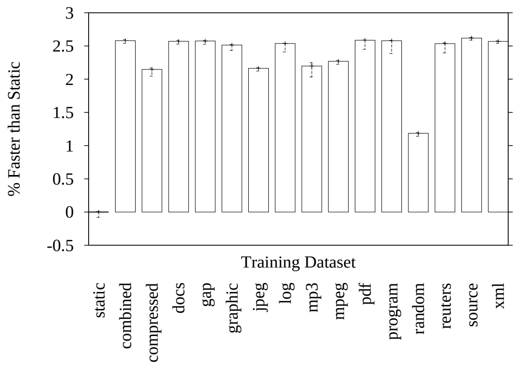


(b) Geometric

Figure 4.30: Performance of `gzip` with inlining on the Itanium



(a) Arithmetic

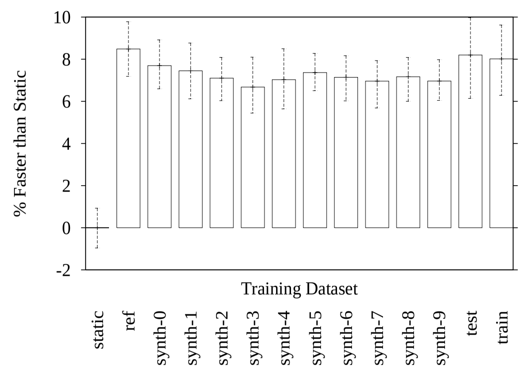


(b) Geometric

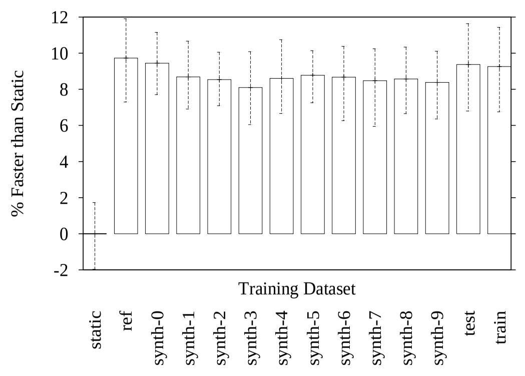
Figure 4.31: Performance of `gzip` with inlining on the Itanium 2

processing time. Furthermore, there is no visible connection between the metric scores and observed performance.

Figure 4.30 shows fairly consistent performance improvements for `gzip` on the Itanium. docs has a very large mean difference score, more than four times larger than any other input. However, this large difference does not correspond to any impact on performance. The performance improvements on the Itanium 2 are small, as shown in Figure 4.31. Training on random produces a markedly smaller improvement in performance than training on the other inputs on both processors. random also has a mean difference score nearly twice as large as any other input on the Itanium 2, and about 50% larger than all inputs except docs on the Itanium. Recall that random's low alignment score was due to it inlining a significantly different set of callsites than other logs. The performance results reveal that these different decisions result in inferior performance on our workload. Therefore, the selection of random as the training input for `gzip` does result in different inlining decisions that do result in different performance than training on other inputs.

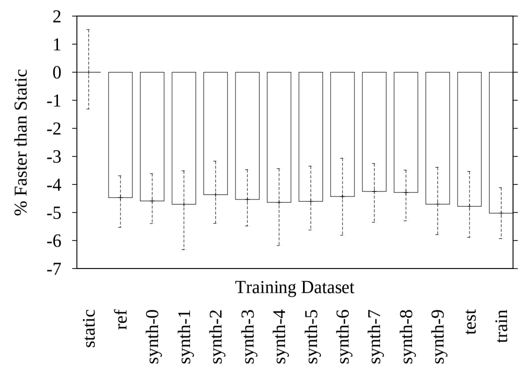


(a) Arithmetic

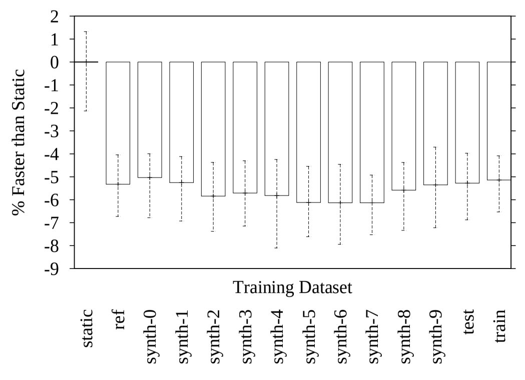


(b) Geometric

Figure 4.32: Performance of MCF with inlining on the Itanium

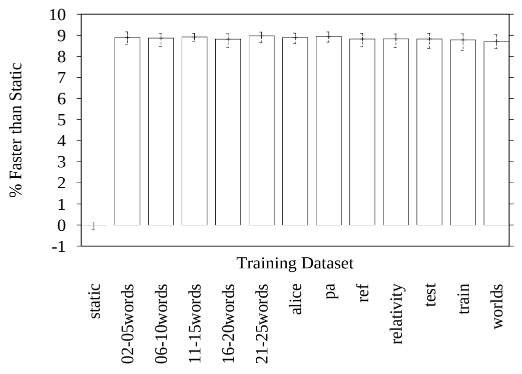


(a) Arithmetic

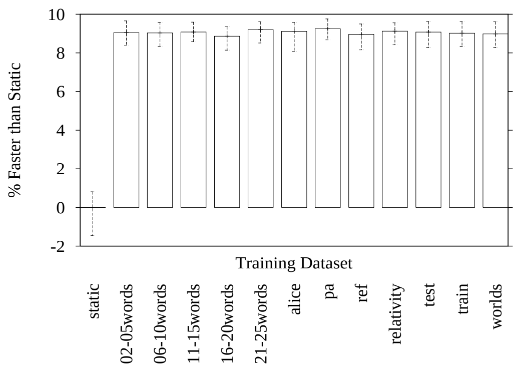


(b) Geometric

Figure 4.33: Performance of MCF with inlining on the Itanium 2

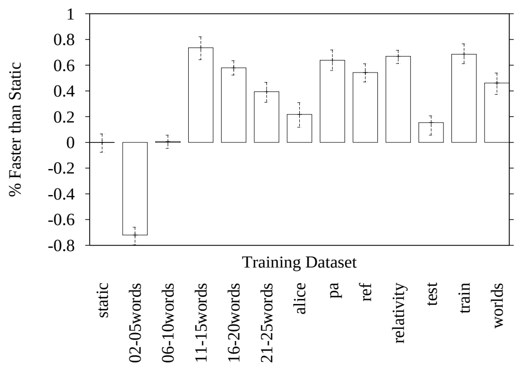


(a) Arithmetic

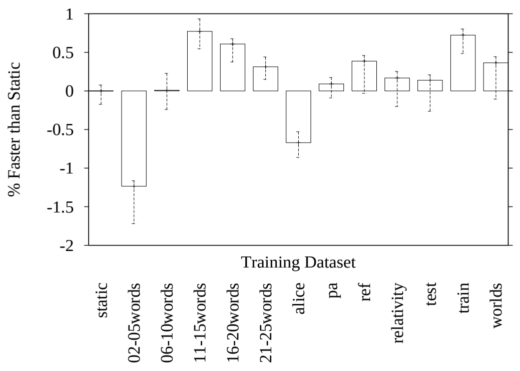


(b) Geometric

Figure 4.34: Performance of `parser` with inlining on the Itanium



(a) Arithmetic



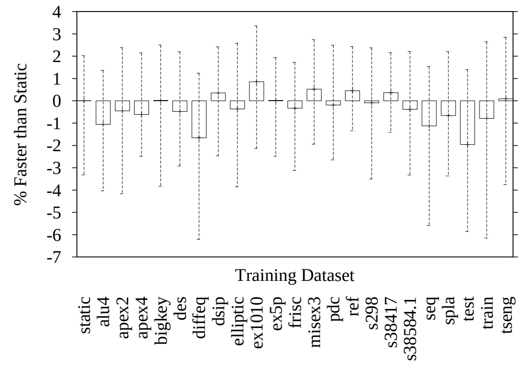
(b) Geometric

Figure 4.35: Performance of `parser` with inlining on the Itanium 2

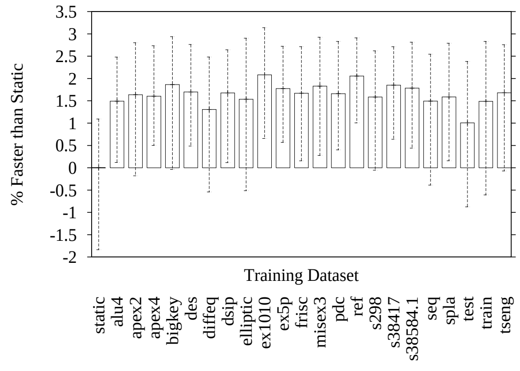
FDO inlining results in equivalent performance regardless of the training input chosen for `MCF` for both processors. Performance is improved about 7-8% on the Itanium, see Figure 4.32, but reduced by 4-5% on the Itanium 2, see Figure 4.32. Since the mean difference scores for all inputs on both processors are very small, these similarities in performance across training inputs are expected. Recall that the metric scores for `MCF` are identical for the same training input on the two processors, and that the inlining decisions are identical as well. Clearly, these decisions are effective at improving performance on the Itanium, but are inappropriate for the Itanium 2.

Figure 4.34 illustrates that the large mean difference scores for `parser` on the Itanium do not impact performance, as performance is improved uniformly by 9%. Furthermore, alice has a much larger mean difference score and significantly lower alignment score than the other inputs, but achieves identical performance. Therefore, the differences between the logs must be for infrequently executed callsites.

In Figure 4.35 different training inputs clearly result in different levels of performance for

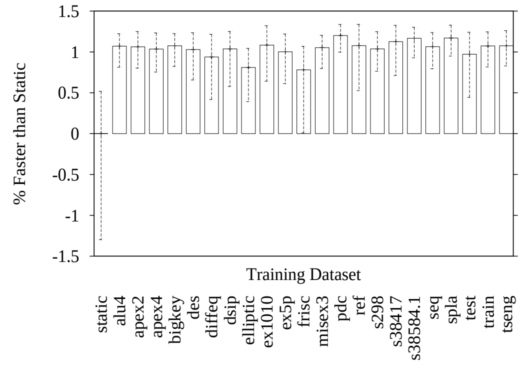


(a) Arithmetic

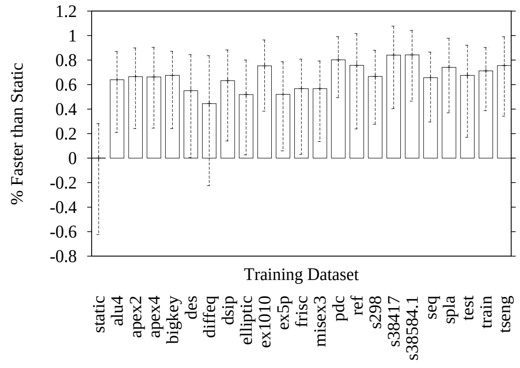


(b) Geometric

Figure 4.36: Performance of VPR (place) with inlining on the Itanium



(a) Arithmetic

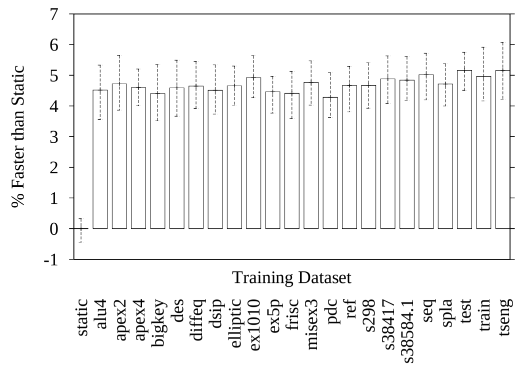


(b) Geometric

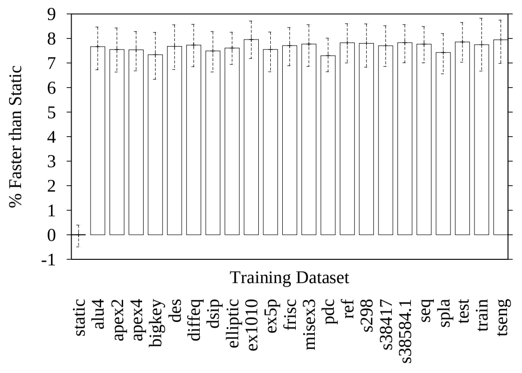
Figure 4.37: Performance of VPR (place) with inlining on the Itanium 2

parser on the Itanium 2. Most inputs achieve a distinct level of performance from the other inputs. Unfortunately, the performance impacts are small, with improvements and reductions all less than 1%, which limits the practical significance of these results. Nonetheless, there does appear to be a weak correlation between alignment and performance. Conversely, there is likely a weak inverse correlation between performance and mean difference score. 02-05words has the worst performance, the highest mean difference score, and the lowest alignment. Meanwhile, 11-15words has the best performance, the second largest alignment score and a low mean difference score. Therefore, the better-performing binaries likely have more inlined callsites than the worse-performing binaries. The effectiveness of training on alice varies between the arithmetic and geometric measures. Unlike the two inputs for GAP, training on alice results in a reduction in processing time for the entire workload, compared to static. Despite this fact, the average per-input performance of parser is reduced compared to static.

FDO inlining usually has a small impact on performance for VPR. The routing component of

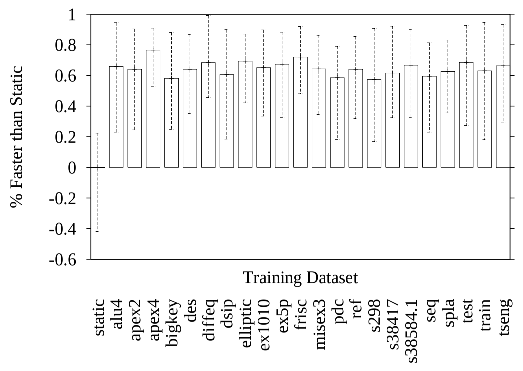


(a) Arithmetic

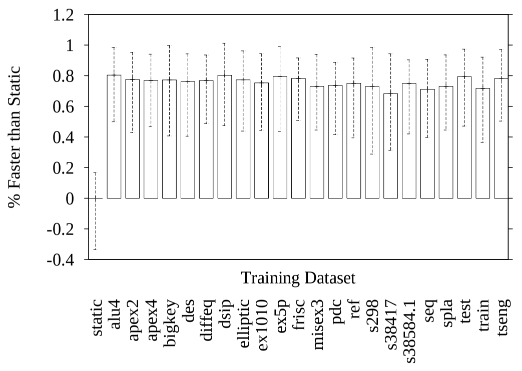


(b) Geometric

Figure 4.38: Performance of VPR (route) with inlining on the Itanium



(a) Arithmetic



(b) Geometric

Figure 4.39: Performance of VPR (route) with inlining on the Itanium 2

VPR on the Itanium is the only exception, where performance is improved by more than 4%. It is not surprising that performance is nearly identical regardless of the training input used. The maximum difference between logs for routing on both processors is 4, indicating that the optimization decisions made during compilation are virtually identical regardless of the training input. The difference scores are a bit larger for the placement component of VPR. However, the scores are still fairly small, and there are no significant differences in performance between the binaries. The benefit of FDO inlining for routing on the Itanium is likely due to a small number of callsites that are easily identified as important using any training input.

4.2.3 Conclusions

The experimental results indicate that training on different inputs does lead to different decisions by the compiler, and that there are often performance differences between binaries trained on different inputs. Ideally, there would be a correlation between the alignment and/or difference metrics and performance. Visually comparing the graphs for `if` conversion from Section 4.1.1 with the tables from Section 4.2.1, there is no obvious correlation. A similar situation exists for inlining. There may be a slight correlation between alignment and performance for `bzip2` on the Itanium, where xml and docs have reduced alignment and reduced performance. However, the GAP snf900 and the `gzip` docs inputs on the Itanium both have elevated alignment and difference scores compared to the other inputs, but no apparent corresponding variation in performance. There are no visually identifiable trends or anomalies in the Itanium 2 data.

In order to further examine a possible correlation between alignment and performance, we graphed the alignment score of each training input against its performance on the workload for each benchmark. These graphs do not suggest any correlation between alignment and performance. For completeness, these graphs can be found in Appendix B.

A significant finding of this performance study is that while profile-guided inlining usually does not reduce performance (the main exceptions being `bzip2` and MCF on the Itanium 2), the same cannot be said of profile-guided `if` conversion. `If` conversion almost always reduces performance on the Itanium (though `bzip2` is improved about 4% and parser is improved about 1.5%), and reduces performance on the Itanium 2 for every training input for every program in our study. Furthermore, performance is reduced by more than 5% on average. An improved design of `if` conversion in the ORC with respect to the way that profile information is used may correct the performance degradation caused by this transformation. Similarly, there is also potential to improve inlining for the Itanium 2.

4.3 Resubstitution

An important question when using FDO is whether or not the compiler makes good use of the profile information. More precisely, does the accuracy of the profile impact the resulting performance of the optimizations? Resubstitution is the practice of using the same input for both the training and evaluation runs. While running a program on identical input multiple times would seldom, if ever, be done in practice (since the results of the computation would be known after the first run), resubstitution allows for the evaluation of how well the compiler uses so-called "perfect information." Since the profile contains only branch and callsite frequency counts, instead of full path frequency counts, the information is not perfect. However, no other input could produce a profile that is more accurate than resubstitution.

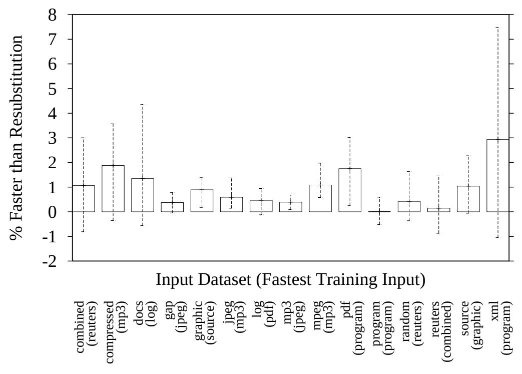
Ideally, a compiler that makes good use of profile information will produce the fastest binary for a given input when resubstitution is used. If this is not the case, then:

- The collected profile may be insufficient to capture important program behaviors, or the information may not be sufficient to be representative of the actual program behavior in certain situations.
- The compiler may not properly use profile information. Heuristics that use the profile information might not make correct decisions, or perhaps the machine model is insufficient or inaccurate.
- Performance improvements may arise unexpectedly under another input due to complex interactions between optimizations.

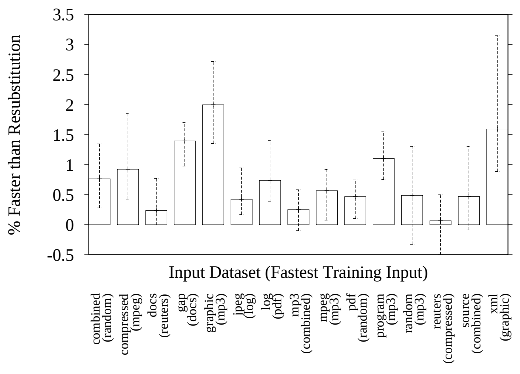
Whatever the reason, the use of FDO can be questioned if it does not consistently result in performance improvements when provided with an ideally accurate profile. Resubstitution should not be expected to produce the fastest binary in every instance. If this were the case, the optimization would likely perform poorly in non-resubstitution cases. However, if resubstitution does not perform well, there is no reason to expect that providing the compiler more accurate information about program behavior (*i.e.*, via profiling) should result in improved program performance.

We present *Aestimo*'s resubstitution results in a similar manner to the performance results of Section 4.2. Instead of calculating the speedup over static compilation, *Aestimo* calculates the performance improvement between the fastest FDO binary for each input and the resubstitution case. The training input that resulted in the fastest binary is indicated in parenthesis beside the resubstitution training data below the graph. For example, in Figure 4.40(a) resubstitution on the xml input with `if` conversion is about 3% slower than the fastest binary, which was trained on the program input.

The execution performance results suggest that the ORC generally makes good use of profile information. While resubstitution seldom results in the fastest average execution time, it is often the fastest or nearly the fastest when the range of run times is considered. Resubstitution is less than 2% slower than the fastest FDO binary in 81% of cases. However, there are also several cases where resubstitution is substantially slower than the fastest FDO binary. In one case, the performance difference is over 17%.

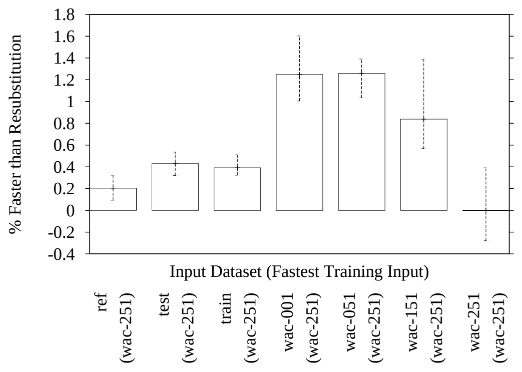


(a) Itanium

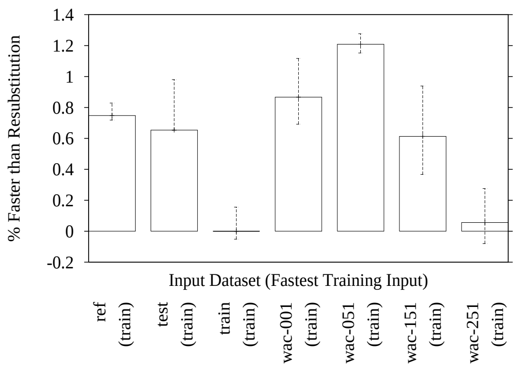


(b) Itanium 2

Figure 4.40: Resubstitution for `if` conversion on `bzip2`



(a) Itanium



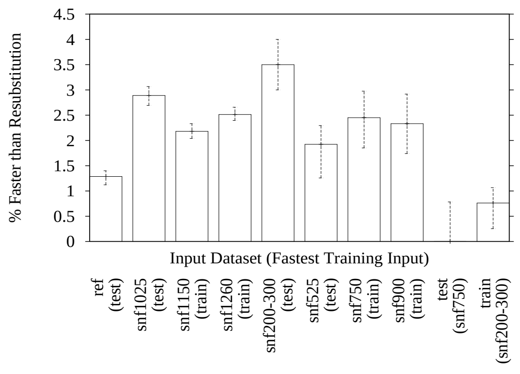
(b) Itanium 2

Figure 4.41: Resubstitution for `if` conversion on `crafty`

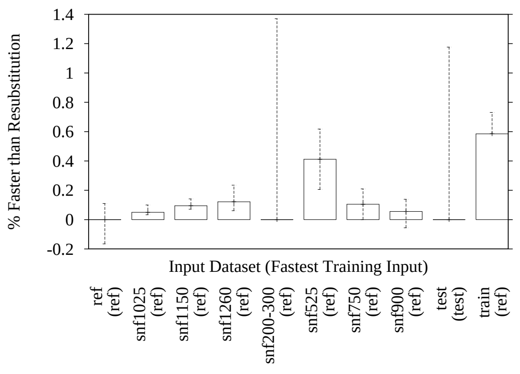
4.3.1 `If` conversion

Figure 4.40 shows resubstitution results for `bzip2`. On the Itanium, the ORC usually uses profile information effectively. In most cases, resubstitution is less than 2% slower than the fastest binary, and with 10 of the 15 inputs resubstitution leads to the fastest binary when range of runtime is considered. On the Itanium 2, the ORC makes good use of perfect information, and resubstitution is less than 2% slower than the fastest binary for all inputs. Furthermore, resubstitution achieves the fastest execution, when the range of run times is considered, for 5 of the 15 inputs.

With `crafty` resubstitution consistently results in nearly the same performance as the fastest FDO binaries. On both architectures, resubstitution is slower than the fastest binary by about 1.2% in the worst cases, as shown in Figure 4.41. Interestingly, training on the wac-251 input on the Itanium, and training on the train input on the Itanium 2, always results in the fastest binary. This suggests that these two inputs result in a particularly good profile for `if` conversion on their re-

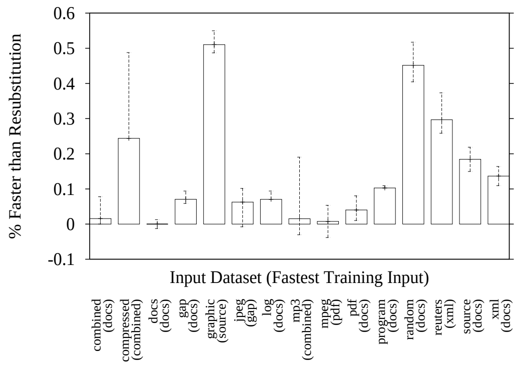


(a) Itanium

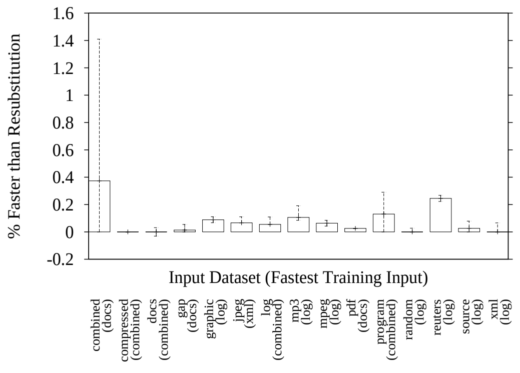


(b) Itanium 2

Figure 4.42: Resubstitution for `if` conversion on `GAP`



(a) Itanium



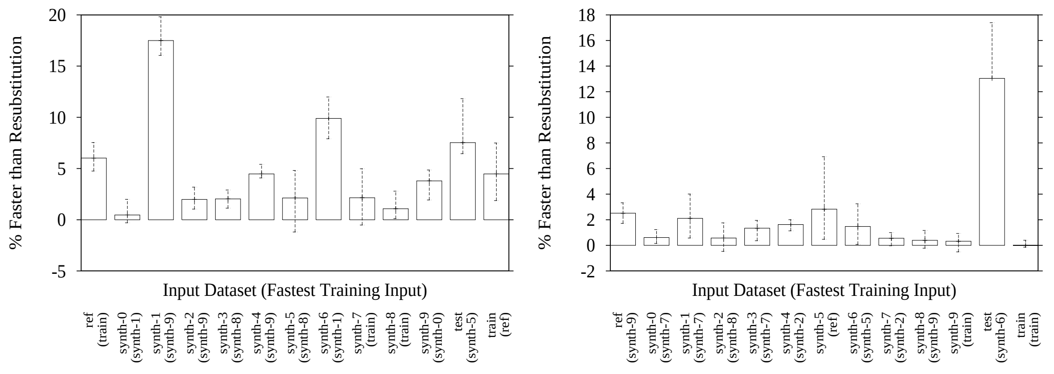
(b) Itanium 2

Figure 4.43: Resubstitution for `if` conversion on `gzip`

spective platforms, and manage to capture some aspect of program behavior that is not effectively represented or exploited in the profiles based on other inputs.

Figure 4.42 presents the resubstitution result for the GAP benchmark. On the Itanium, resubstitution usually produces a binary that is more than 2%, and as much as 3.5%, slower than the fastest binary. Accurate information does not result in increased performance in these cases. However, accurate information does result in competitive levels of performance for the same benchmark on the Itanium 2. In this case, resubstitution is no more than 0.2% slower than the fastest binary for all but two inputs. In another case where training on a single input consistently outperforms resubstitution, the ref input results in the fastest binary for all inputs except test. However, since the differences between resubstitution and training on ref are so small, this phenomenon could be coincidental. The worst-case performance is for train, where resubstitution is less than 0.6% slower than ref.

`Gzip` also demonstrates effective use of accurate profile information. As shown in Figure 4.43, resubstitution is usually within 0.2% of the fastest binary. The worst-case resubstitution performance



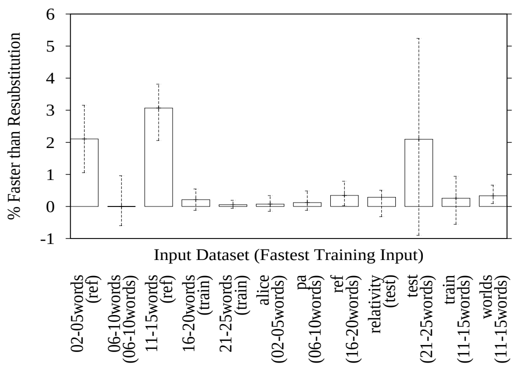
(a) Itanium (b) Itanium 2

Figure 4.44: Resubstitution for `if` conversion on `MCF`

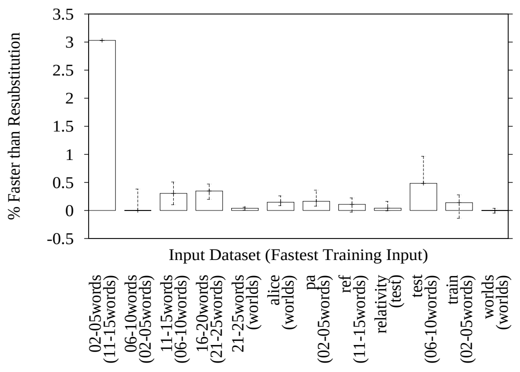
occurs on the Itanium, where resubstitution on graphic is slightly more than 0.5% slower than using the binary trained on source.

Figure 4.44 presents both effective and extremely ineffective use of accurate feedback information with `MCF`. On the Itanium, there are 4 cases where resubstitution performs as well as the fastest binary when the run time range is considered. However, `MCF` on the Itanium also results in the worst resubstitution performance in our study. Resubstitution on the synth-1 input is more than 17% slower than running the same input on the binary trained on synth-9. At the same time, synth-1 produces the fastest binary for synth-6, which is more than 10% faster than resubstitution. Interestingly, synth-9 and train most frequently produce the fastest binaries, while synth-9 achieves the best performance on the workload (refer back to Figure 4.15 in Section 4.2.1), but neither is close to achieving the best performance under resubstitution. Unfortunately, greater understanding of these inputs and their effect on program behavior is required to speculate on why these performance results are observed. This issue is discussed in more detail in Section 6.1.

Resubstitution for `MCF` on the Itanium 2 fares better. Here, resubstitution is within 2% of the fastest binaries in most cases, and within 3% in all cases expect for test. In fact, seven inputs result in best performance when the range of run times are considered. However, resubstitution performance on the test input is particularly poor, about 13% slower than the binary trained on synth-6. In this particular case, it may be that the test input is not sufficient to generate a useful profile, and that a profile from a longer running input captures additional information that can improve the performance of even a short-running input. However, this explanation does not apply to the results on the Itanium. Refer back to the run times reported in Table 3.3. The test input only runs for 0.21 seconds. However, all the other inputs run for at least 30 seconds, and for several minutes in most cases. Therefore, the synth-1 input on the Itanium and the test input on the Itanium 2 are likely candidates to discover a scenario where the `if` conversion heuristics fail to make the right decision. Further analysis of this scenario could lead to a better understanding of `if` conversion, and a better



(a) Itanium



(b) Itanium 2

Figure 4.45: Resubstitution for `if` conversion on `parser`

`if` conversion heuristic.

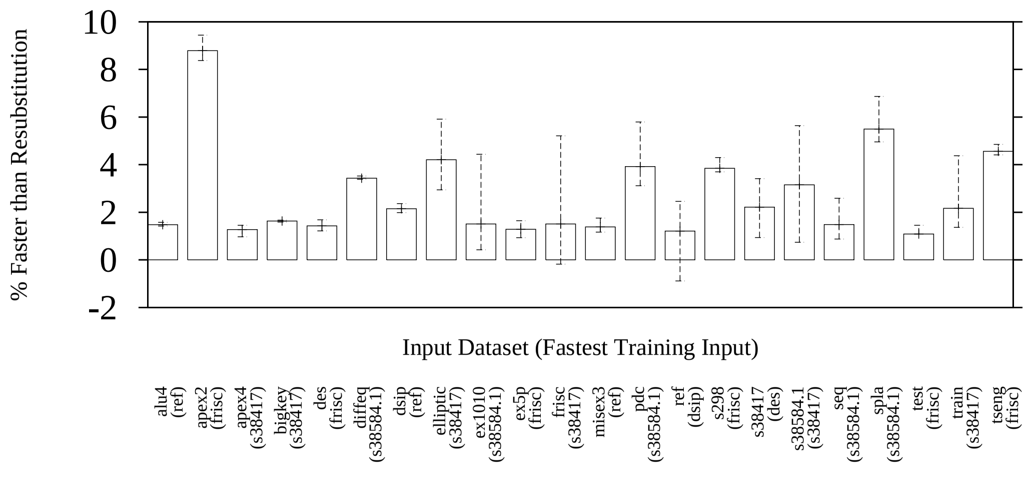
The results for `parser` in Figure 4.45 show effective use of accurate information. On both processors, resubstitution is within 0.5% of the fastest binary, and results in the best performance in the majority of cases. On the Itanium, the key exceptions are for 02-05words and 11-15words, where resubstitution is about 2% and 3% slower than the fastest binary, respectively. The only exception on the Itanium 2 is 02-05words, where resubstitution is 3% slower than training on 11-15words.

Figure 4.46 presents results for the placement task of VPR. Resubstitution performance is mixed on the Itanium. Half of the inputs result in resubstitution performance less than 2% slower than the fastest binary. However, among these, only three are as fast as the fastest binary. Also, resubstitution is more than 4% slower than the fastest binary in 4 cases, and nearly 9% slower than the fastest binary on the apex2 input. Also, it is curious that the best FDO performance on dsip is obtained by training on ref, but conversely the best performance on ref is obtained by training on dsip. The run-time range running on ref is large enough that resubstitution may be as fast as using the binary trained on dsip. Therefore, it seems likely that the binary trained on ref is fastest on the dsip input, and that the binaries trained on the two inputs achieve equivalent performance on the ref input. On the Itanium 2, resubstitution universally leads to high levels of performance compared the fastest FDO binaries, which are always less than 1.5% faster than resubstitution.

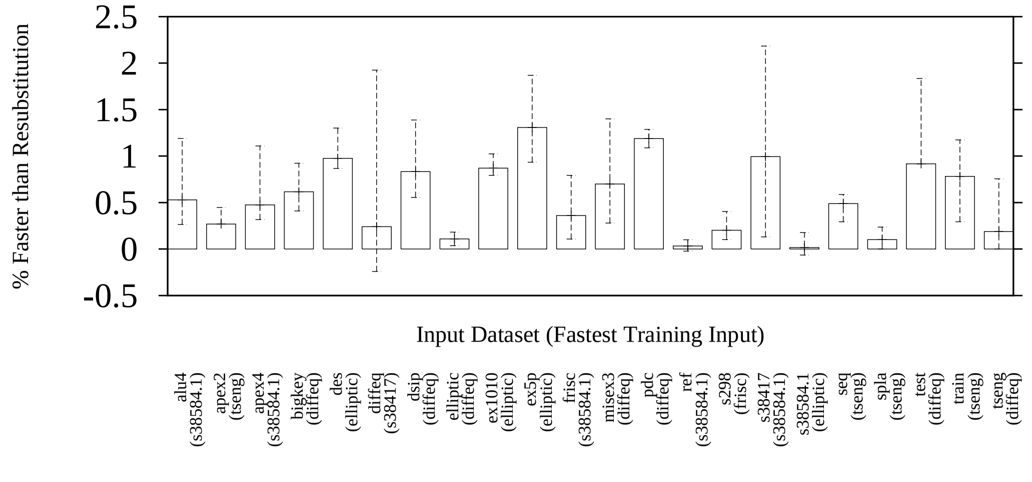
Resubstitution performs well for the routing task of VPR, as seen in Figure 4.47. Resubstitution is never more than 1.5% slower than the fastest binary on the Itanium, and no more than 0.6% slower than the fastest binary on the Itanium 2.

Rank Analysis

Despite the encouraging resubstitution performance results discussed above, it is possible that the range of performance among FDO binaries is frequently small, and that this situation results in

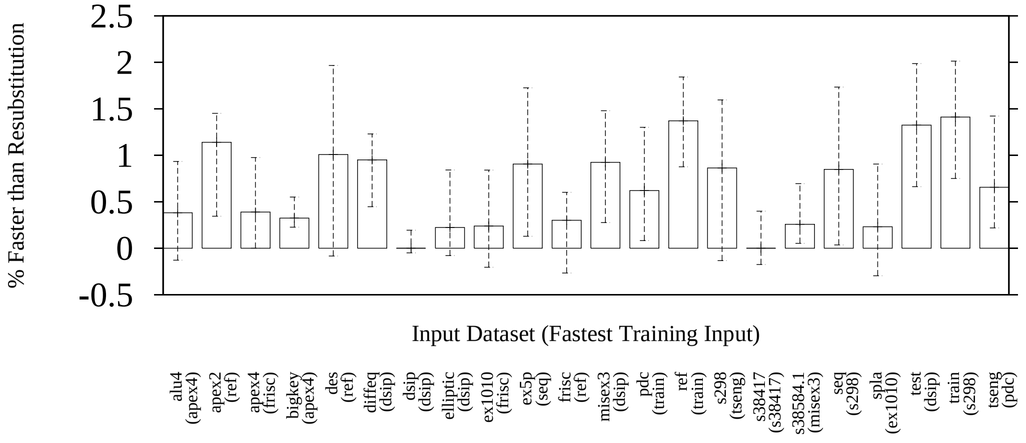


(a) Itanium

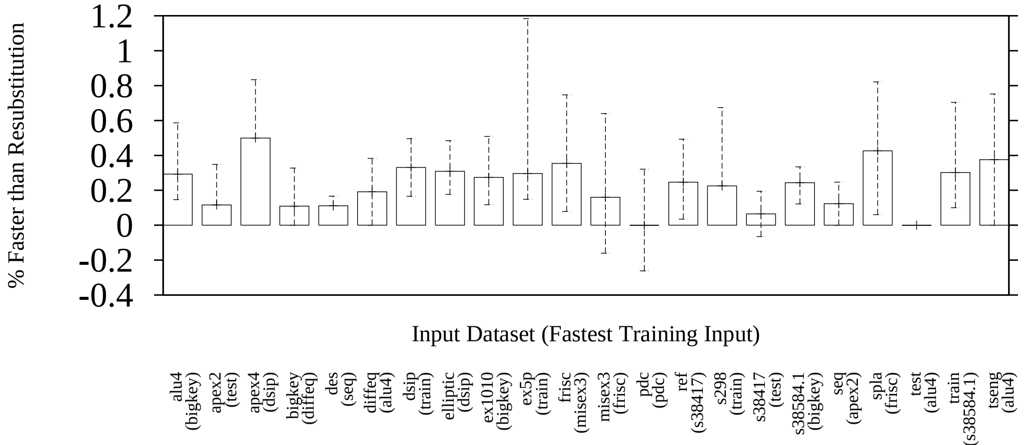


(b) Itanium 2

Figure 4.46: Resubstitution for `if` conversion on `VPR` (place)



(a) Itanium



(b) Itanium 2

Figure 4.47: Resubstitution for `if` conversion on `VPR` (route)

the small performance differences between resubstitution and the fastest FDO binaries. In order to investigate this possibility, *Aestimo* also provides the rank of each binary according to performance on each input. A rank of 1 indicates that a binary is the fastest on a particular input. Likewise, a rank of 2 indicates that the binary achieved the second-best performance for the input.

Tables 4.40 through 4.47 list each input in the program workloads. For each input, and for each processor, the rank of the `if` conversion resubstitution binary for the input is listed, along with the performance difference between the resubstitution binary and the rank-1 FDO binary. For instance, the first row of Table 4.40 show that among the FDO binaries for `bzip2` on the Itanium, the binary trained on combined is the 8^{th} fastest when evaluated using the combined input. Furthermore, the binary trained on combined was 1.06% slower than the fastest FDO binary.

Except for `MCF` (see Table 4.44), the differences in performance between resubstitution and the rank-1 binary are small, regardless of the rank of the resubstitution binary. With `MCF`, the performance differences between the rank-1 binary and the resubstitution binary vary greatly, but the performance differences are not correlated to the rank of the resubstitution binary. For example, on the Itanium, the resubstitution binaries for the synth-2 and synth-3 inputs both have a rank of 12, and result in performance about 2% slower than the rank-1 binaries. On the other hand, resubstitution with either ref or synth-0 results in a rank of 5. However, resubstitution for ref is 6% slower than the rank-1 binary, while resubstitution for synth-0 is less than 0.5% slower than the rank-1 binary.

The rank results are quite similar for all the benchmarks. While resubstitution achieves a low rank on some inputs for every program, resubstitution achieves very high ranks with similar frequency. For every benchmark, on both processors, there is an input where resubstitution gets a rank of 1 or 2, but also an input where resubstitution gets the highest or second-highest possible rank. Furthermore, resubstitution ranks are scattered across the possible range of ranks for each program.

Therefore, the rank analysis suggests that the frequently good performance of resubstitution binaries compared to their peers is due to small performance differences among the FDO binaries, and not due to the more accurate information provided by resubstitution allowing the compiler to make better optimization decisions. In fact, the rank results show that many other FDO binaries are often faster then the resubstitution binary.

4.3.2 Inlining

Inlining is an important optimization that yields large performance gains. Consequently, it has been the focus of many studies. Therefore, the heuristics for inlining in a mature compiler should be finely tuned, and resubstitution should perform well. While inlining resubstitution does, in general, perform better than `if` conversion, there are several cases where inlining resubstitution results in significantly reduced performance compared to the fastest FDO binaries.

Resubstitution on `bzip2` performs fairly well. Figure 4.48 shows that resubstitution is as fast

Input	Itanium		Itanium 2	
	Rank	Slower(%)	Rank	Slower(%)
combined	8	1.06	11	0.76
compressed	3	1.88	11	0.93
docs	7	1.35	3	0.24
gap	6	0.37	14	1.40
graphic	10	0.89	10	2.00
jpeg	3	0.59	5	0.43
log	3	0.47	9	0.74
mp3	8	0.39	2	0.25
mpeg	14	1.08	7	0.57
pdf	15	1.75	10	0.47
program	1	0.00	10	1.11
random	9	0.42	2	0.49
reuters	3	0.15	3	0.07
source	11	1.04	6	0.47
xml	13	2.93	6	1.60

Table 4.40: Rank of resubstitution binaries for `if` conversion on `bzip2`

Input	Itanium		Itanium 2	
	Rank	Slower(%)	Rank	Slower(%)
ref	2	0.20	5	0.75
test	2	0.43	3	0.65
train	3	0.39	1	0.00
wac-001	6	1.25	6	0.87
wac-051	6	1.26	7	1.21
wac-151	5	0.84	3	0.61
wac-251	1	0.00	2	0.06

Table 4.41: Rank of resubstitution binaries for `if` conversion on `crafty`

Input	Itanium		Itanium 2	
	Rank	Slower(%)	Rank	Slower(%)
ref	3	1.29	1	0.00
snf1025	6	2.89	2	0.05
snf1150	9	2.18	2	0.09
snf1260	3	2.51	9	0.12
snf200-300	8	3.50	1	0.00
snf525	6	1.92	2	0.41
snf750	7	2.45	4	0.10
snf900	7	2.33	2	0.06
test	1	0.00	1	0.00
train	9	0.76	10	0.58

Table 4.42: Rank of resubstitution binaries for `if` conversion on `GAP`

Input	**Itanium**		**Itanium 2**	
	Rank	**Slower(%)**	**Rank**	**Slower(%)**
combined	2	0.02	14	0.37
compressed	9	0.24	1	0.00
docs	1	0.00	1	0.00
gap	3	0.07	4	0.01
graphic	10	0.51	6	0.09
jpeg	6	0.06	3	0.07
log	10	0.07	13	0.05
mp3	3	0.02	15	0.11
mpeg	2	0.01	7	0.06
pdf	5	0.04	8	0.03
program	8	0.10	10	0.13
random	12	0.45	1	0.00
reuters	11	0.30	11	0.24
source	10	0.18	5	0.03
xml	15	0.14	1	0.00

Table 4.43: Rank of resubstitution binaries for `if` conversion on `gzip`

Input	**Itanium**		**Itanium 2**	
	Rank	**Slower(%)**	**Rank**	**Slower(%)**
ref	5	6.02	13	2.51
synth-0	5	0.46	5	0.61
synth-1	13	17.50	12	2.11
synth-2	12	1.99	10	0.57
synth-3	12	2.04	9	1.34
synth-4	5	4.47	10	1.62
synth-5	5	2.13	13	2.82
synth-6	12	9.90	10	1.47
synth-7	7	2.14	8	0.56
synth-8	4	1.08	4	0.39
synth-9	11	3.79	3	0.32
test	8	7.53	10	13.04
train	11	4.47	1	0.00

Table 4.44: Rank of resubstitution binaries for `if` conversion on `MCF`

Input	**Itanium**		**Itanium 2**	
	Rank	**Slower(%)**	**Rank**	**Slower(%)**
02-05words	8	2.11	4	3.03
06-10words	1	0.00	1	0.00
11-15words	12	3.07	7	0.30
16-20words	2	0.21	12	0.35
21-25words	2	0.05	5	0.04
alice	6	0.07	11	0.15
pa	7	0.12	4	0.16
ref	10	0.34	5	0.11
relativity	5	0.29	6	0.04
test	9	2.10	4	0.48
train	8	0.26	3	0.14
worlds	10	0.33	1	0.00

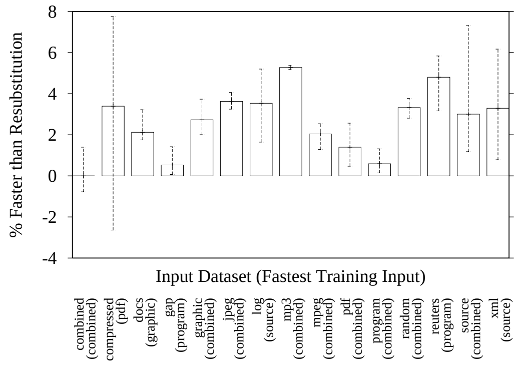
Table 4.45: Rank of resubstitution binaries for `if` conversion on `parser`

Input	Itanium		Itanium 2	
	Rank	Slower(%)	Rank	Slower(%)
alu4	13	1.48	7	0.53
apex2	18	8.79	6	0.27
apex4	13	1.27	5	0.48
bigkey	6	1.63	7	0.62
des	9	1.43	19	0.98
diffeq	17	3.43	4	0.24
dsip	11	2.15	21	0.83
elliptic	17	4.21	4	0.11
ex1010	3	1.51	9	0.87
ex5p	11	1.29	20	1.31
frisc	12	1.51	5	0.36
misex3	12	1.38	7	0.70
pdc	19	3.92	18	1.19
ref	6	1.21	4	0.03
s298	21	3.85	10	0.20
s38417	8	2.22	17	0.99
s38584.1	8	3.15	2	0.02
seq	14	1.48	5	0.49
spla	22	5.50	3	0.10
test	5	1.09	5	0.92
train	16	2.17	17	0.78
tseng	19	4.56	5	0.19

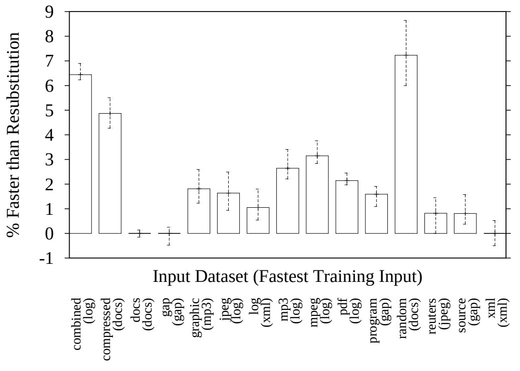
Table 4.46: Rank of resubstitution binaries for `if` conversion on `VPR` (place)

Input	Itanium		Itanium 2	
	Rank	Slower(%)	Rank	Slower(%)
alu4	9	0.38	7	0.29
apex2	22	1.14	2	0.12
apex4	4	0.39	15	0.50
bigkey	8	0.32	5	0.11
des	20	1.01	10	0.11
diffeq	18	0.95	5	0.19
dsip	1	0.00	9	0.33
elliptic	2	0.22	16	0.31
ex1010	7	0.24	11	0.27
ex5p	19	0.91	5	0.30
frisc	12	0.30	18	0.35
misex3	20	0.92	2	0.16
pdc	14	0.62	1	0.00
ref	21	1.37	7	0.25
s298	19	0.86	2	0.22
s38417	1	0.00	2	0.06
s38584.1	13	0.26	16	0.24
seq	20	0.85	6	0.12
spla	7	0.23	19	0.43
test	13	1.32	1	0.00
train	22	1.41	10	0.30
tseng	11	0.66	15	0.38

Table 4.47: Rank of resubstitution binaries for `if` conversion on `VPR` (route)

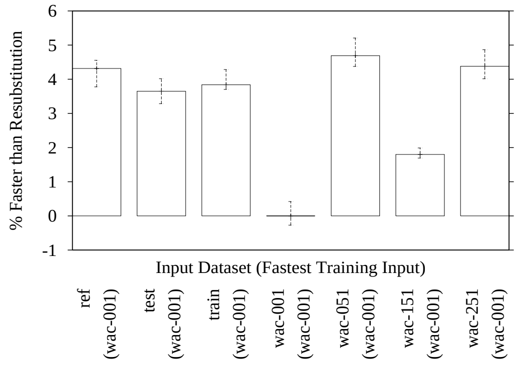


(a) Itanium

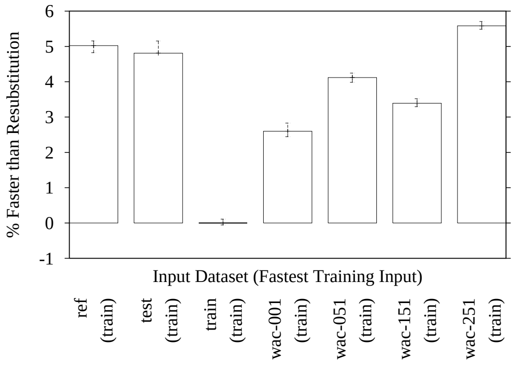


(b) Itanium 2

Figure 4.48: Resubstitution for inlining on `bzip2`



(a) Itanium

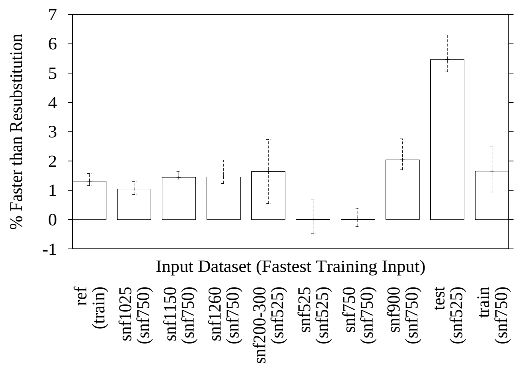


(b) Itanium 2

Figure 4.49: Resubstitution for inlining on `crafty`

as the fastest binary for 8 of the 13 inputs on the Itanium. However, there are also 3 inputs where other training inputs result in performance gains of more than 4% over resubstitution. Results are similar for the Itanium 2. However, in this case, training on the program input is nearly 8% faster than resubstitution for the mp3 and jpeg inputs. The binary trained on program is also more than 3% faster than resubstitution on pdf. The fact that program is a SPEC reference input raises the possibility that inlining heuristics in the ORC may be over-fit to these inputs. However, there is no supporting evidence for this hypothesis in the Itanium case. Section 4.3.3 will revisit this issue.

Unlike the `if`-conversion case, resubstitution performs poorly with inlining for `crafty`. In Figure 4.49 resubstitution is slower than the fastest binary, by over 3% in most cases, for both architectures. As in the `if`-conversion case, there is a single fastest binary on each architecture. On the Itanium, wac-001 is always the fastest (wac-251 was fastest for `if` conversion), while train is again the fastest in all cases on the Itanium 2. As expected, these two inputs achieve the best performance on the workload.


(a) Itanium

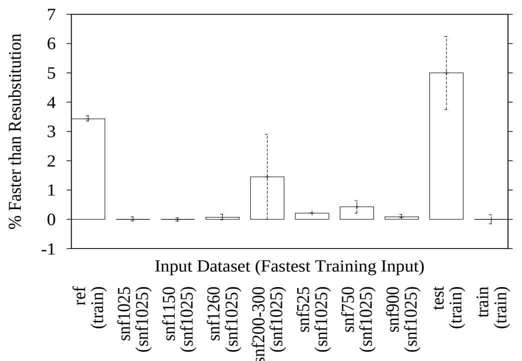
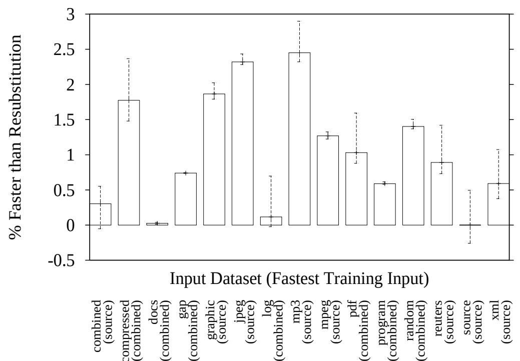

(b) Itanium 2

Figure 4.50: Resubstitution for inlining on GAP


(a) Itanium

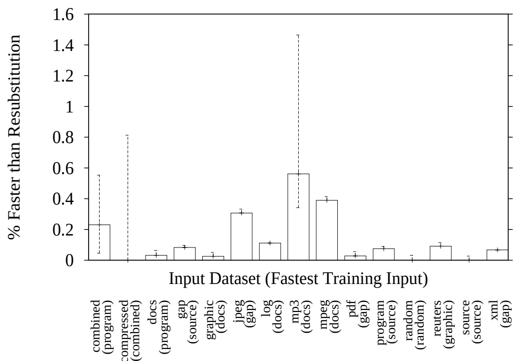
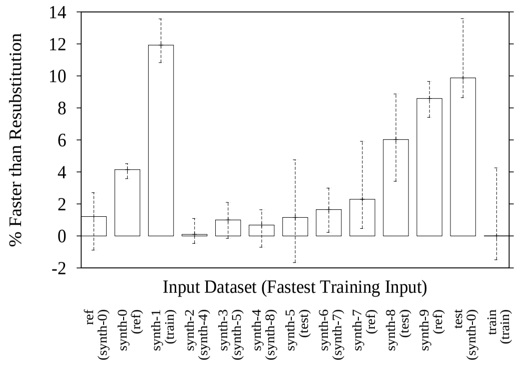

(b) Itanium 2

Figure 4.51: Resubstitution for inlining on gzip

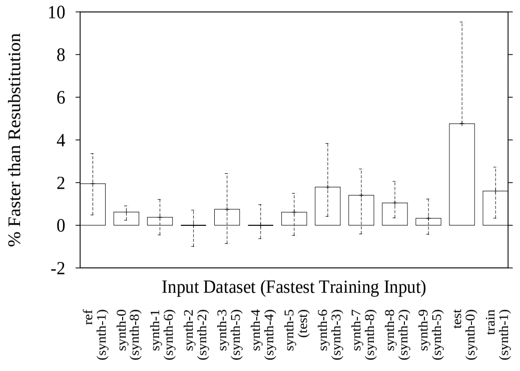
Figure 4.50 show resubstitution results for GAP. On the Itanium, resubstitution only achieves the fastest performance in two cases, though resubstitution is usually less than 2% slower than the fastest binary. The test input is anomalous, where training on snf525 is more than 5% faster than resubstitution. Performance on the Itanium 2 is generally better, with resubstitution usually as fast or nearly as fast as the fastest binary. However, training on the train input outperforms resubstitution on ref by more than 3%, and nearly 5% on test.

Resubstitution performs fairly well for gzip. In Figure 4.51(a), resubstitution is never more than 2.5% slower than the fastest binary. Resubstitution is always within 0.6% of the fastest binary on the Itanium 2. Note that on the Itanium, all the fastest binaries were trained on SPEC inputs.

Figure 4.52 show that MCF is problematic for inlining resubstitution. While 6 of 13 inputs produce resubstitution binaries as fast as those trained on other inputs, resubstitution is more than 5% slower than the fastest binaries in 5 cases. In particular, the binary trained on the train input is nearly 12% faster than resubstitution on the synth-1 input. Results are better on the Itanium 2,

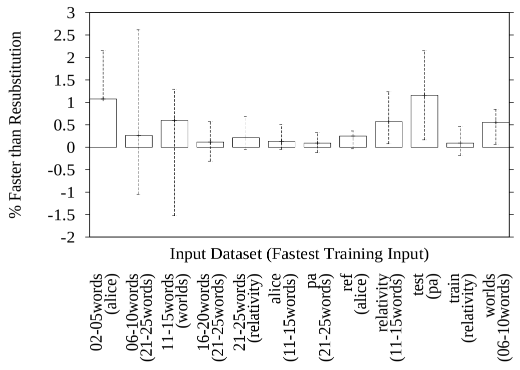


(a) Itanium

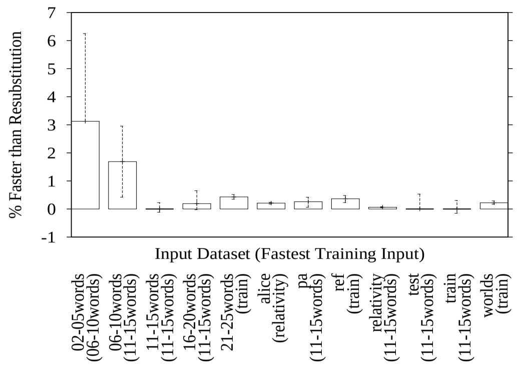


(b) Itanium 2

Figure 4.52: Resubstitution for inlining on MCF



(a) Itanium

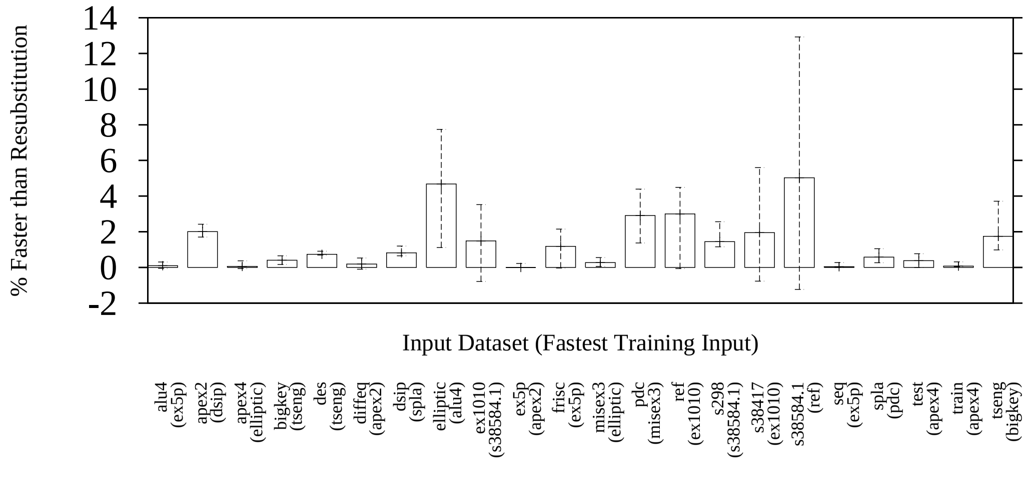


(b) Itanium 2

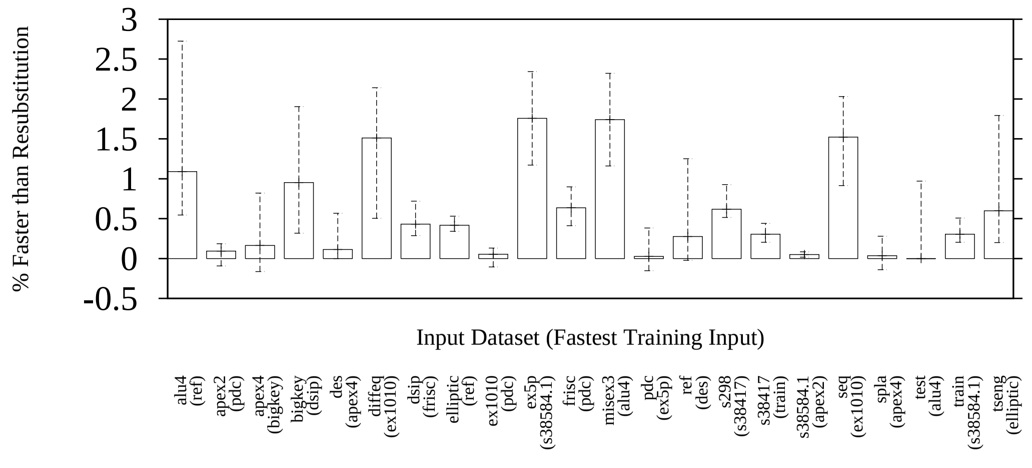
Figure 4.53: Resubstitution for inlining on `parser`

where resubstitution is fastest for 7 inputs. Test is the only input where resubstitution is more than 2% slower than the fastest binary, at nearly 5% slower than binary trained on synth-0.

Accurate profile information is used effectively for inlining with `parser`. Figure 4.53 shows that on the Itanium, 9 of the 12 inputs result in best performance using resubstitution. In the worst case, resubstitution is slightly more than 1% slower than the fastest binary. Results are almost as good on the Itanium 2. Here, resubstitution is within 0.5% of the fastest binary for 10 of the 12 inputs. However, performance degrades for the inputs containing the shortest sentences. Resubstitution is more than 3% slower for 02-05words, and almost 2% slower for 06-10words. Interestingly, in both these cases the fastest training input was the input containing the next-shortest sentences. Since the run times for these inputs is very small (see Table 3.5), it might be the case that the profile generated by those inputs does not contain sufficient information to achieve the best performance. The slightly larger inputs probably have fairly similar program behavior (since they have only slightly longer sentences), but gather more information in their profiles, which allow the



(a) Itanium

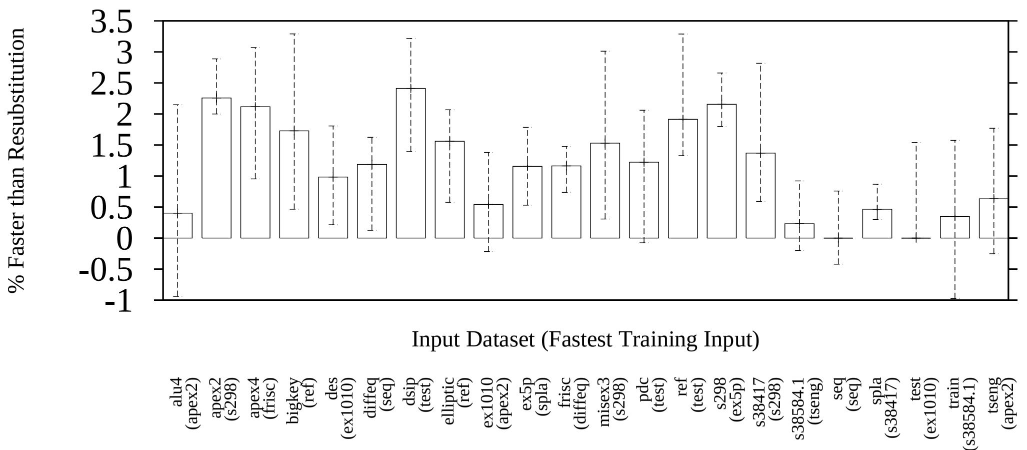


(b) Itanium 2

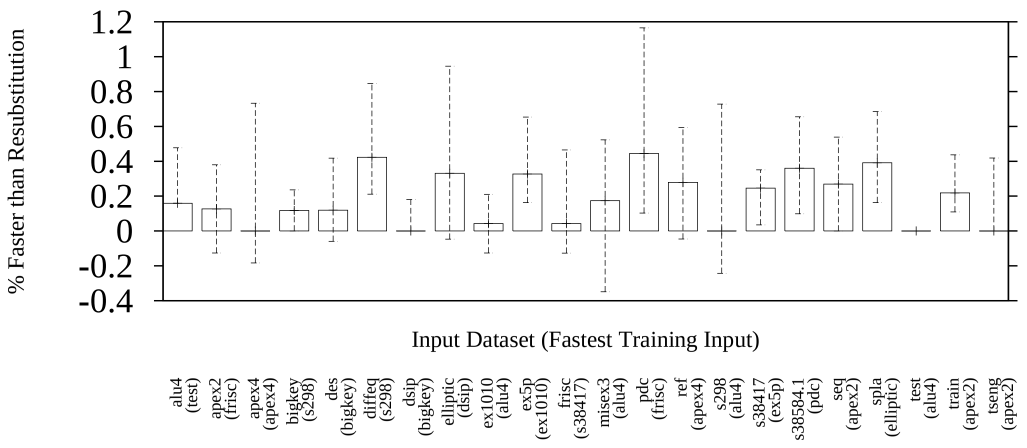
Figure 4.54: Resubstitution for inlining on VPR (place)

compiler to make better decisions. This hypothesis warrants further investigation.

Resubstitution results are mixed for the placement task of VPR. Figure 4.54 shows that for 13 cases on the Itanium, resubstitution performs as well as the fastest FDO binary. Nonetheless, it is over 4% slower than the fastest binary for the elliptic input. Results are better on the Itanium 2. While only 8 inputs achieve the best performance using resubstitution, resubstitution is always within 2% of the fastest binary. Resubstitution is also effective for the routing task, particularly on the Itanium 2. On the Itanium, 8 of the 22 inputs resulted in best performance using resubstitution. On the remaining inputs, resubstitution was within 2% of the fastest binary for all but 4 inputs, and these exceeded 2% only slightly. On the Itanium 2, resubstitution obtained the best performance in 14 cases, and was always within 0.5% of the fastest binary.



(a) Itanium



(b) Itanium 2

Figure 4.55: Resubstitution for inlining on VPR (route)

Input	**Itanium**		**Itanium 2**	
	Rank	**Slower(%)**	**Rank**	**Slower(%)**
combined	1	0.00	14	6.44
compressed	10	3.40	12	4.87
docs	8	2.12	1	0.00
gap	4	0.53	1	0.00
graphic	9	2.73	6	1.81
jpeg	7	3.63	9	1.63
log	11	3.54	5	1.05
mp3	12	5.28	10	2.64
mpeg	3	2.04	10	3.15
pdf	2	1.40	8	2.14
program	5	0.59	4	1.59
random	8	3.32	13	7.23
reuters	11	4.80	5	0.82
source	8	3.01	4	0.81
xml	12	3.30	1	0.00

Table 4.48: Rank of resubstitution binaries for inlining on `bzip2`

Input	**Itanium**		**Itanium 2**	
	Rank	**Slower(%)**	**Rank**	**Slower(%)**
ref	5	4.32	6	5.02
test	3	3.65	5	4.81
train	6	3.84	1	0.00
wac-001	1	0.00	2	2.60
wac-051	3	4.69	4	4.12
wac-151	2	1.80	2	3.39
wac-251	7	4.38	7	5.58

Table 4.49: Rank of resubstitution binaries for inlining on `crafty`

Rank Analysis

The Tables 4.48 through 4.55 provide the rank of each resubstitution binary for inlining. These results are similar to those presented for `if` conversion in Section 4.3.1. Once again, the ranks of resubstitution binaries are scattered across the possible range of ranks. Resubstitution achieves both high and low ranks for inputs for every program.

However, the performance differences between resubstitution and the rank-1 binary are significantly larger for inlining than for `if` conversion. While there is no clear correlation between rank and performance overall, a lower rank is usually associated with a smaller performance difference compared to the rank-1 binary for both `bzip2` (Table 4.48) and `crafty`(Table 4.49). For these programs, cases where resubstitution achieves good performance compared to the rank-1 binary are more likely to correspond to situations where better feedback information results in better inlining decision. However, the small number of low-rank resubstitution binaries suggests that the FDO system seldom makes effective use of more accurate feedback information.

Input	Itanium		Itanium 2	
	Rank	Slower(%)	Rank	Slower(%)
ref	8	1.31	3	3.43
snf1025	4	1.04	1	0.00
snf1150	5	1.44	1	0.00
snf1260	6	1.45	3	0.07
snf200-300	3	1.64	7	1.45
snf525	1	0.00	4	0.21
snf750	1	0.00	6	0.43
snf900	8	2.04	2	0.08
test	9	5.46	10	5.00
train	8	1.66	1	0.00

Table 4.50: Rank of resubstitution binaries for inlining on `GAP`

Input	Itanium		Itanium 2	
	Rank	Slower(%)	Rank	Slower(%)
combined	2	0.30	10	0.23
compressed	5	1.78	1	0.00
docs	2	0.03	4	0.03
gap	9	0.74	4	0.08
graphic	5	1.87	5	0.03
jpeg	6	2.32	11	0.31
log	2	0.12	6	0.11
mp3	9	2.45	14	0.56
mpeg	3	1.27	12	0.39
pdf	4	1.03	3	0.03
program	4	0.59	7	0.07
random	5	1.40	1	0.00
reuters	4	0.89	9	0.09
source	1	0.00	1	0.00
xml	7	0.59	4	0.07

Table 4.51: Rank of resubstitution binaries for inlining on `gzip`

Input	Itanium		Itanium 2	
	Rank	Slower(%)	Rank	Slower(%)
ref	5	1.21	13	1.95
synth-0	2	4.14	7	0.62
synth-1	5	11.92	3	0.37
synth-2	3	0.10	1	0.00
synth-3	8	1.00	7	0.75
synth-4	5	0.68	1	0.00
synth-5	7	1.15	5	0.61
synth-6	4	1.65	10	1.79
synth-7	7	2.29	12	1.40
synth-8	8	6.03	10	1.05
synth-9	3	8.59	4	0.32
test	11	9.88	3	4.76
train	1	0.00	13	1.61

Table 4.52: Rank of resubstitution binaries for inlining on `MCF`

Input	Itanium		Itanium 2	
	Rank	Slower(%)	Rank	Slower(%)
02-05words	3	1.08	10	3.13
06-10words	4	0.26	8	1.69
11-15words	8	0.60	1	0.00
16-20words	2	0.11	2	0.19
21-25words	6	0.21	8	0.43
alice	8	0.13	7	0.21
pa	2	0.09	3	0.26
ref	10	0.25	7	0.36
relativity	12	0.57	2	0.06
test	10	1.16	1	0.00
train	3	0.09	1	0.00
worlds	12	0.56	7	0.22

Table 4.53: Rank of resubstitution binaries for inlining on `parser`

Input	Itanium		Itanium 2	
	Rank	Slower(%)	Rank	Slower(%)
alu4	4	0.10	19	1.09
apex2	9	2.01	3	0.09
apex4	2	0.06	6	0.16
bigkey	6	0.41	16	0.95
des	9	0.74	6	0.11
diffeq	7	0.19	22	1.51
dsip	15	0.82	12	0.43
elliptic	22	4.68	11	0.42
ex1010	11	1.49	3	0.05
ex5p	1	0.00	22	1.76
frisc	19	1.18	22	0.64
misex3	8	0.28	22	1.74
pdc	19	2.91	2	0.03
ref	10	3.00	15	0.28
s298	22	1.45	16	0.62
s38417	12	1.95	18	0.31
s38584.1	20	5.03	6	0.05
seq	3	0.04	20	1.52
spla	10	0.58	5	0.03
test	11	0.38	1	0.00
train	4	0.08	6	0.30
tseng	22	1.75	13	0.60

Table 4.54: Rank of resubstitution binaries for inlining on VPR (place)

Input	Itanium		Itanium 2	
	Rank	Slower(%)	Rank	Slower(%)
alu4	2	0.40	2	0.16
apex2	15	2.26	2	0.13
apex4	15	2.12	1	0.00
bigkey	15	1.73	3	0.12
des	11	0.98	6	0.12
diffeq	9	1.19	8	0.42
dsip	18	2.41	1	0.00
elliptic	21	1.56	21	0.33
ex1010	10	0.54	6	0.04
ex5p	12	1.16	4	0.33
frisc	13	1.16	2	0.04
misex3	20	1.53	9	0.17
pdc	18	1.22	16	0.44
ref	20	1.91	7	0.28
s298	20	2.16	1	0.00
s38417	11	1.37	19	0.25
s38584.1	3	0.23	21	0.36
seq	1	0.00	4	0.27
spla	11	0.47	19	0.39
test	1	0.00	1	0.00
train	6	0.35	10	0.22
tseng	15	0.63	1	0.00

Table 4.55: Rank of resubstitution binaries for inlining on VPR (route)

4.3.3 SPEC Inputs

The SPEC CINT2000 benchmarks are the most frequently used programs and inputs for the evaluation of compiler optimizations. Therefore, we were curious if the consistent use of the SPEC-provided inputs may have unintentionally biased the ORC's heuristics toward the profiles produced by these inputs. Stated differently, have the compiler's heuristics been over-fit to the SPEC inputs, to the detriment of other inputs? Since most compiler designers use at least a significant portion of the suite to evaluate their work, optimizations and heuristics should be generally applicable, and not tailored to any particular program. However, it is possible that the SPEC inputs do not present the full spectrum of possibilities that exist in alternate inputs, and that heuristics may not deal with these unencountered situations properly.

Table 4.56 presents a summary of information from the graphs in Section 4.3. For each benchmark, we list the number of inputs in the workload, the number of those inputs provided by SPEC, and the number of cases where training on a SPEC input resulted in the fastest binary for a single input. Ideally, resubstitution would always produce the fastest binary, though this is often not the case. If it were the case, it would be reasonable to expect that the proportion of times that SPEC inputs produce the fastest FDO binary would be equal to the proportion of SPEC inputs in the workload, assuming that the selected inputs are spread across the spectrum of possible inputs for each

Benchmark	Inputs		Itanium		Itanium2	
	Total	SPEC	If-Conversion	Inlining	If-Conversion	Inlining
bzip2	15	7	7	13	7	5
crafty	7	3	0	0	7	7
GAP	10	3	8	1	10	3
gzip	15	7	3	15	10	8
MCF	13	3	4	7	3	1
parser	12	3	5	0	1	3
VPR (Place)	22	3	2	1	0	3
VPR (Route)	22	3	5	5	5	1
Total	116	32 28%	38 33%	42 36%	43 37%	31 27%

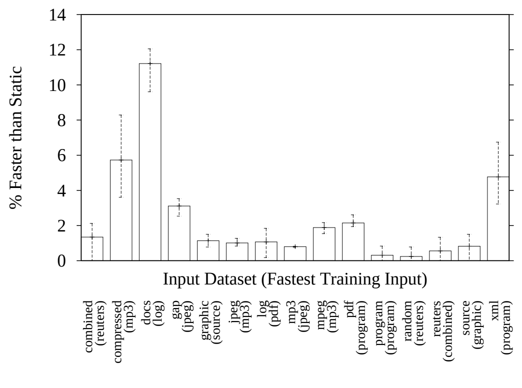
Table 4.56: Number of cases where training on SPEC-provided inputs results in best FDO performance

program. Unfortunately, we do not yet have a methodology to adequately characterize inputs, nor to describe the space from which benchmark inputs are selected. However, the selected inputs were chosen with care, with the intent to create a workload as varied as possible while still using realistic inputs for each program.

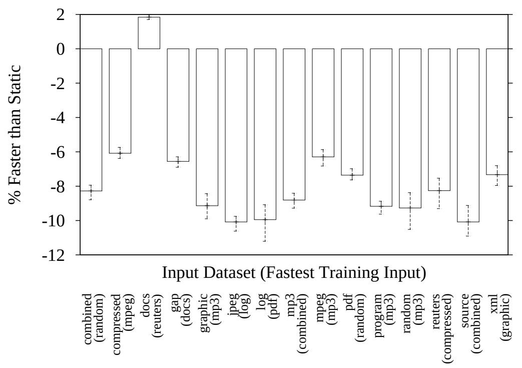
The potential limitations of the inputs notwithstanding, the results in Table 4.56 show that the experimental results are not too far from the idealized expectation. SPEC inputs account for 28% of the entire workload, while binaries trained on SPEC inputs accounted for the fastest times on 31% of the inputs. Furthermore, the results in Section 4.4 show that performance on the SPEC inputs is frequently poor compared to other inputs in the workload. Therefore, while the selected inputs are not guaranteed to span the space of possible inputs or follow any particular distribution in that space, there does not appear to be any reason to be concerned about the ORC's heuristics being over-fit to the SPEC inputs.

4.3.4 Conclusions

Overall, resubstitution with the ORC leads to high levels of performance compared to other FDO cases. However, rank analysis shows that resubstitution does not consistently result in faster-than-average FDO binaries. Therefore, the more accurate profile information provided by resubstitution is not used effectively to make better optimization decisions. This result is in agreement with the conclusion in Section 4.2 which stated that profile-guided inlining, and particularly `if` conversion, require more attention to become effective optimizations, since compiling without profile information often results in substantially better performance than even the best FDO binaries. Additionally, an overview of the experimental results suggests that there is little evidence that compiler heuristics are unintentionally tailored toward the profiles generated by the SPEC inputs.



(a) Itanium



(b) Itanium 2

Figure 4.56: Static vs. FDO performance for `if` conversion on `bzip2`

4.4 Feedback-Directed Optimization

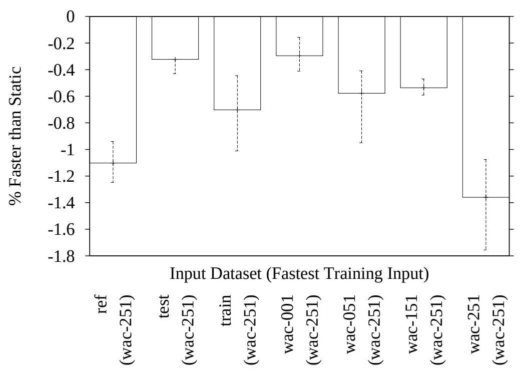
Feedback-Directed Optimization is intended to improve program performance above that obtainable by static optimization by providing compiler heuristics with accurate information about dynamic program behavior. Therefore, an effective FDO system should be able to meet or exceed the performance of static optimization for the majority of programs and inputs. In this section the performance of statically optimized binaries is compared to the performance of the fastest FDO binaries for each input of every program in the study. These measures represent the best case performance of FDO recorded by *Aestimo*, and as such represent an upper bound on FDO performance given the inputs selected for each program.

4.4.1 `If` conversion

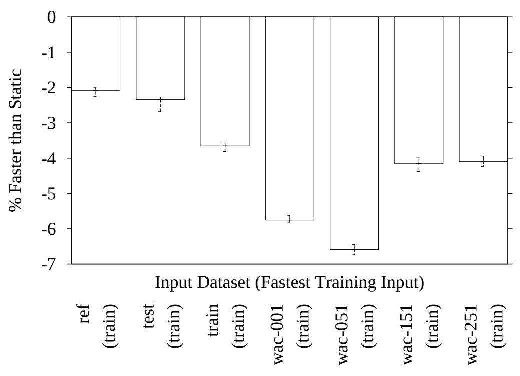
Section 4.2.1 showed that profile-guided `if` conversion seldom improves workload performance on the Itanium, and always reduced workload performance on the Itanium 2. Similar results are obtained when best-case FDO performance is compared to the performance of the statically optimized binary.

Figure 4.56 shows large differences in best-case FDO `if`-conversion performance for both the Itanium and the Itanium 2. On the Itanium, performance is often improved by less than 2% over static optimization. These cases may indicate inputs where the additional information provided by the profile is not required to make good `if` conversion decisions. However, there are three inputs where FDO `if` conversion increases performance by more than 4%, with performance nearly 12% faster than static on the `docs` input. This impressive gain highlights the potential of FDO. On the other hand, FDO results in performance reductions on the Itanium 2 as large as 10%. FDO only results in a performance gain on the `docs` input, and the gain is only 2% over static.

In Figure 4.57 best-case FDO performance approaches the performance of static optimization

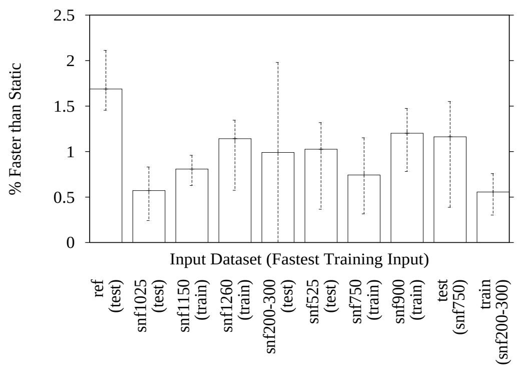


(a) Itanium

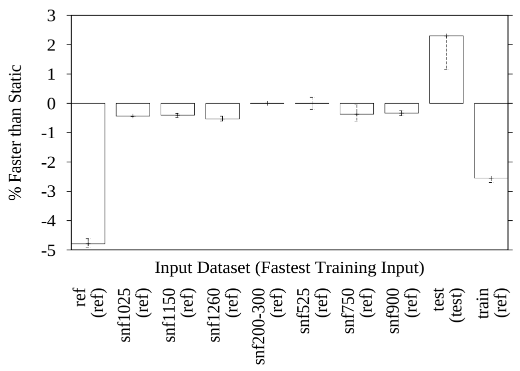


(b) Itanium 2

Figure 4.57: Static vs. FDO performance for `if` conversion on `crafty`



(a) Itanium

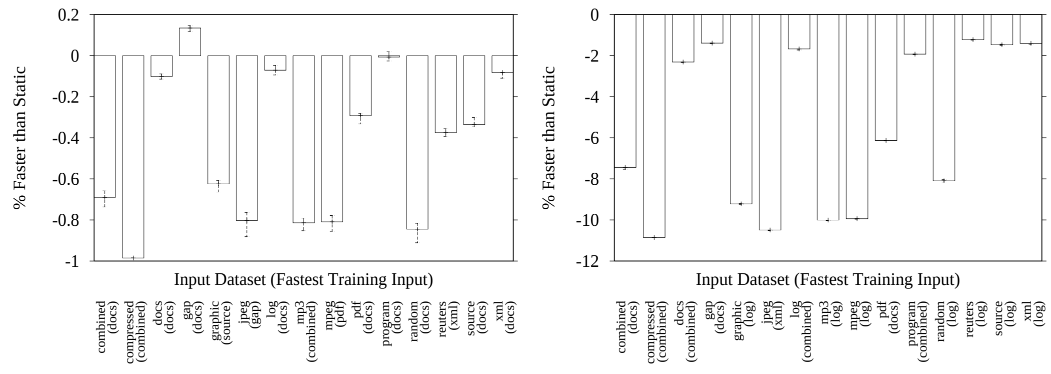


(b) Itanium 2

Figure 4.58: Static vs. FDO performance for `if` conversion on GAP

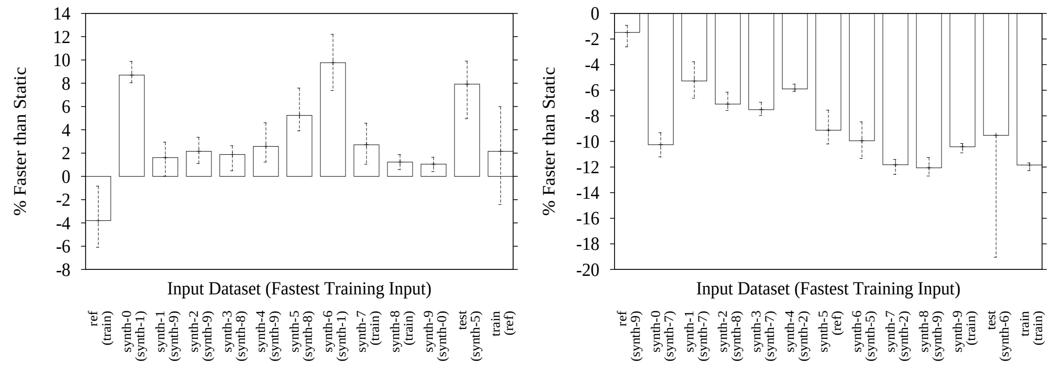
on the Itanium, but is significantly slower than static optimization on the Itanium 2. Since the fastest FDO binary is always the same on each processor, the graph also shows that testing performance on different inputs can lead to different conclusions. If FDO performance on the Itanium 2 is evaluated by training on the SPEC train input and evaluated using the SPEC ref input, performance is only reduced by 2%, which may be acceptable if the optimization is more successful for most other programs. However, if the same FDO binary is evaluated using the wac-051 input, a performance reduction of more than 6% is observed. It is much less likely that such a large performance penalty would be acceptable to a compiler designer. Furthermore, recall that training on train resulted in the best performance on wac-051. Training on other inputs results in even larger performance penalties.

In the best case, FDO `if` conversion results in performance similar to static optimization for GAP. However, Figure 4.58(b) shows that even the fastest FDO binary for the ref inputs results in nearly a 5% performance degradation, while best-case FDO degrades performance on the train input by over 2.5%.



(a) Itanium (b) Itanium 2

Figure 4.59: Static vs. FDO performance for `if` conversion on `gzip`

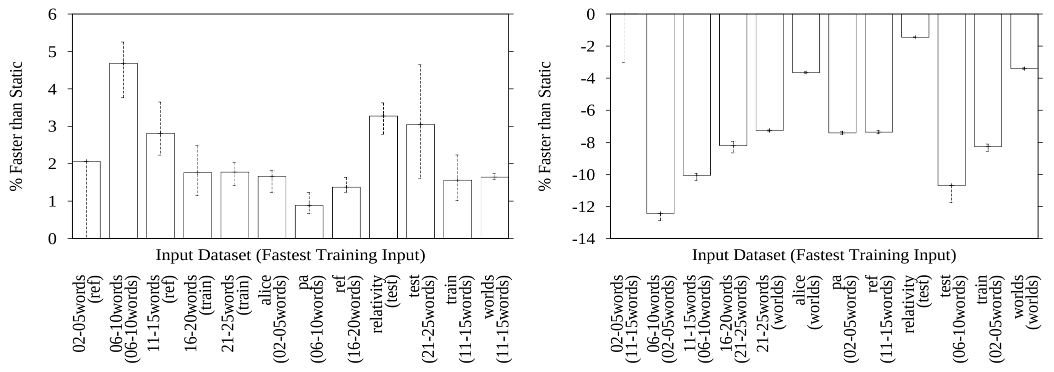


(a) Itanium (b) Itanium 2

Figure 4.60: Static vs. FDO performance for `if` conversion on `MCF`

FDO `if` conversion also negatively impacts performance for `gzip`. While performance on the Itanium nearly matches the performance of static optimization, even the best-case FDO binaries result in large reduction in performance on the Itanium 2. FDO reduces performance by at least 6% on 8 of the 15 inputs, and by over 10% on 4 inputs. However, performance is within 2% of static on 6 other inputs. Interestingly, the same training inputs created the fastest binaries for inputs where FDO had both large and small performance reductions, strengthening the hypothesis that the choice of inputs used for evaluation is important.

For `MCF`, FDO generally results in performance gains on the Itanium. Figure 4.60 shows performance improvements exceeding 8%. On the Itanium 2, the fastest FDO binaries are at least 5% slower than the static binary in all but one case, and almost 12% slower in the worst case. Recall that all the FDO binaries for `MCF` have very similar workload performance, and were produced by nearly identical optimization logs. It is therefore likely that the performance of the FDO binaries does not vary significantly on a given input. Consequently, the observed performance of FDO `if`



(a) Itanium (b) Itanium 2

Figure 4.61: Static vs. FDO performance for `if` conversion on `parser`

conversion for `MCF` on the Itanium 2 could vary by more than 10% depending on which input is used for evaluation.

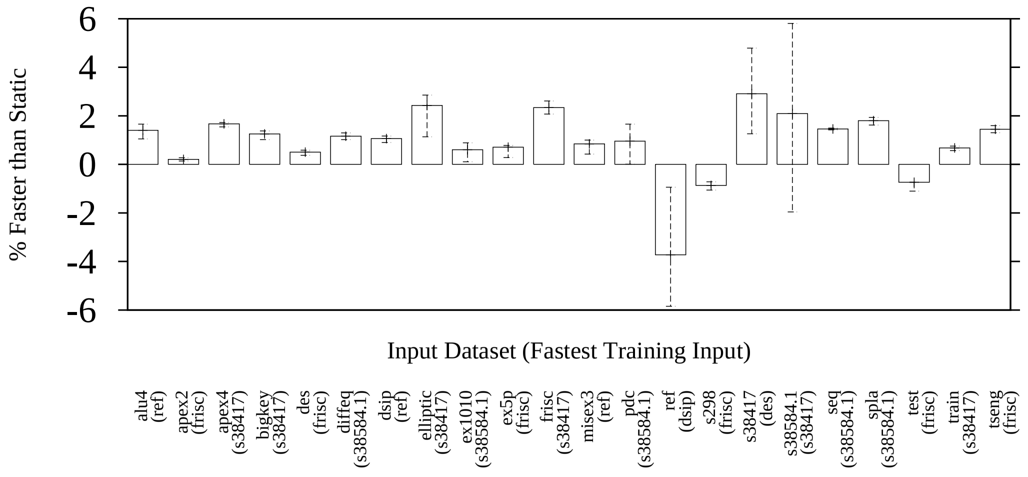
Best-case FDO `if` conversion barely outperforms static optimization for all inputs to `parser` on the Itanium, while FDO `if` conversion always reduces performance on the Itanium 2. Even with the fastest FDO binaries, performance is degraded by more than 12% on the `06-10words` input, and by at least 7% on all but 4 inputs.

According to Figures 4.62 and 4.63, best-case FDO `if` conversion is able to match the performance of static optimization on the Itanium, and is usually within 2.5% for the placement task on the Itanium 2. However, routing on the Itanium 2 shows the typical failure of FDO `if` conversion to approach the performance of static optimization. Performance of the fastest FDO binary is between 4% and 12% slower than static on every input in the workload.

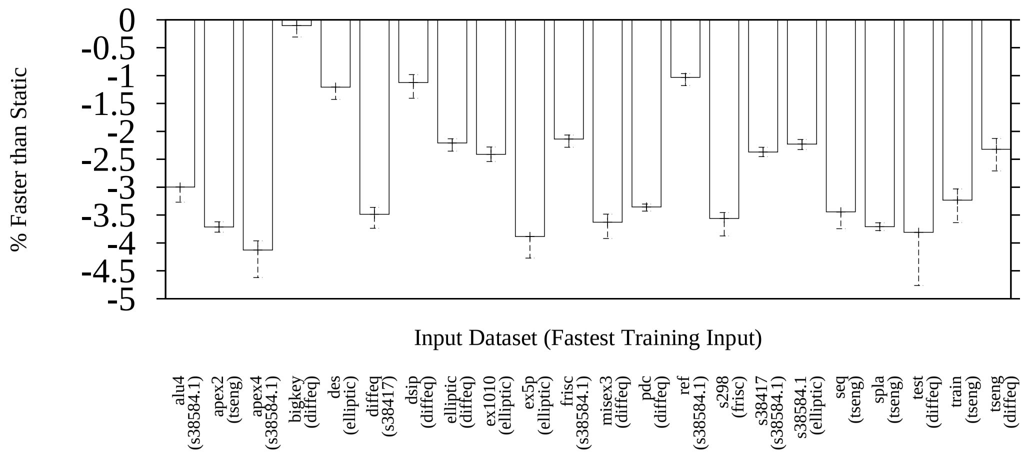
Results for FDO `if` conversion on the Itanium are generally unimpressive. Performance improvements are mostly small, but the occasional performance reductions are also fairly insubstantial. Nonetheless, there are a few cases, such as for `bzip2` and `MCF`, where FDO `if` conversion displays its potential to have a significant positive impact on program performance. Selecting the fastest FDO binary for each input is optimistic, but could help compensate for the deficiencies in the FDO system suggested by Section 4.3.

Based on the results in this section, feedback-directed `if` conversion, as implemented in the ORC 2.1 compiler, does not correctly use profile information to improve performance on the Itanium 2. Even when the fastest FDO binary for each input is used, performance is reduced compared to static optimization in 111 out of 116 cases. Furthermore, there is an input for each program where FDO `if` conversion reduced performance by at least 5%, while reductions in performance in excess of 10% are not uncommon.

We suspect that the `if` conversion heuristics, originally designed for the Itanium, were not modified to deal with the architectural differences of the Itanium 2. Considering that `if` conversion

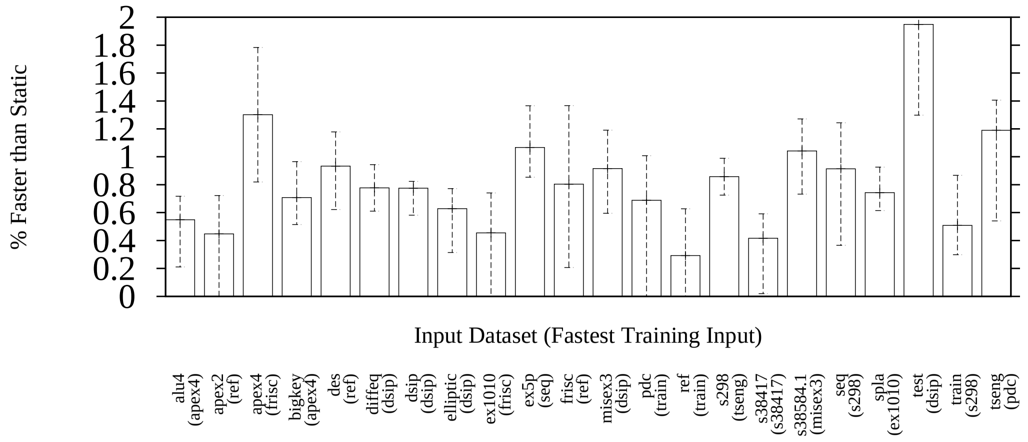


(a) Itanium

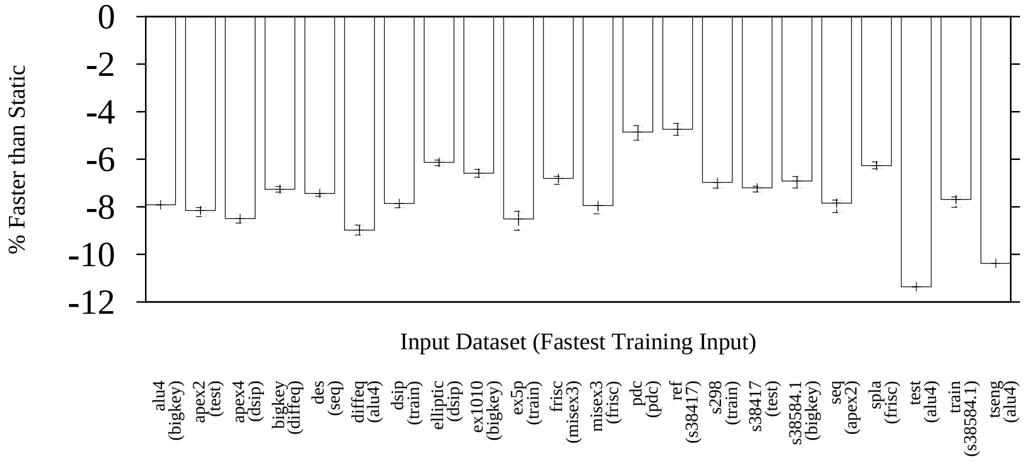


(b) Itanium 2

Figure 4.62: Static vs. FDO performance for `if` conversion on `VPR` (place)

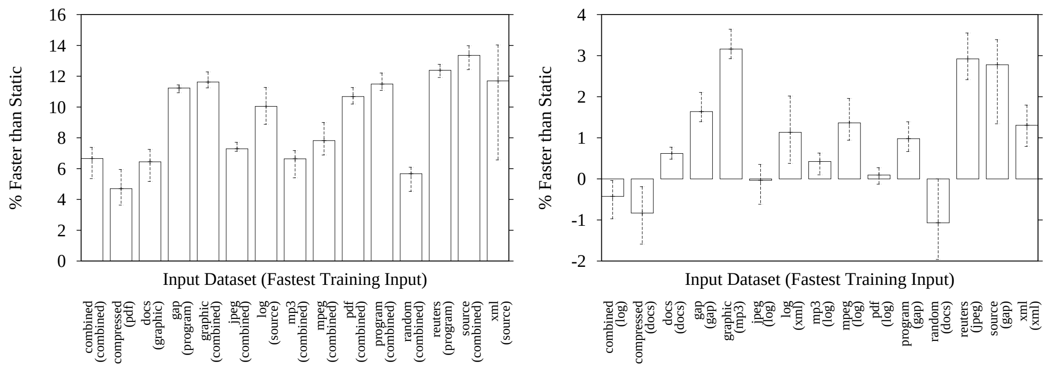


(a) Itanium



(b) Itanium 2

Figure 4.63: Static vs. FDO performance for `if` conversion on VPR (route)



(a) Itanium (b) Itanium 2

Figure 4.64: Static vs. FDO performance for inlining on `bzip2`

is commonly acknowledged to have little impact on performance, Amdahl's Law [23] (pp. 40-42) would dictate that limited compiler-developer resources should be used to address more significant issues. However, the results presented in this section suggest that FDO `if` conversion may have a more significant impact on program performance than previously expected, and that more investigation into this optimization may be warranted. Unfortunately, the importance of `if` conversion to performance does not appear to be in its potential to reduce program run times, but rather in the potential for poor `if` conversion decisions to significantly degrade performance.

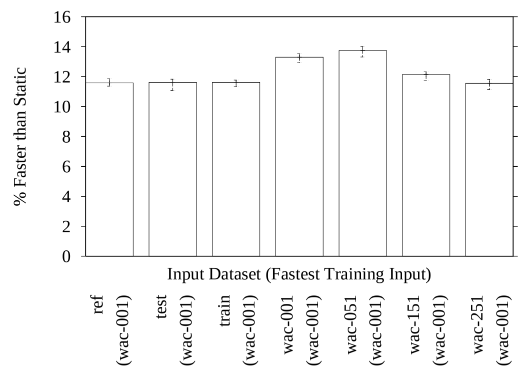
4.4.2 Inlining

Profile-guided inlining performs well against static optimization, particularly on the Itanium, where performance gains in excess of 10% are common. In the best case, performance is improved over static optimization by more than 20%.

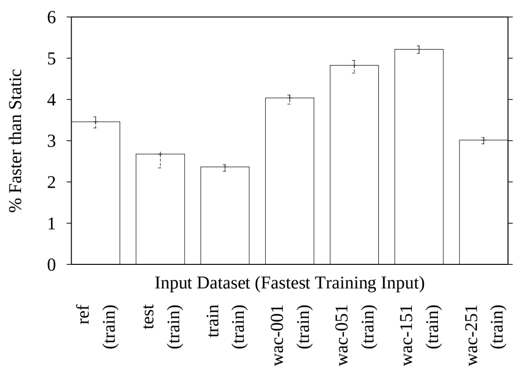
Figure 4.64 presents best-case FDO inlining for `bzip2`. Performance on the Itanium is very good, with a minimum improvement of about 4% and a maximum gain of about 13%. On the Itanium 2, performance gains are small, at most 3% faster than static. However, three inputs suffer a small loss in performance, but the loss is only 1% in the worst case.

The fastest FDO inlining binaries perform well on `crafty`, as shown in Figure 4.65. On the Itanium, FDO inlining is more than 11.5% faster than static optimization for all inputs. Results on the Itanium 2 are also positive, though the performance gains of the fastest FDO binary over static are much smaller. As was the case for `if` conversion, one FDO binary is the fastest for all inputs on each processor. The selection of the evaluation input could change the reported performance improvement over static optimization by 2.5%.

In Figure 4.66 best-case FDO inlining always improves performance for `GAP`. On the Itanium, improvements range from more than 3% to over 9%, while on the Itanium 2 improvements range from around 1% to 8%. Binaries trained on `snf525` and `snf750` account for most of the fastest

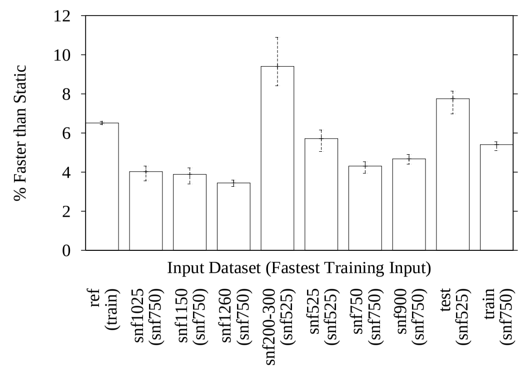


(a) Itanium

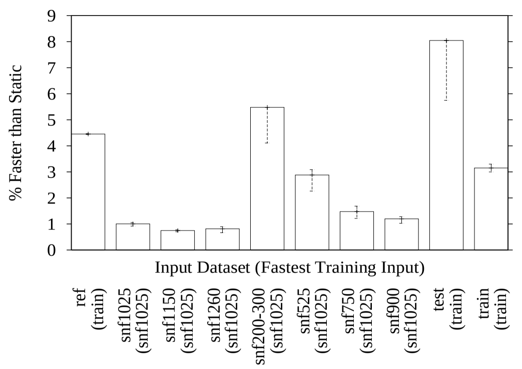


(b) Itanium 2

Figure 4.65: Static vs. FDO performance for inlining on `crafty`

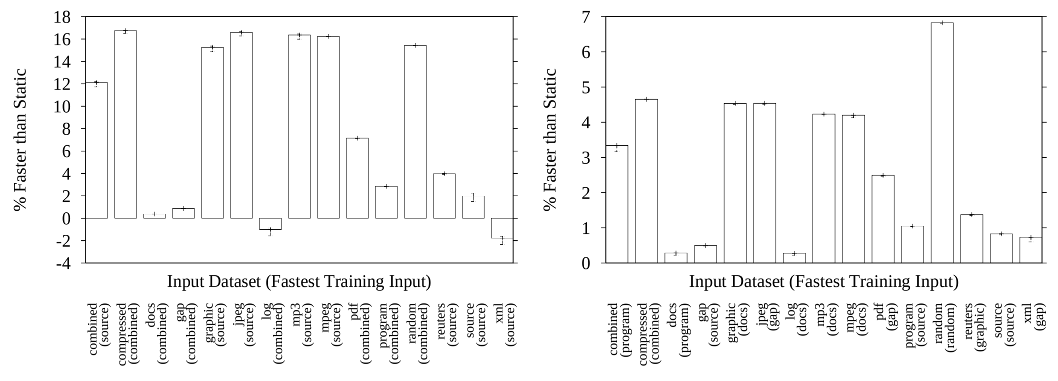


(a) Itanium



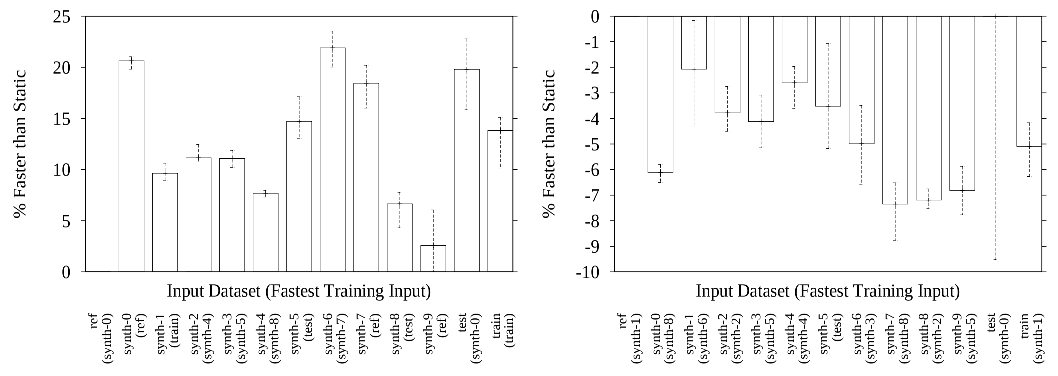
(b) Itanium 2

Figure 4.66: Static vs. FDO performance for inlining on `GAP`



(a) Itanium (b) Itanium 2

Figure 4.67: Static vs. FDO performance for inlining on `gzip`



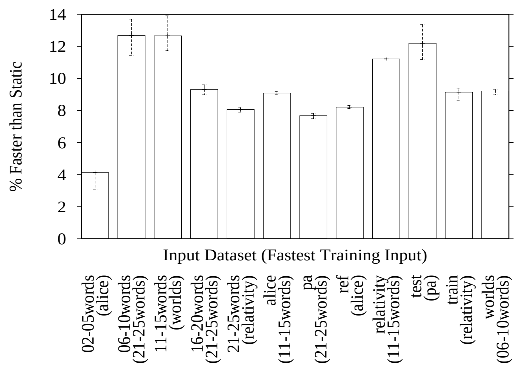
(a) Itanium (b) Itanium 2

Figure 4.68: Static vs. FDO performance for inlining on `MCF`

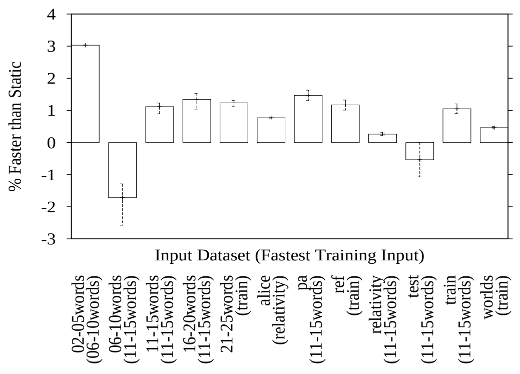
binaries on the Itanium, while `snf1025` produces most of the fastest binaries for the Itanium 2. Depending on which inputs are used to evaluate the performance of these binaries, performance varies by 3%, 2%, or 4.5%, respectively.

Figure 4.67 presents best-case FDO inlining results for `gzip`. On the Itanium, there are performance gains of over 12% for 7 of the 15 inputs, with gains over 16% on 4 inputs. However, there are also several inputs with small performance gains, and 2 which experience slightly reduced performance. It is noteworthy that the `log` input is provided by SPEC, and thus commonly used to evaluate performance, but still does not gain performance from the fastest FDO inlining binary. Results are similar on the Itanium 2. All inputs have improved performance compared to static optimization, though the gain is less than 1% for 5 inputs. However, 6 inputs have performance gains over 4%, with the maximum gain approaching 7%.

`MCF` displays the most dramatic results with FDO inlining, and highlights FDO's potential for performance improvement, as shown in Figure 4.68. In the worst cases on the Itanium, performance



(a) Itanium



(b) Itanium 2

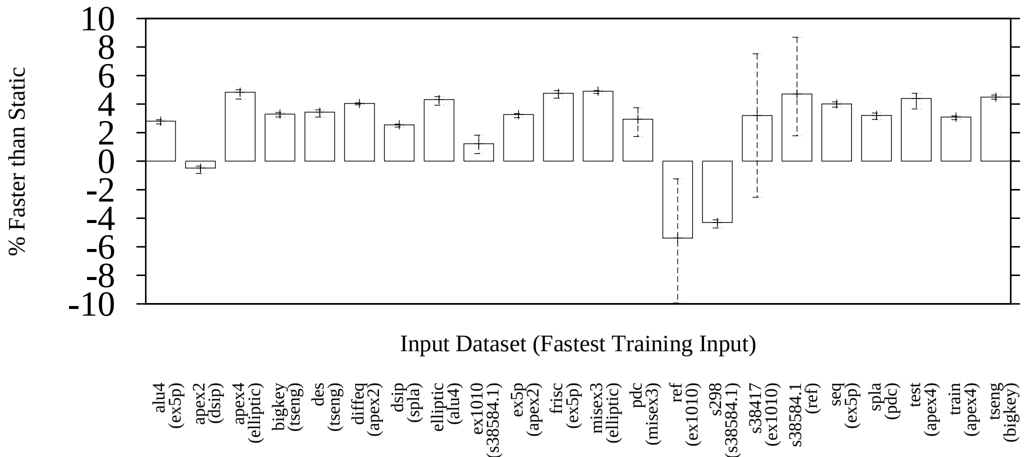
Figure 4.69: Static vs. FDO performance for inlining on `parser`

on `ref` is degraded by a small amount, while performance on `synth-9` is about 2.5% faster than static. For the rest of the workload, performance is at least 5% faster than with static inlining. In three cases, performance is improved by 20% or more. On the Itanium 2, these results are inverted. `MCF` on the Itanium 2 is the only case where even the fastest FDO inlining binaries consistently degrade performance. Static is faster than the best FDO inlining by at least 2% in all by two cases, and is more than 7% faster in the worst cases.

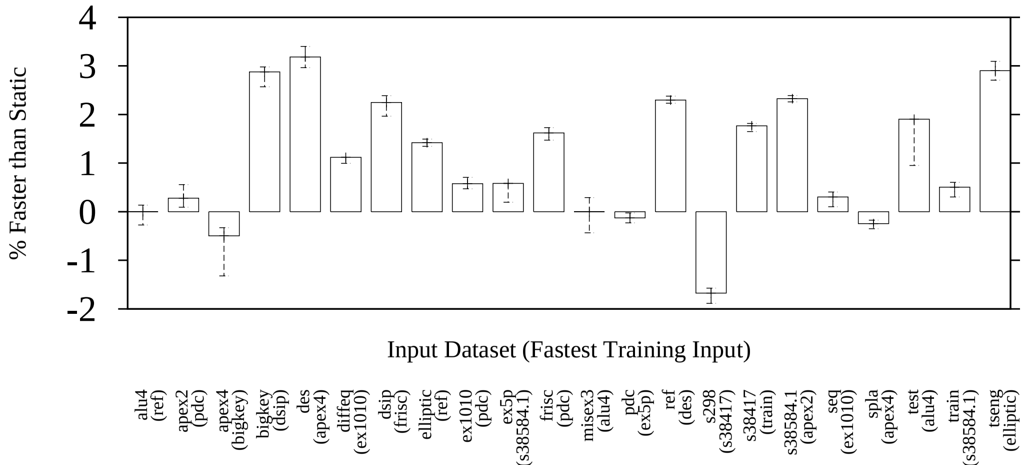
Results for parser are shown in Figure 4.69. The fastest FDO inlining binaries achieve significant performance gains over static for all inputs on the Itanium. On the other hand, while the fastest FDO binaries match the performance of static optimization on the Itanium 2, they do not result in any significant improvements in performance.

The fastest FDO inlining binaries usually exceed the performance of static optimization for the placement component of `VPR`, and achieve significant performance gains for `VPR` routing on the Itanium. However, performance for both the `ref` and `s298` inputs for placement is more than 4% slower than static. The fastest FDO inlining binaries generally achieve slightly better performance that static on the Itanium 2. However, there appears to be little potential for FDO inlining to improve performance compared to static optimization on the Itanium 2.

Overall, on the Itanium, FDO inlining exhibits the potential to significantly improve performance of each benchmark program on nearly every input in the workload. The best-case FDO performance is slower than static in only 6 out of 116 cases. Furthermore, the fastest FDO inlining binary is more than 10% faster than static in 41 cases. However, on the Itanium 2, while the fastest FDO inlining binaries match or exceed the performance of static optimization for 96 inputs, performance is only more than 3% faster than static in 19 of the 116 cases.

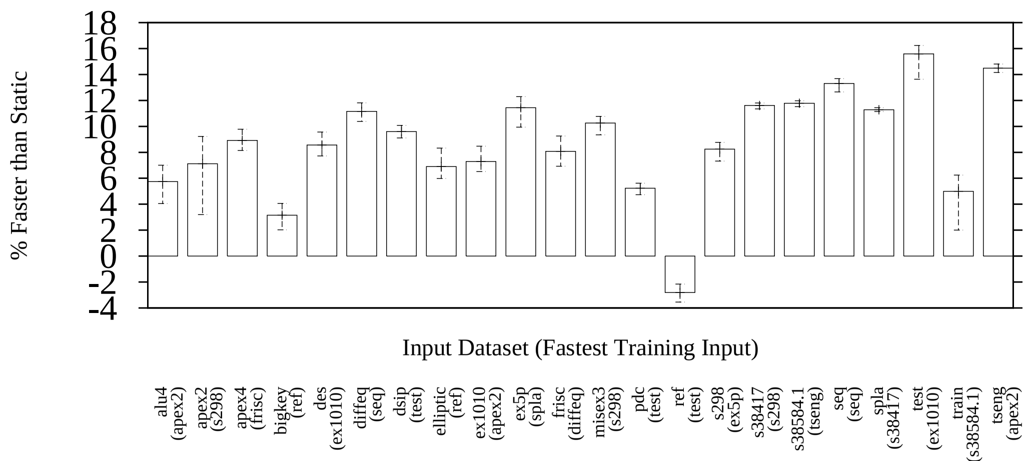


(a) Itanium

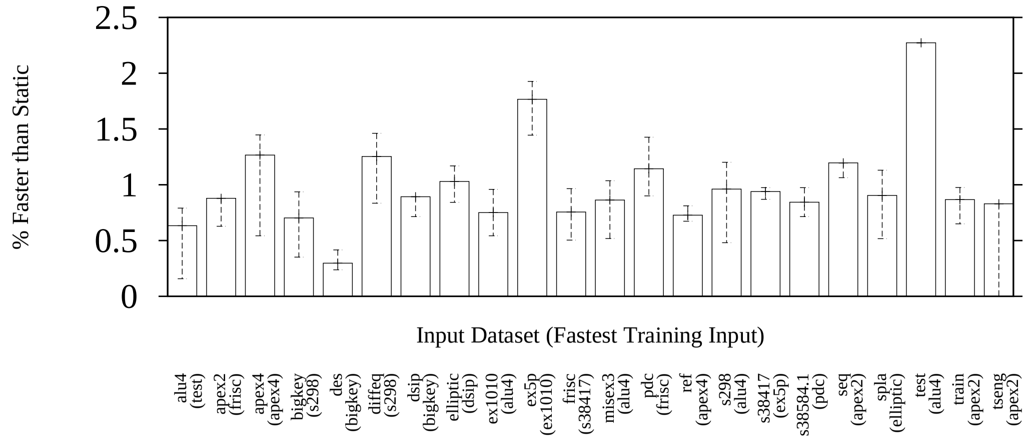


(b) Itanium 2

Figure 4.70: Static vs. FDO performance for inlining on VPR (place)



(a) Itanium



(b) Itanium 2

Figure 4.71: Static vs. FDO performance for inlining on VPR (route)

4.4.3 Conclusions

FDO appears to have the potential to substantially improve program performance on the Itanium. On the other hand, even the fastest FDO binaries are often significantly slower than static optimization on the Itanium 2. In particular, fastest FDO inlining binaries for MCF on the Itanium 2 are significantly slower than static for nearly all inputs. We find this fact surprising, given that both MCF and inlining are frequently studied. The performance results reported here are consistent with the performance results measured on the entire workload in Section 4.2. Possible explanations for better FDO results on the Itanium are that it has more resource limitations than the Itanium 2, and thus more potential for performance improvement, while the Itanium also benefits from a more mature code base since it is an older processor.

One unanticipated result of this study comes from the observation of the performance of the fastest FDO binaries for the crafty benchmark. For both if conversion and inlining, on both processors, one FDO binary had the best performance for all inputs. However, the performance of these binaries compared to static is not consistent across the workload. In particular, significantly different performance results are obtained for if conversion on the Itanium 2. If the SPEC training input train is used, performance on the SPEC evaluation input ref is 2% slower static, the best result from the workload. However, if the wac-051 input is used for evaluation, performance is reduced by more than 6%. This result confirms that evaluating optimization using a single evaluation input can lead to conclusions about performance that do not generalize to other program inputs.

Chapter 5

Related Work

5.1 Input Selection and Benchmarking

Input selection and benchmark creation are difficult, but important, tasks. Compiler writers, hardware designers, and system vendors all use benchmarks. However, the goals and requirements of these communities differ. System vendors may favor codes that are hard to optimize to help ensure fairer comparisons between systems, while compiler designers wish to investigate how a compiler feature affects the performance of typical programs. Where system vendors and compiler designers run programs on larger inputs to reduce measurement errors and better represent full system behavior on real-world problems, architecture researchers strive for the smallest representative inputs to limit simulation times.

In [21], Eeckhout *et al.* attempt to find a minimal set of representative programs and inputs for architecture research. They cluster program-input combinations using principal-component analysis based on low-level program behavior such as cache misses and branch mispredictions. They found that while different inputs to the same program were often clustered together, there were several cases where different inputs to the same program resulted in data points in separate clusters. This finding supports our conclusion that the input to a program does have an impact on program behavior.

Phansalkar *et al.* survey the four generations of the SPEC benchmark suite and investigate how the suite has evolved [34]. They measure low-level architecture-independent program behaviors such as instruction mix, basic-block size, various branch statistics, and locality. They use principal-component analysis to cluster and compare the benchmark programs. The benchmarks are found to have changed little in terms of static instruction count, branch behavior, or ILP. However, temporal locality has lessened in more recent benchmarks. The authors suggest that, based on their clustering, several benchmarks in the SPEC suites are redundant. Based on their overall characteristics, `bzip2` and `gzip` form the entirety of one cluster. Looking back to Chapter 4, *Aestimo* finds significantly different results for `bzip2` and `gzip` in nearly every case. Therefore, we caution compiler designers against omitting programs from a benchmark suite based on clustering analysis of low-level program behaviors.

MinneSPEC proposes reduced inputs to the SPEC CPU2000 benchmarks based on function-level execution profiles and instruction mix profiles to reduce simulation time for architecture research [26]. For more than half of the program-input pairs the reduced inputs have function profiles that are statistically similar to the original inputs, while they have instruction mixes similar to the original inputs in nearly every case. However, the authors warn that memory behavior may be substantially different with the reduced inputs. MinneSPEC inputs should not be considered equivalent to the original inputs supplied by SPEC. Eeckhout *et al.* analyze program behavior on the reduced inputs suggested by MinneSPEC [20]. They use a larger mix of behavior measures that are more closely related to program performance than those used to create the MinneSPEC inputs. PCA and clustering show that while the MinneSPEC set of large (lgred) inputs remain similar to the original SPEC inputs from which they are derived, the medium (mdred) and small (smred) input sets generally lead to dissimilar program behavior.

Citron has investigated the use of the SPEC benchmarks by research reported in computer architecture conferences [15]. He found that while the SPEC benchmarks are very commonly used, the suite is seldom used as intended. Two important issues are failure to use all the benchmark programs from the integer or floating-point collections, and infrequent use of the floating-point benchmarks. When reported results are adjusted by assuming that the reported techniques have no effect on missing benchmark programs from the collection used, large speedups were reduced to moderate speedups. For example, one reported speedup of 1.76 was reduced to 1.15. Our results compound this problem. We have shown that the training input used with FDO as well as the testing input used to evaluate performance can significantly vary the observed performance impact of an optimization. The common practice of using only the inputs supplied with the SPEC benchmarks is likely to further obscure the true performance impact of a technique when used outside the lab.

5.2 Feedback-Directed Optimization

Cohn and Lowney investigate FDO in Compaq's compiler tools for the Alpha processor using the SPEC CINT95 benchmarks [16].

They report the performance impacts when several FDO optimizations are applied individually. In particular, they find that FDO inlining improves performance by up to 45%, and by 10% on average over static inlining. While these results are similar to ours, they report that FDO inlining never results in a performance penalty. However, differences in compiler, architecture, and benchmark programs makes meaningful comparisons between the performance results impossible.

Langdale also investigates the sensitivity of FDO to the training data used [29]. The programs and inputs from the SPEC95 and SPEC2000 benchmark suites are used in conjunction with Digital's GEM compiler and the Alto link-time optimizer for the Alpha architecture. The study concludes that there is a statistically significant difference in performance when different training inputs are used. Our study expands on this work in two ways. First, we have used a large number of additional non-

SPEC inputs for both training and evaluation. Second, we have investigated individual optimizations that benefit from FDO rather than considering the entire FDO system as a whole. In our study, we have also observed variations in performance when different training inputs are used. However, the differences in performance in our study are much larger, and can be observed without resorting to statistical techniques. Langdale also investigates resubstitution, and concludes that profile accuracy is not tightly coupled to performance gains. We have also observed a general failure of resubstitution to achieve the best performance. However, given the frequently poor performance of FDO compared to static optimization, we believe that further improvements to the FDO system must be made before we can provide a final verdict on the usefulness of perfect information.

5.3 Compiler-Decision Optimization

Several researchers have used iterative compilation techniques to improve program performance. Iterative compilation is the pinnacle of FDO: a program is compiled and run repeatedly, while statistics collected at run time improve performance. However, in many cases, iterative compilation systems do not consider the impact of data inputs on the performance changes observed between different compilations. They often use a single input for both training and evaluation, and do not evaluate the performance of the final binary on any additional inputs.

Pan and Eigenmann break a program into regions, called Tuning Sections (TS), and attempt to find an optimal optimization strategy for each TS [33]. They compare the performance of multiple versions of each TS using three methods. Context-Based Rating is used if the same TS is executed frequently in the same execution context. In this case, versions of the TS can be swapped to determine their performance during a single run of the program. Model-Based Rating applies mathematical relationships between contexts to enable comparisons between versions of a TS executed in different contexts. Finally, Re-execution-Based Rating restores state and restarts execution at the beginning of a TS when different versions of a TS would be otherwise incomparable. Using these techniques, their offline compilation system based on GCC is able to improve performance on four SPEC 2000 benchmarks by an average of 26%, while reducing tuning time by 80%. Tuning is performed by running on the SPEC train inputs, while final performance evaluation uses the SPEC ref inputs. If the ref input is resubstituted instead, much larger performance gains are observed on two of the benchmarks. The performance improvement obtained by this approach is often small compared to the performance variations we have seen between inputs, or compared to the benefits of the usual FDO inlining used in our study. In 5 of 8 cases, the largest performance gain for a benchmark is less than 4%, and is less than 10% in another two cases. Average performance is inflated by the remaining case, where the technique improves performance by more than 170%. Therefore, we suspect that normal FDO should provide a larger and more consistent benefit when applied across a larger collection of programs.

Cooper *et al.* [17] and Kulkarni *et al.* [28] find solutions to the problem of ordering the phases in a compiler using genetic algorithms. Iterative compilation obtains performance improvements for the given program and input data. We have observed that testing a version of a program on different inputs can lead to different conclusions regarding the performance of that version of the program. Therefore, using a single input for training and performance evaluation during iterative compilation may result in a final program that does not have the best performance in general.

Stephenson *et al.* also uses genetic algorithms to learn compiler heuristics for hyperblock formation, register allocation, and data prefetching. They observe significant performance differences between running resubstitution and non-resubstitution cases for some programs, which indicates that over-fitting heuristics to input data is a danger. This result compounds the implications of our findings, and further cautions against the use of a single (or small sample) of inputs when evaluating FDO techniques.

Cooper and Waterman use iterative compilation to determine the optimal blocking size for a matrix of a fixed size with a matrix multiplication kernel [18]. Execution time is significantly improved as the matrix dimensions grow because the profile-guided compiler heuristic fails to consider cache size. It is unfortunate that rather than correcting this deficiency in the compiler, they propose an iterative compilation technique that bypasses the problem. In our study, we also discovered a deficiency in compiler heuristics regarding `if` conversion. We suggest that the `if` conversion heuristics of the ORC should be amended before any other technique uses them as the basis for comparison.

Chapter 6

Conclusion

6.1 Future Work

This study has raised several new questions. First, is there a metric that can link compile-time decisions to performance impacts? Most likely, such a metric will need to be able to identify "important" choices, but it is unclear if there is a way to estimate the importance of a choice short of some sort of iterative compilation framework. After all, compiler heuristics already attempt to make the best decisions for the most important choices in order to maximize program performance. However, it may be possible to further analyze our data to determine decision importance "after the fact", and then use this information to evaluate and possibly augment the existing compiler heuristics. In particular, since feedback-directed `if` conversion reduces performance compared to static `if` conversion, there is clearly an opportunity to improve the `if` conversion heuristics.

In a similar vein, we would like to understand how our inputs differ, and how these differences impact optimization decisions. PCA and clustering techniques could be applied, but we have seen in Chapter 5 that it can be difficult to get meaningful results from these techniques. Furthermore, these techniques rely on a set of aggregate measure to characterize program performance. These measures are well-known to be important for architecture research where these techniques have been used.

However, a different approach is required in the realm of compilers and FDO. While a particular architecture must use the same branch-prediction mechanism for every branch encountered during the execution of every program, a compiler must make an `if`-conversion decision for each branch in a program, and makes each decision individually. Furthermore, for many optimizations only a small number of choices have a significant impact on program performance. It is therefore doubtful that aggregate measures can adequately characterize inputs for use by a compiler. In fact, it is likely that the failure of the difference and alignment metrics to correspond in any consistent way to performance is due to this same problem.

Therefore, it would be advantageous to develop input comparison techniques that work at a level similar to that present in a compiler. These techniques should work at the control-flow graph and call-graph levels, rather than using low-level measures like ILP and cache miss rates. We have started development of a tool called `ProfEdit` that is an initial step in this direction. At present, `ProfEdit` is an interactive program that allows a user to view and modify the frequency counts stored in an ORC profile file. However, the profile does not contain information about the structure of the program. Thus, it is impossible to maintain the consistency of profile information. An extension of `ProfEdit` would allow the frequencies recorded in different profile files to be compared. However, the volume of information being compared necessitates the use of a summarization technique for the results to be manageable by a human compiler designer. It is unclear what sort of summarization would reduce the quantity of such data to an understandable volume without unacceptably compromising its usefulness.

Another problem regarding input selection is that the space of possible profiles, as well as the location of a profile from a particular input in this space, is unknown. If program structure information can be integrated into `ProfEdit`, it could be used as part of a system to automatically explore the space of possible profiles, without the need to find actual inputs that correspond to any of the particular profiles used for exploration. If the profile-space of a program is characterized, and inputs can be mapped into this profile space, then the distance between inputs in this space can be determined, and the distribution of inputs in an evaluation workload can be measured. Furthermore, if certain areas in the profile space are found to be "interesting," the feasibility of real inputs mapping into that area can be investigated.

Finally, more study is needed. Similar experiments should be run using different compilers and different architectures, and should investigate a larger range of optimizations and programs in order to increase the generality of any conclusions about FDO's sensitivity to training inputs.

6.2 Conclusions

Our extensive experimental study provides important insights into feedback-directed optimization. Most significantly, training on different inputs does lead to different optimization decisions and different levels of performance in the FDO binaries in most cases. Training on different inputs results in as much as a 5% difference in performance with `if` conversion, and as much as a 6% difference in performance with inlining, on a workload of inputs. On the other hand, evaluating FDO performance on different inputs can lead to substantially different performance results. We observe differences in the best case FDO performance on different inputs for the same program larger than 13% for `if` conversion, and larger than 20% for inlining. Therefore, the selection of training inputs for FDO does impact performance. Furthermore, the measured performance for any particular binary is dependent on the inputs used for testing. Consequently, performance evaluations that use multiple training inputs as well as multiple evaluation inputs will result in more reliable performance measures than typical compiler and architecture evaluations that use a single training input and a single evaluation input.

Furthermore, these results enable an assessment of the FDO infrastructure of the ORC. Resubstitution often results in the fast binaries on a given input. However, rank analysis shows that the high levels of performance of resubstitution binaries compared to the fastest FDO binaries are a result of small difference in performance between FDO binaries. The rank of resubstitution binaries cover the full spectrum of possible ranks, from best performance to worst performance, for each program in this study. Furthermore, there are several cases were resubstitution is substantially slower than training on a different input. Resubstitution is more than 17% slower than the fastest binary for `if` conversion, and nearly 12% slower than the fastest binary for inlining in the worst cases. Therefore, the FDO system in the ORC does not make effective use of the more accurate profile information provided by resubstitution.

In general, feedback-directed inlining is effective at increasing performance on both a workload and on individual inputs. However, we also observe that feedback-directed `if` conversion seldom improves performance. In fact, it always reduces performance on the Itanium 2, which strongly suggests that further work is required for the `if` conversion heuristics to use profile information effectively.

Bibliography

[1] Open research compiler for Itanium^{TM} processor family. http://ipf-orc.sourceforge.net/. Latest release: ORC 2.1, July 15, 2003.

[2] Open64 compiler. http://sourceforge.net/projects/open64/, http://sourceforge.net/projects/open64/. Latest release: Open64 0.16, March 21, 2003. Page maintainer: Alban Douillet (douillet@capsl.udel.edu).

[3] ORC performance on Itanium 2/Linux. http://ipf-orc.sourceforge.net/orc2-1_itanium2-Perf.ppt. Powerpoint Graph.

[4] ORC performance on Itanium/Linux. http://ipf-orc.sourceforge.net/orc2-1_itanium1-Perf.ppt. Powerpoint Graph.

[5] SGI pro64 compiler. http://oss.sgi.com/projects/Pro64/. webpage copyright 1993-2003 Silicon Graphics, Inc.

[6] Thomas Ball and James R. Larus. Branch prediction for free. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation (PLDI 93)*, volume 28, pages 300–313, June 1993.

[7] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, July 1994.

[8] Thomas Ball and James R. Larus. Efficient path profiling. In *MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 46–57, Washington, DC, USA, 1996. IEEE Computer Society.

[9] Vaughn Betz. FPGA place-and-route challenge. http://www.eecg.toronto.edu/∼vaughn/challenge/challenge.html. University of Toronto, Department of Electrical and Computer Engineering.

[10] Lewis Carroll. *Alice's Adventures in Wonderland*. Project Gutenberg, January 1991. http://www.gutenberg.org/etext/11.

[11] John Cavazos, J. Eliot, and B. Moss. Inducing heuristics to decide whether to schedule. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 183–194, New York, NY, USA, 2004. ACM Press.

[12] Pohua P. Chang, Scott A. Mahlke, and Wen mei W. Hwu. Using profile information to assist classic code optimizations. *Software – Practice and Experience*, 21(12):1301–1321, 1991.

[13] Youngsoo Choi, Allan Knies, Luke Gerke, and Tin-Fook Ngai. The impact of if-conversion and branch prediction on program execution on the Intel Itanium processor. In *34th Annual International Symposium on Microarchitecture (MICRO'01)*, pages 182–191, 2001.

[14] Kingsum Chow and Youfeng Wu. Feedback-directed selection and characterization of compiler optimizations. In *MICRO 32*, Isreal, Nov 1999.

[15] Daniel Citron. MisSPECulation: Partial and misleading use of SPEC CPU2000 in computer architecture conferences. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA'03)*, pages 52–59, 2003.

[16] Robert Cohn and P. Geoffrey Lowney. Feedback directed optimization in Compaq's compilation tools for Alpha. In *2^{nd} ACM Workshop on Feedback-Directed Optimization*, Haifa, Israel, November 1999.

[17] Keith D. Cooper, Devika Subramanian, and Linda Torczon. Adaptive optimizing compilers for the 21st century. *J. Supercomput.*, 23(1):7–22, 2002.

[18] Keith D. Cooper and Todd Waterman. Investigating adaptive compilation using the MIPSpro compiler. In *Los Alamos Computer Science Institute Symposium*, 2003.

[19] Standard Performance Evaluation Corporation. SPEC: The standard performance evaluation corporation. http://www.spec.org/.

[20] Liven Eeckhout, Hans Vandierendonck, and Koen De Bosschere. Designing computer architecture research workloads. In *IEEE Computer*, volume 36, pages 65–71, February 2003.

[21] Liven Eeckhout, Hans Vandierendonck, and Koen De Bosschere. Quantifying the impact of input data sets on program behavior and its applications. *Journal of Instruction-Level Parallelism*, 5:1–33, 2 2003.

[22] Albert Einstein. *Relativity : the Special and General Theory*. Project Gutenberg, January 2004. http://www.gutenberg.org/etext/5001.

[23] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 3rd edition, 2003.

[24] Apple Computer Inc. Apple iTunes. http://www.apple.com/itunes/. version 4.7.

[25] Toru Kisuki, Peter M. W. Knijnenburg, and Michael F. P. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *IEEE PACT*, pages 237–248, 2000.

[26] AJ KleinOsowski and David J. Lilja. MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research. In *Computer Architecture Letters*, volume 1, June 2002.

[27] Mike Krahulk and Jerry Jerry Holkins. http://www.penny-arcade.com/.

[28] Prasad Kulkarni, Stephen Hines, Jason Hiser, David Whalley, Jack Davidson, and Douglas Jones. Fast searches for effective optimization phase sequences. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 165–198, 2004.

[29] Geoff Langdale. *The Effect of Profile Choice and Profile Gathering Methods on Profile-Driven Optimization Systems*. PhD thesis, Carnegie-Mellon University, 2004.

[30] David D. Lewis. Reuters-21578 text categorization test collection. http://www.daviddlewis.com/resources/testcollections/reuters21578/, May 2004. Distribution 1.0.

[31] Scott Alan Mahlke. *Exploiting Instruction Level Parallelism in the Presence of Conditional Branches*. PhD thesis, University of Illinois at Urbana-Champaign, 1996.

[32] Hewlett Packard. Inside the Intel Itanium 2 processor. Technical report, Hewlett Packard Developer & Solution Partner Program, July 2002. http://h21007.www2.hp.com/dspp/ddl/ddl_Download_File_TRX/1,1249,952,00.pdf.

[33] Zhelong Pan and Rudolf Eigenmann. Rating compiler optimizations for automatic performance tuning. In *ACM/IEEE Conference on High Performance Networking and Computing (SC04)*, pages 14–23, November 2004.

[34] Aashish Phansalkar, Ajay Joshi, Lieven Eeckhout, and Lizy K. John. Measuring program similarity: Experiments with SPEC CPU benchmark suites. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2005.

[35] Martin Schoenert and Steve Linton. Re: [GAP support] additional inputs for 254.gap. Personal email correspondence, April 2005.

[36] Michael D. Smith. Overcoming the challenges to feedback-directed optimization. In *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo'00)*, pages 1–11, Boston, MA, January 2000.

[37] Mark Stephenson, Saman Amarasinghe, Martin Martin, and Una-May O-Reilly. Meta optimization: Improving compiler heuristics with machine learning. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 77–90, 2003.

[38] Reinhold Weicker and Kaivalya Dixit. (osgcpu-10955) re: Your question to SPEC about input data selection for benchmarks. Personal email correspondences, July 2004.

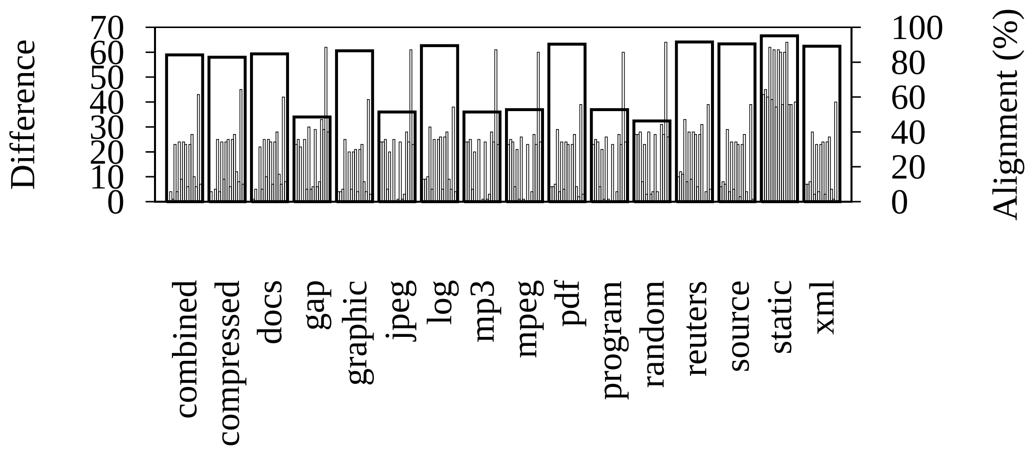
[39] Herbert George Wells. *The War of the Worlds*. Project Gutenberg, October 2004. http://www.gutenberg.org/etext/36.

[40] Peng Zhao and José Nelson Amaral. To inline or not to inline? Enhanced inlining decisions. In *Languages and Compilers for Parallel Computing: 16th International Workshop*, October 2003.

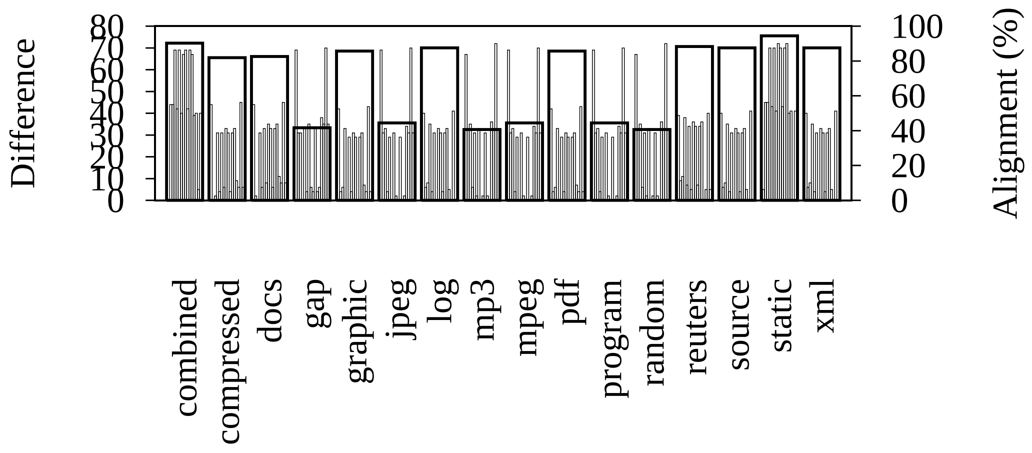
Appendix A

Metric Values

This appendix presents the raw data for the difference and alignment metrics defined in Chapter 3. These graphs are similar to the first one in Figure A.1(a). The name of the training input who's log is used to calculate the metrics is listed below the graph. The wide bar represents the alignment score (as a percent), and encompasses the narrow bars which show pairwise difference scores. The difference bars are in the same order as the coverage bars. For example, the log for compressed has a coverage score just over 80%, while GAP has a coverage score just over 50%. Within the large bar for compressed coverage, we see that the first difference score, $\delta(compressed, compressed)$, is 0, and the second difference score, $\delta(compressed, docs)$, is about 2.25.

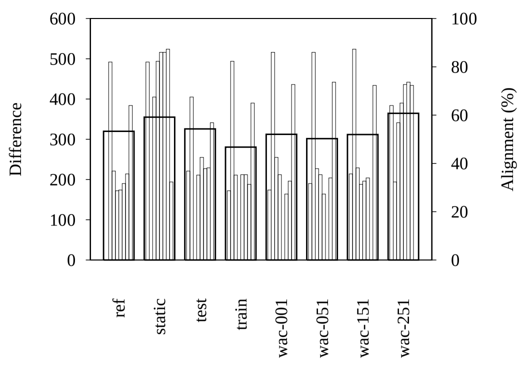


(a) Itanium

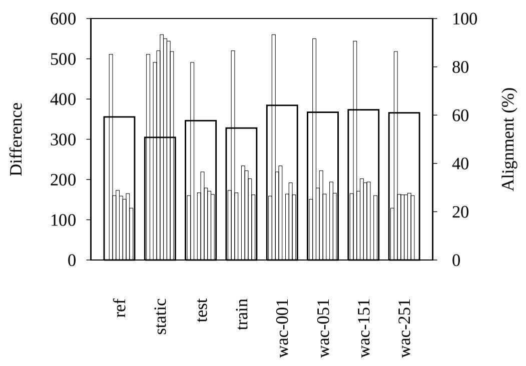


(b) Itanium 2

Figure A.1: Metric scores for `if` conversion on `bzip2`

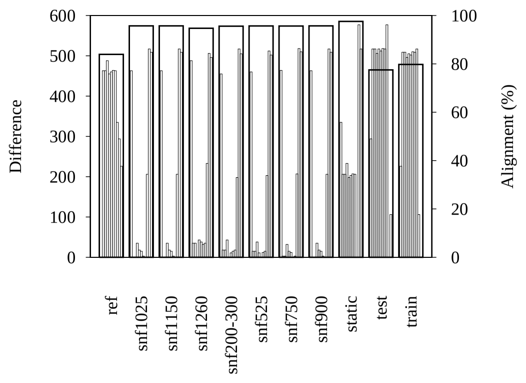


(a) Itanium

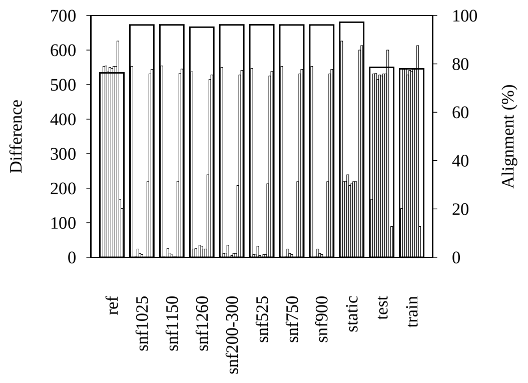


(b) Itanium 2

Figure A.2: Metric scores for `if` conversion on `crafty`

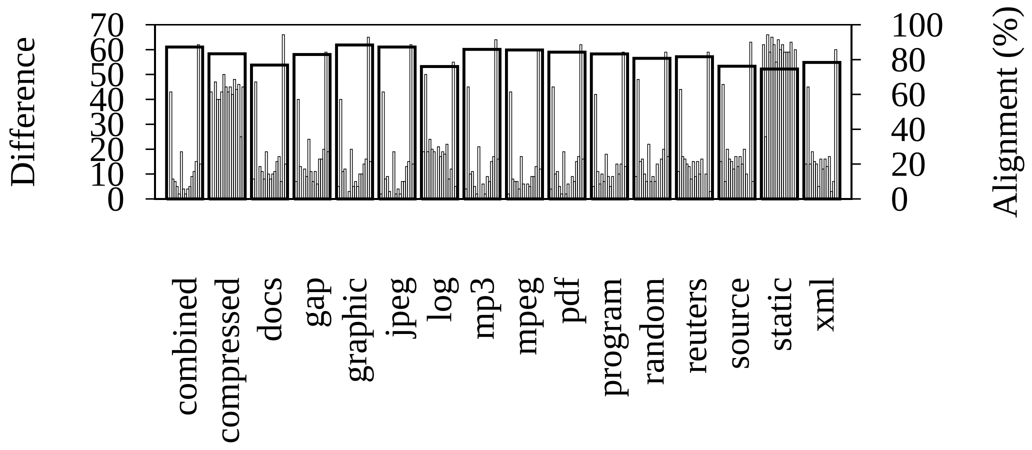


(a) Itanium

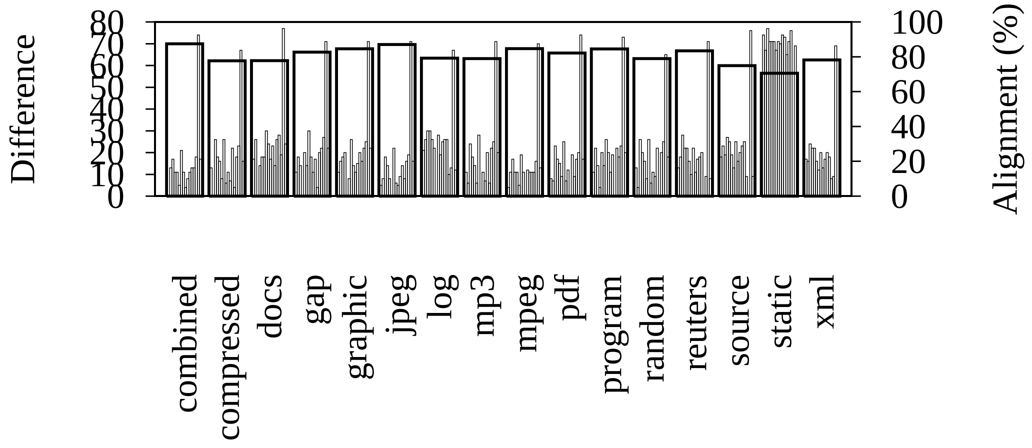


(b) Itanium 2

Figure A.3: Metric scores for `if` conversion on `GAP`

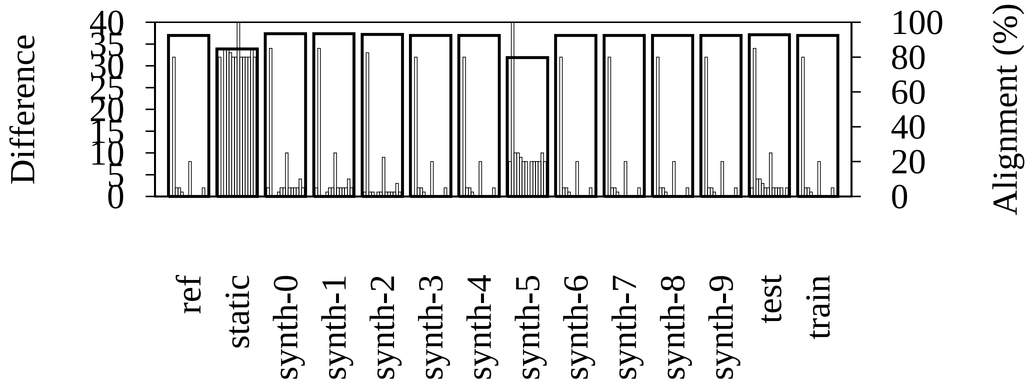


(a) Itanium

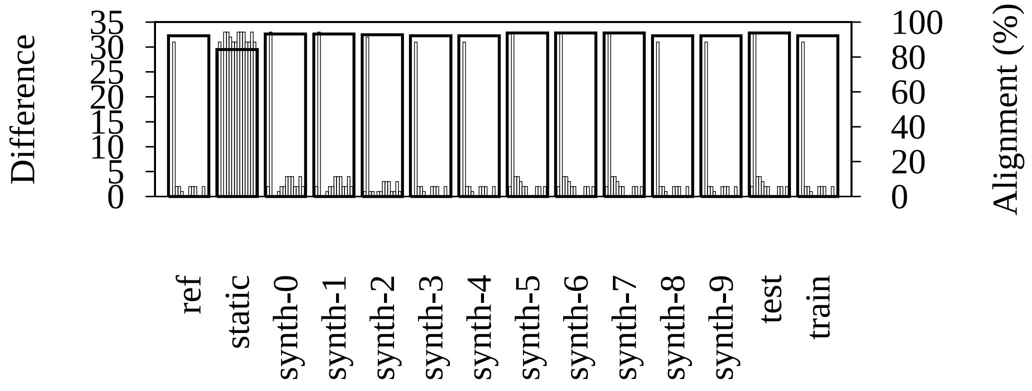


(b) Itanium 2

Figure A.4: Metric scores for `if` conversion on `gzip`

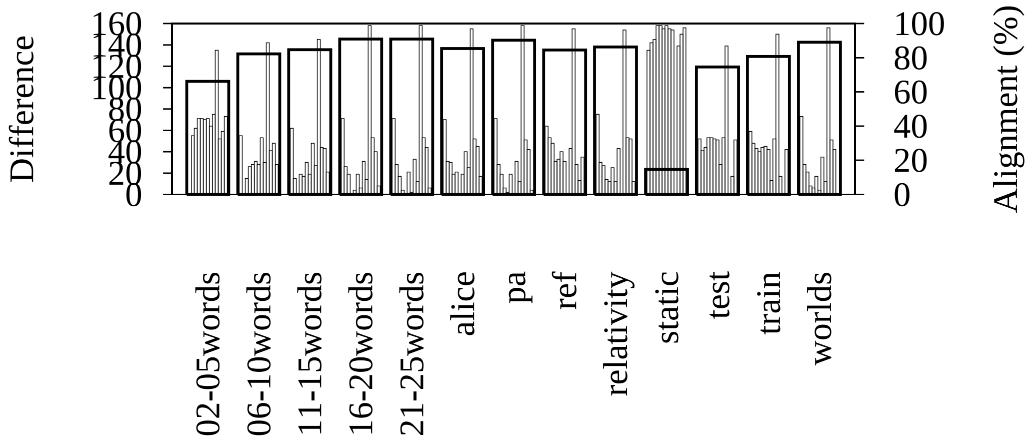


(a) Itanium

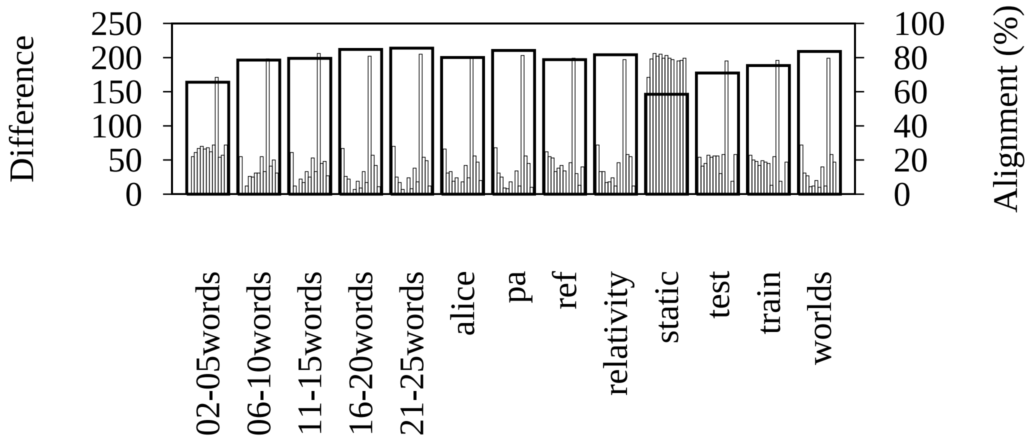


(b) Itanium 2

Figure A.5: Metric scores for `if` conversion on `MCF`

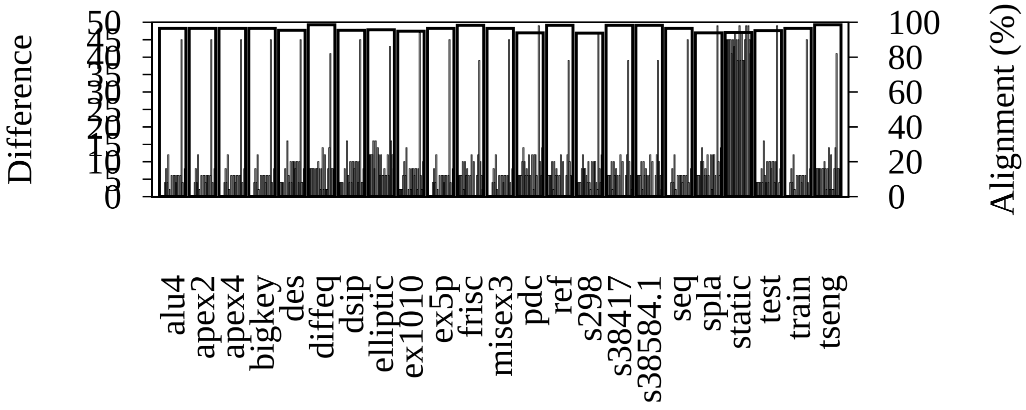


(a) Itanium

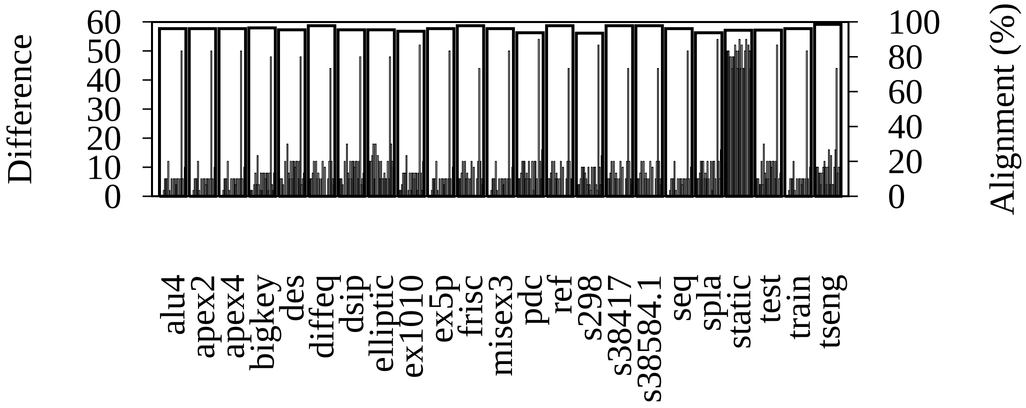


(b) Itanium 2

Figure A.6: Metric scores for `if` conversion on `parser`

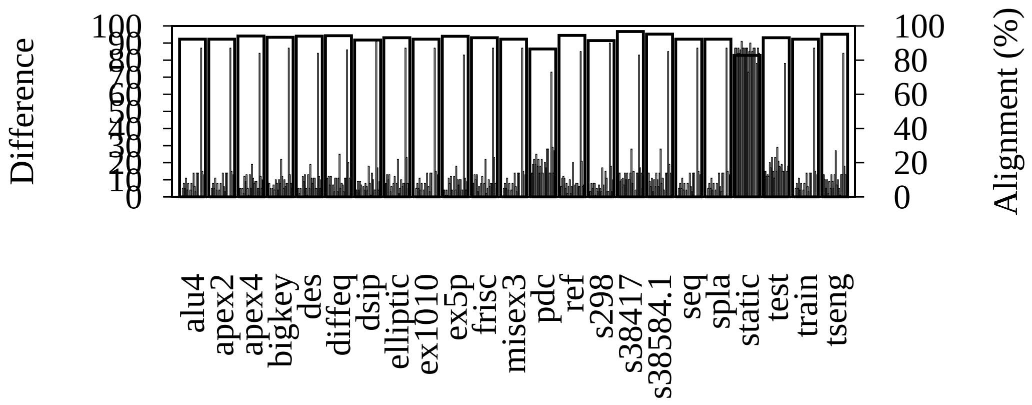


(a) Itanium

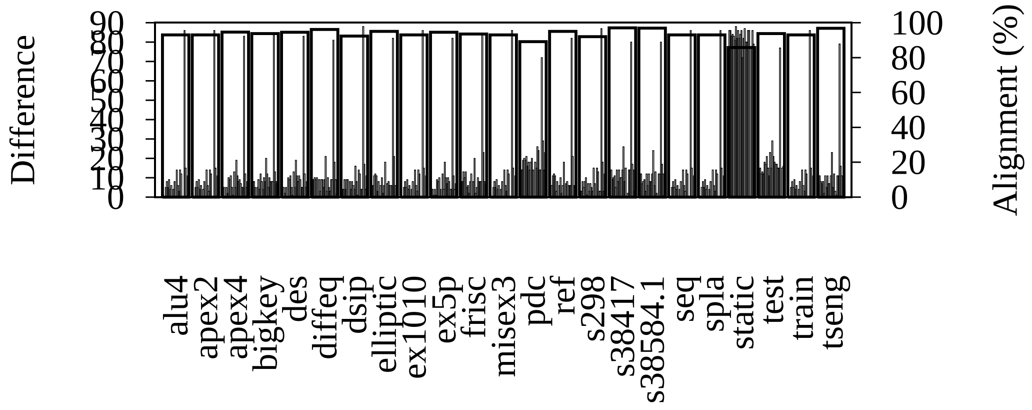


(b) Itanium 2

Figure A.7: Metric scores for `if` conversion on VPR (place)

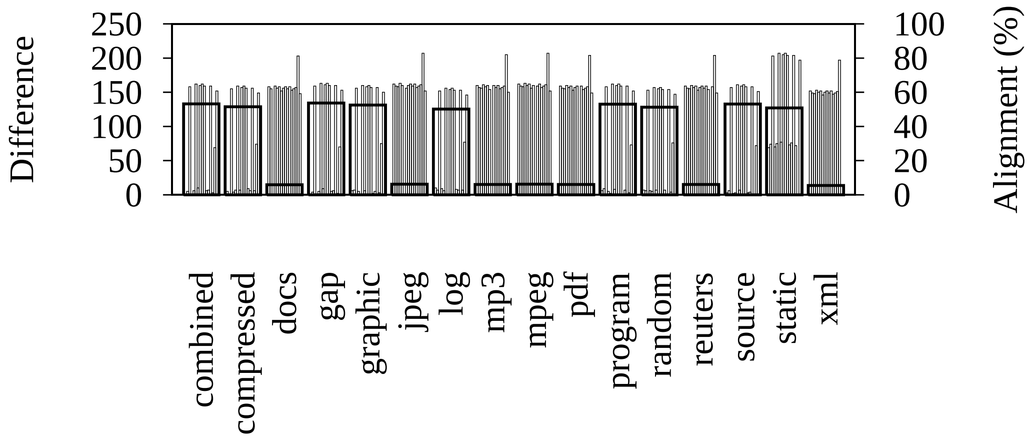


(a) Itanium

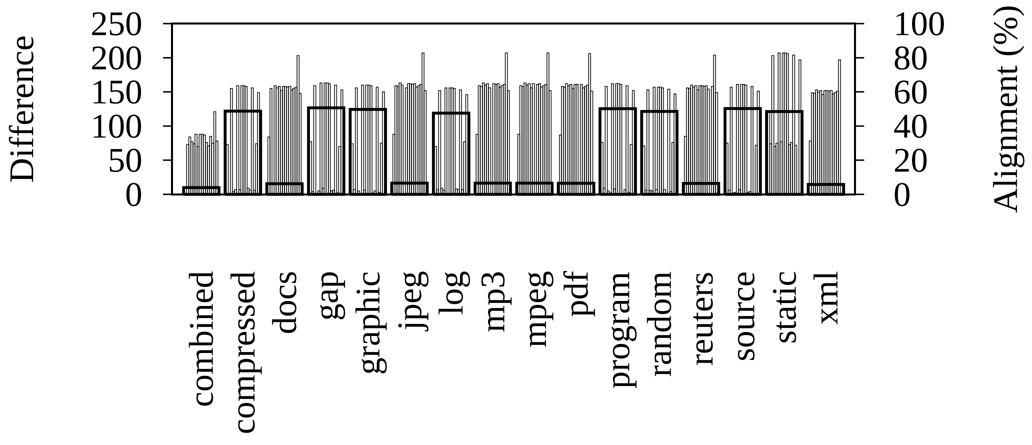


(b) Itanium 2

Figure A.8: Metric scores for `if` conversion on VPR (route)

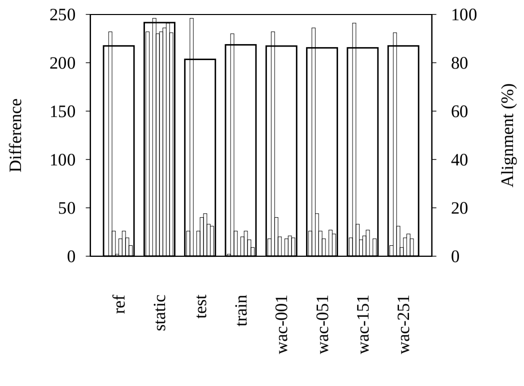


(a) Itanium

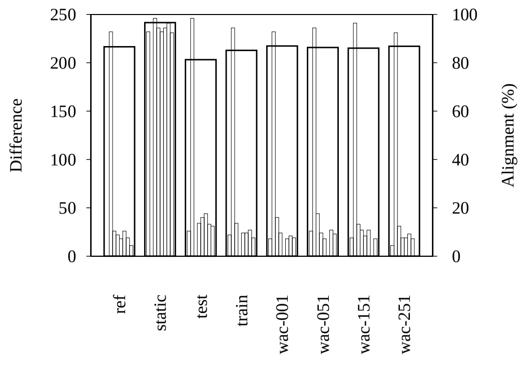


(b) Itanium 2

Figure A.9: Metric scores for inlining on `bzip2`

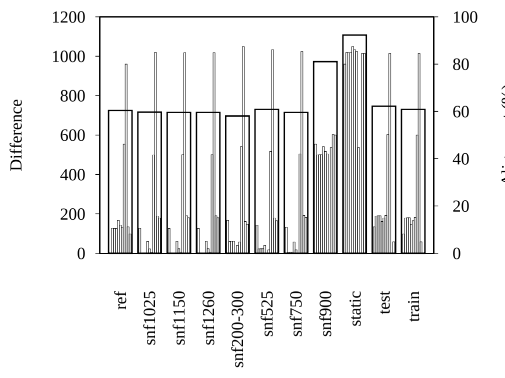


(a) Itanium

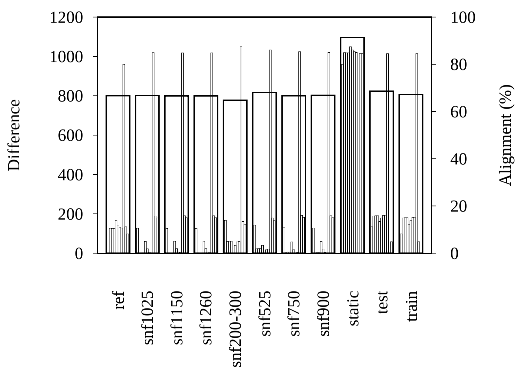


(b) Itanium 2

Figure A.10: Metric scores for inlining on `crafty`

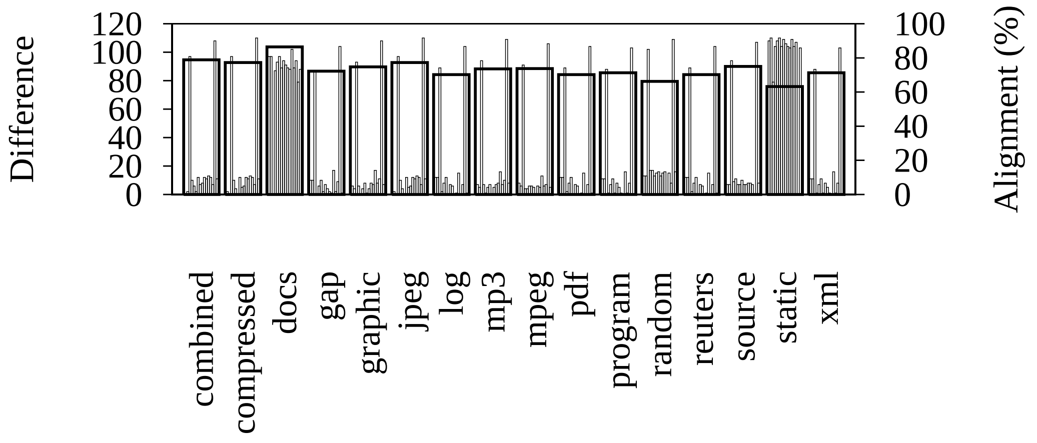


(a) Itanium

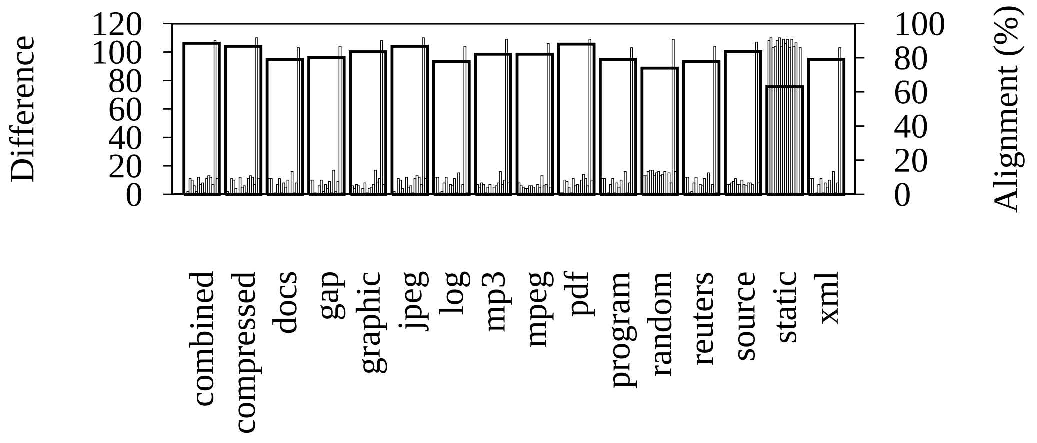


(b) Itanium 2

Figure A.11: Metric scores for inlining on GAP

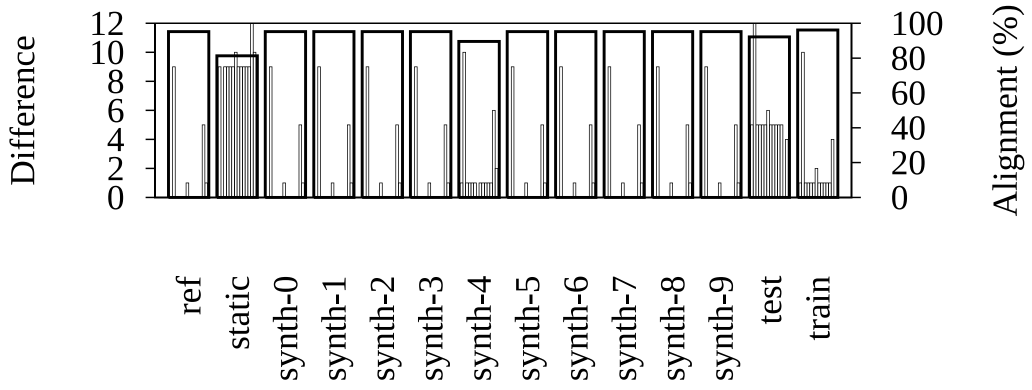


(a) Itanium

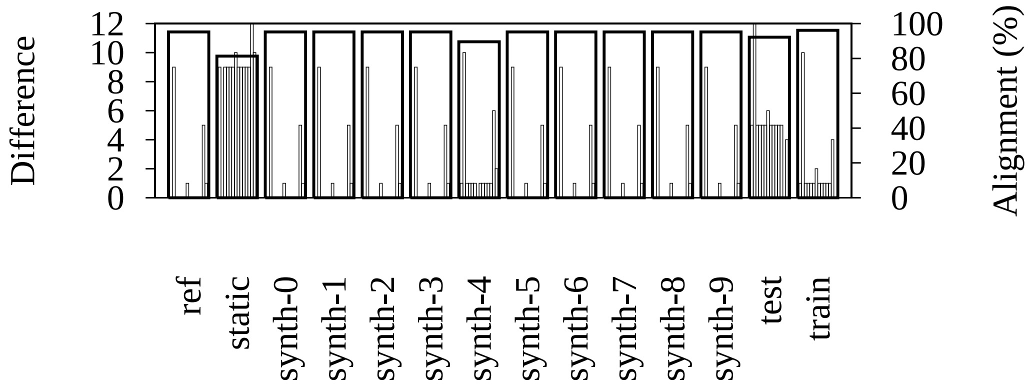


(b) Itanium 2

Figure A.12: Metric scores for inlining on `gzip`

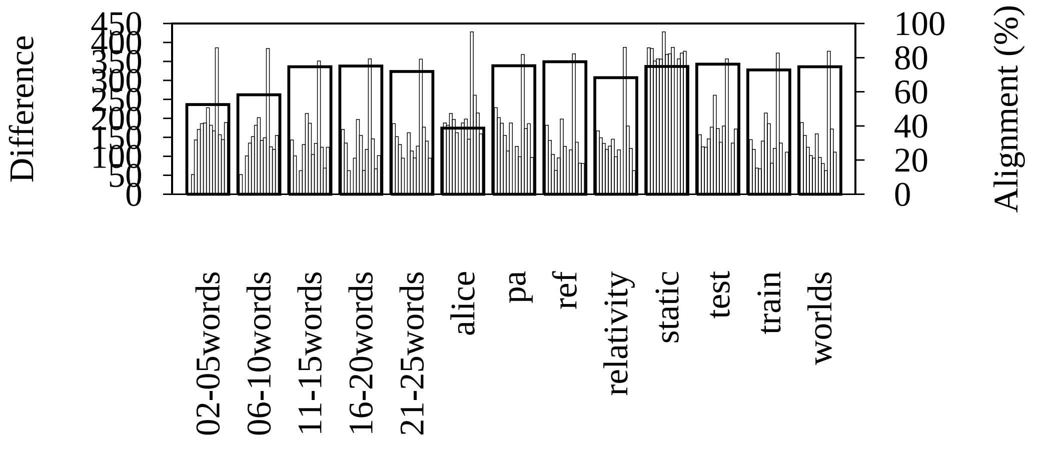


(a) Itanium

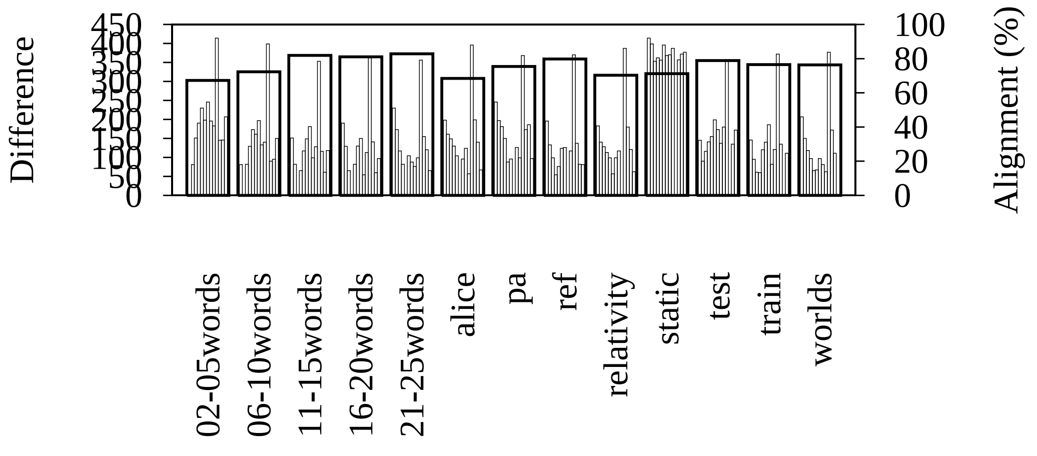


(b) Itanium 2

Figure A.13: Metric scores for inlining on MCF

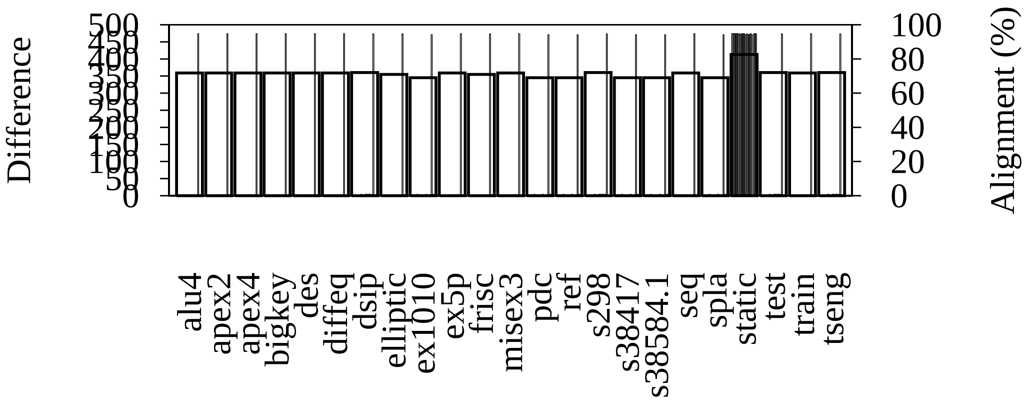


(a) Itanium

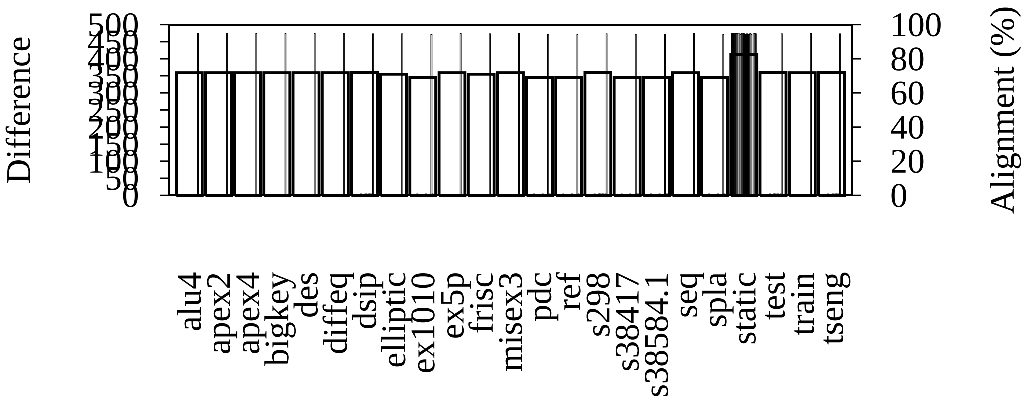


(b) Itanium 2

Figure A.14: Metric scores for inlining on `parser`

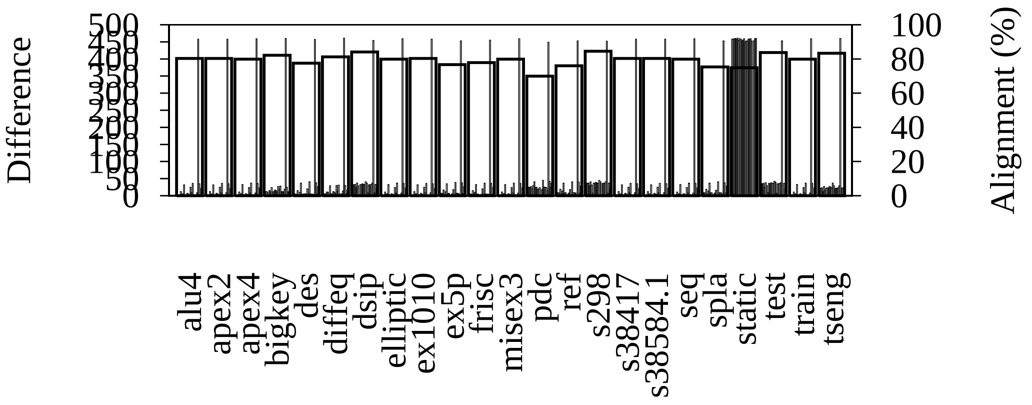


(a) Itanium

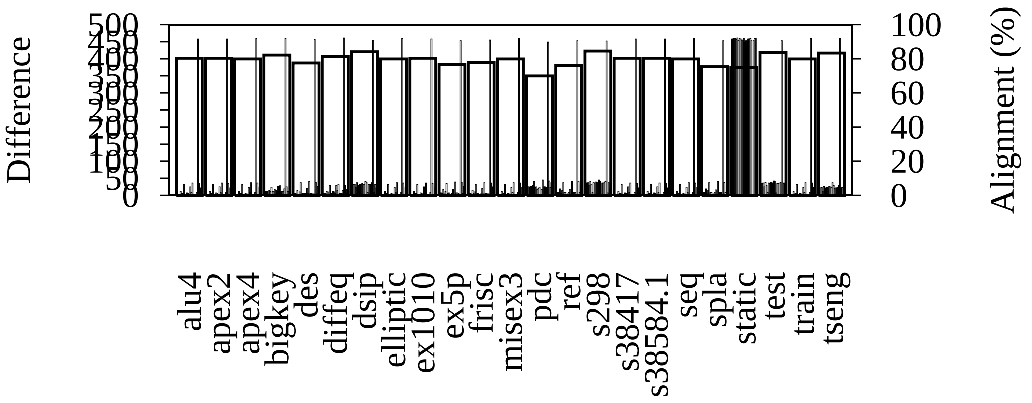


(b) Itanium 2

Figure A.15: Metric scores for inlining on VPR (place)



(a) Itanium



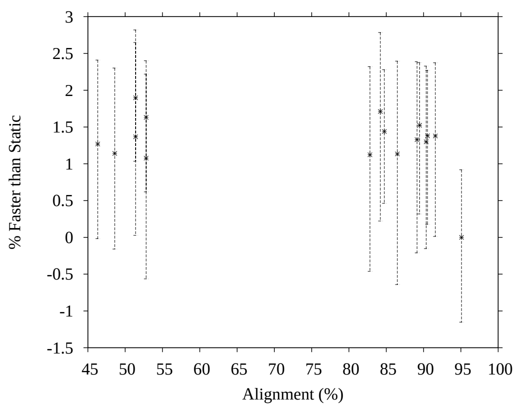
(b) Itanium 2

Figure A.16: Metric scores for inlining on VPR (route)

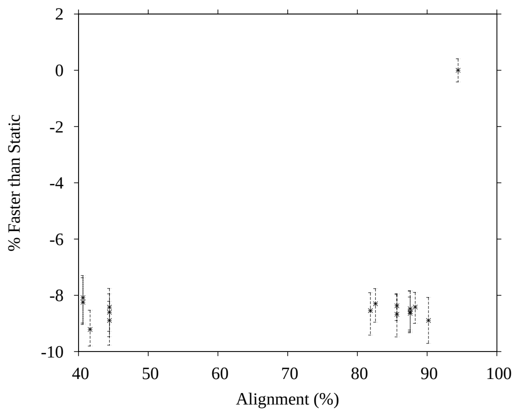
Appendix B

Alignment vs Performance

In the following graphs, there is one point for each of the binaries for a benchmark. The x-axis of the graph represents the alignment score for the optimization log used to create the binary. The y-axis represents the performance of the binary on the workload as a percent faster than the static binary, while the error-bars show the variance in performance. Performance is computed using the arithmetic measure exactly as in Chapter 4. There is one point for the static binary, which always has a performance value of 0% faster than static. This point shows the alignment score for static, as well as the variance in performance of the static binary.

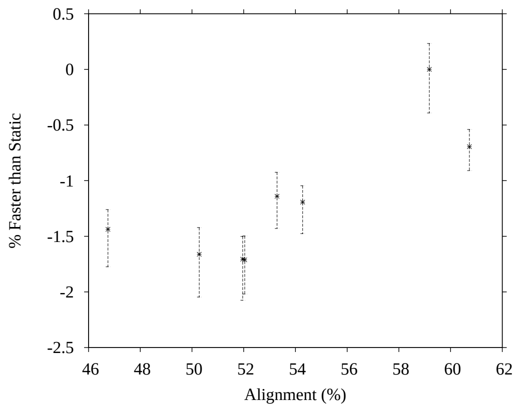


(a) Itanium

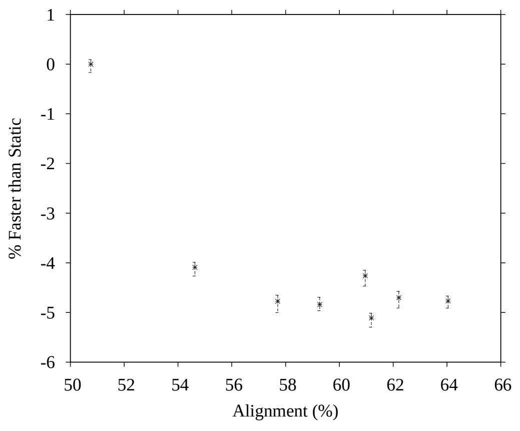


(b) Itanium 2

Figure B.1: Alignment vs. performance for `if` conversion on `bzip2`

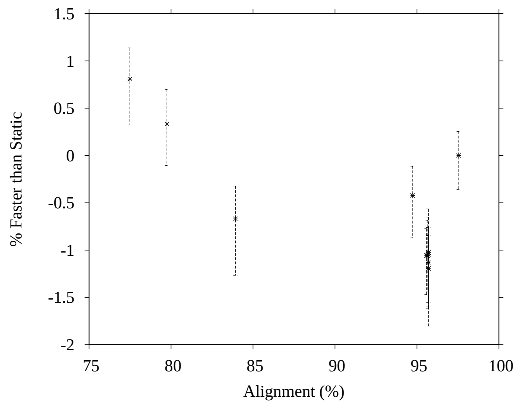


(a) Itanium

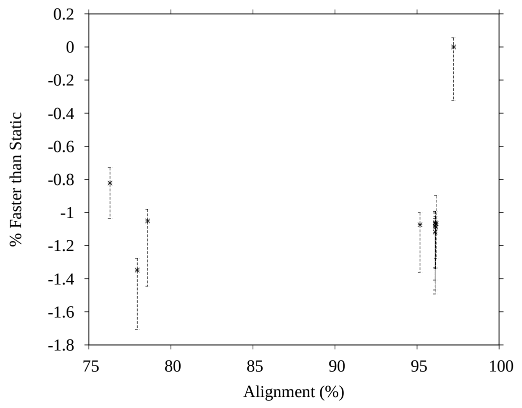


(b) Itanium 2

Figure B.2: Alignment vs. performance for `if` conversion on `crafty`

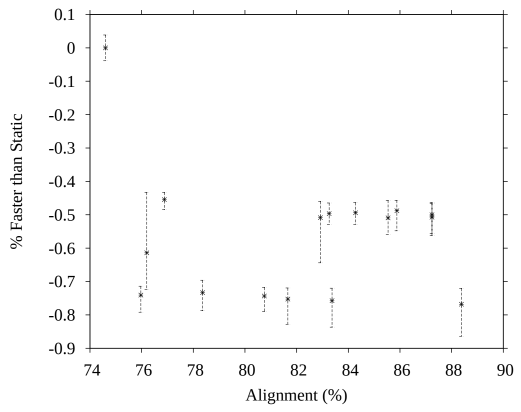


(a) Itanium

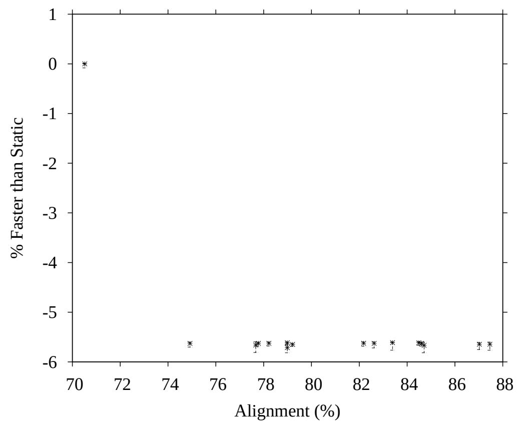


(b) Itanium 2

Figure B.3: Alignment vs. performance for `if` conversion on `GAP`

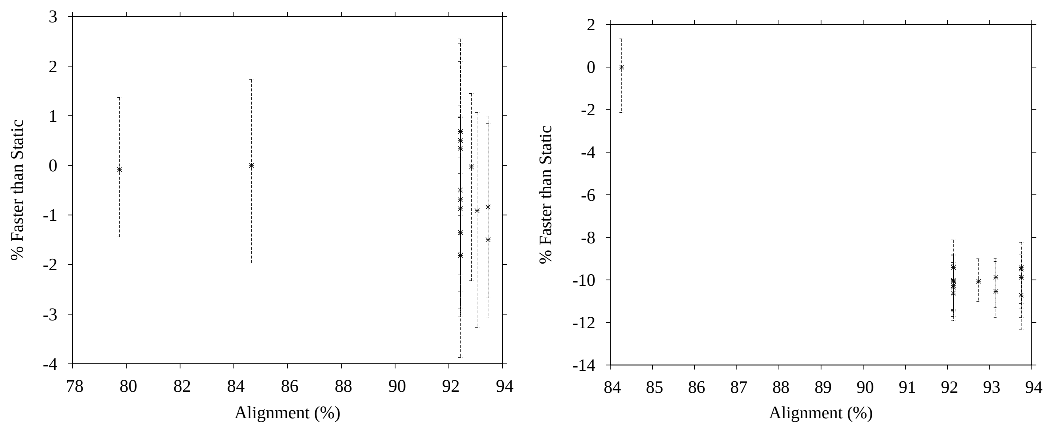


(a) Itanium



(b) Itanium 2

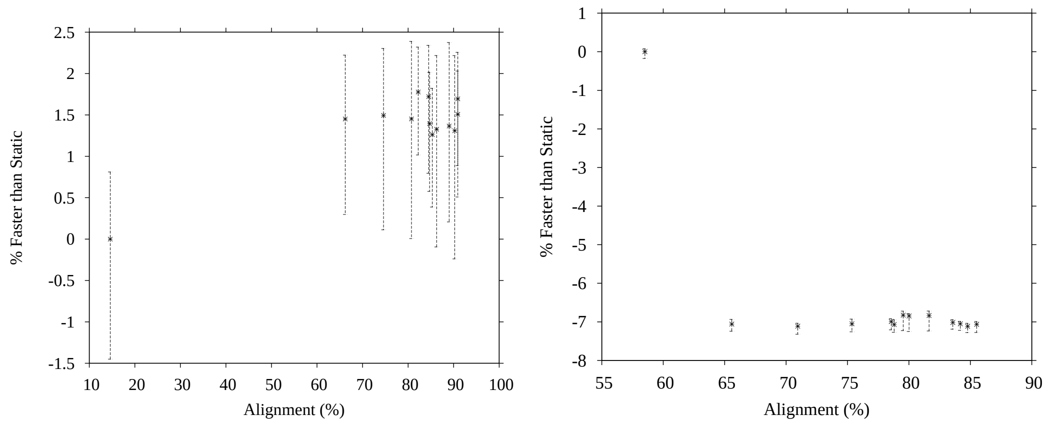
Figure B.4: Alignment vs. performance for `if` conversion on `gzip`



(a) Itanium

(b) Itanium 2

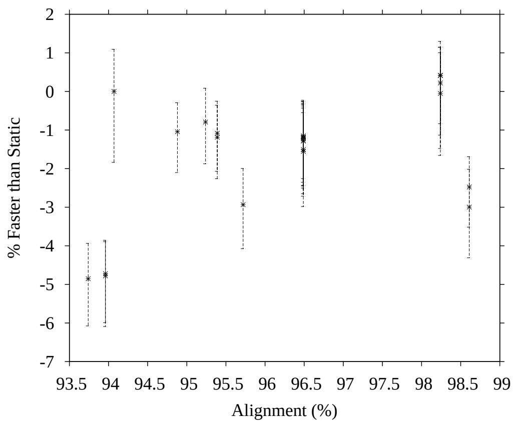
Figure B.5: Alignment vs. performance for `if` conversion on `MCF`



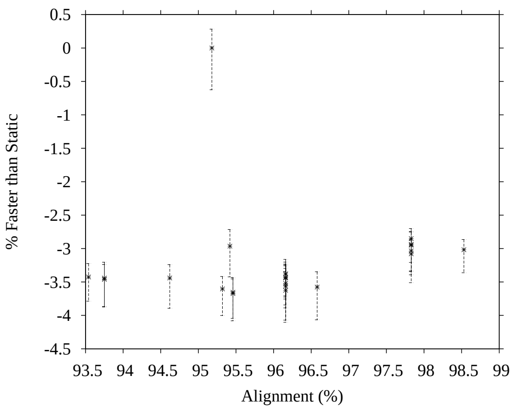
(a) Itanium

(b) Itanium 2

Figure B.6: Alignment vs. performance for `if` conversion on `parser`

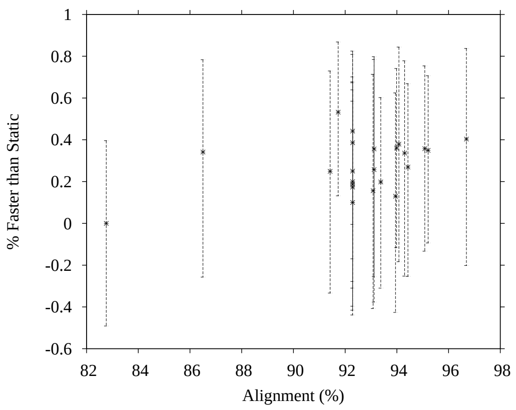


(a) Itanium

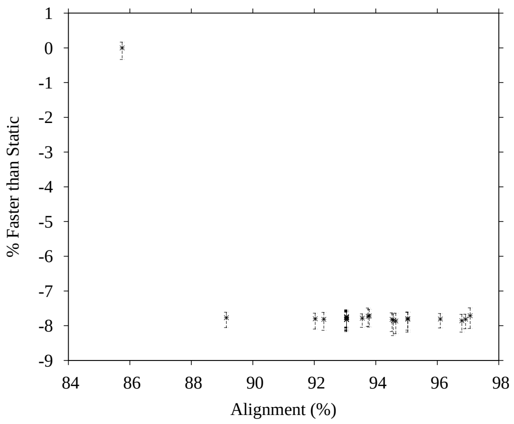


(b) Itanium 2

Figure B.7: Alignment vs. performance for `if` conversion on VPR (place)

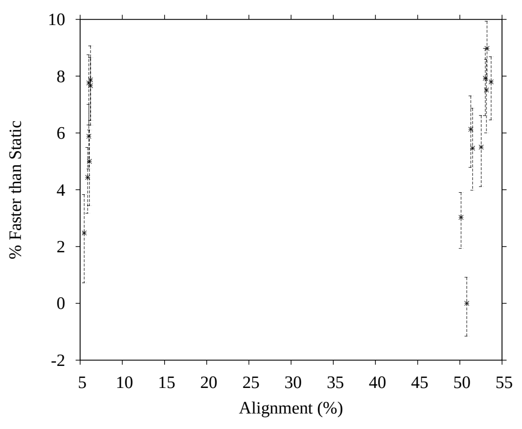


(a) Itanium

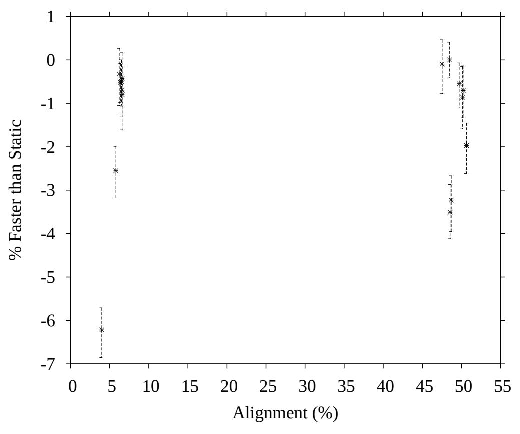


(b) Itanium 2

Figure B.8: Alignment vs. performance for `if` conversion on VPR (route)

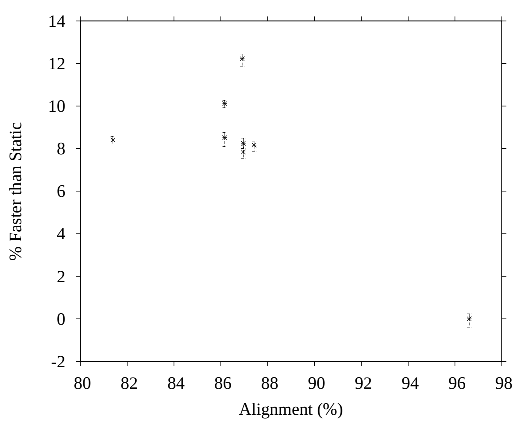


(a) Itanium

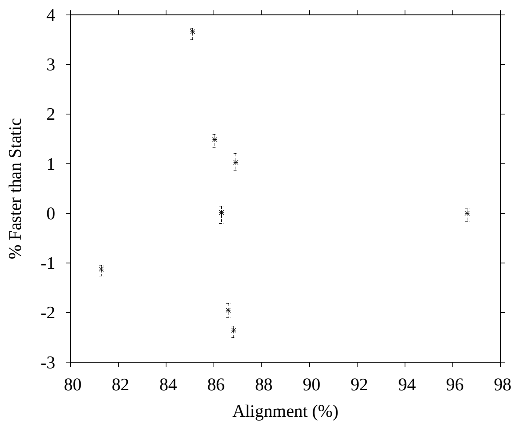


(b) Itanium 2

Figure B.9: Alignment vs. performance for inlining on `bzip2`

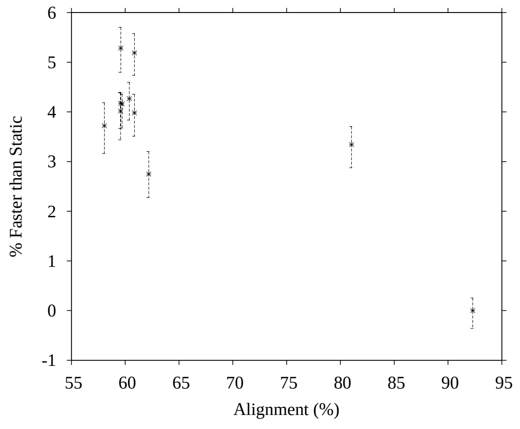


(a) Itanium

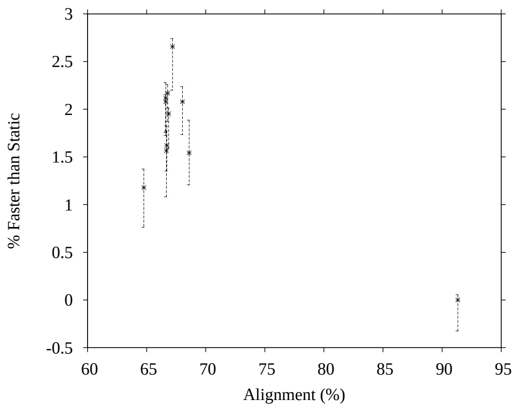


(b) Itanium 2

Figure B.10: Alignment vs. performance for inlining on `crafty`

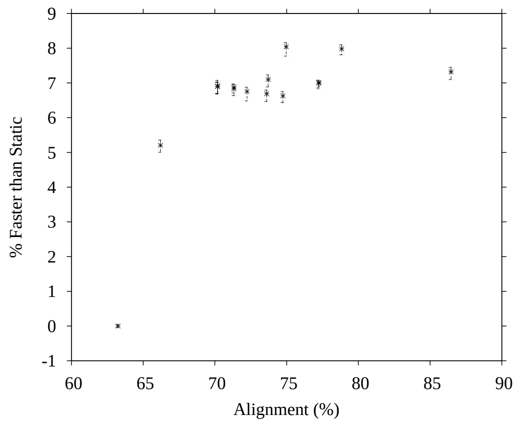


(a) Itanium

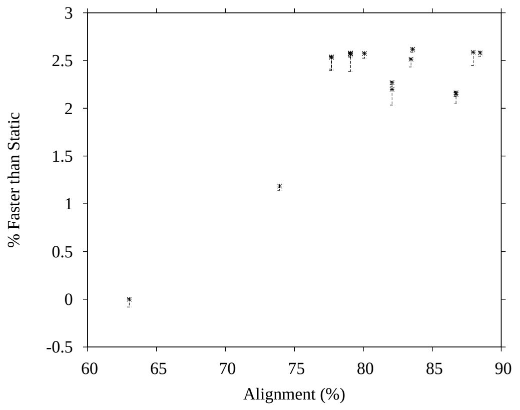


(b) Itanium 2

Figure B.11: Alignment vs. performance for inlining on `GAP`



(a) Itanium



(b) Itanium 2

Figure B.12: Alignment vs. performance for inlining on `gzip`

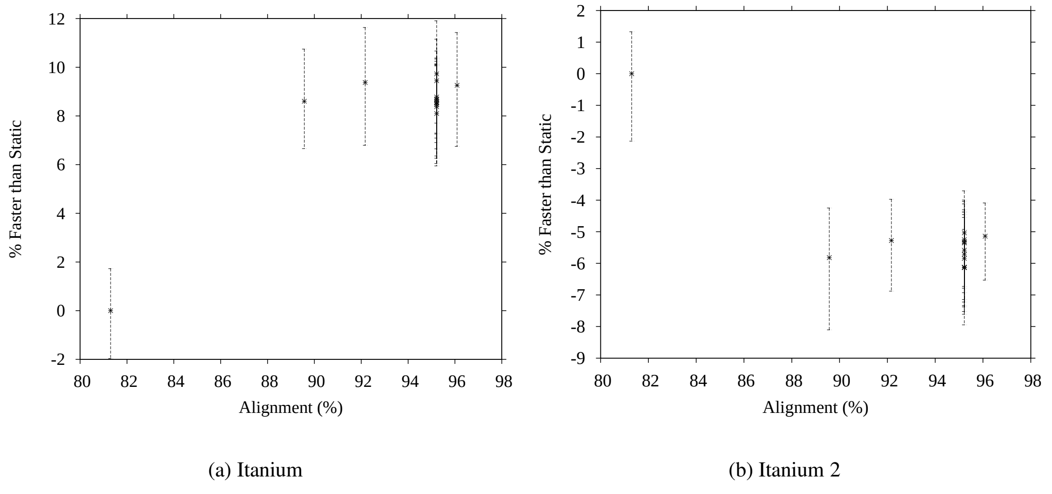


Figure B.13: Alignment vs. performance for inlining on `MCF`

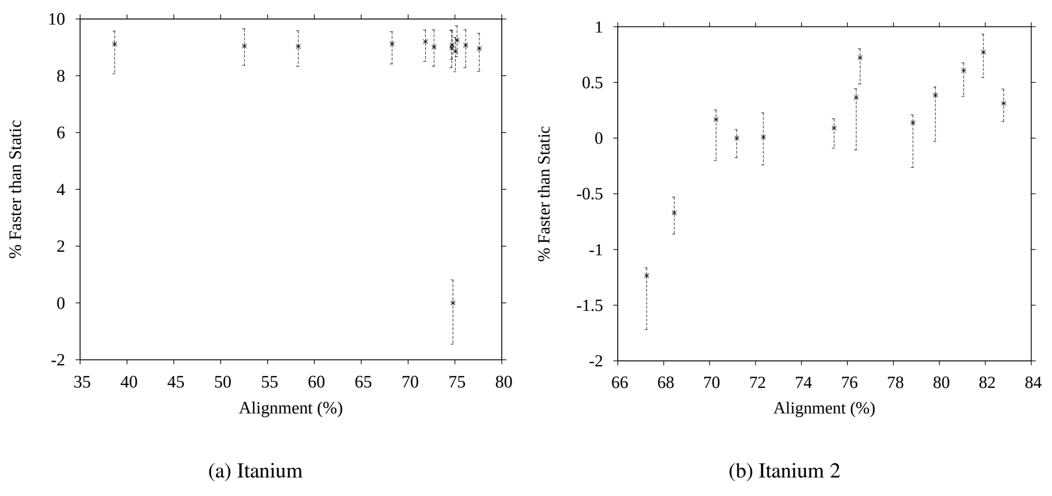
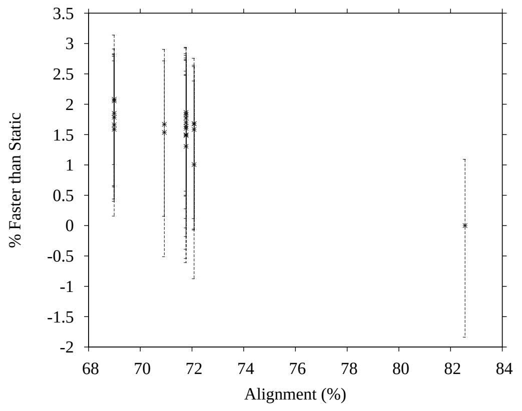
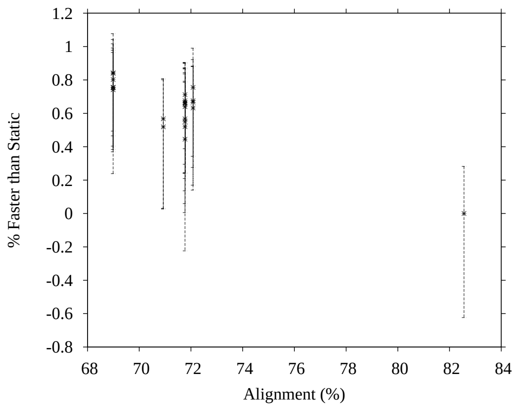


Figure B.14: Alignment vs. performance for inlining on `parser`

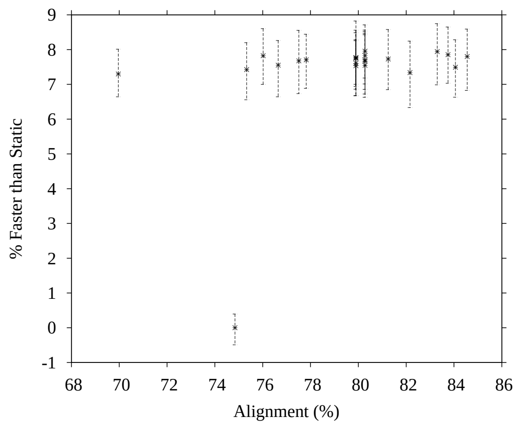


(a) Itanium

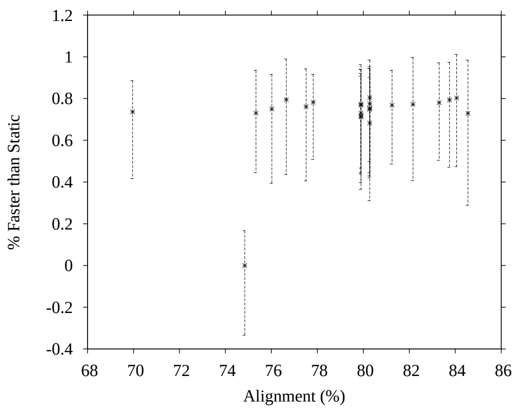


(b) Itanium 2

Figure B.15: Alignment vs. performance for inlining on VPR (place)



(a) Itanium



(b) Itanium 2

Figure B.16: Alignment vs. performance for inlining on VPR (route)