Aestimo: A Feedback-Directed Optimization Evaluation Tool

Paul Berube, José Nelson Amaral*
Dept. of Computing Science, University of Alberta
Edmonton, Alberta, T6G 2E8, Canada

Abstract

Published studies that use feedback-directed optimization (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. Thus an important question is if the FDO results published in the literature are sensitive to the training and testing input selection.

Aestimo is a new evaluation tool that uses a workload of inputs to evaluate the sensitivity of specific code transformations to the choice of inputs in the training and testing phases. Aestimo uses optimization logs to isolate the effects of individual code transformations. It incorporates metrics to determine the effect of training input selection on individual compiler decisions.

Besides describing the structure of Aestimo, this paper presents a case study that uses SPEC CINT2000 benchmark programs with the Open Research Compiler (ORC) to investigate the effect of training/testing input selection on inlining and if-conversion. The experimental results indicate that: (1) training input selection affects the compiler decisions made for these code transformation; (2) the choice of training/testing inputs can have a significant impact on measured performance.

1. Introduction

Feedback-directed optimization (FDO), also known as profile-guided optimization, may enhance the optimization decisions in a compiler [6]. FDO can be viewed as a spectrum of performance-enhancing techniques that rely on measurements of run-time program behavior [22]. In this paper a more traditional definition of FDO is considered. When FDO is used, the program is first compiled with additional instrumentation code to record statistics about runtime program behavior into a *profile* or *feedback* file. Then, this *instrumented binary* is run on a *training input* to generate a profile. Finally, the program is recompiled, and the compiler reads the profile and replaces its static estimates of program behavior with the values recorded in the profile.

The training input used with FDO is an important component of the FDO process. The success of FDO depends on the selection of training inputs that are representative of the majority of common uses of an application. It is therefore important to determine the significance of training-input selection on code transformations that use profile information, both in terms of the decisions made at compile time, and in terms of the performance of the resulting binaries. *Aestimo* is a tool developed to facilitate these investigations.

Studies that use FDO techniques may use either a single input for both training and performance evaluation, or a single input for training and a single input for evaluation [5, 23, 17, 14, 7, 20, 9, 25]. Few studies have investigated the impact of the training input used in FDO on the performance of the resulting binary, or methods to effectively select training inputs.

An important question remains open: How important is the selection of training data for FDO? The answer to this question is not constant across all transformations 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 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. The reality is that optimizations have varied measures of input selection sensitivity.

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

*This research is supported by fellowships and grants from the Natural Sciences and Engineering Research Council of Canada (NSERC), the Informatics Circle of Research Excellence (iCORE), and the Canadian Foundation for innovation (CFI).

This paper presents *Aestimo*, a new tool to investigate FDO systems, and reports on an initial exploratory investigation that provides the following contributions:

- Introduces an experimental methodology to investigate the impact of input selection on individual code transformations, both in terms of compiler decisions and program performance.
- Uses a large selection of varied training input for the SPEC CINT2000 benchmark programs to demonstrate that training input selection does impact code transformation decisions and the resulting program performance. Additionally, the study shows that the selection of evaluation inputs can significantly alter the results of performance evaluation.

Material in this paper has been previously presented in an extended form in a thesis with the same title [2]. Section 2 provides an overview of the experimental methodology used in this study, and details the operation and functions of *Aestimo*. Section 3 follows with a summary of the results from an extensive experimental study using *Aestimo*, and presents a case study of inlining for `bzip2` to demonstrate the information provided by *Aestimo*. Section 4 presents some related work, and Section 5 concludes.

2. Experimental Methodology

In order to investigate the sensitivity of individual feedback-directed code optimizations, we created *Aestimo*[1]. *Aestimo* is a performance evaluation tool that automates the process of compiling, executing, and evaluating programs on workloads composed of several program inputs. Figure 1 provides an overview of *Aestimo*.

2.1. Compilation Process

The experiments performed by *Aestimo* required the creation of a large number of binaries. *Aestimo* distinguishes a *program*, which is the algorithm encoded in the source code, from a *binary*, which is one compiled instance of the program. In particular, changing the training input used with FDO results in a different binary. A flow diagram for *Aestimo*'s compilation process is presented in Figure 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

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

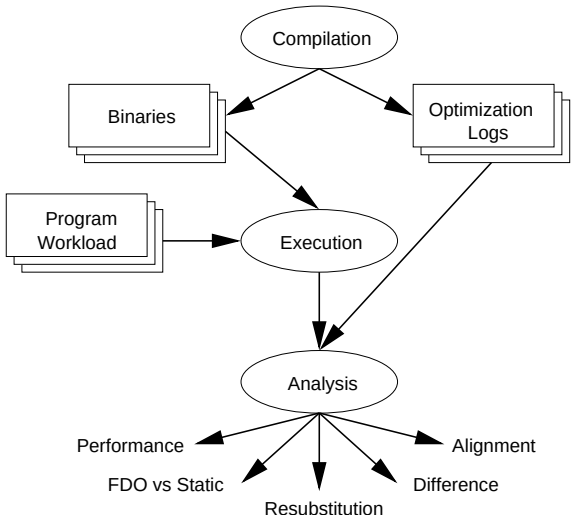

Figure 1. Overview of *Aestimo*

of flags that refer to the profile file. In this study, the flags used are -O3 and -ipa for inlining, along with any flags for defines required by the particular SPEC benchmark. Only one instrumented binary is created for each program. However, the remaining steps in the flow diagram are performed for each optimization/input pair for each program.

When *Aestimo* is investigating an optimization P, it produces binaries that only use profile-guided decisions for P. For each benchmark B, a training run executes the instrumented binary on a training input. Then, B is compiled using the generated profile data, and an optimization log L is emitted for P. The binary produced at this point is discarded. *Aestimo* recompiles B statically using L to instruct the compiler to make the same decisions for P 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 P. 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 L. For example, if P is `if` conversion, there may be a function that is not inlined without profile guidance. In that case, any `if` conversion listed in L for the inlined code in 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 L forces a callsite to be inlined, any static optimizations applicable to the inlined code will still be applied. Thus, *Aestimo* ensures that any opportunity to apply P will result in the same decision as in the full feedback-directed case, while not ignoring cascading effects due to the interrelatedness of optimizations. Nonetheless, the interactions between optimization are complex and generally unpredicatble. Therefore, the impact of training data selection on program performance discovered by this technique is only an estimate. Similarly, the combined im-

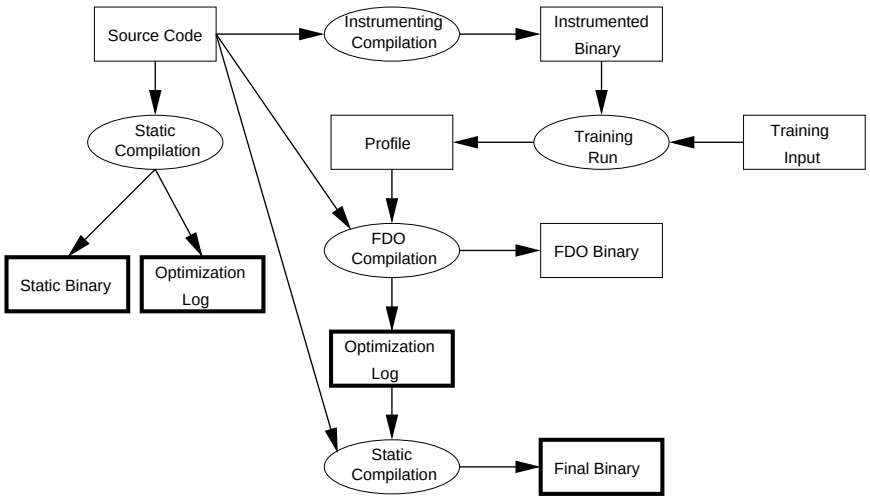

Figure 2. Compilation process

pact of several optimizations is unlikely to be equal to the accumulated impact of the optimizations measured individually.

2.2. Performance Evaluation

After the compilation process, *Aestimo* executes each of the FDO binaries on each of the inputs in the program workload five times. *Aestimo* then analyzes the program run times and the optimization logs — calculating difference and alignment metrics — and reports the results.

Aestimo uses two methodologies to report results: an arithmetic mean and a geometric sum of run times. The arithmetic mean aggregates the raw run times for each of the inputs in the workload for a given binary and reports this sum as a percent faster than the same measure for the statically optimized binary. The geometric sum is similar, but, for each input, it normalizes the run times against the 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 j.

Aestimo also compares the performance of the statically optimized binary with the performance of the fastest FDO binary for each input in the program workload. This measure represents the best case performance of FDO recorded by *Aestimo*, and as such provide an upper bound on FDO performance for the inputs in W.

Resubstitution is the practice of using the same input for both the training and evaluation runs [18]. Ideally, a compiler that makes good use of profile information will produce the fastest binary for a given input when resubstitution is used. *Aestimo* calculates the rank of each FDO binary on each input. A rank of 1 indicates that a binary is the fastest on a particular input. Thus, if more accurate profile information is used effectively during FDO, resubstitution should produce binaries with low ranks.

2.3. Workload Selection

This study uses benchmarks, and datasets, from the SPEC CINT2000 suite[2] [10]. SPEC provides three sets of inputs for each benchmark: test is a very small input that allow easy verification; train is a set of small or medium-sized inputs for training with FDO; ref (reference) is the input set used for performance evaluation. This study uses all the SPEC inputs plus additional inputs chosen to be representative of the benchmark's typical workload.

Benchmark authors have been consulted, for their expert knowledge of the program, to inform the selection of inputs. For `GAP` and `crafty`, the benchmark authors provided additional inputs for use in this study. Inputs for `bzip2` and `gzip` are selected as a collection of files in common formats. Inputs for `parser` are taken from the Project Gutenberg ebooks collection [4, 13, 24], web-pages [16], and the Reuters-21578 text categorization test collection [19]. A synthetic generator is used to create problem instances for `MCF` with parameters similar to the SPEC ref input. The placement and routing tasks of `VPR` are considered individually, and use the FPGA Place-and-Route Challenge [3] problems in the program workload. In total, 116 inputs are

[2]The following benchmarks are omitted because they could not be compiled with the appropriate flags in the ORC: `perlbmk`, `vortex`, `twolf`, `GCC` and `eon`

```
void foo() {}

void bar() {
  foo();
}

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

Figure 3. Callsites in a simple program

callsite	log 1	log 2	log 3	log 4
bar.foo	yes	no	yes	no
main.foo	no	no	no	yes
main.bar	no	yes	yes	yes
main.bar.foo		yes	yes	yes

Figure 4. Some possible inlining logs

used, of which only 32 are provided by SPEC[3].

2.4. Metrics

Does profiling on different training inputs result in different optimization decisions in the compiler? To address this question, we propose 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. A particular instance where a decision is made is a *choice*. 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. For instance, Figure 3 shows the callsites of a simple program. Three possible inlining logs are presented in Figure 4. The notation `caller.callee` is used to name callsites.

Log files record the compiler's choices and decisions for an optimization during a single compilation. All the logs for a given program 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 positive non-zero value is recorded for an affirmative decision (choosing to perform the optimization), while a 0 is recorded in the vector for a non-affirmative decision (choosing not to perform an optimization). In the case where a choice is not present in one or more logs, a default value of 0 is recorded. This

[3]The additional inputs used in this study can be found at http://www.cs.ualberta.ca/∼berube/compiler/fdo/

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 5. Log files converted to vectors

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

Table 1. Values for the difference metric

situation may arise when the existence of one decision depends on a previous affirmative decision. For example, the *main.bar.foo* callsite does not exist in log 1 in Figure 4, so it is assigned the default value of 0 in the vectors in Figure 5.

The difference metric is defined as the squared length of the difference vector between two log vectors $\vec{v}_i$ and $\vec{v}_j$: $\delta(\vec{v}_i, \vec{v}_j) = |\vec{v}_i - \vec{v}_j|^2$. Where decisions are recorded in the vectors as 0s and 1s, $\delta(\vec{v}_i, \vec{v}_j)$ is simply the Hamming distance between the vectors[4]. δ grows with the number of choices that result in different decisions in the two logs. Thus, δ indicates when a different selection of training input results in different optimization decisions during compilation. Difference values for the example are given in Table 1.

FDO is based on the premise that a representative input used for profiling reflects the runtime behavior of other common inputs. Thus, optimization logs based on profiles from different representative training inputs should not vary significantly. The difference metric does not indicate how much 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.

Aestimo first calculates the *combined total* vector for a set of logs: $\vec{T} = \sum_i \vec{v}_i$. $\vec{T}$ is a measure the of agreement between all the logs. A choice that frequently results in an affirmative decision will have a high value recorded at its index in $\vec{T}$, while a decision that is usually decided non-affirmatively 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

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

accumulation of the element-wise products of $\vec{T}$ and $\vec{v}_i$, α is large only if $\vec{v}_i$ has positive values (*i.e.*, affirmative decisions) at the same indexes as many other logs. If a log has no affirmative decisions, α will be 0. On the other hand, if a log has an affirmative decision for every choice for which any log records a affirmative decisions, α will be 100%. A high alignment score does not necessarily indicate more effecive optimization. Some decisions may be harmful, while many logs may miss an important optimization. A low alignment may indicate that a log contains better decisions that do not agree with most logs. In the example, $\alpha_1 = \frac{2}{9} = 22\%$, $\alpha_2 = \frac{6}{9} = 67\%$, $\alpha_3 = \frac{8}{9} = 89\%$, and $\alpha_4 = \frac{7}{9} = 78\%$.

It is important to note that the difference and alignment metrics do not consider the performance of the binaries corresponding to the optimization logs. Consequently, these metric scores do not reflect the quality of a training input. Furthermore, the metrics only measure the similarity between optimization logs. If the decisions recorded in most logs are poor, then a log with high difference scores and a low alignment may in fact record many different decision that lead to improved performance. A high alignment score may indicate that a log contains a "representative" set of decisions, but this does not suggest that these decisions correspond to a faster program.

2.5. Compiler Infrastructure

This study uses version 2.1 of the Open Research Compiler (ORC), an open-source compiler based on the code base of SGI's Pro64 compiler [1]. The ORC focuses on producing high-performance code, and is frequently used for compiler research. ORC has a rich profiler to support its FDO infrastructure. The IPF processor family is the only target for the ORC. ORC combines a mature code base with state-of-the-art compiler technology.

3. Evaluating FDO

This section summarizes the results of a case study that uses *Aestimo* to study the `if` conversion and inlining transformations in the ORC compiler, targeting the Itanium and Itanium 2 processors [2].

Figure 3 presents the arithmetic-mean performance of the Itanium FDO binaries for each benchmark program. The results are mixed, but on average FDO `if` conversion has little effect on performance.

On the other hand, the experimental results show that there are performance benefits from feedback-directed inlining. Furthermore, there are several cases where the selection of training input has a significant impact on performance. Figure 3 shows the arithmetic-mean performance of FDO inlining on each program. FDO inlining improves

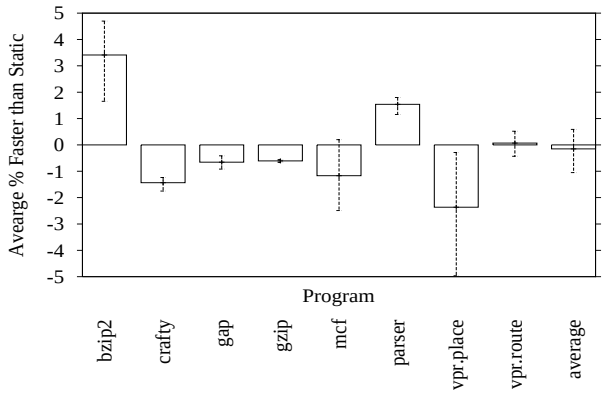

Figure 6. Average FDO `if` conversion performance on Itanium

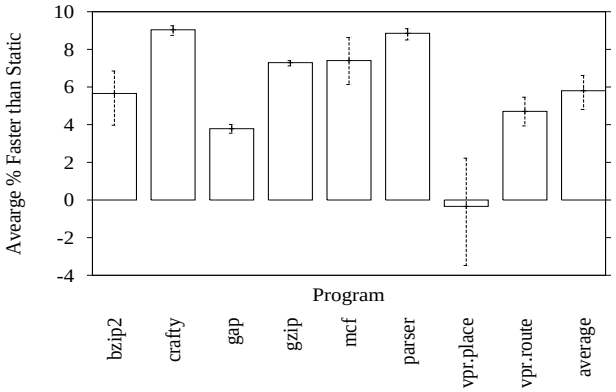

Figure 7. Average FDO inlining performance on Itanium

performance by 6% on average on the Itanium. The best-case FDO inlining performance is slower than static in only 6 out of 116 cases. Furthermore, the fastest FDO inlining binaries are more than 10% faster than static in 41 cases.

The results for the Itanium 2 are disappointing: `if` conversion almost always result in performance degradation and inlining produces mixed results. Extensive results for the experiments with Itanium 2 are presented in [2].

3.1. Case Study: Inlining for `bzip2`

Tables 2 through 4 present difference and alignment scores for the FDO inlining logs of `bzip2`. 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 for the FDO log listed in the first column paired with all the other FDO logs. The Max

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 2. 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 3. Inlining metric scores for `bzip2` low cut group on the Itanium

Input	Mean	Std Dev	Max	Static	Alignment
combined	5.57	3.48	10	69	91.74
compressed	6.14	3.12	9	74	88.78
gap	5.00	3.08	8	71	91.03
graphic	5.00	2.74	7	75	90.55
log	7.57	3.34	10	77	86.42
program	5.86	3.49	9	73	91.38
random	6.14	2.79	7	76	88.43
source	4.14	2.38	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. Inlining metric scores for `bzip2` high cut group on the Itanium

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. Additional relevant information is recorded in the last four rows of each table. 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, including the static 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.

The consensus values for `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.

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 3 and 4 show the results of cutting the logs into two groups.

The inputs in the high cut group (which originally had alignment scores greater than 45%) are quite similar. Inputs in this group have low 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, the inputs in the low cut group are significantly different from each other. 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 group, and there is 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 only contain between 82 and 93 affirmative 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 inputs 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.

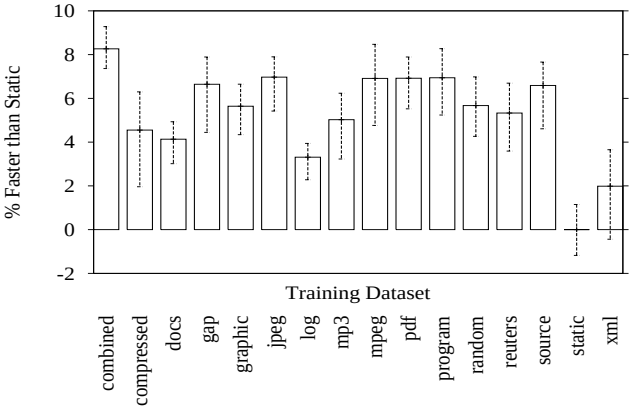

Figure 8. FDO Inlining performance: `bzip2` on Itanium

Figure 8 shows the performance impact, using the arithmetic mean, of different training inputs on `bzip2`. Each input in the workload is used as a training input for one binary. The training input used is listed below each bar in the graph. The bars represent the average run times of 5 trials on the entire workload, while the error bars correspond to the minimum and maximum times from those 5 trials.

Despite the large ranges of run times between trials, Figure 8 shows that, for the Itanium, training on the combined input results in performance gains of about 8%, while training on xml improves performance by only 2%.

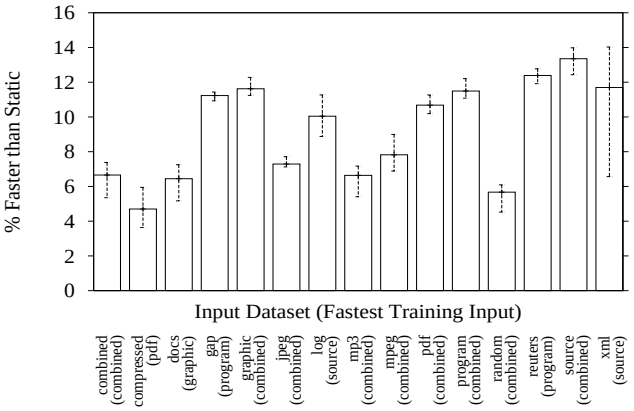

Figure 9. Static vs. FDO Inlining performance: `bzip2` on Itanium

Comparing best-case FDO performance to static optimization is an optimistic measure that can identify poten-

tial for FDO to improve performance. Figure 9 presents best-case FDO inlining for `bzip2`. Below the graph, in parenthesis beside the names of the evaluation inputs, we record the training input used to create the fastest FDO binary for each evaluation input. Performance on the Itanium is very good, with a minimum improvement of about 4% and a maximum gain of about 13%. These results highlight the potential for FDO to improve performance.

The binary trained on combined is often the fastest binary in Figure 9. However, the performance of this binary is not consistent across the inputs where it achieves best performance. This result indicates that the common practice of using a single input for the performance evaluation of code transformations is liable to produce unreliable results.

In an effective FDO system, more accurate feedback information should result in a faster-running binary. The most accurate information can be obtained by resubstitution, that is, using the same input for both training and evaluation. Therefore, the rank calculated by *Aestimo* for resubstitution binaries from an effective FDO system should be low.

Unfortunately, the ORC does not appear to use feedback information effectively for the programs and inputs used in this study. Table 5 lists each input in the `bzip2` workload. For each input, and for each processor, the rank of the inlining 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 5 show that among the FDO binaries for `bzip2` on the Itanium 2, the binary trained on combined is the 14^{th} fastest (of 15) when evaluated using the combined input. Furthermore, the binary trained on combined was 6.44% slower than the fastest FDO binary.

A lower rank is usually associated with a smaller performance difference compared to the rank-1 binary for `bzip2`. 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 scarcity of such highly-ranked resubstitution binaries suggests that the FDO system seldom makes effective use of more accurate feedback information for either processor. In fact, the ranks of resubstitution binaries are fairly evenly distributed across the range of possible ranks. This result suggests that there is no relationship between the quality of feedback information and the performance of the resulting binary.

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 5. Rank of resubstitution binaries for inlining on `bzip2`

4. Related Work

4.1. Input Selection and Benchmarking

Eeckhout *et al.* attempt to find a minimal set of representative programs and inputs for architecture research [12]. They cluster program-input combinations using principal-component analysis (PCA) of low-level program behavior such as cache misses and branch mispredictions. They find that while different inputs to the same program were often clustered together, in several cases different inputs to the same program result 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 [21] using PCA on low-level, architecture-independent program behaviors such as instruction mix, basic-block size, branch statistics, and locality. Their clustering suggests that several benchmarks in the SPEC suites are redundant. Based on their overall characteristics, `bzip2` and `gzip` form the entirety of one cluster. However, in our study, *Aestimo* finds significantly different results for `bzip2` and `gzip`. Therefore, while clustering based on low-level program behaviors may identify redundancy for architectural studies, we caution compiler designers against omitting programs from a benchmark suite based on this technique.

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 [15]. Eeckhout *et al.* analyze program behavior on the reduced inputs suggested by MinneSPEC [11]. 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 shows 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. The MinneSPEC inputs, derived from SPEC inputs with the intent of maintaining program behavior, have limitted success at achieving this goal. Therefore, it is not surprising that *Aestimo* finds that alternate training inputs, which are intended to be different from the SPEC inputs, also result in different program behavior, and consequently different compile-time decisions and different levels of performance in the resulting FDO binaries.

Citron investigatea the use of the SPEC benchmarks by research reported in computer architecture conferences [8] and finds that while commonly used, the suite is seldom used as intended. The use of only selected programs from the benchmark suite is common, and can dramatically inflate reported results. 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 a code transformation. The common practice of using only the inputs supplied with the SPEC benchmarks is likely to further obscure the true impact of a technique when used outside the lab.

4.2. Feedback-Directed Optimization

Cohn and Lowney investigate FDO in Compaq's compiler tools for the Alpha processor using the SPEC CINT95 benchmarks [9]. 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. *Aestimo* finds much smaller gains for FDO inlining from the ORC. Furthermore, unlike the Compaq compiler, FDO inlining with the ORC degraded performance in some cases. However, the differences in compiler, architecture, and benchmark programs makes meaningful comparisons between the performance results difficult.

Langdale also investigates the sensitivity of FDO to the training data used [18]. 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.

4.3 Iterative Optimization

Iterative compilation can be used to giude a search through the space of possible program transformations. A metric computed on the binary produced at one iteration guides the compilation of subsequent iterations.

Pan and Eigenmann break a program into regions, called Tuning Sections (TS), and attempt to find an optimal optimization strategy for each TS [20]. Their GCC-based system is able to improve performance on four SPEC 2000 benchmarks by an average of 26%. Tuning is giuded by the performance of binaries run on the SPEC `train` inputs, while evaluation uses the SPEC `ref` inputs. In contrast to our results, 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 their 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 non-iterative FDO may provide a more consistent benefit when applied across a larger collection of programs and inputs.

5. Conclusion

Aestimo is a tool to investigate and evaluate individual optimizations in FDO systems. *Aestimo* introduces a methodology to investigate compiler decisions for individual transformations and measures their performance consequences. Furthermore, the difference and alignment metrics quantitatively measure the differences in compile-time decisions made based on different training inputs. Additionally, we select a large number of additional inputs for SPEC CINT2000 benchmark programs to create representative workloads with a substantially larger degree of variation than the small evaluation workloads provided by SPEC.

The results of an extensive experimental study using *Aestimo* are illustrated by a case study of inlining for the `bzip2` benchmark program. Selecting different training inputs results in substantially different inlining decisions for

many of the inputs in the workload. Furthermore, there are significant performance variations on the workload depending which training input is used. Using the fastest FDO binaries for each input reveals the potential for FDO inlining to substantially improve performance on the Itanium.

Acknowledgments

We are very grateful to Bradley Calder and David August for their many detailed comments that improve the writing in this paper.

References

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

[2] P. Berube. *Aestimo*: A feedback-directed optimization evaluation tool. Master's thesis, University of Alberta, October 2005.

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

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

[5] J. 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.

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

[7] K. Chow and Y. Wu. Feedback-directed selection and characterization of compiler optimizations. In *MICRO 32*, Isreal, Nov 1999.

[8] D. 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.

[9] R. Cohn and P. G. Lowney. Feedback directed optimization in Compaq's compilation tools for Alpha. In *2nd ACM Workshop on Feedback-Directed Optimization*, Haifa, Israel, November 1999.

[10] S. P. E. Corporation. SPEC: The standard performance evaluation corporation. http://www.spec.org/.

[11] L. Eeckhout, H. Vandierendonck, and K. D. Bosschere. Designing computer architecture research workloads. In *IEEE Computer*, volume 36, pages 65–71, February 2003.

[12] L. Eeckhout, H. Vandierendonck, and K. D. Bosschere. Quantifying the impact of input data sets on program behavior and its applications. *Journal of Instruction-Level Parallelism*, 5:1–33, 2 2003.

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

[14] T. Kisuki, P. M. W. Knijnenburg, and M. F. P. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *IEEE PACT*, pages 237–248, 2000.

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

[16] M. Krahulk and J. J. Holkins. http://www.penny-arcade.com/. A popular video-game news and webcomic website. Text from news posts from December 29, 2004 through May 6, 2005.

[17] P. Kulkarni, S. Hines, J. Hiser, D. Whalley, J. Davidson, and D. Jones. Fast searches for effective optimization phase sequences. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 165–198, 2004.

[18] G. Langdale. *The Effect of Profi le Choice and Profi le Gathering Methods on Profi le-Driven Optimization Systems*. PhD thesis, Carnegie-Mellon University, 2004.

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

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

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

[22] M. 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.

[23] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O-Reilly. Meta optimization: Improving compiler heuristics with machine learning. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 77–90, 2003.

[24] H. G. Wells. *The War of the Worlds*. Project Gutenberg, October 2004. http://www.gutenberg.org/etext/36.

[25] P. Zhao and J. N. Amaral. To inline or not to inline? Enhanced inlining decisions. In *Languages and Compilers for Parallel Computing: 16th International Workshop*, October 2003.