2011

Plackett and Burman analysis to select effective compiler optimizations

Dustin Larson
Michigan Technological University

A PLACKETT AND BURMAN ANALYSIS TO SELECT EFFECTIVE COMPILER OPTIMIZATIONS

By

Dustin Larson

A REPORT

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

(Computer Science)

MICHIGAN TECHNOLOGICAL UNIVERSITY

2011

This report, "A Plackett and Burman Analysis to Select Effective Compiler Optimizations," is hereby approved in partial fulfillment of the requirements for the Degree of MASTER OF SCIENCE IN COMPUTER SCIENCE.

Computer Science

Signatures:

Report Advisor _____
Dr. Steven Carr

Report Co-Advisor _____
Dr. Zhenlin Wang

Committee Member _____
Dr. Martin Thompson

Department Chair _____
Dr. Steven Carr

Date _____

1 Abstract

Compiler optimizations help to make code run faster at runtime. When the compilation is done before the program is run, compilation time is less of an issue, but how do on-the-fly compilation and optimization impact the overall runtime? If the compiler must compete with the running application for resources, the running application will take more time to complete.

This paper investigates the impact of specific compiler optimizations on the overall runtime of an application. A foldover Plackett and Burman design is used to choose compiler optimizations that appear to contribute to shorter overall runtimes. These selected optimizations are compared with the default optimization levels in the Jikes RVM. This method selects optimizations that result in a shorter overall runtime than the default O0, O1, and O2 levels. This shows that careful selection of compiler optimizations can have a significant, positive impact on overall runtime.

2 Introduction

How is overall runtime affected by virtual machines that compile code on-the-fly? When code is compiled ahead of time, performing many powerful optimizations on the code is a logical choice to decrease the running time of the program later on. However, when code is compiled on-the-fly, this code compilation time adds to the overall runtime of the application. Thus, performing many optimizations (or perhaps a few expensive ones) may actually cause the application to run for a longer period of time. If instead, a few very good optimizations (or maybe even none at all) are chosen, the overall runtime can decrease and is a better option. The question arises: which optimizations should be selected to improve overall runtime?

To answer this question, we can ask several more. Which optimizations generally contribute to a longer overall runtime? Which ones shorten the runtime? Do particular optimizations significantly increase compilation time while having a relatively small effect on the execution time (resulting in an increase in overall runtime)? A method for constructing a select group of optimizations that gives better overall runtime performance would be helpful. This method should eliminate from consideration any optimizations that increase compile time enough to actually increase overall runtime of the application.

In this report, section 3 contains background information on the foldover Plackett and Burman Design, the Jikes RVM system, and the DaCapo Benchmark suite.

Section 4 describes the Plackett and Burman setup and gives short explanations for each tested compiler optimization. Section 5 gives the results and compares the selected optimizations to the default O0, O1, and O2 levels in the Jikes RVM. Section 6 provides a brief analysis and a breakdown of compile time phases. Section 7 concludes the paper.

3 Background

3.1 Plackett and Burman

The immediate issue that arises in any problem that measures some effect is the number of experiments that must be run in order to determine the contribution of each parameter toward the overall effect. In our case, 23 different optimizations are tested. Testing every combination of optimizations is infeasible, as 2^n experiments are needed to determine the effects of n parameters, as well as all interactions between parameters.

The Plackett and Burman (PB) Design [1] is a method of finding which parameters have the most significant effect in a system without running 2^n experiments. In fact, only about n experiments are needed. Yi et. al. use a variation on the PB Design, described shortly, to estimate the effects of different processor components and their values, such as cache size or ALU latency [2]. We apply the same idea to compiler optimizations in this experiment. The PB Design assigns each parameter of interest a "high" value or a "low" value (represented by +1 or -1), then tests how the parameter assignments change the overall effect. These high and low values should be "just outside of the 'normal' range of values." [2] Since the compiler optimizations tested are selected as either on or off, we consider on to be "high" and off to be "low".

The design of the PB experiment must be set up specifically according to [1]. This special setup will make each column vector orthogonal to every other column vector in the matrix, which is necessary for calculating the effect of each optimization. Each experiment is a vector (the set of compiler optimizations) where each vector component is a parameter (a specific optimization). These experiment vectors are the rows in the PB matrix. If there are n parameters to vary, then it is necessary to run X tests, where X is the next multiple of four strictly greater than n. By following the setup in [1], we set each parameter to its high value for half of the experiments and to its low value for the other half.

An improvement upon the PB Design, the "foldover" PB design [3], requires $2X$ tests, where X is defined as above. The matrix is set up as in the PB design for the first X tests. Table 1 shows the first $X = 24$ experiments. The first row is given directly in [1]. The following 22 rows shift the values from the previous row one position to the right. The 24th row shown is a row of all low (-1) values, as specified in [1]. The second $X = 24$ tests simply reverse the high and low values from X tests given in the table.

field_analysis	inline	inline_guarded	inline_guarded_interfaces	inline_preex	simplify_float_ops	simplify_tib_ops	simplify_field_ops	local_constant_prop	local_copy_prop	local_cse	control_static_splitting	escape_scalar_replace_aggregates	escape_monitor_removal	reorder_code	reorder_code_ph	h2l_inline_write_barrier	h2l_inline_primitive_write_barrier	l2m_handler_liveness	regalloc_coalesce_moves	regalloc_coalesce_spills	osr_guarded_inlining	osr_inline_policy
1	1	1	1	1	-1	1	-1	1	1	-1	-1	1	1	-1	-1	1	-1	1	-1	-1	-1	-1
-1	1	1	1	1	1	-1	1	-1	1	1	-1	-1	1	1	-1	-1	1	-1	1	-1	-1	-1
-1	-1	1	1	1	1	1	-1	1	-1	1	1	-1	-1	1	1	-1	-1	1	-1	1	-1	-1
-1	-1	-1	1	1	1	1	1	-1	1	-1	1	1	-1	-1	1	1	-1	-1	1	-1	1	-1
-1	-1	-1	-1	1	1	1	1	1	-1	1	-1	1	1	-1	-1	1	1	-1	-1	1	-1	1
1	-1	-1	-1	-1	1	1	1	1	1	-1	1	-1	1	1	-1	-1	1	1	-1	-1	1	-1
-1	1	-1	-1	-1	-1	1	1	1	1	1	-1	1	-1	1	1	-1	-1	1	1	-1	-1	1
1	-1	1	-1	-1	-1	-1	1	1	1	1	1	-1	1	-1	1	1	-1	-1	1	1	-1	-1
-1	1	-1	1	-1	-1	-1	-1	1	1	1	1	1	-1	1	-1	1	1	-1	-1	1	1	-1
-1	-1	1	-1	1	-1	-1	-1	-1	1	1	1	1	1	-1	1	-1	1	1	-1	-1	1	1
1	-1	-1	1	-1	1	-1	-1	-1	-1	1	1	1	1	1	-1	1	-1	1	1	-1	-1	1
1	1	-1	-1	1	-1	1	-1	-1	-1	-1	1	1	1	1	1	-1	1	-1	1	1	-1	-1
-1	1	1	-1	-1	1	-1	1	-1	-1	-1	-1	1	1	1	1	1	-1	1	-1	1	1	-1
-1	-1	1	1	-1	-1	1	-1	1	-1	-1	-1	-1	1	1	1	1	1	-1	1	-1	1	1
1	-1	-1	1	1	-1	-1	1	-1	1	-1	-1	-1	-1	1	1	1	1	1	-1	1	-1	1
1	1	-1	-1	1	1	-1	-1	1	-1	1	-1	-1	-1	-1	1	1	1	1	1	-1	1	-1
-1	1	1	-1	-1	1	1	-1	-1	1	-1	1	-1	-1	-1	-1	1	1	1	1	1	-1	1
1	-1	1	1	-1	-1	-1	1	1	-1	-1	-1	1	-1	-1	-1	-1	-1	1	1	1	1	-1
-1	1	-1	1	1	-1	-1	1	1	-1	-1	1	-1	1	-1	-1	-1	-1	1	1	1	1	1
1	-1	1	-1	1	1	-1	-1	1	1	-1	-1	1	-1	1	-1	-1	-1	1	1	1	1	1
1	1	-1	1	-1	1	1	-1	-1	1	1	-1	-1	1	-1	1	-1	-1	-1	-1	1	1	1
1	1	1	-1	1	-1	1	1	-1	-1	1	1	-1	-1	1	-1	1	-1	-1	-1	-1	1	1
1	1	1	1	-1	1	-1	1	1	-1	-1	1	1	-1	-1	1	-1	1	-1	-1	-1	-1	1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

Table 1: The PB Experiment

We measure the overall runtime of an application as the effect in this experiment. After the overall runtime of an application is measured for each set of compiler optimizations, the effect of each parameter is calculated as follows. Let the "effect column" be a column vector with the same height as the columns in the PB foldover design (one component per set of compiler optimizations). This effect column is

populated with the overall running time for the corresponding set of compiler optimizations. The contribution of a specific optimization towards the total overall runtime is then calculated by taking the dot product of that optimization's column (of +1's and -1's) and the effect column. The resulting value gives the insight into how the optimization is believed to affect overall runtime. Since all the columns in the matrix are orthogonal, the contribution of other optimizations is assumed to be negligible when calculating a specific optimization's effect. If the effect is large and positive, then in general, when the optimization is turned on, the overall runtime increases. If the value is large and negative, then in general, when the optimization is on, the overall runtime decreases.

3.2 Jikes RVM

Jikes RVM is an open source Java virtual machine, initially developed by researchers at IBM [4]. The optimizing compiler provides an extensive selection of command line compiler optimization options. This allows the optimizations to easily be turned on and off for different tests.

Jikes RVM does support adaptive compilation [4]. It works by selectively choosing "hot" methods that are frequently executed and further optimizing those methods. While this is an interesting feature, it addresses different research questions. In this experiment, we seek to find optimizations that tend to decrease overall runtime when used across the entire program. Thus, for these experiments, adaptive recompilation is deactivated. The initial compiler is set to be the optimizing compiler so that the selected optimizations are performed. Enabling the adaptive compilation system would introduce inconsistencies in the measurements.

3.3 DaCapo Benchmarks

We use the applications in the DaCapo Benchmark suite [5], version 2006-10-MR2, in this experiment. Each benchmark is tested individually to determine which compiler optimizations are best for that particular benchmark.

The DaCapo benchmarks are chosen over other benchmark suites for this experiment because of the ability to easily distinguish whether or not a benchmark passes or fails. Certain compiler optimization selections actually result in exceptions being thrown by the Jikes RVM optimizing compiler, causing undesired failures. The assurance of a one-line pass or fail at the end of each benchmark run increases the confidence that the results are trustworthy.

4 The PB Experiment

4.1 Experiment Setup

The experiment consists of two phases. First, we run the foldover PB Design with $n = 23$ ($X = 24 > 23$) to calculate the estimated effect of individual compiler optimizations on overall runtime.

Then, compiler optimizations that show a desirable effect (a decrease in overall runtime) are enabled, and all other compiler optimizations are disabled. For each DaCapo benchmark that is tested, specific optimizations are chosen on a per-benchmark basis. Also tested are the Jikes RVM default settings for O0, O1, and O2 per benchmark, as well as nearly all optimizations turned off (see the section on "Testing Decisions" below).

4.2 Compiler Options

4.2.1 Compiler Optimization Descriptions

The documentation for some of the Jikes RVM compiler optimizations are a bit vague and lack intuitive descriptions as to what they actually do. This section presents the options that are considered in this experiment, along with the actions they are believed to perform.

field_analysis: Perform field analysis on class fields. See [8]. According to a comment in the code, only private fields are currently analyzed. [4]

inline: Perform inlining. Note that "trivial, unguarded inlines" are the only inlines that are done at optimization level 0 (this experiment was run at optimization level 2) [4]

inline_guarded: Inline methods with guards when necessary. A guard is typically a cheaper instruction (cheaper than a call) that can decide whether or not the inlined code is executed.

inline_guarded_interfaces: If there is a single implementation of an interface in the current class hierarchy, then consider inlining the single implementation of the interface's method that is being called. [4]

inline_preex: If the target of a virtual call currently exists, then the code can be inlined, and an inline guard is not necessary. [4]

simplify_float_ops: Simplify operations containing floats.

simplify_tib_ops: Simplify operations containing type information blocks (TIBs). According to a comment in the Jikes RVM source code, "at runtime it is an array with Object elements." [4]

simplify_field_ops: Simplify operations containing fields.

local_constant_prop: Perform local constant propagation.

local_copy_prop: Perform local copy propagation.

local_cse: Perform local common subexpression elimination.

control_static_splitting: Statically determine code paths and basic blocks that will not be frequently used. Then, eliminate the merge point between infrequent code paths and the frequent code paths by duplicating code. [4]

escape_scalar_replace_aggregates: Perform escape analysis on pointers. If the pointer does not escape the current procedure (it is not accessible outside the procedure), then it may not be necessary to keep a particular object as one contiguous block in memory. Instead, this optimization breaks up the object so that certain values can be stored in registers instead.

escape_monitor_removal: Perform escape analysis on pointers. If the pointer can not escape the current thread of execution (no other threads can access it), then the code that is meant to perform synchronization related with the pointer can be safely removed.

reorder_code: As the code is executing, the frequency of use of basic blocks is calculated. Basic blocks that are found to be infrequently used are moved to the end of the code order. [4]

reorder_code_ph: Basic blocks are reorganized according to the Pettis and Hansen Algo2 [4] [7]

h2l_inline_write_barrier: In this context, a write barrier contains code that performs additional memory operations that are necessary for garbage collection. This compiler option decides whether or not the code performing the memory operations will be considered for inlining. [4]

h2l_inline_primitive_write_barrier: This option performs the same as the above, except it deals with primitive types.

l2m_handler_liveness: A PEI is a "potentially excepting instruction." According to a comment in the Jikes RVM code, "Performing live analysis may reduce dependences between PEIs and stores." This optimization makes modifications to the dependence graph in order to reduce these dependencies. [4]

regalloc_coalesce_moves: When performing register allocation, try coalescing in order to minimize the number of register moves that are needed. By definition, this optimization attempts to allocate variables to registers that contain no-longer-needed values before allocating to registers that contain values that are still needed.

regalloc_coalesce_spills: When performing register allocation, attempt to reuse areas on the stack that have already been allocated before allocating new areas on the stack to spill registers. [4]

osr_guarded_inlining: OSR stands for on stack replacement. It is used to swap different versions of compiled methods on the fly. OSR points can be placed at guarded inlines in place of a call, in the case of a failed guard. [4] [6]

osr_inline_policy: According to the documentation, this option will "use OSR knowledge to drive more aggressive inlining". However, this does not appear to be used in the source code at this time. [4]

simplify_long_ops: Simplify operations containing longs.

simplify_double_ops: Simplify operations containing doubles.

simplify_integer_ops: Simplify operations containing integers.

simplify_ref_ops: Simplify operations containing references. [4]

simplify_chase_final_fields: Since the value of a final field does not change, eliminate loads of these fields at runtime by getting the value of the final field at compile time. [4]

h2l_inline_new: This stage occurs as part of the H2L (high intermediate representation to low intermediate representation) conversion process. When activated, allocation of new scalars and arrays are inlined in the code. According to a code comment, these instructions are "implemented as calls to runtime service methods", so inlining would eliminate a call. [4]

4.2.2 Testing Decisions

Not all of the compiler optimizations available with Jikes RVM are tested. We made this decision because the Jikes RVM documentation mentions that several compiler optimizations are believed to be broken. We initially ran larger PB experiments as attempts to enable these questionable options. However, enabling some of these options resulted in failed benchmarks, so we decided to just test the optimizations enabled by default for optimization level O2. All other optimizing compiler options remain at their default values.

Of the remaining 29 optimizations, the initial PB experiments still revealed many failed benchmarks with particular on/off combinations of compiler optimizations. After analyzing the data, we found that four options seem to cause failures when they are disabled. These options are simplify_long_ops, simplify_double_ops, simplify_integer_ops, and simplify_ref_ops. After fixing these optimizations to always be on, nearly all failures caused by different selections of optimizations were eliminated.

An additional observation made in the initial PB experiments is that the compilation time for benchmarks increases significantly whenever simplify_chase_final_fields is disabled and h2l_inline_new is enabled. It is clear that these two settings should never be set as stated, so these two optimizations are removed from the PB experiment. In order to still measure the other three combinations possible for these two options, they are manually varied. There are three runs of the final $X = 24$ PB experiment, one for each of the three remaining combinations for simplify_chase_final_fields and h2l_inline_new. Following in this paper, references to "FF", "TF" or "TT" in different experiments correspond to the respective settings of simplify_chase_final_fields and h2l_inline_new.

Each row of the $X = 24$ experiment displayed in Table 1 represents one combination of on/off settings for the specified compiler optimizations. Each row is run three times, we use the best overall time in determining the impact of the optimizations. We assume that the best overall time represents the time with minimal disturbance from background processes running on the system.

4.3 Timing

We have collected our measurements using the "Linux time" command and the built-in Jikes RVM compilation timing system.

The user mode time spent by the Jikes RVM process, measured by the "time" command, is our measurement for total runtime. We found that the total process running time, measured by the "time" command, was significantly greater than the combined total of user and kernel time. Since the sum of user and kernel time should be equal to the total running time, this indicated that something else was happening in the background on the machine, causing the total time spent to increase. Every DaCapo benchmark has a built in timing mechanism that measures the runtime of the benchmark. This mechanism is subject to the same issue, which is why it is not used.

The option "-X:vm:measureCompilation=true" is used to collect compilation time. This makes it possible to divide the runtime into both compile time and execution time. The option "-X:vm:measureCompilationPhases=true" provides data on what compilation phases take the most time.

5 PB Results

We measure the effect of each compiler optimization on each benchmark. If an optimization shows a negative effect, it means that the overall effect of that optimization across all the tests is a decrease in runtime. Conversely, the optimizations showing positive effects contribute to longer runtimes. The final selected optimizations for each individual benchmark are all optimizations where the PB experiment indicates a decreased runtime contribution. The optimizations chosen for each benchmark are listed in Table 2.

5.1 Select Optimization Results

Each benchmark is run three separate times using the selected optimizations, and the best runtime is taken as the sample. In general, we observed that the selected optimizations outperform the optimizations for the O0, O1, and O2 optimization levels. Table 2 shows which options are on and which options are off.

The benchmarks Luindex and Chart were not tested. Luindex is excluded due to persistent failures, despite the fixed compiler optimizations. The reason for the failures remains uninvestigated. Chart is not tested because the machine that the benchmarks are run on does not have the necessary packages installed for the benchmark to run.

All other benchmarks in the DaCapo benchmark suite are run with the optimizations in Table 2. Of the remaining rows in the chart, the first three rows are run to give a baseline before any of the selected optimizations are applied. The next three rows are the default selected optimizations for the O0, O1, and O2 optimization levels in Jikes RVM. Note that O2 turns on all optimizations that are tested, so it is unnecessary to run a test that turns all optimizations on for comparison. The next three rows are the average rows. The optimizations for these average tests are calculated from all three runs of all the benchmarks (not just the best runs).

field_analysis	inline	inline_guarded	inline_guarded_interfaces	inline_preex	simplify_float_ops	simplify_tib_ops	simplify_field_ops	local_constant_prop	local_copy_prop	local_cse	control_static_splitting	escape_scalar_replace_aggregates	escape_monitor_removal	reorder_code	reorder_code_ph	h2l_inline_write_barrier	h2l_inline_primitive_write_barrier	l2m_handler_liveness	regalloc_coalesce_moves	regalloc_coalesce_spills	osr_guarded_inlining	osr_inline_policy	simplify_long_ops	simplify_double_ops	simplify_integer_ops	simplify_ref_ops	simplify_chase_final_fields	h2l_inline_new	
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	1	1	1	1	-1	-1	All Off Except Needed Four FF
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	1	1	1	1	1	-1	All Off Except Needed Four TF
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	1	1	1	1	1	1	All Off Except Needed Four TT
1	1	1	1	1	1	1	1	1	1	1	-1	-1	-1	1	-1	-1	-1	-1	1	1	-1	-1	1	1	1	1	1	1	O0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	-1	1	1	1	1	1	1	1	1	1	1	O1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	O2 (Equivalent to AllOn)
-1	-1	-1	-1	-1	-1	-1	1	-1	-1	1	-1	-1	-1	-1	-1	-1	1	-1	-1	-1	-1	-1	1	1	1	1	-1	-1	Average FF Select
-1	-1	-1	-1	-1	-1	-1	1	-1	-1	1	-1	-1	-1	-1	-1	-1	1	-1	-1	1	-1	-1	1	1	1	1	-1	-1	Average TF Select
-1	-1	-1	-1	-1	-1	-1	1	-1	-1	1	-1	-1	-1	-1	-1	-1	1	-1	-1	1	-1	-1	1	1	1	1	1	1	Average TT Select
-1	-1	-1	-1	-1	1	-1	1	-1	-1	-1	1	-1	-1	-1	-1	-1	1	-1	-1	1	1	-1	1	1	1	1	-1	-1	Antlr FF Select
-1	-1	-1	-1	-1	1	-1	1	-1	-1	-1	1	-1	-1	-1	-1	-1	1	-1	-1	1	1	1	1	1	1	1	1	-1	Antlr TF Select
-1	-1	-1	-1	-1	1	-1	1	-1	-1	-1	1	-1	-1	-1	-1	-1	-1	-1	-1	1	1	-1	1	1	1	1	1	1	Antlr TT Select
-1	-1	-1	-1	-1	-1	-1	1	1	-1	1	-1	-1	-1	-1	-1	-1	1	-1	-1	-1	1	-1	1	1	1	1	-1	-1	Bloat FF Select
1	-1	-1	-1	-1	-1	-1	1	1	-1	1	-1	-1	-1	-1	-1	-1	1	-1	-1	1	-1	-1	1	1	1	1	1	-1	Bloat TF Select
-1	-1	-1	-1	-1	-1	-1	1	1	-1	-1	-1	-1	-1	-1	-1	-1	1	-1	-1	1	-1	1	1	1	1	1	1	1	Bloat TT Select
-1	-1	-1	-1	-1	-1	-1	1	1	-1	1	-1	-1	-1	-1	-1	-1	1	-1	-1	1	-1	-1	1	1	1	1	-1	-1	Eclipse FF Select
-1	-1	-1	-1	-1	-1	-1	1	1	-1	1	-1	-1	-1	-1	-1	-1	1	-1	-1	1	-1	-1	1	1	1	1	1	-1	Eclipse TF Select
-1	-1	-1	-1	-1	-1	-1	1	-1	-1	1	-1	-1	-1	-1	-1	-1	1	-1	-1	1	-1	-1	1	1	1	1	1	1	Eclipse TT Select
-1	-1	-1	1	-1	-1	-1	1	1	-1	-1	-1	-1	1	-1	-1	-1	1	-1	-1	1	-1	-1	1	1	1	1	-1	-1	Fop FF Select
-1	-1	-1	1	-1	-1	-1	1	-1	1	1	-1	-1	-1	1	-1	-1	1	-1	-1	1	-1	-1	1	1	1	1	1	-1	Fop TF Select
-1	-1	-1	1	-1	-1	-1	1	-1	-1	-1	-1	1	-1	-1	-1	-1	1	-1	-1	1	1	-1	1	1	1	1	1	1	Fop TT Select
1	-1	-1	-1	-1	1	-1	1	-1	-1	-1	-1	-1	-1	1	-1	-1	1	-1	-1	-1	1	-1	1	1	1	1	-1	-1	Hsqldb FF Select
-1	-1	-1	-1	-1	-1	-1	1	-1	-1	-1	1	-1	1	1	-1	-1	-1	-1	-1	1	-1	-1	1	1	1	1	1	-1	Hsqldb TF Select
-1	-1	-1	-1	1	-1	-1	1	-1	-1	-1	1	-1	-1	1	-1	-1	1	-1	-1	1	-1	-1	1	1	1	1	1	1	Hsqldb TT Select
-1	-1	-1	1	-1	-1	-1	1	1	1	1	-1	-1	-1	-1	-1	-1	1	-1	-1	-1	-1	-1	1	1	1	1	-1	-1	Jython FF Select
1	-1	-1	1	-1	-1	-1	1	1	1	1	-1	1	-1	-1	-1	-1	1	-1	-1	1	-1	-1	1	1	1	1	1	-1	Jython TF Select
-1	-1	-1	1	-1	-1	-1	1	1	1	1	-1	-1	-1	-1	-1	-1	1	-1	-1	-1	1	-1	1	1	1	1	1	1	Jython TT Select
-1	1	1	-1	1	-1	-1	1	1	1	-1	-1	1	1	1	1	-1	1	-1	1	-1	1	-1	1	1	1	1	-1	-1	Lusearch FF Select
-1	1	1	1	1	1	-1	1	1	1	-1	1	1	-1	1	1	1	1	-1	-1	-1	1	1	1	1	1	1	1	-1	Lusearch TF Select
1	1	1	-1	1	1	-1	1	-1	-1	-1	-1	1	1	1	1	1	1	-1	-1	1	-1	1	1	1	1	1	1	1	Lusearch TT Select
-1	-1	-1	-1	-1	-1	-1	1	1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	1	1	1	1	1	-1	Pmd FF Select
-1	-1	-1	-1	-1	-1	-1	1	-1	1	-1	1	-1	-1	-1	-1	-1	1	-1	-1	-1	-1	1	1	1	1	1	1	-1	Pmd FF Select
-1	-1	-1	-1	-1	-1	-1	1	-1	-1	-1	-1	-1	1	-1	-1	-1	1	-1	-1	1	-1	1	1	1	1	1	1	1	Pmd TT Select
-1	-1	-1	1	1	1	-1	1	-1	-1	-1	-1	-1	-1	1	-1	-1	1	-1	1	1	-1	1	1	1	1	1	-1	-1	Xalan FF Select
-1	-1	-1	-1	1	1	-1	1	-1	-1	-1	1	-1	1	-1	-1	-1	-1	-1	-1	1	-1	1	1	1	1	1	1	-1	Xalan TF Select
-1	-1	-1	-1	1	1	-1	1	-1	1	-1	-1	-1	-1	1	-1	-1	1	-1	-1	-1	1	-1	1	1	1	1	1	1	Xalan TT Select

Table 2: Selected Optimizations for each Benchmark. Enabled optimizations are shaded for convenience.

The graphs on the following pages show the total overall time for each benchmark under different compiler optimizations. The first three bars represent all optimizations turned off, except for the ones listed in the respective first three rows of Table 2. The second three bars are the optimizations that are selected for that particular benchmark. The last three bars show the performance of the default Jikes RVM compiler optimizations for O0, O1, and O2. All the graphs are normalized to the "All Off Except 4 FF" bar for each benchmark, the setting with the fewest compiler options. Each bar is split into compilation time and execution time, with compilation time on the bottom. Both of these values are expressed as the percentage of the overall time that the "All Off Except 4 FF" test spent running.

Summarizing the results, the selected optimizations outperform the selected optimizations for O0, O1, and O2 in all cases. The selected optimizations for Antlr TF result in a 621% improvement over O0, a 688% improvement over O1, and a 716% improvement over O2 optimizations. Fop TT gives a 135% improvement over O0, a 174% improvement over O1, and a 184% improvement over O2. The least improvement occurs in Lusearch, yet Lusearch TF still outperforms O0, O1, and O2 by 11.3%, 10.3%, and 12.1% respectively.

In some cases, it turns out that turning nearly all optimizations off outperforms the selected optimizations in terms of overall runtime, when compared with the respective FF, TF, and TT tests. This occurs in the following cases: Eclipse FF and TF, Fop FF and TF, Hsqldb TT, Pmd TF and TT, Xalan TF. The worst relative performers in this respect were Pmd TT and Pmd TF, but they only perform about 2.65% and 2.48% worse than the corresponding All Off Except Four TT and TF cases. All the other cases have less than 2% of a difference.

It appears that of the FF, TF, and TT selected options, TF is the preferred selection. TT results nearly always perform the worst, with the exceptions being Eclipse and Jython TT. In Eclipse, the TT test outperforms both the FF and TF tests, and in Jython, the TT test slightly outperforms the FF test. The FF tests only outperform the TF tests in Pmd and Xalan. As mentioned previously, early experimentation discovered that the FT combination (simplify_chase_final_fields=false and h2l_inline_new=true) incurs a significantly large increase in overall runtime. It is possible that certain combinations of compiler optimizations that are tested in this experiment happen to have a significant impact on overall runtime when used in a particular combination with one another. It should be noted that this experiment does not necessarily capture all of these interactions. Interdependency among the different compiler optimizations tested has not been explored in this experiment.

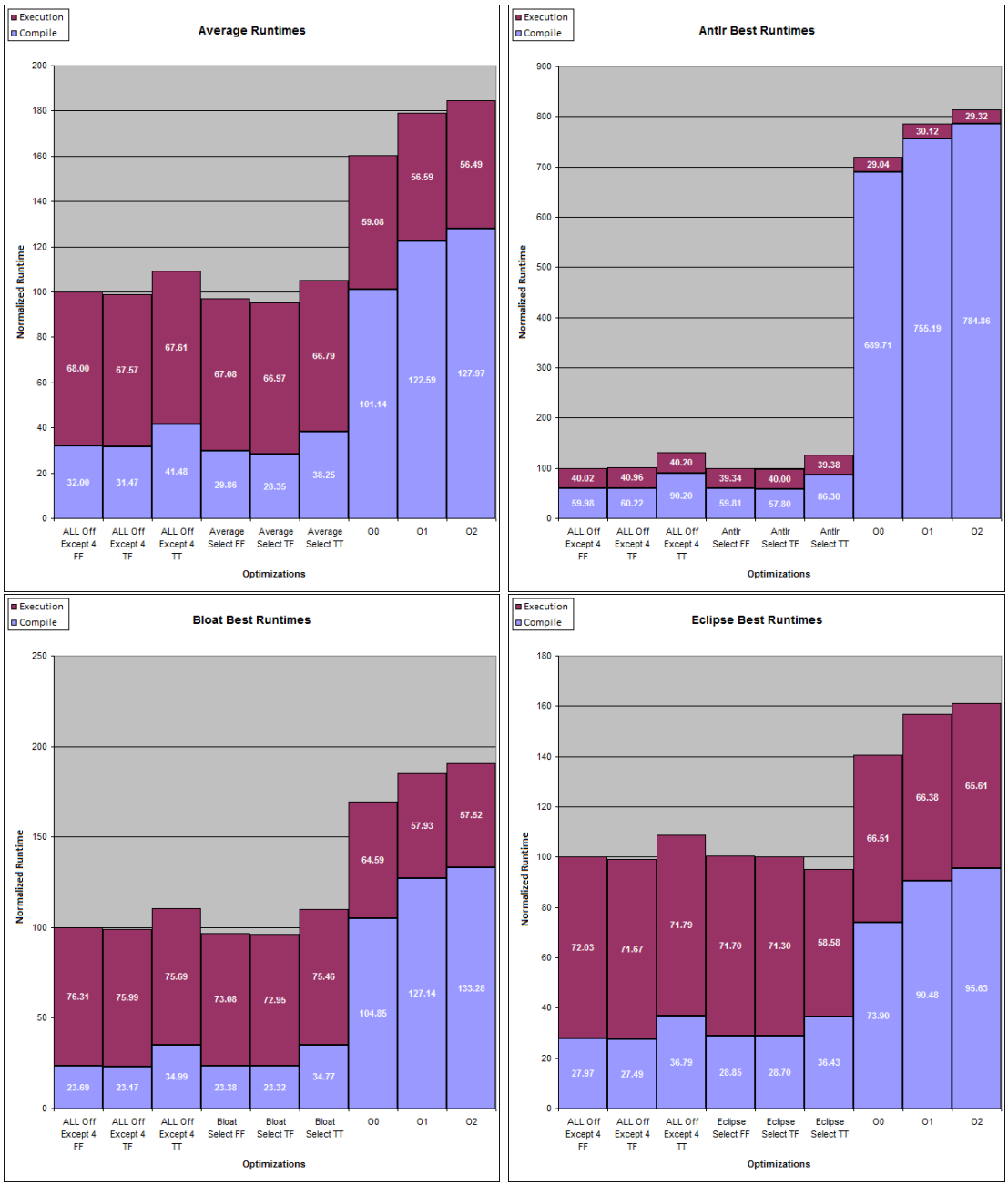

Figure 1: Normalized values for the Average running times of all benchmarks, and the best runs of the Antlr, Bloat, and Eclipse benchmarks.

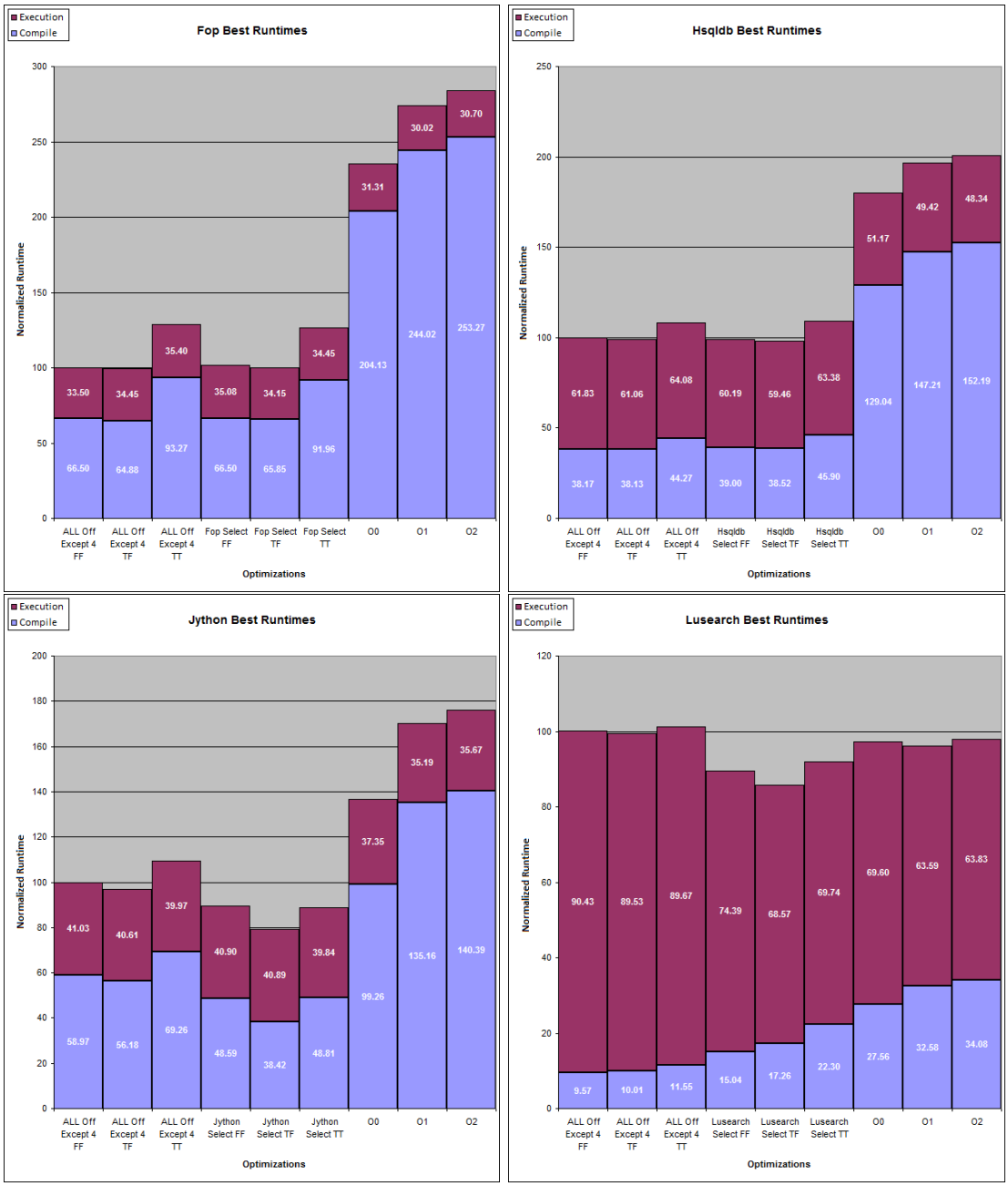

Figure 2: Normalized values for the Fop, Hsqldb, Jython, and Lusearch benchmarks.

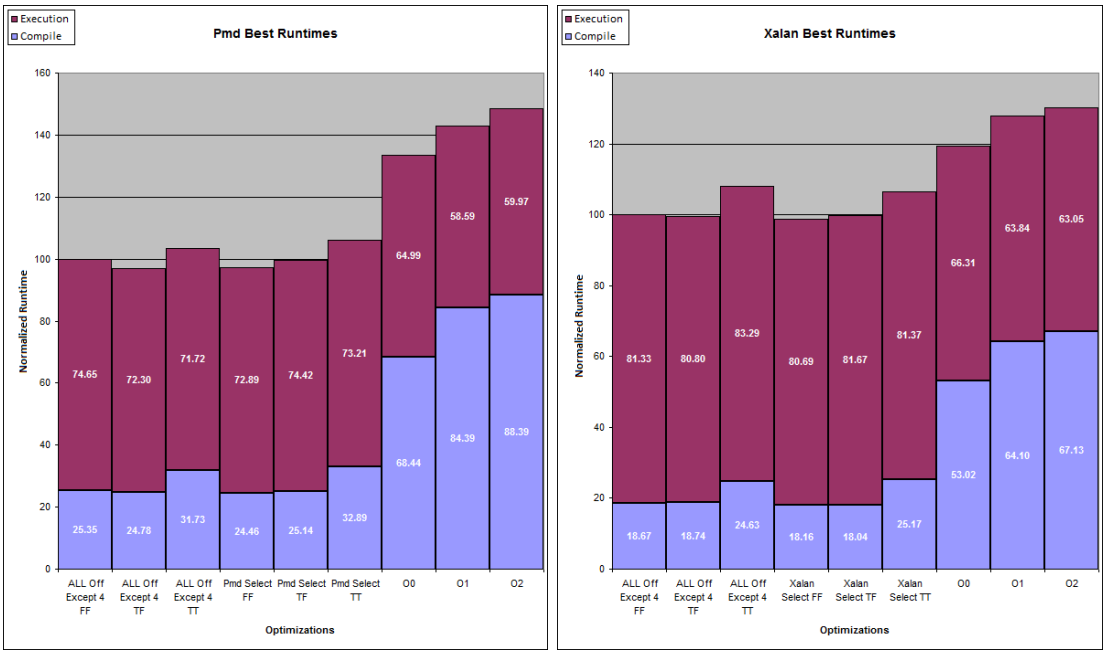

Figure 3: Normalized values for the Pmd and Xalan benchmarks.

6 Data and Analysis

To provide an insight as to why simplify_chase_final_fields turned off and h2l_inline_new turned on results in a horrible runtime, we consulted the Jikes RVM documentation. The former option, when activated, will get the value of final fields at compile time, eliminate the load operation that is initially needed to get that value, and replace it with a simpler operation. The latter option inlines the allocation of scalar and array values. When the first is off and the second is on, this combination would presumably result in many inlined loads added to the code.

Our PB experiment selects simplify_field_ops for every single benchmark. This comes to no surprise, as this optimization targets member variables in objects. Specifically, it simplifies the code needed to get the length of an array and the code needed to check if a given index is within the bounds of an array. It also simplifies call instructions. When simplify_chase_final_fields is activated as well, the compiler propagates final field values to eliminate the load for the field. These observations were all made after Jikes RVM source code analysis. [4] There are undoubtedly a sizable number of applicable instances where this optimization is worthwhile.

There are some factors that could lead to a biased analysis either toward or away from certain optimizations. One factor is the implementation of particular optimizations. If a simpler but more costly algorithm is used to perform a certain optimization, it is less likely to result in an improvement in overall running time. There may be some cases in the Jikes RVM source code where the best algorithms are not used.

Some optimizations may actually enable or disable other optimizations. For example, after the running of this experiment, we discovered that reorder_code and reorder_code_ph have an interesting interaction. By code analysis, it appears that neither has an effect if reorder_code is disabled. However, if both are enabled, then one particular algorithm is executed, whereas if only reorder_code is enabled, a different algorithm is executed. [4] Thus, two of the four combinations of these two options have the same effect, which would bias the results in whatever direction the effect happens to be.

It is questionable whether or not testing osr_guarded_inlining and osr_inline_policy help the experiment. After further code analysis, it appears that the adaptive recompilation system needs to be activated in order to take full advantage of these two optimizations. Since adaptive recompilation is disabled, these optimizations may do some initial work without using the results.

It should also be noted that some of the optimizations have a dependency on the optimization level (0, 1, 2, 3) selected. For example, only trivial and unguarded inlining is performed when the optimization level is set to 0 [4]. When the level is set higher, the compiler attempts more complex inlines. Thus, varying the optimization level varies the behavior of the inlineoptimization. To provide greater consistency, the optimization level is set to level 2 for all experiments (including the experiments that test the default O0 and O1 optimizations). By fixing the level at 2, it eliminates variability of specific optimizations based on the current optimization level.

Following are the compile-time breakdowns of the best performing All Off, Select, O0, O1, and O2 tests for each benchmark. Note that most of the compilation time, in all cases, is spent in the conversion process from one language to another: Java bytecode to HIR (high intermediate representation), HIR to LIR (low intermediate representation), LIR to MIR (machine intermediate representation), and MIR to machine code. A significant portion of time is also spent during the register mapping phase.

Antlr Compilation Time %	AllOff FF	AllOff TF	AllOff TT	Select FF	Select TF	Select TT	O0	O1	O2
Convert Bytecodes to HIR	7.40%	7.59%	5.32%	8.16%	8.55%	5.73%	9.86%	9.84%	9.33%
CFG Transformations	2.27%	2.51%	1.55%	2.57%	2.36%	1.82%	1.75%	1.95%	1.63%
CFG Structural Analysis	1.15%	1.10%	0.74%	1.08%	1.13%	0.76%	0.75%	0.63%	0.72%
Simple Opts	1.40%	1.38%	0.91%	1.38%	1.43%	0.98%	1.06%	1.08%	0.87%
Escape Transformations	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.93%	1.05%
Branch Optimizations	0.73%	0.73%	0.48%	0.86%	0.74%	0.50%	0.49%	0.45%	0.45%
Local CopyProp	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.17%	0.11%	0.10%
Local ConstantProp	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.40%	0.24%	0.23%
Local CSE	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.60%	0.39%	0.43%
Field Analysis	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.05%	0.04%	0.04%
Convert HIR to LIR	4.55%	4.59%	30.54%	4.66%	4.50%	29.20%	29.21%	29.18%	28.39%
Local CopyProp	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.35%	0.39%	0.51%
Local ConstantProp	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.70%	0.88%	0.89%
Local CSE	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.65%	0.78%	0.72%
Simple Opts	1.91%	1.99%	1.72%	1.84%	2.31%	1.84%	1.46%	1.41%	1.31%
Basic Block Frequency Estimation	1.61%	1.44%	1.94%	1.41%	1.56%	1.77%	2.08%	1.88%	1.85%
Code Reordering	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.09%	1.98%	1.94%
Branch Optimizations	1.15%	1.14%	1.80%	1.12%	1.22%	1.15%	1.21%	1.69%	1.35%
Convert LIR to MIR	33.64%	35.00%	18.50%	33.78%	33.10%	18.57%	21.17%	20.30%	23.41%
Register Mapping	38.14%	36.04%	30.94%	36.81%	36.95%	32.63%	23.86%	22.29%	21.18%
Branch Optimizations	1.32%	1.69%	1.20%	1.37%	1.48%	1.24%	0.92%	0.77%	0.87%
Generate Machine Code	4.71%	4.82%	4.37%	4.95%	4.69%	3.82%	3.17%	2.79%	2.72%

Table 3: Antlr compile time percentage breakdowns.

Bloat Compilation Time %	AllOff FF	AllOff TF	AllOff TT	Select FF	Select TF	Select TT	O0	O1	O2
Convert Bytecodes to HIR	8.14%	7.83%	7.15%	8.59%	9.01%	5.37%	8.42%	7.79%	7.28%
CFG Transformations	2.39%	2.48%	1.58%	2.49%	2.47%	1.94%	2.02%	1.66%	1.38%
CFG Structural Analysis	1.20%	1.25%	0.79%	1.24%	1.21%	0.79%	0.92%	0.60%	0.49%
Simple Opts	1.81%	1.70%	0.99%	1.51%	1.69%	1.09%	1.06%	0.77%	0.77%
Escape Transformations	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	1.10%	1.06%
Branch Optimizations	0.76%	0.82%	0.50%	1.05%	0.80%	0.50%	0.49%	0.41%	0.39%
Local CopyProp	0.00%	0.00%	0.00%	0.00%	0.32%	0.00%	0.20%	0.12%	0.12%
Local ConstantProp	0.00%	0.00%	0.00%	0.18%	0.18%	0.13%	0.18%	0.14%	0.13%
Local CSE	0.00%	0.00%	0.00%	1.68%	1.67%	0.00%	0.57%	0.47%	0.44%
Field Analysis	0.00%	0.00%	0.00%	0.00%	0.16%	0.00%	0.07%	0.06%	0.06%
Convert HIR to LIR	4.96%	4.79%	26.23%	4.98%	5.32%	26.81%	30.21%	31.80%	31.09%
Local CopyProp	0.00%	0.00%	0.00%	0.00%	0.71%	0.00%	0.48%	0.53%	0.49%
Local ConstantProp	0.00%	0.00%	0.00%	0.36%	0.36%	0.28%	0.38%	1.09%	0.72%
Local CSE	0.00%	0.00%	0.00%	1.82%	1.74%	0.00%	0.68%	0.65%	0.66%
Simple Opts	2.63%	2.07%	1.90%	2.32%	2.46%	1.90%	1.65%	1.70%	1.37%
Basic Block Frequency Estimation	2.07%	2.04%	2.11%	1.94%	1.99%	1.88%	2.03%	2.20%	1.73%
Code Reordering	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.12%	1.73%	1.59%
Branch Optimizations	1.21%	1.50%	1.07%	1.19%	2.94%	1.14%	1.35%	1.83%	1.57%
Convert LIR to MIR	25.01%	25.75%	17.20%	22.75%	20.42%	17.16%	17.64%	15.99%	20.33%
Register Mapping	42.79%	42.67%	35.16%	41.07%	40.01%	35.87%	27.34%	25.72%	24.63%
Branch Optimizations	1.36%	1.73%	1.03%	1.43%	1.31%	1.04%	1.11%	0.91%	0.86%
Generate Machine Code	5.68%	5.38%	4.30%	5.39%	5.24%	4.09%	3.07%	2.75%	2.83%

Table 4: Bloat compile time percentage breakdowns.

Eclipse Compilation Time %	AllOff FF	AllOff TF	AllOff TT	Select FF	Select TF	Select TT	O0	O1	O2
Convert Bytecodes to HIR	8.45%	8.33%	7.02%	9.05%	8.13%	6.46%	10.88%	10.51%	9.51%
CFG Transformations	3.09%	4.25%	2.36%	3.13%	3.00%	2.35%	2.53%	2.08%	1.86%
CFG Structural Analysis	1.58%	1.70%	1.28%	1.66%	2.46%	1.35%	1.04%	0.91%	0.82%
Simple Opts	1.82%	1.94%	1.36%	1.86%	2.26%	1.41%	1.42%	1.14%	1.17%
Escape Transformations	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	1.29%	1.21%
Branch Optimizations	0.95%	1.11%	0.74%	1.01%	1.01%	0.79%	0.75%	0.57%	0.57%
Local CopyProp	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.26%	0.22%	0.18%
Local ConstantProp	0.00%	0.00%	0.00%	0.25%	0.24%	0.00%	0.35%	0.23%	0.20%
Local CSE	0.00%	0.00%	0.00%	1.25%	1.24%	0.99%	0.83%	0.63%	0.59%
Field Analysis	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.12%	0.09%	0.09%
Convert HIR to LIR	8.04%	7.55%	23.63%	7.13%	7.29%	24.99%	24.62%	29.72%	27.95%
Local CopyProp	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.63%	0.45%	0.71%
Local ConstantProp	0.00%	0.00%	0.00%	0.33%	0.34%	0.00%	0.41%	0.95%	0.95%
Local CSE	0.00%	0.00%	0.00%	1.32%	1.43%	1.25%	0.82%	0.79%	0.77%
Simple Opts	2.64%	2.48%	2.02%	2.22%	2.30%	2.22%	2.31%	1.67%	1.64%
Basic Block Frequency Estimation	3.50%	2.51%	2.19%	2.55%	2.40%	2.26%	1.98%	1.92%	2.05%
Code Reordering	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.15%	1.52%	1.40%
Branch Optimizations	1.57%	1.57%	1.51%	1.48%	1.44%	1.39%	1.48%	1.89%	1.94%
Convert LIR to MIR	21.30%	20.34%	18.96%	19.87%	19.98%	17.89%	16.73%	15.13%	19.78%
Register Mapping	38.65%	39.21%	32.77%	39.01%	39.09%	30.72%	27.77%	24.09%	22.58%
Branch Optimizations	1.48%	2.50%	1.26%	2.42%	1.61%	1.30%	1.19%	0.91%	0.85%
Generate Machine Code	6.94%	6.49%	4.90%	5.48%	5.77%	4.63%	3.73%	3.29%	3.16%

Table 5: Eclipse compile time percentage breakdowns.

Fop Compilation Time %	AllOff FF	AllOff TF	AllOff TT	Select FF	Select TF	Select TT	O0	O1	O2
Convert Bytecodes to HIR	8.53%	10.27%	6.24%	9.23%	9.03%	6.99%	9.02%	8.54%	8.52%
CFG Transformations	3.00%	3.01%	2.15%	2.99%	3.12%	2.19%	2.80%	2.04%	1.77%
CFG Structural Analysis	1.55%	2.19%	1.11%	1.55%	1.55%	1.11%	0.86%	0.96%	0.71%
Simple Opts	1.80%	1.80%	1.36%	1.81%	1.81%	1.36%	1.37%	1.08%	1.07%
Escape Transformations	0.00%	0.00%	0.00%	2.32%	0.00%	1.67%	0.00%	1.55%	1.35%
Branch Optimizations	0.86%	0.88%	0.62%	0.90%	0.86%	0.68%	0.61%	0.52%	0.51%
Local CopyProp	0.00%	0.00%	0.00%	0.00%	0.27%	0.00%	0.27%	0.17%	0.15%
Local ConstantProp	0.00%	0.00%	0.00%	0.25%	0.00%	0.00%	0.19%	0.26%	0.17%
Local CSE	0.00%	0.00%	0.00%	0.00%	2.19%	0.00%	0.62%	0.57%	0.56%
Field Analysis	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.11%	0.09%	0.08%
Convert HIR to LIR	5.41%	5.10%	24.14%	5.52%	5.48%	25.33%	25.38%	29.08%	29.98%
Local CopyProp	0.00%	0.00%	0.00%	0.00%	0.48%	0.00%	0.44%	0.40%	0.45%
Local ConstantProp	0.00%	0.00%	0.00%	0.36%	0.00%	0.00%	0.51%	0.86%	0.92%
Local CSE	0.00%	0.00%	0.00%	0.00%	1.59%	0.00%	0.80%	0.76%	0.73%
Simple Opts	2.01%	2.03%	1.80%	2.05%	2.30%	2.03%	1.56%	1.60%	1.69%
Basic Block Frequency Estimation	1.97%	1.99%	2.17%	2.02%	2.87%	2.11%	1.67%	1.72%	1.61%
Code Reordering	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.14%	1.42%	1.47%
Branch Optimizations	1.25%	1.27%	1.15%	3.40%	1.24%	2.21%	1.23%	1.64%	1.61%
Convert LIR to MIR	29.36%	26.26%	22.30%	22.90%	23.87%	20.07%	23.42%	21.03%	22.36%
Register Mapping	36.93%	36.64%	31.18%	37.03%	36.01%	28.03%	24.56%	21.65%	20.46%
Branch Optimizations	1.38%	1.38%	1.12%	1.83%	1.35%	1.55%	1.13%	0.79%	0.79%
Generate Machine Code	5.95%	7.18%	4.67%	5.84%	5.97%	4.66%	3.28%	3.27%	3.03%

Table 6: Fop compile time percentage breakdowns.

Hsqldb Compilation Time %	AllOff FF	AllOff TF	AllOff TT	Select FF	Select TF	Select TT	O0	O1	O2
Convert Bytecodes to HIR	7.35%	7.91%	6.74%	19.03%	7.42%	8.16%	7.94%	9.63%	9.97%
CFG Transformations	2.82%	2.34%	2.40%	2.44%	2.50%	1.99%	2.67%	1.79%	1.50%
CFG Structural Analysis	1.14%	1.15%	1.08%	1.14%	1.14%	0.95%	0.91%	0.58%	0.55%
Simple Opts	1.43%	1.46%	1.23%	1.46%	1.43%	1.55%	1.52%	0.85%	1.05%
Escape Transformations	0.00%	0.00%	0.00%	0.00%	1.91%	1.44%	0.00%	1.11%	1.09%
Branch Optimizations	0.77%	0.77%	0.66%	0.77%	1.01%	0.66%	0.60%	0.44%	0.46%
Local CopyProp	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.30%	0.18%	0.18%
Local ConstantProp	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.17%	0.14%	0.13%
Local CSE	0.00%	0.00%	0.00%	0.00%	0.00%	1.66%	0.74%	0.60%	0.60%
Field Analysis	0.00%	0.00%	0.00%	0.14%	0.00%	0.00%	0.07%	0.05%	0.05%
Convert HIR to LIR	6.24%	17.37%	20.96%	6.78%	6.05%	20.89%	21.05%	23.04%	22.48%
Local CopyProp	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.31%	0.31%	0.36%
Local ConstantProp	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.34%	0.58%	0.52%
Local CSE	0.00%	0.00%	0.00%	0.00%	0.00%	2.89%	0.81%	0.67%	0.60%
Simple Opts	12.71%	2.39%	1.88%	1.86%	1.84%	2.08%	1.29%	1.22%	1.44%
Basic Block Frequency Estimation	1.95%	1.78%	1.90%	1.99%	1.78%	2.20%	1.44%	1.72%	1.46%
Code Reordering	0.00%	0.00%	0.00%	0.15%	0.15%	0.00%	0.10%	1.16%	1.18%
Branch Optimizations	1.36%	1.22%	1.22%	1.32%	3.52%	1.42%	1.08%	1.41%	4.10%
Convert LIR to MIR	22.13%	22.62%	22.61%	21.17%	28.91%	18.24%	19.66%	17.67%	23.04%
Register Mapping	32.87%	34.97%	32.40%	35.44%	36.37%	30.31%	35.63%	33.60%	26.24%
Branch Optimizations	1.19%	1.44%	1.11%	1.44%	1.17%	1.49%	0.72%	0.83%	0.63%
Generate Machine Code	8.04%	4.57%	5.81%	4.88%	4.80%	4.06%	2.65%	2.45%	2.37%

Table 7: Hsqldb compile time percentage breakdowns.

Jython Compilation Time %	AllOff FF	AllOff TF	AllOff TT	Select FF	Select TF	Select TT	O0	O1	O2
Convert Bytecodes to HIR	3.44%	3.51%	3.09%	4.16%	5.36%	4.57%	14.79%	11.40%	10.76%
CFG Transformations	1.30%	1.92%	1.17%	1.62%	2.30%	1.51%	1.99%	1.50%	1.44%
CFG Structural Analysis	0.69%	0.66%	0.62%	0.80%	2.10%	0.77%	0.76%	0.60%	0.56%
Simple Opts	0.93%	0.92%	0.73%	1.07%	1.37%	1.53%	1.26%	1.03%	0.92%
Escape Transformations	0.00%	0.00%	0.00%	0.00%	1.82%	1.40%	0.00%	1.11%	1.04%
Branch Optimizations	0.46%	0.42%	0.34%	0.59%	0.60%	0.47%	0.56%	0.38%	0.41%
Local CopyProp	0.00%	0.00%	0.00%	1.19%	0.81%	0.63%	0.42%	0.29%	0.28%
Local ConstantProp	0.00%	0.00%	0.00%	0.14%	0.18%	0.15%	0.22%	0.13%	0.13%
Local CSE	0.00%	0.00%	0.00%	1.24%	1.58%	1.04%	0.85%	0.59%	0.58%
Field Analysis	0.00%	0.00%	0.00%	0.00%	0.12%	0.00%	0.10%	0.07%	0.07%
Convert HIR to LIR	2.78%	3.15%	12.55%	3.49%	4.89%	16.73%	18.34%	26.91%	25.33%
Local CopyProp	0.00%	0.00%	0.00%	0.26%	0.43%	0.57%	0.45%	0.31%	0.30%
Local ConstantProp	0.00%	0.00%	0.00%	0.22%	0.29%	0.23%	0.27%	0.73%	0.63%
Local CSE	0.00%	0.00%	0.00%	1.82%	2.07%	1.58%	1.01%	0.91%	0.82%
Simple Opts	1.25%	1.32%	1.59%	1.68%	1.91%	1.77%	1.83%	1.40%	1.43%
Basic Block Frequency Estimation	1.31%	1.59%	1.48%	1.92%	2.23%	1.94%	1.97%	1.97%	1.94%
Code Reordering	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.14%	1.55%	1.50%
Branch Optimizations	0.76%	0.70%	0.71%	0.87%	0.98%	0.96%	1.09%	1.41%	1.47%
Convert LIR to MIR	56.03%	55.03%	51.30%	44.99%	28.77%	26.92%	22.57%	18.77%	21.73%
Register Mapping	26.87%	26.64%	22.89%	29.40%	36.31%	32.41%	26.72%	25.23%	24.77%
Branch Optimizations	0.82%	0.87%	0.68%	1.14%	1.46%	1.13%	1.13%	0.80%	0.92%
Generate Machine Code	3.36%	3.27%	2.82%	3.37%	4.40%	3.71%	3.54%	2.88%	2.98%

Table 8: Jython compile time percentage breakdowns.

Lusearch Compilation Time %	AllOff FF	AllOff TF	AllOff TT	Select FF	Select TF	Select TT	O0	O1	O2
Convert Bytecodes to HIR	13.09%	10.45%	9.34%	18.41%	16.05%	11.93%	12.41%	18.00%	17.04%
CFG Transformations	2.39%	5.31%	2.66%	3.03%	2.37%	1.95%	3.98%	1.96%	1.10%
CFG Structural Analysis	1.12%	1.06%	1.30%	1.07%	1.25%	0.72%	0.58%	0.61%	0.46%
Simple Opts	1.16%	1.10%	1.19%	2.03%	1.25%	1.15%	0.80%	0.95%	0.64%
Escape Transformations	0.00%	0.00%	0.00%	2.06%	1.93%	1.85%	0.00%	1.08%	0.89%
Branch Optimizations	0.65%	0.61%	0.54%	1.00%	0.78%	0.62%	0.42%	0.50%	0.34%
Local CopyProp	0.00%	0.00%	0.00%	0.21%	0.20%	0.00%	0.30%	0.10%	0.10%
Local ConstantProp	0.00%	0.00%	0.00%	0.16%	0.13%	0.00%	0.11%	0.07%	0.07%
Local CSE	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	1.61%	1.13%	0.96%
Field Analysis	0.00%	0.00%	0.00%	0.00%	0.00%	0.09%	0.07%	0.05%	0.05%
Convert HIR to LIR	7.66%	6.97%	22.56%	7.71%	16.37%	28.97%	18.74%	20.40%	21.34%
Local CopyProp	0.00%	0.00%	0.00%	0.62%	0.21%	0.00%	0.27%	0.24%	0.38%
Local ConstantProp	0.00%	0.00%	0.00%	0.24%	0.47%	0.00%	0.20%	0.33%	0.49%
Local CSE	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	2.04%	1.23%	1.10%
Simple Opts	1.57%	1.43%	1.40%	2.85%	1.99%	1.41%	1.22%	1.12%	0.91%
Basic Block Frequency Estimation	1.36%	1.45%	3.68%	1.52%	1.59%	3.36%	1.27%	1.06%	1.15%
Code Reordering	0.00%	0.00%	0.00%	1.70%	1.27%	1.38%	0.10%	1.39%	0.96%
Branch Optimizations	1.10%	0.88%	2.60%	3.04%	1.55%	1.55%	1.04%	1.27%	1.20%
Convert LIR to MIR	26.44%	25.74%	18.08%	16.65%	18.47%	13.63%	28.19%	24.64%	28.02%
Register Mapping	37.45%	38.48%	30.88%	32.41%	29.53%	26.79%	22.70%	19.42%	19.52%
Branch Optimizations	1.68%	0.89%	0.81%	0.97%	1.06%	1.03%	0.72%	0.54%	0.52%
Generate Machine Code	4.32%	5.64%	4.97%	4.32%	3.53%	3.56%	3.25%	3.89%	2.77%

Table 9: Lusearch compile time percentage breakdowns.

Pmd Compilation Time %	AllOff FF	AllOff TF	AllOff TT	Select FF	Select TF	Select TT	O0	O1	O2
Convert Bytecodes to HIR	9.91%	9.81%	7.51%	9.83%	10.37%	7.88%	11.81%	10.13%	9.55%
CFG Transformations	4.05%	3.70%	3.04%	4.22%	3.90%	3.21%	3.52%	2.08%	2.09%
CFG Structural Analysis	2.37%	1.89%	1.49%	1.91%	2.01%	1.49%	1.17%	1.03%	0.82%
Simple Opts	1.97%	1.93%	2.77%	2.38%	1.99%	1.80%	1.28%	1.07%	1.01%
Escape Transformations	0.00%	0.00%	0.00%	0.00%	0.00%	1.95%	0.00%	1.53%	1.25%
Branch Optimizations	1.13%	1.11%	0.86%	1.12%	1.13%	0.90%	0.71%	0.58%	0.55%
Local CopyProp	0.00%	0.00%	0.00%	0.00%	0.41%	0.00%	0.38%	0.18%	0.17%
Local ConstantProp	0.00%	0.00%	0.00%	0.31%	0.00%	0.00%	0.37%	0.25%	0.23%
Local CSE	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.99%	0.57%	0.55%
Field Analysis	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.11%	0.08%	0.08%
Convert HIR to LIR	6.32%	6.79%	20.34%	6.49%	6.23%	19.48%	20.09%	26.67%	25.88%
Local CopyProp	0.00%	0.00%	0.00%	0.00%	0.65%	0.00%	0.45%	0.36%	0.52%
Local ConstantProp	0.00%	0.00%	0.00%	0.35%	0.00%	0.00%	0.32%	0.60%	0.66%
Local CSE	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.84%	0.78%	0.79%
Simple Opts	2.42%	2.21%	2.06%	2.50%	2.39%	3.77%	1.57%	1.56%	1.55%
Basic Block Frequency Estimation	4.11%	2.70%	2.31%	2.79%	3.79%	2.48%	1.76%	1.77%	1.79%
Code Reordering	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.15%	1.46%	1.43%
Branch Optimizations	1.57%	1.64%	1.64%	1.57%	1.58%	1.41%	1.37%	1.71%	1.53%
Convert LIR to MIR	23.66%	24.48%	18.74%	23.35%	22.37%	18.14%	24.46%	20.93%	24.80%
Register Mapping	34.25%	35.63%	32.80%	35.41%	34.75%	30.29%	23.92%	22.28%	20.87%
Branch Optimizations	2.19%	1.55%	1.24%	1.50%	2.37%	1.80%	0.95%	0.94%	0.77%
Generate Machine Code	6.06%	6.55%	5.19%	6.27%	6.08%	5.41%	3.78%	3.41%	3.11%

Table 10: Pmd compile time percentage breakdowns.

Xalan Compilation Time %	AllOff FF	AllOff TF	AllOff TT	Select FF	Select TF	Select TT	O0	O1	O2
Convert Bytecodes to HIR	10.09%	9.82%	7.51%	10.44%	10.26%	8.18%	10.04%	9.48%	8.70%
CFG Transformations	3.41%	3.23%	2.38%	3.44%	3.50%	3.84%	2.92%	1.92%	1.82%
CFG Structural Analysis	1.77%	1.60%	1.74%	3.21%	1.63%	1.19%	0.99%	0.70%	0.68%
Simple Opts	2.41%	1.86%	1.41%	1.95%	1.93%	1.40%	1.27%	1.45%	0.98%
Escape Transformations	0.00%	0.00%	0.00%	0.00%	2.30%	0.00%	0.00%	1.45%	1.27%
Branch Optimizations	1.01%	0.98%	0.73%	1.06%	1.07%	0.78%	0.74%	0.55%	0.53%
Local CopyProp	0.00%	0.00%	0.00%	0.00%	0.00%	0.34%	0.36%	0.28%	0.35%
Local ConstantProp	0.00%	0.00%	0.00%	0.30%	0.00%	0.00%	0.21%	0.17%	0.16%
Local CSE	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.95%	0.64%	0.50%
Field Analysis	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.11%	0.08%	0.08%
Convert HIR to LIR	6.06%	7.04%	22.71%	5.67%	6.24%	21.31%	22.59%	26.82%	26.60%
Local CopyProp	0.00%	0.00%	0.00%	0.00%	0.00%	0.70%	0.47%	0.39%	0.48%
Local ConstantProp	0.00%	0.00%	0.00%	0.34%	0.00%	0.00%	0.38%	0.83%	0.70%
Local CSE	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.81%	0.84%	0.76%
Simple Opts	3.74%	2.19%	2.03%	2.22%	2.43%	1.88%	1.66%	1.47%	1.45%
Basic Block Frequency Estimation	2.48%	2.24%	2.17%	2.11%	2.42%	2.21%	1.85%	1.83%	1.72%
Code Reordering	0.00%	0.00%	0.00%	0.20%	0.00%	0.17%	0.14%	1.41%	1.32%
Branch Optimizations	1.42%	1.38%	1.26%	1.55%	3.43%	1.34%	1.41%	2.24%	1.56%
Convert LIR to MIR	23.34%	24.64%	18.66%	22.78%	20.43%	17.50%	23.15%	19.21%	23.42%
Register Mapping	35.54%	36.42%	32.36%	36.38%	35.60%	31.71%	24.60%	23.53%	22.55%
Branch Optimizations	1.65%	1.58%	1.32%	1.40%	1.54%	1.12%	0.90%	0.90%	0.82%
Generate Machine Code	7.07%	7.02%	5.72%	6.95%	7.23%	6.32%	4.46%	3.79%	3.58%

Table 11: Xalan compile time percentage breakdowns.

7 Conclusion

We ran this experiment in the hopes of determining an optimal subset of compiler optimizations that yields a lower overall runtime. Our results show that by specifically choosing "good" optimizations that are believed to decrease overall runtime, the runtimes of the benchmarks outperform the runtimes when the default O0, O1, or O2 optimization levels are selected. The tests with our chosen optimizations outperform the tests with very few optimizations in many cases.

The method is not perfect. The fact that the minimal compiler option tests outperform the selected optimization tests on some of the benchmarks shows that the foldover PB experiment alone is not a guaranteed way to find the best optimizations to perform for a specific application. It is ideal to select an optimal set of optimizations. However, finding this combination of optimizations may require running a significant number of additional tests. An alternative option is to group the optimizations by type and vary all combinations of these groups, turning all options in a group either on or off. This is more feasible than varying all optimizations individually.

8 References

[1] R. Plackett and J. Burman, "The Design of Optimum Multifactorial Experiments", Biometrika, Vol. 33, Issue 4, June 1956, Pages 305-325.

[2] Yi, J.J.; Lilja, D.J.; Hawkins, D.M.; , "A statistically rigorous approach for improving simulation methodology," High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on , vol., no., pp. 281- 291, 8-12 Feb. 2003 doi: 10.1109/HPCA.2003.1183546 URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1183546&isnumber=26557

[3] D. C. Montgomery, "Design and Analysis of Experiments", Third Edition, Wiley 1991.

[4] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. 2000. The Jalapeño virtual machine. IBM Syst. J. 39, 1 (January 2000), 211-238. DOI=10.1147/sj.391.0211 http://dx.doi.org/10.1147/sj.391.0211

[5] Blackburn, S. M., Garner, R., Hoffman, C., Khan, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J. E. B., Phansalkar, A., Stefanovic, D., VanDrunen, T., von Dincklage, D., and Wiedermann, B. "The DaCapo Benchmarks: Java Benchmarking Development and Analysis", OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications, (Portland, OR, USA, October 22-26, 2006)

[6] Stephen J. Fink and Feng Qian. 2003. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization (CGO '03). IEEE Computer Society, Washington, DC, USA, 241-252.

[7] Karl Pettis and Robert C. Hansen. 1990. Profile guided code positioning. SIGPLAN Not. 25, 6 (June 1990), 16-27. DOI=10.1145/93548.93550 http://doi.acm.org/10.1145/93548.93550

[8] Sanjay Ghemawat, Keith H. Randall, and Daniel J. Scales. 2000. Field analysis: getting useful and low-cost interprocedural information. In Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation (PLDI '00). ACM, New York, NY, USA, 334-344. DOI=10.1145/349299.349343 http://doi.acm.org/10.1145/349299.349343