JIT Technology with C/C++: Feedback-Directed Dynamic Recompilation for Statically Compiled Languages

DORIT NUZMAN, REVITAL ERES, SERGEI DYSHEL, MARCEL ZALMANOVICI,
and JOSE CASTANOS, IBM Research

The growing gap between the advanced capabilities of static compilers as reflected in benchmarking results and the actual performance that users experience in real-life scenarios makes client-side dynamic optimization technologies imperative to the domain of static languages. Dynamic optimization of software distributed in the form of a platform-agnostic Intermediate-Representation (IR) has been very successful in the domain of managed languages, greatly improving upon interpreted code, especially when online profiling is used. However, can such feedback-directed IR-based dynamic code generation be viable in the domain of statically compiled, rather than interpreted, languages? We show that fat binaries, which combine the IR together with the statically compiled executable, can provide a practical solution for software vendors, allowing their software to be dynamically optimized without the limitation of binary-level approaches, which lack the high-level IR of the program, and without the warm-up costs associated with the IR-only software distribution approach. We describe and evaluate the fat-binary-based runtime compilation approach using SPECint2006, demonstrating that the overheads it incurs are low enough to be successfully surmounted by dynamic optimization. Building on Java JIT technologies, our results already improve upon common real-world usage scenarios, including very small workloads.

Categories and Subject Descriptors: D.3.4 [Programming Languages]: Processors—*Compilers, Optimization, Run-time environments*; D.3.2 [Programming Languages]: Language Classifications—*Statically compiled languages*

General Terms: Design, Performance

Additional Key Words and Phrases: Dynamic optimization, just-in-time compilation

ACM Reference Format:

Nuzman, D., Eres, R., Dyshel, S., Zalmanovici, M., and Castanos, J. 2013. JIT technology with C/C++: Feedback-directed dynamic recompilation for statically compiled languages. *ACM Trans. Architect. Code Optim.* 10, 4, Article 59 (December 2013), 25 pages.
DOI: <http://dx.doi.org/10.1145/2555289.2555315>

1. INTRODUCTION

Static compilers provide highly sophisticated program transformations for performance, driven by extremely deep and accurate analysis, as well as advanced feedback-directed techniques. These compiler optimizations are always used by computer system vendors when publishing benchmark numbers, yielding tens of percents of performance improvement. On top of that, sophisticated users, especially those who deal with High-Performance Computing (HPC), further optimize their code by hand, beyond what the compiler can provide. In these scenarios, the capabilities of the most up-to-date

Authors' addresses: D. Nuzman, R. Eres, S. Dyshel, and M. Zalmanovici, IBM Haifa Research Lab, Haifa University Campus, Haifa, Israel; J. Castanos, IBM Thomas J Watson Research Center - Yorktown Heights, New York.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481 or permissions@acm.org.

© 2013 ACM 1544-3566/2013/12-ART59 \$15.00

DOI: <http://dx.doi.org/10.1145/2555289.2555315>

hardware can be unleashed, providing the highest possible performance. However, this is not the most common usage scenario. More commonly, developers, users building open-source software, and Independent Software Vendors (ISVs) building software in production do not enable the highest optimization levels, nor do they use hardware-specific optimizations or Feedback-Directed Optimizations (FDOs). The implications of such highly optimizing build processes on software production costs are such that only a single-step build with moderate optimizations is actually used in practice. The outcome is an inherent gap between the potential performance that compilers can provide and the performance that users actually experience.

Dynamic optimization provides an opportunity to gain back this lost performance benefit of unused optimizations as part of the computer system, rather than as part of the software production process. A dynamic compilation environment can exploit the features of the exact CPU model, thereby solving the HW-specific optimization issue and eliminating the current large time gap between when a processor is introduced and when it gets used efficiently. It also solves the FDO problem by essentially performing it online, transparently. It can detect the most relevant execution paths for the current invocation and guide aggressive code specialization, whereas static techniques, even using offline FDO, need to be more conservative, using fit-for-all heuristics, often based on profiles averaged over several possible inputs. In addition, dynamic recompilation can optimize programs repeatedly, continuously adapting the optimization strategy to the program's changing behavior [Arnold et al. 2011].

1.1. Why Now

More than ever, recent trends motivate a re-examination of the application of dynamic compilation technologies to statically compiled languages, for several reasons:

- The growth of single thread performance has slowed down. Hardware alone no longer provides the performance boost at rates seen in the past. This puts a greater burden on software to drive performance improvement, using all the means possible.
- The vast move to cloud and virtual environments results in increased abstraction of performance-critical information from the static compilation environment. Only upon runtime deployment do the actual physical resources become known, and these can change during program execution due to workload migration and consolidation considerations, requiring continuous online adaptation of programs.
- The proliferation of multicore designs, and the limited ability of programs (and programmers) to take advantage of all the abundant available parallelism, motivates the use of some of these resources for recompilation in parallel to program execution.
- C and C++ are still among the most dominant programming languages used in newly compiled codes.¹ Leveraging technologies that were originally developed for dynamically compiled languages, for the benefit of C/C++ programs, is therefore extremely relevant.

1.2. Existing Solutions

The main existing solutions for client-side dynamic code generation for static languages are based on either (1) an **IR-only** based approach, in which a common platform-neutral IR such as LLVM [Lattner and Adve 2004] or CLI [Campanoni et al. 2008] is shipped and interpreted and/or Just-In-Time (JIT) compiled on the target platform, or (2) a **binary-only** approach, in which a native binary is shipped and dynamically optimized. The first approach, while promoting portability, incurs high startup costs and requires that vendor software be ported to the common-IR tool chain. The second

¹TIOBE Programming Community Index, <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>.

approach, while capable of transparently optimizing nonrecompilable code, is limited in its optimization to the low-level information available in machine-level code.

A major challenge of profile-driven dynamic recompilation in either approach is the various costs it incurs, due to monitoring/instrumentation that interferes with/slow down program execution. In the domain of interpreted languages, where JIT technology has traditionally been applied, dynamic FDO has been successfully used, as driven by the interpreter, to boost performance compared to interpreted execution. In the context of static languages, however, while applied with some success under the binary-only approach [Bala et al. 2000], runtime profile-guided recompilation is rarely used in IR-only scenarios. Even when it is used, in the general-purpose domain it has had very limited success, confined to long-running programs and shared libraries, the only scenarios in which the costs of dynamic optimization could be amortized [Kistler and Franz 2003]. Indeed, since compilers operate on an intermediate representation at a much higher level than native machine code (as opposed to binary optimizers), generating machine code requires many costly compilation passes, and the associated overheads may offset any code optimization improvement.

1.3. The Fat Binary Solution

In light of these limitations, we propose to revisit a third, hybrid approach, based on the Continuous Program Optimizer (CPO) idea of Stoodley et al. [2006]. We present a selective dynamic recompilation system for static languages, based on *fat binaries*. A preceding static compilation stage builds the intermediate representation (IR) and provides it along with the initial native executable. As the program starts executing, that IR is used to selectively recompile methods at runtime. Other than the initial IR, everything is done online, in parallel to program execution, including profile collection and the decisions concerning what, when, and how to compile. Optimized versions are constructed on the fly in the background and substituted in place. To achieve that, we adapted a Java JIT compiler to the C/C++ domain, and we reuse its dynamic compilation capabilities, driven by a new custom runtime engine that we built. The fat binary approach combines the advantages of a high-level IR amenable for aggressive optimization with a native binary that can be executed in parallel, thereby avoiding system warm-up costs associated with generating native code from the IR. Most importantly, it incurs almost no costs from the software vendor, requiring only a simple IR dump while keeping the same software build and distribution process. As such, it is both practical and has the prospects to provide high-quality code.

1.4. Our Work

The main goal of this work is to provide an initial insight to these performance prospects. Our implementation does not yet exploit the full potential of dynamic FDO but can already be used to bring important insights about the feasibility of the approach. Our main focus, and main achievement, is the ability to demonstrate an execution environment with overheads that are low enough to already provide performance speed-ups to moderately optimized programs—the primary target scenario of dynamic optimization. Ten out of 12 benchmarks are improved, with speed-ups up to 19% and degradations less than 2%, constituting an overall 8% average improvement, all overheads included. Furthermore, the overheads are also low enough to largely avoid a negative performance effect even in scenarios of highly optimized binaries. These results are well scalable with increasing benchmark code size, and degradations are generally very limited even for tiny inputs, making this technology relevant for a large range of applications and workloads. In the domain of dynamic recompilation of static languages, these are new results. Previous approaches driven by online profiling were unable to avoid huge degradations compared to the statically compiled code in the

case of small workloads [Kistler and Franz 2003]. This article makes the following contributions:

- We describe a robust runtime recompilation system for statically compiled languages based on fat binaries and driven by dynamic profiling while discussing its design and implementation, as well as advantages compared to other alternatives.
- We provide detailed evaluation of the quality of the profiling information we gather, revisiting known Java techniques in the context of C/C++ programs.
- We provide a thorough evaluation of our compilation approach on the full SPECint2006 suite, also compared to alternative solutions, demonstrating its viability in terms of the overheads it incurs.

Specifically, we present the following insights:

- An existing unmodified IR of a static compiler and a Java JIT compiler with minimal repurposing to C/C++, with the same profiling and instrumentation mechanisms and thresholds, can be leveraged almost as is to support a feedback-directed fat-binary-based dynamic optimization approach for static languages.
- Profiling overheads do not pose a significant limiting factor, even though the system uses recompilation-based instrumentation and does not rely on hardware support for lightweight profiling or on interpreter-based profiling. System start-up costs, the only overhead that grows linearly with program size, allow supporting workloads as short as 3 to 52 seconds (from the smallest to the largest of SPECs benchmarks in terms of fat binary size) without incurring a slowdown.
- Cross-module inlining is key to unleashing the power of FDO in SPECint2006. On the other hand, longer sampling periods do not guarantee better FDO.
- In terms of performance, the inability to run native code in parallel to dynamic code generation is the key limiting factor behind the IR-only and binary-only approaches (represented by LLVM/CLR and Dynamo/DynamoRio, respectively). While all three systems are challenged by increased program code footprint size, the fat-binary-based approach will incur fewer slowdowns than its counterparts, which interpret the program while populating their code cache with optimized native code. At the same time, our system also successfully copes with small workloads, including SPEC's *gcc*, which has challenged the other approaches.

The rest of the article is organized as follows: Section 2 presents a real-world motivating scenario and discusses the premise of the fat binary solution compared to other alternatives. Section 3 describes our runtime compilation framework and online instrumentation-based profiling mechanisms. Next we provide extensive experimental evaluation of our implementation, with the experimental methodology in Section 4, the main results in Section 5, and a discussion of additional evaluation perspectives in Section 6. Section 7 discusses related work, and Section 8 concludes.

2. A CASE FOR FAT-BINARY-BASED DYNAMIC OPTIMIZATION FOR STATIC LANGUAGES

Here's a typical scenario faced by computer system vendors (e.g., IBM): a customer who uses a third-party application owned by some ISV. The customer has performance issues with that software on IBM's platform (e.g., a Power780 server); however, the ISV refuses to increase the platform-target level to something more modern. The ISV is also uninterested in increasing the optimization level of the Power version of the binary and is unwilling to apply FDO as it doesn't think training for one customer's interests warrants the cost/effort on its behalf. There is very little IBM can do here to help the customer boost the performance of that software on its machine.

2.1. The Static Optimization Usage Problem

Why is it so costly or difficult for ISVs to highly optimize their software? There are several possible reasons:

- High optimization levels beyond *-O3* require several compilation hours to obtain a SPEC run and in general are much slower to compile than the *-O2/-O3* optimization levels due to the aggressive interprocedural and link-time optimizations that they employ. They are also more prone to fail than the more basic optimization levels, and beyond the *-O2* level, they often do not guarantee to preserve strict semantics.
- Hardware-specific optimizations imply a specialized production process for each specific software/platform combination on which the software is shipped. This means huge development and testing costs and a prolonged production cycle. As a result, ISVs prefer to compile for the lowest-common-denominator platform (e.g., a generic Power machine) and rarely take advantage of HW-specific optimizations and features introduced in new processors.
- Feedback-Directed Optimizations (FDOs) require finding a representative input and performing extra compilation and profile gathering stages. But such representative input is not always available. Software products like DB2, SAP, and compilers inherently do not have a single representative workload. These products are also typically huge with an already complicated build process. Further complicating it with additional profiling runs incurs too much burden. FDO in these scenarios, if applied in actual distribution, is almost never done with profiles specific to a single customer, but rather with a heterogeneous collection of inputs from various customers. More commonly, FDO is used for these applications only for benchmarking. The GCC compiler, for example, is included as a benchmark in SPEC where it can be optimized with FDO, whereas the actual build process of GCC in real life does not include FDO, not by the compiler developers themselves (who are sophisticated users who care about performance), and certainly not by ISVs who distribute it as a binary.

2.2. Solutions and Tradeoffs

The inherent contradiction ISVs face today, between the need to shorten their production cycle and minimize costs on one hand and the desire to provide high-performing software on the other hand, could be resolved using client-side dynamic optimization. ISVs could focus on developing, building, testing, and distributing their software, leaving performance tuning to the computer system vendors, who would in turn be able to control the optimization level best suitable to exploit the full potential of their machines. In this scenario, users are also more accepting of aggressive optimizations they otherwise avoid when compiling themselves, offline. When optimizations are applied by the ISV, the responsibility for supporting failing code falls on them, as opposed to when optimizations are done on the system vendor's side (by the JIT).

With this ISV scenario in mind, what can computer system vendors do to allow dynamic optimization of third-party software on their platforms? Theoretically, they could choose one of two well-known solutions: (1) Convince the ISV to switch to some existing standard common IR (such as CLI or LLVM) to be optimized on the system vendor side by a JIT compiler (the system vendor would in turn invest in improving that JIT for their platform), or (2) not change anything on the ISV side, but just continue to produce the same platform-specific binaries using the same compiler (while the system vendor would develop a native binary optimizer for its platform). Another alternative, advocated in this article, is a hybrid of the two previous solutions: (3) Convince the ISV to allow the compiler (the same compiler already employed in its software production process) to embed its IR into the native binary. The system vendor would develop a fat binary optimizer along with a JIT for this specific IR.

This section compares these three approaches focusing on two main aspects: (a) the practicality of the solution in terms of what is required of the ISVs and (b) the prospects for performance improvement upon native binaries that ISVs currently generate.

2.2.1. IR-only Approach. This approach is driven mainly by portability: software is provided in a platform-neutral representation, which is executed on the actual hardware via a Virtual Machine (VM). Inherent in this approach is **startup cost**, associated with the need to interpret (and/or JIT compile) the IR, as opposed to the fat binary approach, which can kick off native execution immediately. There is a lot of evidence for the severity of this issue in the Java literature [Gartley et al. 2013; Kulkarni et al. 2007; Arnold et al. 2001], attesting to the fact that compiling all the functions in the application, or even just all the functions that get executed, can cause a dramatic performance loss, both in the case of short-running programs and in the case of startup times of large middle-ware applications. The advantage of having a native executable along with the IR is clear in this context.

In terms of the requirements from the ISV, the cost of adopting the IR-only approach depends on which IR is chosen. If the goal is to leverage existing open JITs of standard common IRs such as LLVM or CLI while also gaining the benefits of portability, the ISVs would have to be willing to port their entire software to compilers that can generate these IRs, a major and costly technology shift. If the goal is minimal cost transition, then a more suitable approach would be to ship the IR produced by the compiler already in use (that compiler may need to be enhanced with the ability to dump the IR, a minor change with almost no implications on the production process). In this case, however, it is not clear what the advantage is of shipping only the IR over the fat binary approach: the software will be penalized with startup cost while gaining no benefits compared to the fat binary approach.

2.2.2. Binary-only Approach. Dynamic optimization has also been applied to native binaries with works such as DAISY [Ebcioglu and Altman 1997], Dynamo [Bala et al. 2000], ADORE [Lu et al. 2004], and more recently [Böhm et al. 2011]. The native binary optimization approach, especially behind the Dynamo project, shares the same motivation as this work: deliver client-side performance transparently, requiring no user/ISV involvement. However, it differs from the fat binary recompilation approach in the level of semantic information available to it. Dynamic binary optimizers have to deal with the challenges of optimizing machine-level code, which lacks high-level semantic information about the program, making many things harder or even impossible to do. Issues that our compiler can easily cope with, such as detecting nontrivial induction-variable constructs and dealing with exceptions and synchronous signal handling, can be very challenging at the binary level, not to mention advanced analyses such as aliasing. From a qualitative perspective, IR-based dynamic recompilation therefore has far more promising prospects for performance impact than that of binary optimizers.

2.2.3. Fat Binary Approach. In light of this, the main qualitative tradeoffs between the fat binary approach and the alternative are the following:

Performance versus Transparency: Fat binaries trade off the transparency of the fully native binary approach for the performance potential of more aggressive IR-based optimization. The implication is that this approach cannot optimize legacy, or otherwise nonrecompilable code, but has the prospects to provide superior performance advantages compared to the native binary approach.

Performance versus Portability: Fat binaries trade off the portability of the fully IR-based approach for the performance benefit of avoiding startup costs associated with executing the IR. In fact, native binary-based optimizers are often also based on interpretation of the native code while populating a code cache with optimized code

Fig. 1. **The static and dynamic compilation flows.** The static flow (a) is used as a comparison point for experimental evaluation.

traces [Bala et al. 2000], and as such also incur a warm-up cost that has severe implications on software response time (recent work [Mahlke et al. 2013] on prepopulating the code cache for this very reason can attest to the severity of this issue).

Performance versus Security: The IR-based approaches (both IR only and fat binary) require that ISVs be willing to ship the IR, thereby potentially revealing sensitive information about their code. In our experience, even relatively conservative vendors, such as those from the banking domain, accept this solution. The various banking ISVs who use a Java-based system can already attest to that. Extremely conservative ISVs have the option to provide the IR only for the less sensitive parts of their software, in which case only these parts will be amenable for aggressive IR-based reoptimization. The rest of their software can be optimized only using the more limited purely binary-based optimization. We believe that the expected inability of hardware to continue to provide the same rate of single-thread performance boost will strongly motivate ISVs to provide the dynamic optimization system with as much of their software IR as they can, because the performance of the rest of their software will be limited by what the purely native binary optimizers could provide.

Ease of Adoption by ISVs: A primary requirement for the adoption of any of the approaches is for it to incur close to zero cost on the software production side. Creating the fat binary as we propose is cheap and safe: it only involves adding one extra dump and does nothing more than preserve some information that is typically thrown away at the end of the compilation stage. No additional analyses are applied, and no additional information is generated beyond what the compiler already uses. Indeed, other fat-binary-based approaches that rely on significant extra analysis and compile time and/or memory requirements (such as Yardimci and Franz [2009]) are less likely to be adopted in practice. Lastly, whatever is running in current systems will still run under the fat binary optimizer, including legacy functions from old libraries. These would not get optimized, but the system will not break.

3. THE RUNTIME OPTIMIZATION SYSTEM

3.1. Overview

Our runtime optimization system is based on a pre-existing IBM in-house *split-compilation* tool chain, in which part of the compilation is deferred to runtime. It includes a static compiler, a dynamic compiler, and a custom virtual machine, as illustrated in Figure 1(b). The static compiler consists of a machine-independent,

high-level optimizer, followed by a special backend that can create a fat binary, a native executable accompanied with a *split IR*, and an intermediate representation of the program that provides extra semantic information that can be used for recompilation at runtime.² The dynamic compiler is based on an existing Java JIT compiler repurposed to support statically compiled languages. The static compilation stage applies the major optimizations that require expensive analysis without runtime feedback, whereas the dynamic compiler performs simpler, less expensive, intramethod optimizations, as well as inlining within a single compilation unit. The runtime engine orchestrates the monitoring and selective recompilation of the fat binary at runtime simultaneously with the execution of the program. Method selection and optimization are guided by profiling information that is gathered at runtime, using a combination of a timer-based sampling profiler and an instrumentation-based profiler.

Our current implementation is based on IBM's compilers, but the fat binary approach is general and applicable to other tool chains. For example, the LLVM compiler can generate a native executable and a serialized IR (LLVM bitcode³), as can GCC, which can store its GIMPLE IR for the purpose of link-time optimization (LTO⁴), or alternatively generate CLI-compliant byte code with the gcc4cli CLI backend [Costa et al. 2007].

3.2. The Dynamic JIT Compiler

Our dynamic compiler was developed on top of the Testarossa JIT compiler [Sundaresan et al. 2006], a mature multiplatform Java compiler. It supports a variety of optimization levels, which trade optimization complexity against speed. The JIT's optimizer and code generator are generally reused, whereas the IR generator is new, translating the split IR into the JIT compiler's IR.

The JIT compiler is attached to the runtime engine as a shared library and is invoked at the granularity of methods. JIT compilation is triggered when the method reaches a predefined "hotness" threshold and is currently invoked twice for each method that is selected for recompilation. The first dynamic recompilation instruments the selected method. The second recompilation optimizes the method based on the profile gathered by the instrumented code (see Figure 2(a)). Currently, methods are selected for recompilation only once, but there is no inherent limitation on that. Additional recompilation cycles may be useful to consider in the future (e.g., where adaptation of the optimization to the changing program behavior is required).

The profile information we focused on is Basic Block (BB) frequencies, as described in Section 3.4. These are used to improve code layout (in terms of locality and number of taken branches) by optimizations such as BB reordering and BB splitting, and also used to guide the heuristics of other optimizations such as inlining, register allocation, partial redundancy elimination, and loop unrolling.

The choice to base our dynamic recompiler on a Java JIT compiler is natural, since in both scenarios the compilation time is part of the execution time, a setting that requires different considerations than a static compiler and makes some aggressive algorithms (such as coloring register allocator, sophisticated dependence analysis, heavy loop-nest optimizations) unaffordable. The profiling infrastructure is also different, having to use carefully tuned instrumentation techniques with low dynamic overhead versus not worrying about overhead at all when collecting profile offline. The drawback of this choice is that much adaptation is still required. Since the compiler was originally

²The split IR is the regular IR of XLC. As it is IBM proprietary, we cannot disclose any details about it; however, no modifications were made to this IR by this work to facilitate better runtime compilation.

³<http://llvm.org/docs/BitCodeFormat.html>.

⁴<http://gcc.gnu.org/onlinedocs/gccint/LTO.html>.

Fig. 2. (a) The stages of runtime profiling and compilation. (b) Execution modes for experimental evaluation. Both (a) and (b) are shown for a single recompiled method (the times “ t_1 ” to “ t_8 ” are per recompiled method). **dyn-base**: loads the statically compiled executable, builds the mapping between the object code and the IR, and then proceeds to run the executable as is, with no further steps (no sampling, no initiation of the recompilation thread, etc.). **dyn-no-recomp**: like the aforementioned mode, and in addition the runtime activates the sampling interrupts for method hotness and the recompilation thread; however, the method hotness threshold is set so high that no method ever gets recompiled. **dyn-instr**: like the aforementioned, but the method hotness threshold is set to the default value, to allow recompilation. Once the second recompilation stage completes, we drop the optimized version and switch back to the original version of the method. **dyn**: like the aforementioned, but we remain with the optimized version after the second recompilation completes. This is the normal dynamic execution mode, which includes the entire dynamic execution time with all overheads included.

designed for Java, although it can compile any language that can be converted to our IR, it still contains many “javaisms” in terms of the language features it has specialized in optimizing. For example, in the C/C++ fat binary context, Testarossa is invoked following the static optimizer, which is very probable to unroll and inline and otherwise increase the code size. As a result, Testarossa is faced with code that is much larger than what it is ever faced with in the Java interpreter-based context. We are still in the process of repurposing the compiler to its new usage scenario.

3.3. The Runtime Manager

While our framework borrows the JIT compiler from the Java compilation infrastructure, our runtime environment is new and is not based on the JVM. For minimal interference with the program execution, the runtime consists of two separate threads: one for program execution and the other for recompilation, as shown in Figure 2(a).

Runtime environment startup. The runtime engine is implemented as an executable that, after initialization, transfers the control to the program. At startup, a custom loader is used to load the program onto the heap area and map function and data symbols between the split IR and the native binary. The runtime engine then initiates the compiler thread, turns on sampling, and calls the program’s *main*.

Method selection. Hot methods are identified based on event-based profiling. The current heuristic for method selection is a simplified version of the JVM’s selection mechanism. First, a method has to accumulate $T1 = 90$ samples to be considered

a candidate for selection, thus filtering methods that are associated with program startup. After that, a candidate method has to accumulate its next $T2 = 10$ samples quickly enough (within the next $X = 100$ global samples) to be elected.

Recompilation management. Once a method is determined to be hot, it is put in the recompilation queue, along with an indication of which version should be generated: the instrumented or the optimized one. At that point, the recompilation thread, which is waiting idle as long as the queue is empty, is signaled and the compiler is invoked. Recompiled versions of methods are stored in a software-managed contiguous memory area called *Code Cache*. We haven't implemented any special code-caching-related mechanisms so far. Our experiments so far had only modest space requirements for recompilation (11–144KB per benchmark for SPECint2006). In addition, since in our approach nonrecompiled code is already available in native form and executes in parallel to recompilation, our system doesn't rely on smart code caching as heavily as other dynamic compilation systems that try to avoid slow, interpreted code execution [Hazelwood and Smith 2004; Bouakaz et al. 2011].

For control redirection from the original method to the newly compiled one, a patching mechanism is used. The mechanism creates a trampoline at the beginning of the original function body. The trampoline directs execution to a new stub code, which retrieves the call instruction (from the link register), and in case it is a direct call, the stub code changes its target to the address of the new recompiled version of the function and jumps to it. This way, for direct function calls, only the first call from a call site incurs the redirection overhead, and all subsequent calls from the call site will directly jump to the new address. If, however, it is an indirect call, the stub code merely jumps to the new recompiled code without changing the original call site.

3.4. The Runtime Profiling Mechanisms

We use runtime feedback as the primary means to decide what, and how, to recompile in our framework. Two main mechanisms are used:

3.4.1. Timer Interrupts. We use event-based profiling to guide the decision concerning which methods to recompile. Timer interrupts are used to sample the program counter at a rate of 10ms (the maximum rate supported for nonroot processes on AIX). In our experience so far, the overhead from sampling is negligible (it is presented in Figure 5, included within the “sampling and synchronization” overhead). Upon each sample, the counter of the respective containing method is incremented. The method is identified using a segment tree search of all registered method boundaries (as prepared during startup time; the overhead associated with this preparation is included within the “loading and mapping” overhead of Figure 5).

3.4.2. Instrumentation. We use instrumentation-based profiling to guide the decision concerning how to optimize the code. It is currently applied once for each selected method. While much more costly than event-based sampling, compiler-inserted instrumentation makes it much easier to match sampled addresses back to the compiler IR/CFG than with event-based sampling. The first compilation stage inserts basic block counters, which are updated using a relatively costly {load,increment,store} sequence to track how many times a basic block is visited. During the second, optimizing, recompilation phase, the compiler reads the basic block profile and attempts to infer edge frequencies.

We experimented with two instrumentation techniques that trade off accuracy, temporal coverage, and overhead. The first, **exhaustive** instrumentation, records all the occurrences that a basic block was visited during the profile-gathering period (which lasts 10ms, the duration between two consecutive time interrupts). The second,

cloning-based instrumentation [Arnold and Ryder 2001], tries to lower the overhead of exhaustive instrumentation by introducing two versions of the code. The first, *checking* version, is the original version along with code to transfer execution to the second, *cloned* version, which contains basic block counters. The slower, cloned version is executed once per N invocations of the faster, checking code, up to a total of M times (before switching back to executing solely the checking version till the end of the instrumentation phase). The overall “budget” (total count value) that the counters can accumulate is therefore exactly M ($M = 10K$ in our configuration, three orders of magnitude less than the value the counters reach in the exhaustive scheme). This way, a sampling instrumentation is achieved, incurring a carefully controlled overhead.

In VMs that rely on interpretation, an alternative to event-based sampling and JIT-based instrumentation is to use interpreter-based profiling [Gartley et al. 2013]. The interpreter controls the execution of the program and can profile it without the need for an additional JIT compilation stage. Interpreter-based profiling is thus available immediately upon initial translation, thereby incurring less overhead than JIT-based instrumentation. Our system does not rely on interpretation and has to use one of the other profiling techniques described previously.

4. EXPERIMENTAL METHODOLOGY

We performed our experiments on a 3.5GHz 64GB RAM Power7 processor running AIX6.1. The dynamic execution time includes all overheads, including startup of the runtime environment (as opposed to some experimental setups that allow sufficiently long warmup time until steady state is reached, assuming long-running programs). We use two primary optimization modes:

(1) **The BaseOpt** optimization mode emulates the real-world scenario, which we set as our primary target goal, in which users (and ISVs) prefer to compile code using a single small set of generic moderate optimization switches and a single-step make process. In this mode, we enable the `-O2` option for the high-level static XL-C optimizer and *warm* optimization level for Testarossa static backend. The balance that the `-O2` optimization level provides, between code quality, compilation speed, debuggability, and guarantee of strict semantics (which `-O3` can break), makes it a popular choice, for both users and ISVs, and a representative of common “out-of-the-box” scenarios.

(2) **The HighOpt** optimization mode is used for a synthetic evaluation of the overheads and limits of our system. Here we enable the options `-O3 -qhot` for the high-level XL-C optimizer, and the highest optimization level of Testarossa: *scorching*. `-qhot` enables more aggressive optimization of loops, and *scorching* is the optimization level that was designed for the very hottest methods selected for the most aggressive optimization. This scenario evaluates the dynamic execution environment in the challenging scenario in which the same aggressive optimization level that is used by the dynamic optimizer had already been applied by the static compiler, but without profiling information. This is a higher optimization level than “average” and is meant to “stress test” our runtime system and demonstrate its feasibility and scalability.

We create the statically compiled executables using the high-level optimizer of XL-C followed by a backend compiler that was extended to support the creation of fat binaries, namely, Testarossa (TR) in its static mode (the default backend of XL-C, TOBEY, is not yet fat binary aware). The same statically compiled executable is used both as the input to the dynamic execution system and as the point of comparison to evaluate the dynamic system. The performance gap between the two static backends is shown in Table I column 8. This gap, currently 10% and gradually closing, is largely because TOBEY has been tuned for C/C++ on Power for a longer period than Testarossa. Note, however, that the dynamic execution time cannot be compared to

Table I. SpecInt2006 Benchmark Suite Statistics

1. Benchmark (+ LOC)	2. # inputs ref/train	3. opt. potential stat vs. stat	4. TH = 10: #funcs, impact dyn vs. stat	5. TH = 2: #funcs, impact dyn vs. stat	6. TH = 0 #funcs, impact dyn vs. stat	7. size of fatbin (MB) (bin:IR) ratio	8. default XLC vs. XLC+TR stat vs. stat
<i>SQLite</i>	N/A	-16%	12, -13%	40, -15%	106, -16%	10 (2:1)	0.96
<i>astar</i> (6K)	2/2	-8%	6, -9%	7, -10%	340, -8.4%	1 (1:1)	0.82
<i>bzip2</i> (8K)	6/3	-2%	8, -2%	10, -1%	34, +0.5%	1 (1:2)	0.92
<i>gcc</i> (521K)	9/1	N/A	6, +2%	9, +2%	N/A	42 (1:2)	0.87
<i>gobmk</i> (197K)	5/8	-5%	3, +1%	20, -2%	166, -1.0%	39 (1:4)	0.92
<i>h264ref</i> (51K)	3/1	-11%	5, -8%	12, -10%	104, -9.4%	9 (1:2)	0.99
<i>hmmer</i> (34K)	3/1	-12%	2, -9%	2, -10%	8, -10%	4 (1:3)	0.94
<i>libquant</i> (4K)	1/1	-17%	6, -19%	7, -19%	27, -20%	1 (1:1)	0.85
<i>mcf</i> (3K)	1/1	-1%	6, -1%	8, +1%	26, -0.8%	0.3 (1:2)	0.97
<i>omnetpp</i> (48K)	1/1	-2%	6, -1%	25, -2%	N/A	19 (1:2)	0.94
<i>perl</i> (170K)	3/5	-20%	3, -11%	9, -10%	131, -11%	14 (1:2)	0.93
<i>sjeng</i> (14K)	1/1	+2%	9, -1%	22, -3%	64, -2%	1.5 (1:2)	0.86
<i>xalan</i> (554K)	1/1	-19%	6, -9%	14, -10%	305, -8%	175 (1:3)	0.78
average		-8.6%	-7.2%	-8.2%	-7%		0.90

Columns: **col1**: Benchmark and lines of code. **col2**: Number of inputs under the *ref/train* datasets, respectively. **col3**: Impact of the HighOpt vs. BaseOpt optimization level when applied statically to the entire program. **cols4/5/6**: Number of methods compiled and runtime improvement upon the statically BaseOpt compiled program, when using the default/aggressive/zero method hotness threshold, respectively (see Section 5.1.1). **col7**: Size of fat binary in MB and the distribution ratio to its two constituent parts (executable size : IR size). **col8**: Execution time of the program when compiled with XLC using its default static backend vs. when using Testarossa as static backend.

the execution time of the TOBEY compiled programs, as the dynamic optimization is applied to the Testarossa compiled programs.⁵

We used the full CINT2006 benchmark suite for our experiments. These benchmarks were developed from actual end-user applications and span a range of code sizes (Table I column 1) and execution times, up to orders of magnitude apart, thereby facilitating an evaluation of the scalability of the system overheads with changing code sizes and dataset sizes. We report the average of three runs for each benchmark. For benchmarks that are invoked with several inputs (Table I column 2), we present the sum of all per-input runs. Note that this incurs extra costs on the dynamic execution side, as the startup costs are incurred for each individual invocation of the benchmark. While our implementation supports Fortran, we focus on the Integer rather than the FP component of SPEC since it is more representative of general-purpose commercial codes that are likely to fall into the underoptimized usage case. The Integer benchmarks are also more challenging because their profile is flatter and less loop intensive (therefore, execution time is less concentrated in small portions of the code). The SPECcpu suite in general poses an already high bar for reoptimization on top of static compilation, even in the moderate optimization usage case, since static compilers have been thoroughly tuned for SPEC's benchmarks. As such, this suite is highly suitable to explore the feasibility aspects of our approach. The first non-SPEC application we have experimented with is *SQLite*⁶ and is included here as well, running a stream of instances of query #1 from TPC-H using a database filled with 1GB TPC-H tables.

⁵The two backends also present different opportunities altogether for dynamic optimization. Though counterintuitive, often more aggressively optimized executables present more potential for subsequent FDO, as we also show in Section 5.3.

⁶<http://www.sqlite.org>.

This section presents the overall effect of dynamic execution and examines the different factors that constitute this overall effect under different optimization levels, datasets, benchmark sizes, and system settings. Specifically, we explore the effect of compiler optimizations and number of functions recompiled (Section 5.1), the dynamic execution overheads and the limits of our system (Section 5.2), and the quality of the dynamically gathered profile (Section 5.3).

The overall effect of the dynamic optimizer under the BaseOpt scenario (our primary target scenario) is shown in Figure 3(a). The figure shows the overall execution time under the dynamic optimizer relative to execution time of the statically compiled moderately optimized executable. The same executable is invoked by the dynamic optimizer, and only the methods that are selected for recompilation at runtime are optimized at a higher optimization level (*scorching*). To get a sense of how many methods get recompiled, and when, we also plot on the bars of the chart the times at which recompiled methods became ready for execution (the “t8” time of Figure 2(a)).

The primary result this figure shows is that averaged over the entire suite, dynamic execution improves upon the original statically compiled executable by **7%, all overheads included**. Improvements up to 19% and 11% are obtained in *libquantum* and *perlbench*, respectively, and only two benchmarks are degraded, with degradations less than 2%. These improvements reflect the benefit of using a higher optimization level (*scorching*, compared to *warm* in the static compilation) for selected methods, as well

Fig. 4. (a) Coverage of execution time (higher is better) using the *BaseOpt* (moderate) optimization mode. (b) **Col2**: Profile accuracy of our two instrumentation schemes: Higher values (closer to 2) represent higher distance from “perfect” profile gathered over a much longer period of time. **Col3**: Net runtime effect of optimization using cloning-based instrumentation versus using the “perfect” profile.

as feedback-directed optimizations (based on basic-block frequencies). As such, two major factors affect the success of dynamic recompilation: (1) the number of methods that need to be compiled in order to cover a significant enough portion of the program execution time and (2) the net effect of the compiler’s *scorching* optimization level.

5.1.1. Method Recompilation. We experimented with three method recompilation configurations. The first uses our default method selection threshold ($T_2 = 10$), as described in Section 3.3. In order to stress-test the system, we also experimented with a lower, more aggressive method selection threshold ($T_2 = 2$), under which many more methods are selected for runtime optimization. Finally, to examine how the system withstands programs with much flatter profiles (and thus larger code footprints), we experimented with a threshold $T_2 = 0$, in which the optimizer constantly recompiles throughout the entire program execution. Table I lists the number of methods that were dynamically recompiled in each of these modes, along with the respective impact of dynamic optimization, in columns 4, 5, and 6 (for lack of space Figure 3(a) presents visually only the results under the default mode). The difference between the modes, reflecting the effect of the increased recompilation, is small, thanks to the low instrumentation and recompilation overheads, and also probably because the methods selected for compilation by the default setting are the most dominant ones. Figures 3(b) and 4(a) show the accumulative effect of function recompilation on performance and on the portion of execution time covered by recompiled functions, respectively. For example, a 90% coverage is reached after three recompilations in *libquantum*, compared to 20 in *gobmk* and many hundreds in *xalanbmk*. Some benchmarks, however (*mcf*, *bzip2*), reach a

Fig. 5. **Dynamic Execution Overheads Breakdown.** Dynamic execution time is shown relative to static execution time (bars lower than “1” correspond to performance improvement) using the *HighOpt* (aggressive) optimization mode.

high coverage after only a few recompilations and yet are not improved by dynamic optimization. The reason here lies in the net potential of the compiler optimization itself, examined next.

5.1.2. Compiler Optimization. Compared to *warm*, which applies a subset of the compiler’s optimizations and only the intra-basic-block ones, *scorching* applies all the optimizations, including the inter-basic-block ones, does more repetitions of some (e.g., loop canonization, local CSE, global DSE), and uses thresholds (e.g., for inlining) that allow more optimization. Overall, 167/47 optimization passes are applied under the *scorching/warm* level, respectively.⁷ We examined the effect of the *scorching* optimization level when applied statically to all the methods in the program. This reflects the net potential benefit of the optimization itself and is shown in Table I column 3. For most benchmarks, the dynamic effect is close to the static one. The low performance effects of Figure 3(a) can thus be explained by the low benefit of the compiler’s optimization regardless of the dynamic setting. Significant performance gaps between the dynamic and static *scorching* compilation appear in *gobmk*, *xalan*, and *perl*, which can be largely explained by the relatively low coverage of total execution time by recompiled methods (9%, 30%, and 60%, respectively). This reveals the general challenge of dealing with very large and “flat” programs. In terms of overheads, we have shown that our system can cope with the increased recompilation required in such scenarios (see the zero-threshold experiment of Section 5.1.1). Thus, if these larger programs also run

⁷The optimizations added by the *scorching* level include PRE, global copy propagation, loop unrolling, and induction-variable optimization. These additional optimizations do not increase the level of aggressiveness in terms of effect on strict semantics and are standard classic optimizations.

Table II. Computing the Dynamic Execution Effects, Using the Execution Modes of Figure 2(b)

System startup (loading and IR-to-executable mapping): $\text{time}(\text{dyn-no-main}^a)$
Sampling and synchronization with a waiting compiler thread: $\text{time}(\text{dyn-no-recomp}) - \text{time}(\text{dyn-base})$
Recompilation and instrumentation (profile gathering): $\text{time}(\text{dyn-instr}) - \text{time}(\text{dyn-no-recomp})$
Compiler optimization : $\text{time}(\text{dyn}) - \text{time}(\text{dyn-instr})$
Other costs : $\text{time}(\text{dyn-base}) - \text{time}(\text{statically-compiled-executable}) - \text{time}(\text{system-startup})$

^aAn additional execution mode that is not included in the figure, **dyn-no-main**, includes everything up to, but not including, the call to *main* of the statically compiled executable.

longer, recompilation will eventually affect enough portion of execution without incurring much overhead. If, however, the program is very large, very flat, and also short running, it may indeed not be a good candidate for dynamic optimization of any form. Adding more compiler threads to complete more recompilations faster, and carefully tuning the dynamic optimization level, may help in such extreme scenarios.

5.2. Effect of Dynamic Execution: *HighOpt* Scenario

The main focus of our experimental evaluation is to examine the overheads inherent to our optimization approach, since that determines the feasibility of the approach in practice. We implemented several dynamic execution modes to assist in isolating specific factors, as illustrated in Figure 2(b). Using these execution modes, we obtain the impact of the different factors that constitute the overall effect of dynamic execution, as summarized in Table II. Figure 5 shows the overall dynamic execution time relative to that of the statically compiled executable, but this time under the much more challenging *HighOpt* scenario, and in addition we test the system behavior using a very small dataset (*train*). This is a synthetic test, which does not represent a real-world scenario, but is rather meant to test the limits of our approach in terms of the overheads that it incurs, using an artificially small workload.

The figure also indicates the time breakdown into the different overheads of Table II. Averaging in total only 1.3% of the execution time, we observe the following overheads:

- Loading and mapping** (observed under the *dyn-base* execution mode of Figure 2(b)): These are generally insignificant, averaging 0.43% in the *ref* scenario and up to 3% in *gcc*. *gcc* is relatively very large, and at the same time it is invoked on very small inputs (executing less than a minute in the *ref* dataset and just over a second in the *train* dataset), resulting in an extreme case of code-size-to-execution-time ratio.⁸ A solution for such cases would probably need to involve “lazy” mapping: create a mapping only for those methods that are candidates for recompilation, rather than blindly for the entire program up-front.
- Sampling, and synchronization with the recompilation thread** (observed under the *dyn-no-recomp* execution mode): Generally negligible, this overhead is only noticeable in *gcc* and *xalan*, with a 1% effect in each, in both *ref* and *train*. It is worth noting that we are using malloc with a per-thread malloc pool; otherwise, we experience high malloc lock contention in the benchmarks that issue many memory allocation requests (up to 5% overhead in *omnetpp*, *xalan*, and *perlbench* if regular malloc settings are used, due to the presence of the extra compiler thread).
- Recompilation and instrumentation**: Averaging less than 1% with both datasets, this overhead is negligible. The average dynamic compilation time we observe is between 0.09 and 0.15 seconds per function. Since we select for recompilation only a few hot methods, the overall time spent in recompilation is therefore a tiny fraction

⁸The average effect excluding *gcc* is therefore presented as well.

of the overall program execution time and is anyhow applied in parallel to program execution. Indeed, the use of an additional thread has performance implications on the overall system, but as systems today offer more available cores than software knows how to take advantage of, it is a common practice to use additional threads for JIT compilation [Böhm et al. 2011].

- Other overheads:** Other runtime effects that cannot be isolated using the various execution modes are accounted for here, capturing the effect of executing an unchanged program under our runtime environment (e.g., due to loading the program into a different page segment than the default AIX loader).

The startup overhead is the only cost that grows (linearly) with, and can be predicted from, the load size (for fat binary sizes see Table I column 7). For benchmarks smaller than 1MB, the startup overhead is negligible and poses no limit on program execution time (*libquantum*, for example, runs for 3 seconds under the *train* dataset and is not degraded by dynamic execution). Large benchmarks such as *xalan* and *gcc* need to run for at least 52/14 seconds, respectively, in order to overcome the startup cost.

In this *HighOpt* scenario, the only advantage that the dynamic optimizer has over the static optimizer is the runtime profile information (the optimization level is the same), which is currently underutilized by our compiler. As our dynamic optimizer matures, the performance gain that can be achieved by making much more use of runtime profiling information can easily surpass these costs. Yet, we are already able to largely mask the runtime costs, almost entirely in *ref* (the average effect is 0.7% degradation, whereas the median effect is 0.5% improvement), and also get close to that in *train* (10%/2% average/median gap remains, respectively).

5.3. Effect of Profile Quality

The last aspect we examine is the quality of the dynamically gathered profile using the two instrumentation schemes that we implemented (as described in Section 3.4.2). To evaluate the profile accuracy, we compare the “distance” of the profiles gathered by each of the two schemes from a “perfect profile”: a profile that was gathered dynamically using the exhaustively instrumented version through 500ms (50 times longer than our default scheme). We use the so-called perfect profile to represent the complete profile that is gathered over the entire run of the program, similar to what static compilers collect when performing offline FDO. The “perfect” profile is not necessarily equal to the complete profile, but we did verify for most benchmarks that the “perfect profile” we got for the hottest methods are indeed similar to the complete profiles we gathered offline using a static profiling tool.⁹ To compute these “distances,” we used the Basic Block Vector (BBV) [Sherwood et al. 2001] profile representation form. We computed the BBVs that represent the three profiles generated by the three schemes (perfect, exhaustive, and cloning based). A pair of BBVs can then be compared by computing the Manhattan Distance (MD) between them. The MD of two BBVs is the sum of the absolute values of their pair-wise differences. Since all profiles are normalized (represented in the BBV as values between 0 and 1), the resulting MD is in the range of [0,2]. These distances, ranging from 0 (no difference) to 2 (largest distance), are shown in Figure 4(b) column 2 for the hottest method in each benchmark.¹⁰

⁹We don’t use the complete profile gathered by the static profiling tool (FDPR-Pro) as the “perfect” profile because we cannot directly feed it into our compiler, and it is not trivial to create a mapping from the binary-level profile gathered by FDPR-Pro (a postlink binary-level optimizer) to the IR-level profile that our JIT compiler can work with.

¹⁰The process for generating the BBV distance information is partly manual, and therefore it was not feasible for us to produce the BBVs for all the functions of all the benchmarks.

Table III. Effect of FDO: Static vs. Dynamic FDO

	1. Cross-module, offline FDO: FDPR-Pro/XLC PDF vs. XLC w default BE	2. Intramethod, offline FDO: FDPR-Pro/XLC PDF vs. XLC w default BE	3. Intramethod, dynamic FDO: TR JIT vs. XLC w TR BE
Average	0.92/0.95	0.95/1.01	1.01

For lack of space we only show the overall average over the entire SPECint2006 benchmark suite. **Col1:** impact of **cross-module** static FDO (based on complete profile gathered offline), using XLC/FDPR-Pro, respectively, relative to *-O3,-qhot* statically optimized program compiled with XLC using its default static backend. **Col2:** same as col1, but with only **intramethod** static FDO enabled. **Col3:** impact of dynamic FDO relative to the HighOpt statically optimized program compiled with XLC using TR as static backend (this corresponds to Figure 5(a)).

High differences from the perfect profile were observed in *h264ref* and *bzip2* (and to a lesser extent in *libquantum*). In the hottest method of *h264ref*, the program alternates between short bursts in which the condition of an *if* statement in the main loop is true and longer phases in which that condition is false. It so happens that both instrumentation schemes sample the nonrepresentative burst when the condition is true. A similar phenomenon is also observed in *libquantum*. Inaccuracies in *bzip2* are due to the shorter profiling period, reaching lower counters than perfect profiling.

Surprisingly, these inaccuracies usually did not result in significant overall performance differences between the two instrumentation schemes. This is because the profiles, while inaccurate, still identified the warmer blocks, and the resulting basic block ordering was similar to that generated based on the perfect profile. So while the cloning budget was not always able to reflect the true relative “hotness” of all basic blocks, it did manage to identify the hottest basic blocks.

Figure 4(b) also shows the dynamic performance gain when given two different profiles: once the profile was gathered by the default cloning-based instrumentation scheme, and once with “perfect” profile information (gathered ahead of time and saved to a file, as explained earlier) for preselected methods (column 3). In this mode, the system starts immediately from the second recompilation stage, without spending time on identifying warm methods, instrumenting, or gathering profiles. This data point allows us to evaluate the gain that our compiler could currently achieve if applied offline, as a static compiler. As can be seen, the behavior is very similar, indicating that the key factor limiting our optimization effect is not related to profiling overheads or inaccuracies but rather lies in the currently limited FDO capabilities of the repurposed compiler itself, which we are still in the process of adapting. Specific degradations incurred by the compiler optimization such as in *h264ref* were hard to isolate, especially as they were not very significant. Overall, there is no conclusive correlation between the profile accuracies shown in column 2 of Figure 4(b) and the performance differences of column 3 between the “perfect” and default modes (such as observed in *perl*). It may be that profile accuracy, or rather longer sampling period, does not guarantee better performance of FDO, as observed in Wu et al. [2013].

6. ADDITIONAL DATA POINTS

6.1. Static FDO

While we do not aim to boost performance of programs built by users/ISVs who are willing to invest the time and effort in applying offline FDO, this data point can provide insight into the additional potential of FDO yet untapped by our current implementation. Table III summarizes the impact of FDO as obtained by two

state-of-the-art profile-based optimizers: XLC and FDPR-Pro.¹¹ As can be seen, the main factor that contributes to the current advantage of offline FDO is related to interprocedural/intermodule optimizations, which our dynamic optimizer currently does not support. This is not a restriction of the dynamic engine, but rather a limitation in the new JIT-IR generator in the repurposed JIT, which treats the incoming split IR on a per-module basis. This engineering choice can and will be fixed in the future. The offline compiler, on the other hand, by default applies its FDO at link time, across methods and modules, and therefore has much more effect. When we limit the offline compiler to apply FDO only within methods, the benefit disappears almost entirely. This is partly because there are less FDO opportunities when the optimization scope is limited to single methods and partly because the offline compiler is well tuned for SPEC, allowing it to rely on static heuristics, which, in the limited intramethod scope, work well even without feedback information. In the extended scope (across methods/modules), profiling information is more critical as optimizations are much more aggressive.¹² A similar effect as the offline compiler was observed using FDPR-Pro, obtaining an average 8%/5% benefit with/without cross-method and -module FDO, respectively.

In summary, (1) extending the JIT compiler to support cross-module optimizations is important, (2) there still remain some opportunities to improve our basic block frequency-based optimization even within methods, but (3) we are not missing huge amounts of optimization potential. Even two very mature, offline feedback-directed optimizers that use a perfect profile and unlimited optimization time achieve only a few percents of improvement (up to 5%) over the static compiler without FDO. The limited effect of FDO alone (isolated from other optimizations) is not related to dynamic issues (inaccurate profiles, lack of time), but rather to the specific potential in SPECint, especially without cross-module inlining. Indeed, most of SPEC's benchmarks are too small to benefit from the improved code-cache locality that BB ordering can give, and other benefits of BB frequency-based optimizations in SPEC are limited.

6.2. IR-Only and Binary-Only Dynamic Optimization

Section 2.2 presented two known existing alternatives for dynamic optimization of static languages and discussed the qualitative differences between them, but a question that still remains open is what can be said quantitatively about the differences between these schemes.

Indeed, each of the three approaches has its own target domain for which it provides the most suitable solution, be it legacy or nonrecompilable software (most suitable for binary-only optimization), a portable software distribution strategy (most suitable for IR-only optimization), or otherwise software amenable to fat-binary-based optimization. However, assuming a theoretical scenario in which legacy, portability, and transition costs are not a factor, which approach is preferable purely from a performance aspect? Here we bring the closest thing possible to a quantitative comparison of the fat binary approach compared to the alternatives.

6.2.1. Binary only. Pursuing a meaningful quantitative comparison between the native and fat binary optimization schemes, we find that the most relevant candidate for comparison seems to be the Dynamo or DynamoRio systems. Their implementation is available for download; however, it does not support the Power platform. We therefore resort to quoting the results published by these two projects. Dynamo [Bala et al. 2000] reported an average 9% speedup on an HP PA-8000, over binaries “compiled at

¹¹An offline postlink FDO tool: <https://www.research.ibm.com/haifa/projects/systems/cot/fdpr/>.

¹²Indeed, as the interprocedural optimization level rises, the effect of FDO rises as well (XL-C's FDO has more impact on top of -O4/-O5 optimized code than on top of -O2/-O3 optimized code [Wu et al. 2013]).

the `+O2` optimization level using the product *HP C/C++ compiler*” on the SPECint95 benchmark suite **excluding *gcc***, reporting that “This benchmark consists of repeated runs of *gcc* on a number of input files, and the individual runs are too short running to qualify for our performance study.” Later, DynamoRio [Bruening et al. 2003] reported results on a Pentium 4 Xeon using SPEC2000, comparing to binaries compiled with *gcc -O3*. While they were able to obtain an average 12% improvement for SPECfp, they were not able to improve upon SPECint, reporting a **12% degradation**. Specifically, they note that “The largest slowdowns are on *perlbnk* and *gcc*. Both of these consist of multiple short runs with little code re-use. It is difficult to amortize overheads in such conditions. The time spent performing the optimizations outweighs any benefits for these benchmarks.” Obviously, comparing results across different benchmark suite releases and different platforms has limited value but does give an idea on the range of speedups that mature native binary dynamic optimizers were able to obtain, and more importantly informs of the cases in which these optimizers were not able to address, namely, very short- running benchmarks. The ability of our system to withstand even the extremely short inputs of the synthetic “train” dataset of SPECint is therefore a significant achievement and demonstrates the performance advantage of the fat-binary-based approach compared to the binary-only-based approach.

6.2.2. IR only. While abundant in the Java domain, we have not found published numbers that compare the performance of IR-only compilation to native compilation, for example, for VMs/JITs such as CLR or LLVM.¹³ A head-to-head comparison of either of these IR-only systems with our system (on the same Power platform) would have two major drawbacks: (1) The input IR to the dynamic optimizers in the different systems would be generated by entirely different static compilers, and (2) the respective JITs for CIL and LLVM do not yet produce high-quality code for Power and cannot be considered at this point “state-of-the-art” JITs for the Power platform (certainly compared to the IBM JIT on Power). The most meaningful quantitative comparison we could carry out is one that starts from the same IR (our split IR) and uses the same high-quality JIT compiler (our IBM JIT) on the same platform (Power7), operating once in an IR-only mode (in which methods need to be compiled before their code can be executed) and once in a fat binary mode (in which only hot methods are recompiled, in parallel to program execution). We emulated both a strategy in which the JIT compiler is used to compile the entire IR before execution and a strategy in which the JIT compiler is used to compile methods on demand (compiling only functions that are about to be executed, if not yet compiled).¹⁴ Table IV summarizes the results using an extremely conservative evaluation, in which the only overhead considered on the IR-only setups is compilation (disregarding any interpreter/VM-related costs), whereas all the dynamic execution overheads are considered on the fat binary setup. We believe that the performance advantage of having a fat binary compared to shipping an IR-only is clearly evident from the results, especially as the code footprint of the program becomes larger.

7. RELATED WORK

JIT compilation has traditionally been applied to languages whose dynamic nature, which allows essential information about the program to be unknown until runtime, prohibits effective static compilation (such as Smalltalk, Self, Scheme, ML, and today’s scripting languages). JIT compilation has also been motivated by portability

¹³We have also confirmed that with the CLR and LLVM compiler leads.

¹⁴LLVM supports both strategies (its “legacy-JIT” approach compiles on demand, while its new “MCJIT” approach compiles a whole LLVM IR module in a single step). CLR’s default strategy is on-demand compilation. There are additional strategies, such as JIT compiling only hot functions, but neither CLR nor LLVM support profile-driven recompilation.

Table IV. IR-Based JIT Recompilation Approaches: Compilation Statistics and Costs

1. Bench	2. IR-only, JIT compile All: #functions compiled, slowdown factor vs. native	3. IR-only, JIT compile on Demand: #functions compiled, slowdown factor vs. native	4. Fat binary, JIT compile hot: #functions compiled, slowdown factor vs. native
<i>astar</i>	106, 1.04×	96, 1.04×	6, 0.99×
<i>bzip2</i>	134, 1.03×	75, 1.01×	8, 1.02×
<i>gcc</i>	6245, 2.8×	1500, 1.43×	9, 1.04×
<i>gobmk</i>	2839, 1.45×	1943, 1.31×	3, 1.01×
<i>h264ref</i>	734, 1.28×	425, 1.16×	3, 0.99×
<i>hammer</i>	899, 1.43×	204, 1.097×	2, 0.95×
<i>libquantum</i>	183, 1.01×	75, 1.007×	6, 0.99×
<i>mcf</i>	100, 1.03×	22, 1.01×	7, 1.01×
<i>omnetpp</i>	3507, 3.44×	1090, 1.76×	6, 0.99×
<i>perl</i>	2086, 1.46×	124, 1.03×	3, 1.00×
<i>sjeng</i>	201, 1.06×	74, 1.03×	7, 0.95×
<i>xalan</i>	29168, 4.74×	4000, 1.51×	7, 1.03×
Average	1.81×	1.20×	0.99×

For each dynamic optimization approach we present two data points: the number of functions compiled by the JIT compiler and the slowdown (or speedup) factor of the dynamically compiled code compared to the natively compiled code, computed as follows: **col2:** $(X+C1)/X$ where X is the execution time of the statically generated executable when all functions are compiled using the *-O2:scorching* optimization level, and $C1$ is the total JIT compilation time of all functions in the program, using the *scorching* optimization level. No other overheads (VM, interpreter) are considered. **col3:** $(X+C2)/X$ where X is the same as col1 and $C2$ is the total JIT compilation time of the functions that get executed (using the same optimization level). No other overheads (VM, interpreter) are considered. **col4:** dyn/X where X is the same as col1 and dyn is the execution time of the dynamically optimized fat binary including all overheads. Here the number of compiled functions refers to those functions that completed two recompilations (both instrumentation and optimizations). More functions than that are recompiled at least once.

considerations, supporting a “write once,” standard, machine-independent program representation (such as Java). Profile-driven dynamic recompilation has been extensively researched in the context of these languages to avoid the cost of interpretation, to avoid generic dispatch overheads, to reduce the overhead of memory management, and more. However, prior art in the context of statically compiled languages, such as C/C++, has typically relied on alternative means of runtime profile gathering to drive recompilation. This section reviews compiler based dynamic optimization approaches¹⁵ in the context of statically compiled languages, classified according to the information they use to drive their optimization.

Runtime compilation based on load-time information: Some approaches emphasize portability and adaptation to the target platform, rather than adaptation to the runtime characteristics of the program execution, and rely solely on load-time information about the underlying architecture. Examples include dynamic code generators (e.g., VPU [Piumarta 2004], VCODE [Engler 1996]), as well as frameworks for expressing/driving code parallelization such as the C-like OpenCL language, which is JITted by design for portability, and split vectorization [Nuzman et al. 2011], which adapts vectorized code according to available architectural support.

Runtime compilation based on statically prepared optimization strategies: Runtime costs can be reduced by generating different compiled versions or optimization plans ahead of time, such as in Dynamic Feedback [Diniz and Rinard 1997], ADAPT [Voss and Eigenmann 2001], and CoCo [Childers et al. 2003]. Runtime compilation in this class of dynamic optimizers is reduced to simple code expansion/adaptation or choosing between existing versions.

¹⁵Runtime binary optimization is surveyed separately in Section 2.

Runtime compilation based on programmer annotations: Runtime optimizers can be driven by manual programmer annotations to avoid the cost of runtime profiling. DyC [Grant et al. 1999], Tempo [Marlet et al. 1999], tcc [Poletto et al. 1999], and the work by Consel and Noël [1996] rely on program directives to identify runtime invariant variables and, accordingly, the regions of code to be optimized. In ADAPT [Voss and Eigenmann 2001], the programmer drives the generation of different optimization versions. This approach is limited in its applicability due to the extra burden it incurs on the user to manually annotate the source program.

Runtime compilation based on offline profiling: Offline profiling of program inputs is used to drive dynamic optimization in the input-centric approach of Tian et al. [2011], to automatically annotate invariant variables in Calpa [Mock et al. 2000], and to guide static generation of compilation plans in CoCo [Childers et al. 2003]. Offline profiling can be used to guide optimization at various stages in the lifelong optimization framework of LLVM [Lattner and Adve 2004]. Its JIT compiler has been used to dynamically generate code in the VMKit project [Geoffray et al. 2010] and for research on various profiling techniques [Preuss 2010]. However, both projects report that LLVM only supports offline profiling and does not support selective or adaptive optimization of hotspots. This seems to be an inherent limitation in those systems, rather than an incidental one, as “constructing an adaptive compiler would require significant modifications to the LLVM JIT compiler” [Geoffray et al. 2010]. Offline profiling-based approaches rely on users to find representative inputs and perform the offline profiling runs, which limit their applicability in practice. The lack of adoption of ahead-of-time profiling is one of the main reasons to pursue this work.

Runtime compilation based on online profiling: Our work uses runtime feedback as the primary means to decide when, what, and how to recompile. As such, it offers transparent client-side FDO based on the actual current workload rather than relying on profiles gathered in previous runs, or on users/ISVs to enable/drive optimization. Other prior art in this category includes the FORTRAN system of Hansen [1974], which adaptively reoptimizes basic blocks as their frequency increases, and the continuous program optimization of Kistler and Franz [2003], which experimented with dynamic object layout and trace scheduling. The latter reported that the length of program execution time required to compensate for the time spent profiling and optimizing is too large and is feasible only in the context of long running and/or computationally intensive tasks. We successfully avoid large slowdowns by relying on a native executable that can immediately start executing and an IR of the program that can be recompiled in parallel.

An orthogonal body of work is that of online iterative optimization [Chen et al. 2012]. While it also applies dynamic recompilation, it is fundamentally different; its goal is to search for the best combination of compiler optimizations, applying a great many training runs and recompilations with varying options, using the compiler as a black box. Its monitoring is at the level of whole program execution times, as opposed to our fine-grained instrumentation of basic blocks. Recompilation is also applied at the scope of entire programs, rather than feedback-directed recompilation of selected methods. Lastly, they optimize *across runs* of “datacenter applications,” which “run the same workload over and over again, over a sufficiently long period of time.” Our goal is to optimize a *single program instance*, running a *few seconds/minutes*. The fact that dynamic recompilation can be done for regular/short-running programs, rather than long or repetitively running programs, is a new result.

Runtime compilation in specific application domains: While our focus is on the general-purpose domain, it is worth mentioning the large body of work in the domain of dynamic parallelization, which uses dynamic compilation for runtime algorithm selection [Thomas et al. 2005] or to evaluate data dependencies at runtime [Zhuang

et al. 2009; Yardimci and Franz 2009]. The latter also uses a form of fat binary, but as it “requires significant execution time and memory” to create, it is unlikely to be used in practice by ISVs. It also carries statically inserted instrumentation, as opposed to our online code selection and instrumentation approach. Lastly, it represents only a tiny code fraction (preselected loops) and targets only one optimization. This is a very different scenario than that of generic recompilation, which considers large code portions and many optimizations. Also worth noting is the embedded domain, which faces different challenges, such as adapting programs to changing energy needs or memory availability [Ozturk and Kandemir 2009].

8. CONCLUSION

Dynamic optimization has the premise of taking advantage of runtime information to dramatically boost performance; however, in the domain of statically compiled languages, this approach has so far had limited success, due to the costs associated with dynamic profiling and recompilation. Feedback-directed dynamic optimization has been known to work for Java and dynamic languages, improving upon interpreted code. It has also been shown to work for native code in the context of binary-level trace-based optimization. This work has shown that by using fat binaries and a Java JIT compiler like the method-based compilation approach, dynamic recompilation, driven by online profiling, can successfully improve upon native binaries. In the domain of general-purpose C/C++ programs, with regular (non-long-running) workloads, and even synthetically small workloads, this is a new result.

By relying on runtime information on one hand and a semantically rich intermediate representation (IR) of the source program on the other, the system has the potential to support much more aggressive optimizations than have so far been exercised by our initial implementation. Still, even with our current optimization capabilities, we are able to overcome the runtime overheads in the case of moderately optimized executables and provide already 7% improvement on average, and also generally mask the overheads in the case of highly optimized executables.

Minimal impact on the current software production process of ISVs is achieved by relying on the same compilation technology and distribution strategy they already use, with only a small compiler enhancement (to dump the compiler IR for the program). Minimal performance interference with the application is achieved by allowing it to run unmodified, incrementally recompiling only selected methods for which an IR is available. Together, these two features of the fat binary approach constitute a practical solution to the problem of underoptimized binaries, at close-to-zero cost on the software vendor side and with much performance optimization potential on the client side. We believe that the extensive experimentation presented here, under various challenging scenarios in terms of optimization level, dataset sizes, benchmark sizes, number of inputs, and method selection configuration, is a critical first step toward providing a reliable positive answer to the question of the viability of fat-binary-based profile-driven dynamic recompilation for statically compiled languages.

ACKNOWLEDGMENTS

The authors would like to thank Nitzan Peleg for her valuable advice, their colleagues at IBM for their help and support, and the anonymous reviewers for their thoughtful feedback.

REFERENCES

- ARNOLD, M., FINK, S., GROVE, D., HIND, M., AND SWEENEY, P. F. 2011. Adaptive optimization in the jalapeno JVM. *SIGPLAN Not.* 46, 4, 65–83.

- ARNOLD, M., HIND, M., AND RYDER, B. G. 2001. An empirical study of selective optimization. In *Proceedings of the 14th International Workshop on Languages and Compilers for Parallel Computing (LCPC'01)*. Springer-Verlag, London, 49–67.
- ARNOLD, M. AND RYDER, B. G. 2001. A framework for reducing the cost of instrumented code. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'01)*. ACM, New York, NY, 168–179.
- BALA, V., DUESTERWALD, E., AND BANERJIA, S. 2000. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'00)*. ACM, New York, NY, 1–12.
- BÖHM, I., EDLER VON KOCH, T. J., KYLE, S. C., FRANKE, B., AND TOPHAM, N. 2011. Generalized just-in-time trace compilation using a parallel task farm in a dynamic binary translator. *SIGPLAN Not.* 47, 6, 74–85.
- BOUAKAZ, A., PUAUT, I., AND ROHOU, E. 2011. Predictable binary code cache: A first step towards reconciling predictability and just-in-time compilation. In *Proceedings of the 2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'11)*. IEEE Computer Society, Washington, DC, 223–232.
- BRUENING, D., GARNETT, T., AND AMARASINGHE, S. 2003. An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO'03)*. IEEE Computer Society, Washington, DC, 265–275.
- CAMPANONI, S., AGOSTA, G., AND REGHIZZI, S. C. 2008. A parallel dynamic compiler for CIL bytecode. *SIGPLAN Not.* 43, 4, 11–20.
- CHEN, Y., FANG, S., EECKHOUT, L., TEMAM, O., AND WU, C. 2012. Iterative optimization for the data center. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12)*. ACM, New York, NY, 49–60.
- CHILDERS, B., DAVIDSON, J. W., AND SOFFA, M. L. 2003. Continuous compilation: A new approach to aggressive and adaptive code transformation. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03)*. IEEE Computer Society, Washington, DC, 205.1.
- CONSEL, C. AND NOËL, F. 1996. A general approach for run-time specialization and its application to C. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'96)*. ACM, New York, NY, 145–156.
- COSTA, R., ORNSTEIN, A. C., AND ROHOU, E. 2007. CLI back-end in GCC. In *Proceedings of the GCC Developers Summit*. 111–116.
- DINIZ, P. C. AND RINARD, M. C. 1997. Dynamic feedback: an effective technique for adaptive computing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'97)*. ACM, New York, NY, 71–84.
- EBCIOĞLU, K. AND ALTMAN, E. R. 1997. Daisy: Dynamic compilation for 100% architectural compatibility. In *Proceedings of the 24th International Symposium on Computer Architecture (ISCA'97)*. ACM, New York, NY, 26–37.
- ENGLER, D. R. 1996. Vcode: A retargetable, extensible, very fast dynamic code generation system. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'96)*. ACM, New York, NY, 160–170.
- GARTLEY, I., PIRVU, M., SUNDARESAN, V., AND GRCEVSKI, N. 2013. Experiences in designing a robust and scalable interpreter profiling framework. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'13)*.
- GEOFFRAY, N., THOMAS, G., LAWALL, J., MULLER, G., AND FOLLIOT, B. 2010. VMKit: A substrate for managed runtime environments. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'10)*. ACM, New York, NY, 51–62.
- GRANT, B., PHILOPOSE, M., MOCK, M., CHAMBERS, C., AND EGGERS, S. J. 1999. An evaluation of staged run-time optimizations in DyC. *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation (PLDI'99)*. ACM, New York, NY, 293–304.
- HANSEN, G. J. 1974. *Adaptive Systems for the Dynamic Run-time Optimization of Programs*. Ph.D. thesis, Pittsburgh, PA. AAI7420364.
- HAZELWOOD, K. AND SMITH, J. E. 2004. Exploring code cache eviction granularities in dynamic optimization systems. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO'04)*. IEEE Computer Society, Washington, DC, 89.
- KISTLER, T. AND FRANZ, M. 2003. Continuous program optimization: A case study. *ACM Trans. Program. Lang. Syst.* 25, 4, 500–548.
- KULKARNI, P., ARNOLD, M., AND HIND, M. 2007. Dynamic compilation: The benefits of early investing. In *Proceedings of the 3rd International Conference on Virtual Execution Environments (VEE'07)*. ACM, New York, NY, 94–104.

- LATTNER, C. AND ADVE, V. S. 2004. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO'04)*. IEEE Computer Society, 75–88.
- LU, J., CHEN, H., YEW, P.-C., AND CHUNG HSU, W. 2004. Design and implementation of a lightweight dynamic optimization system. *Journal of Instruction-Level Parallelism* 6, 2004.
- MAHLKE, S., MOSELEY, T., HANK, R., BRUENING, D., AND CHO, H. K. 2013. Instant profiling: Instrumentation sampling for profiling datacenter applications. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO'13)*. IEEE Computer Society, Washington, DC, 1–10.
- MARLET, R., CONSEL, C., AND BOINOT, P. 1999. Efficient incremental run-time specialization for free. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'99)*. ACM, New York, NY, 281–292.
- MOCK, M., CHAMBERS, C., AND EGGERS, S. J. 2000. Calpa: a tool for automating selective dynamic compilation. MICRO 33. ACM, New York, NY, 291–302.
- NUZMAN, D., DYSHEL, S., ROHOU, E., ROSEN, I., WILLIAMS, K., YUSTE, D., COHEN, A., AND ZAKS, A. 2011. Vapor SIMD: Auto-vectorize once, run everywhere. *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO'11)*. IEEE Computer Society, Washington, DC, 151–160.
- OZTURK, O. AND KANDEMIR, M. 2009. Using dynamic compilation for continuing execution under reduced memory availability. In *Proceedings of the 2009 Conference on Design, Automation & Test in Europe (DATE'09)*. European Design and Automation Association, Leuven, Belgium, 1373–1378.
- PIUMARTA, I. 2004. The virtual processor: Fast, architecture-neutral dynamic code generation. In *Proceedings of the Virtual Machine Research and Technology Symposium (VM'04)*. USENIX Association, Berkeley, CA, 8–8.
- POLETO, M., HSIEH, W. C., ENGLER, D. R., AND KAASHOEK, M. F. 1999. C and TCC: A language and compiler for dynamic code generation. *ACM Trans. Program. Lang. Syst.* 21, 2, 324–369.
- PREUSS, A. 2010. *Implementation of Path Profiling in the Low-Level Virtual-Machine (LLVM) Compiler Infrastructure*. Technical Report. University of Alberta, Alberta, Canada.
- SHERWOOD, T., PERELMAN, E., AND CALDER, B. 2001. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'01)*. 3–14.
- STOODLEY, K. 2006. Productivity and Performance: Future Directions in Compilers. Retrieved from <http://www.cgo.org/cgo2006/html/program.html>.
- SUNDARESAN, V., MAIER, D., RAMARAO, P., AND STOODLEY, M. 2006. Experiences with multi-threading and dynamic class loading in a java just-in-time compiler. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO'06)*. IEEE Computer Society, Washington, DC, 87–97.
- THOMAS, N., TANASE, G., TKACHYSHYN, O., PERDUE, J., AMATO, N. M., AND RAUCHWERGER, L. 2005. A framework for adaptive algorithm selection in STAPL. In *Proceedings of the 2001 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'05)*. ACM, New York, NY, 277–288.
- TIAN, K., ZHANG, E., AND SHEN, X. 2011. A step towards transparent integration of input-consciousness into dynamic program optimizations. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'11)*. ACM, New York, NY, 445–462.
- VOSS, M. J. AND EIGENMANN, R. 2001. High-level adaptive program optimization with adapt. In *Proceedings of the 2001 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'01)*. ACM, New York, NY, 93–102.
- WU, B., ZHOU, M., SHEN, X., GAO, Y., SILVERA, R., AND YIU, G. 2013. Simple profile rectifications go a long way: Demystifying the influence of sampling errors on feedback driven program optimizations. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'13)*.
- YARDIMCI, E. AND FRANZ, M. 2009. Mostly static program partitioning of binary executables. *ACM Trans. Program. Lang. Syst.* 31, 5, 17:1–17:46.
- ZHUANG, X., EICHENBERGER, A. E., LUO, Y., O'BRIEN, K., AND O'BRIEN, K. 2009. Exploiting parallelism with dependence-aware scheduling. In *Proceedings of the 10th International Conference on Parallel Computing Technologies (PACT'09)*. IEEE Computer Society, Washington, DC, 193–202.

Received June 2013; revised November 2013; accepted November 2013