The Remote Compilation Framework: A Sweetspot Between Interpretation and Dynamic Compilation

UCSB Technical Report #2012-03, May 2012

Nagy Mostafa Madhukar Kedlaya Youngjoon Choi Ben Hardekopf Chandra Krintz

Computer Science Department
University of California, Santa Barbara

Abstract

In this paper, we present the Remote Compilation Framework (RCF), a feedback-directed, phase-aware compilation system "as-a-service" for efficient execution of dynamic scripting language programs. Using RCF, at the client-side, users execute programs using a ubiquitous, memory-efficient interpreter extended with our light-weight sampling support. The interpreters collect samples of annotated calling context trees and communicate them to a remote optimization service. This service combines samples into a phase-based aggregate profiles across users/inputs and uses them to trigger and guide guarded, selective and speculative type specialization of programs via static translation. Users receive new versions of the program from the service as part of software update which they use for future executions of the program. We provide an empirical analysis of RCF and a sensitivity study of its phase-based recompilation strategy. We find that RCF can accurately detect phase shifts in complex applications for different numbers of users and achieves performance benefits similar to that of a popular dynamic compilation system (PyPy) using a significantly simpler and memory efficient client-side runtime system.

1. Introduction

The ubiquity and wide-spread use of interpreter-based runtime systems for dynamic scripting languages is in part due to the use of efficient interpreters implementations [8] written in C/C++ [5, 21, 25]. Such runtime implementations allow language designers to leverage mature and heavily supported tool-chains (e.g. GNU) for static compilation of their interpreters and runtime services on a wide variety of architectures, operating systems, and devices, without having to support complex compilation infrastructures themselves. These implementations also have small runtime memory footprint requirements and expedite support of new and emerging language features since the language semantics (in the interpreter) are implemented independently of and separately from the compilation system.

Alternatively, dynamic compiler systems for scripting languages, especially those that perform feedback-directed specialization [10, 23, 24], have the potential to improve program performance significantly over interpretation. Dynamic compilers, however, are highly complex, difficult to extend, and non-portable – code generation within the runtime must target a particular OS and ISA. They also have much larger memory footprint relative to compiler-free runtimes due to the data structures needed for translation and optimization and to the size of the translated/stored native code. This complexity requires significant manpower for implementation, language evolution, and maintenance. As new versions of interpreters and language specifications emerge frequently, compiler engineers are continuously forced to play catch-up. Such effort can lead to abbreviated projects lifespan [23, 34] or lack of support for the latest language features [12, 13, 22]. Moreover, these systems mix compilation with program execution and thus have a limited compilation time budget. This limitation is exacerbated by the challenges imposed by code generation for the dynamic features that these languages offer (dynamic typing, dynamic objects, polymorphic method dispatch, reflection, etc.).

In this work, we investigate a compilation-based solution that targets an intermediate point in the runtime design space between efficient interpretation and dynamic compilation for dynamic scripting languages. Our goal is to improve the performance of programs written in these languages while maintaining the simplicity, small footprint, and portability of their client-side runtime systems. Our approach is to employ hybrid execution, a combination of compilation and interpretation, but to decouple and separate the two.

Our system, called the Remote Compilation Framework (RCF), employs a simple instrumented interpreter at the client's machine and static, feedback-directed translation offline at a remote *optimization server*. A user's (client-side) interpreter collects (with low-overhead) feedback from her use of a particular application "in-the-wild". The optimization service collects and merges this information from different users and uses it to specialize the application code.

This service then returns the optimized version of the code to all users as part of a software update. Users employ the updated version of the code for future executions of the application. The optimization service compiles and specializes only the most frequently executed methods to ensure that the runtime footprint at the client remains small, while improving program performance. Although techniques similar to RCF's constituent technologies have been previously considered for debugging [15], efficient profiling [18, 33], and per-user online remote compilation of Java [14, 20, 32], RCF combines and extends these technologies into a single framework that targets, for the first time to our knowledge, low-footprint specialization of dynamic scripting languages using phase-aware sampling of multiple users/inputs.

We prototype RCF using the Python programming language as a representative scripting language because of its wide-spread use and similarities to (language and implementation) other popular dynamic languages such as Ruby and JavaScript. At the client-side, users employ a popular Python interpreter (CPython [5]), that we extend with a sample-based profiling mechanism and support for communicating sample to the remote optimization service. Our sampling mechanism collects calling context trees annotated with performance and behavioral data such as call site, execution counts, and data type information.

The remote optimization service is a software system that receives samples from multiple client-side interpreters and aggregates them into a multi-user profile from which it identifies hot methods to specialize. The service employs a phase-based process that enables it to identify when a particular profile is stable (and can be optimized) and when it changes sufficiently to warrant re-consideration (potentially additional optimization). For compilation and optimization, we extract information from the profile about hot methods and data types used in them and perform Python-level optimizations (e.g. inlining and loop optimizations). We then translate to C and type-specialize the code using Cython [6]. We insert guards as part of this process as needed to guarantee correctness. Finally, we compile the code using the GNU tool chain (gcc) for the target (client-side) platform(s). The resulting optimized code can be executed as an extension module to any CPython interpreter.

We extensively exercise RCF in this work to evaluate its overheads and performance potential. We also perform a sensitivity study on the parameters of our phase detection process. We find that the RCF sampling mechanism in CPython imposes less than 2% overhead on program execution at the client-side on average. We also find that for many popular client-side applications, we can shortcut the sampling process to around 5000 total samples because of the similarity in method use behavior (profile convergence) across inputs/users for these programs.

RCF achieves performance improvements over CPython interpretation alone of $1.1\times - 1.7\times$ for real applications, and $1.3\times - 3.4\times$ for community benchmarks, with little increase to memory footprint of the client-side runtime system. We also compare RCF to a popular Python runtime that employs dynamic compilation, feedback-directed optimization, and type and value specialization: PyPy [22]. We find that RCF facilitates performance gains similar to PyPy with a $2.7\times - 7\times$ smaller memory footprint and a significantly simpler and portable client-side runtime system implementation.

For the sensitivity analysis of our multi-user phase-aware profiling mechanism, we employ multi-program workloads as representatives of large-scale multi-component applications from which users execute different parts over time. We consider transitions between such components as phase shifts. Our analysis identifies and explains the range of parameter values required for accurate identification of phase shifts and phase stability (points at which re-compilation and re-optimization should be reconsidered). We also evaluate the impact of using large numbers of users (100-1000) for profile collection. We find that only a single parameter is influenced by the number of users (per-program) participating in the system and that, in our experiments, setting this value equal to the number of participating users is sufficient to eliminate false (premature) recompilations.

In summary, we contribute a new framework that combines interpretation and compilation in a new way that targets the effective, yet memory-efficient, performance optimization of programs written using dynamic scripting language (Python in our case). Our prototype includes new sample-based profile collection and aggregation techniques as well as a phase-driven off-line compilation and type-specialization approach that together improves performance over interpreter-based runtimes while achieving the small footprint of such runtimes on the client-side.

We overview RCF in the next section. We then detail our profiling approach based on calling context trees and across-input sampling (Section 3). We then present the RCF remote optimization service (Section 4) and describe our approach to program specialization. We follow this with a description of our experimental methodology and an extensive empirical evaluation and sensitivity study of RCF. Finally, we discuss the current state of our RCF prototypes and its limitations, present related work, and conclude.

2. Remote Compilation Framework

Our goal with this work is to investigate whether it is possible to develop a system that gleans the benefits of both efficient interpretation and feedback-directed compilation without imposing their drawbacks: poor program performance with the former, and complex client-side runtime and large client-side memory footprint with the latter. Toward this end, we design, implement, and evaluate a *Remote Compilation Framework (RCF)*, a hybrid program optimization system for dynamic scripting languages that decouples and separates interpretation from feedback-directed optimization.

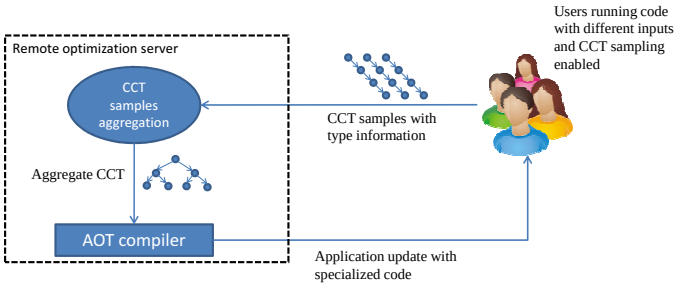

Figure 1. Remote Compilation Framework (RCF) overview.

RCF relies on performance profiles collected from users "in the wild" to trigger and guide phase-aware code optimization remotely using feedback-directed static compilation.

Our key design decisions are three-fold. First, the client-side runtime for dynamic scripting languages remains as it is in its most popular incarnation today: simple, portable, and extensible so that it is easy for language designers and runtime implementers to maintain and extend as the language it implements changes and evolves.

Secondly, we employ a compilation system that executes on a single architecture/system (a remote optimization server) as opposed to all of the possible combinations of architectures and OS of client systems. The optimization server compiles and specializes portions of programs and produces a library from these portions (for all client target platforms). The output of this process is a program that can be executed using any unmodified interpreter: the unoptimized scripting language code calls into the compiled and optimized library extensions for its execution. This model allows the interpreter and compiler to be decoupled and evolved independently of each other. While the unmodified interpreter always supports all recent language features, the compiler need only focus on the subset of features that is important for performance (i.e. features heavily used and perform poorly in the client interpreter). Features unsupported by the compiler can always be interpreted. This allows for incremental and simpler compiler implementation.

Finally, this compilation system is feedback-driven so that it is able to identify the most frequently executing code across users/clients and to specialize the code to reduce the overhead imposed by the use of dynamic scripting languages. We target only most frequently executed (*hot*) code for specialization/compilation to improve program performance without increasing the memory footprint of the client-side runtime. Being feedback-driven, RCF is also able to continuously monitor running programs and adapt to changes in their behavior by recompilation.

Figure 1 depicts RCF. While users execute a program with different inputs and use cases, the interpreter at the client side collects samples randomly about the program's behavior using calling context trees (CCTs) [1]. The client-side runtime communicates the samples to an optimization server which merges them into aggregate profiles that identify frequent and common program and phase behavior across inputs/users. Since samples can come from numerous users, per-user sampling rate can be kept very low. In the sections that follow, we detail the three primary components of RCF: client-side profile collection, the remote optimization service, and the phase detection process.

3. RCF Profiling

RCF employs calling context trees as its profile representation and collects samples randomly from users "in the wild" using modified client-side interpreters.

3.1 Calling Context Tree Profiles

A Calling Context Tree (CCT) [1, 2, 26, 38] is a representation of program behavior as a tree for which nodes represent methods and directed edges represent calls between methods. In RCF, we extend the traditional CCT to distinguish call sites. This means that if *A* calls *B* twice but from different callsites then there will be two different nodes for *B* under the same context. We distinguish nodes by call sites to enable per-callsite and per-context specialization across users/inputs; without it, a method's behavior under a particular callsite or calling-context may appear highly variable (and thus not specializable)

Our sampling system annotates CCT nodes with information about types and execution frequency of the method it represents. The key annotations that we employ are:

- **Execution Count (Time):** To represent the time spent in a method called from a particular call site using the method invocation count plus the number of times a back-edge was taken within the method.
- **Argument Types:** We record a histogram of tuple counts for each set of arguments passed into a method. The higher the number of histogram entries, the more dynamic the method is. We refer to methods (and their callsites) with one unique argument tuple as **single-typed methods/callsites** (vs multi-typed methods/callsites with multiple argument tuples).
- **Bytecode Types:** We also record a tuple of types for objects operated on and returned by the most popular and high overhead bytecodes (operand stack loads/stores and function calls). We refer to bytecodes with a single tuple **single-typed bytecodes** (vs multi-typed bytecodes with multiple tuples).

3.2 Multi-Input Remote Profiling

To collect CCT samples from different users, the users employ a standard CPython interpreter [5] that we have extended to collect sample-based profiles from Python programs. These interpreters can also communicate the samples during execution or store them to disk for off-line communication to an RCF remote optimization service.

The interpreters sample only Python code (CPython bytecodes, not native code). Each CCT sample consists of a se-

quence of CCT nodes describing the calling context at the point during execution at which the sample is collected. A leaf node in the CCT sample represents the sampled method; we annotate only leaf nodes.

The interpreters initiate sampling on a sampling event, i.e. a method call or a backward branch. For each sampling event, the runtime decrements a counter. The counter is set (and reset) to a random value between 5000 and 10000. When the counter expires, the runtime samples the code and collects bytecodes type information, until the next sampling event occurs. When this happens, the runtime resets the counter, terminates the sample, and buffers the data. When the buffer overflows, the interpreter writes the samples to disk or transmits them to the optimization server.

The RCF remote optimization server receives and merges into an aggregate profile the CCT samples from client-side interpreters (users executing the program "in the wild"). The aggregate CCT is similar in spirit to that described in [2] but we do so across inputs and users. To merge a sample, we search for the sample's context in the aggregate CCT, if the context already exists, we augment the type profile of the sample to that of the corresponding aggregate CCT node and increment its frequency. We add unmatched samples to the CCT as new nodes and edges.

4. RCF Remote Optimization Service

RCF extracts the hot callsites from the aggregate profile as candidates for specialization. RCF considers callsites that account for 50% of total execution count (method invocation counts plus taken loop backedges) as hot. For each hot callsite target method, RCF applies Python-level optimization that includes traditional type specialization as well as new techniques for feedback-directed optimization that are particularly useful for dynamic languages, including dispatching and optimization of different method versions (clones) and their context-aware specialization.

4.1 Type Specialization

Type specialization aims to reduce the number of type-checks, unbox operations, indirect calls, and other forms of overhead associated with type-generic code. For example, if an arithmetic operation is performed on two Python objects, then the implementation must first resolve the types of the two objects, find the correct method associated with that operation, and indirectly call it. However, if the types of the two objects are known to be integers, for example, then the compiler can unbox these objects and directly implement (lower) the arithmetic operations. Although it is possible to generate specialized code for different types and have it in an if-else-if ladder, in RCF, we specialize code for single-typed bytecodes only. For multi-typed bytecodes, RCF generates generic code. This simplifies our specialization system and achieves almost zero guard overhead on today's superscalar machines.

4.2 Clones Calling Mechanism

We refer to the method bodies that the optimization service specializes using particular types as *clones*. In RCF, we generate clones for hot, monomorphic (single target) callsites. Although it is possible to have multiple clones for polymorphic and/or multi-typed callsites, RCF only produces one clone per-callsite for simplicity. If a callsite is polymorphic, RCF does not specialize it. If a callsite is monomorphic (one target) and its target is a multi-typed method, RCF specializes only the single-typed bytecodes within the method.

When RCF generates a clone for a callsite, its compiler associates an ID number with the target method object and its clone. At the callsite, the runtime first resolves the method by name. If the method body has the expected ID number, its clone is invoked instead. Each clone has a global guard in its prologue that type-checks the method argument, if any of the checks fails, the generic method is invoked instead. The global guard is redundant in terms of correctness, since every specialized piece of code is already guarded. However, if the argument type checks fail, we have found that it is likely that many guards will fail inside the method body. We thus decide, in such cases, to switch to the generic version early.

4.3 Direct Calls and Inlining of Clones

Since method resolution takes place before the clone is dispatched, RCF emits code to call the clone directly, using the C calling convention, instead of using the slower calling convention that CPython implements. This eliminates the extra work of packing arguments into tuples in the caller and unpacking them in the callee, following the Python calling convention. RCF dispatches and inlines built-in methods directly, without method resolution, since Python disallows built-in methods to be modified. For example, a call to the *append* built-in method for a *List* object receiver can be called directly or inlined, provided that we perform a type check on the receiver.

4.4 Context-aware Specialization

As part of this work, we have discovered that it is effective to partition dynamic callsites by calling context to reveal significantly more static behavior in some programs. Polymorphic callsites in many of the programs we have studied are monomorphic within one or more of its calling contexts. This enables us to use optimization service to more aggressively specialize programs. Context-aware clone resolution is similar to normal clone resolution except that we require an additional comparison on the context identifier as part of the guard that determines if the clone should be dispatched.

We implement the method described in [3] to facilitate low-overhead deterministic hashing to encode calling contexts during execution. Each calling context is encoded based on the sequence of callsites it contains where each callsite is represented as a hash of its method name and bytecode offset. We store the context hash in the corresponding

call-stack frame, so no hash updates are needed on returns. This method is fast but introduces a very small chance of collisions (3 in 10 billion for a 64-bit hash value). Since we guard each clone with a check on the resolved method ID that the runtime checks each time clone is dispatched (which falls back to generic code upon failure), we guarantee that a collision can never lead to incorrect execution. In our experiments, we have observed no context hashes collisions.

5. Phase Detection and Re-Compilation

For simple programs, a single RCF specialization may be sufficient to extract high performance. For such programs, we can turn off RCF profiling at the clients after optimization. For more complex programs that implement a variety of different components and behaviors, continuous profiling and optimization may be needed to achieve additional performance gains when the use of application changes over time. Such applications are increasingly common with the advent and wide spread use of dynamic language frameworks, web services support, and cloud computing technologies (e.g. Python/Django, Python/TurboGears, Ruby/Rails, PHP/Trax, Google AppEngine [11], AppScale [4], ... etc.), among others. To support such continuous optimization, RCF implements a multi-user/input phase detection mechanism for fast phase shift detection and identification of additional optimization opportunities (stable phase behavior).

To capture a program's behavior at a particular point in time, RCF records periodically a copy of the aggregate profile, which we call a *snapshot*. RCF records snapshots every 100 samples; we refer to this duration as the *snapshot interval (SI)*. Note that earlier snapshots are subsets of later snapshots since snapshots accumulate. RCF measures and compares the similarity between two snapshots via their overlap metric [9] to detect when the behavior of an application changes (i.e. when the two snapshots deviate). Overlap measures how well the hot methods (those that cover that top 50% of execution count) in one snapshot cover the the execution count (dynamic method invocations and back edges taken) in the later snapshot. We next discuss how phase detection works in RCF in more detail.

5.1 Convergence Cycle

A naïve approach to detect re-compilation points it to compute the overlap between the latest snapshot and the aggregate profile used for the previous optimization decision. Phase shifts can then be detected when the overlap value falls below a specific threshold . This approach, however, can be very slow in detecting phase shifts and can leave some undetected. This is because the current profile is an aggregate of earlier samples. This means that, over time, the profile becomes heavily skewed toward a previous behavior and requires a long period of new behavior to overcome this skew for the new phase to be detected. To overcome this issue, we

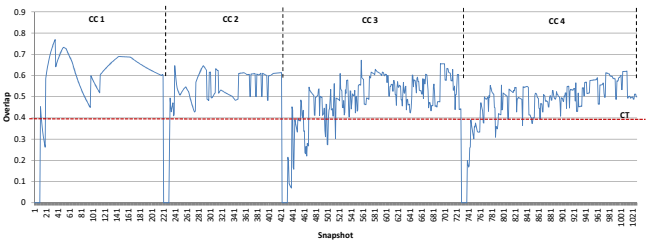

Figure 2. Example of four convergence cycles (CCs). The Y-axis is the overlap with snapshot distance (SD) = 10. The X-axis is the snapshot count. Snapshot interval (SI) is set to 100 samples. The convergence threshold (CT) is marked at 0.4. The overlap fluctuates at the start of every CC but stabilizes after 100 snapshots. CCD is set to 200 after which a new CC is started.

periodically flush the profile. We call this period the *Convergence Cycle (CC)*. A CC always starts with an empty aggregate profile. During a CC, RCF merges received samples and snapshots the profile periodically. When RCF detects that a profile has converged, it flushes the profile, terminates the current CC and starts a new one.

RCF detects CC convergence by repeatedly computing the overlap between the latest snapshot S_i and another snapshot in the past S_{i-d}, where d is the *Snapshot Distance (SD)* between them. If the overlap remains above a *Convergence Threshold (CT)* for a specified number of snapshots (*Convergence Cycle Duration (CCD)*), RCF terminates the CC and starts a new one. Note that if the overlap drops below CT, the *duration* count is reset. Only when *duration* reaches CCD, does RCF declare the CC converged. Figure 2 demonstrates how the overlap varies for four CCs. Note how flushing the profile drops the overlap to zero at the start of every CC. The overlap then rises above CT and remains there for CCD snapshots until the profile is flushed again and a new CC starts.

5.2 Phases

RCF views a phase as a sequence of CCs with homogeneous behavior. Every CC is represented by its *Last Snapshot (LS)*. The LS represents the last recorded snapshot of the converged profile for that CC before it is flushed. To determine similarity between two CCs, RCF computes the overlap between their LSs. To detect need for recompilation, RCF performs the following steps which are showed more formally in Figure 3.

1. **Detect a phase shift:** RCF tracks the LS of the CC where the last compilation occurred. That is the aggregate profile upon which the last compilation was based. For every CC that ends, RCF compares its LS with the LS from the last compilation. If the overlap falls below a threshold (*Phase Threshold (PT)*), RCF detects a phase shift.

```
// state and duration are global variables
// optLs: LS of last optimization
// curLS, prevLS: current and previous LSs
function recompile(optLS, curLS, prevLS)
begin
    if state = phase_stable then
        ol = overlap(optLS, curLS)
        if ol < PT then
            // move to phase shift state
            state = phase_shift
            duration = 0
            return False
        endif
    else if state = phase_shift then
        ol = overlap(prevLS, curLS)
        if (ol > PT) then
            duration = duration + 1
            if duration == PCD then
                // phase stable for long enough
                // move to stable phase state
                state = phase_stable
                duration = 0
                // is it a different phase ?
                ol = overlap(optLS, curLS)
                if ol < PT then
                    return True
                else
                    return False
                endif
            endif
        else
            // another phase shift interrupts
            duration = 0
            return False
        endif
    endif
end
```

Figure 3. Phase detection algorithm. The function takes the last optimization LS, the current LS and the previous LS. The function returns True if recompilation is required, False otherwise.

2. **Wait for a stable phase:** This steps measures the homogeneity of the CCs after the phase shift to detect if the profile has entered a new stable phase or is just going through a transient period. To achieve this, RCF compares the current LS with the previous one. If the overlap remains higher than the PT for a continuous number of CCs (*Phase Convergence Duration (PCD)*), the phase is stable again.
3. **Check for transition to a new phase:** Now what remains is to determine if the currently stable phase is a new phase or just a continuation of the previous stable phase. This is achieved via one last comparison between the newest LS and the LS from the last compilation. If the overlap is below PT, this is a new phase and is different from the previous stable phase that we optimized for. If not, then the behavior shift is a temporary fluctuation of the profile and no recompilation is needed.

6. Experimental Methodology

To investigate the efficacy of the various design decisions we make in RCF and to evaluate its potential and overhead, we use 12 real Python applications. We overview our application suite in Table 1. The last column of the table indicates whether the application uses *C*, a command-line interface (CLI) or *G*, a graphical-user interface (GUI).

Name	Description	G/C	LOC
2to3-2.6	Python 2.x to 3.x translator	C	83469
Brainworkshop-4.8.1	Memory trainer	G	61531
Doctuils-0.7	Text to HTML/Latex converter	C	58327
DrPython-3.11.3	Python IDE	G	16568
Markdown-2.0.3	Text to HTML converter	C	2569
Gourmet-0.15.7	Recipe manager	G	43714
PdfShuffler-0.5.1	PDF documents management tool	G	1031
PyParsing-1.5.5	General grammar parsing tool	C	10581
Solarwolf-1.5	Arcade game	G	5270
TowerDefense-0.5	Arcade game	G	3884
w3af	Web attack and audit framework	G	58133
wapiti	Web apps vulnerability scanner	C	5810

Table 1. Description of 12 real Python applications evaluated. Applications with a graphical user interface are marked with 'G', those with a command-line interface (CLI) are marked with 'C'.

We analyze the performance of RCF optimization using a subset of applications (*docutils, 2to3, markdown, gourmet*, and *pdfshuffler*), chosen arbitrarily. We also employ a set of microbenchmarks (*binarytrees, fannkuch, fasta, meteor, nbody,* and *mandlebrot*) that have been used in other studies on optimization of dynamic scripting languages. These microbenchmarks come from the Programming Language Shootout [31]; we overview their functionality in Table 2.

We also compare the performance of RCF-based optimization with that of a popular dynamic compilation and type specialization system: PyPy [24]. For this study, we use PyPy 1.6 [22], and perform the comparison using the subset of these programs that PyPy is able to run. These programs are *docutils, markdown, fannkuch, fasta,* and *meteor*.

For each program, we consider four different inputs that we have generated as test inputs and through arbitrary use (by different students) of the programs. We chose the inputs to exercise different functionalities of the programs under study. We generate profiles for three of the four inputs, on a per-thread basis, using CPython 2.6.6 [5], extended with our sample-based profiling support. For every application, we combine profiles from three inputs into one global aggregate profile. We use the fourth input to analyze the efficacy of profile-guided optimization across-inputs (for an input not used in the profile aggregation step).

Our optimization step relies on Cython [6], a Python-to-C translator and optimizer that translates Python code with type annotations into efficient C extension modules. We start by applying a set of by-hand Python-level optimizations and type-annotations to hot code guided by the aggregate profile (Section 4). We then compile the optimized Python code to C using Cython which automatically performs the required specialization and adds the necessary guards as needed. We can automate all manual steps of our optimization process but chose not to do so for this paper due to time constraints (and because doing so is purely an engineering exercise that will likely be duplicated by the Cython team). We use Cython 0.14.1 and gcc 4.4.3.

Micro Benchmark	Description
Binarytrees	Allocate and deallocate many binary trees
Fannkuch	Repeatedly access a tiny integer-sequence
Fasta	Generate and write random DNA sequences
Mandelbrot	Generate a Mandelbrot set and write a portable bitmap
Meteor	Search for solutions to shape packing puzzle
Nbody	Perform an N-body simulation of the Jovian planets

Table 2. Description of Python microbenchmarks applications evaluated.

For our experiments, we execute the programs 10 times within a test harness and report the average (with bars for standard error of the mean), across all but the first warmup run. We consider the warmup run separately, for which case we execute the programs without the test harness 10 times and report the average. Our execution platform is a Linux-2.6.32-27 machine running on an Intel Core i5 clocked at 2.67GHz, with 8GB of memory.

7. Empirical Evaluation

We next empirically evaluate RCF. In the subsections that follow, we consider the efficacy of the RCF profiling component, investigate the potential of RCF for improving performance of Python programs, and perform an extensive sensitivity analysis across the parameters of the RCF phase detection system.

7.1 Effect of Profile Aggregation

In this section, we quantify the similarity between the aggregate profile and per-input profiles that contribute to it, in both the amount of recurring type behavior and method hotness. We consider similarity at the method (call-site and body) and the bytecode level. High similarity indicates little information is lost in the aggregation process, and as such, an optimization plan that is guided only by the aggregate profile can benefit programs that employ its constituent inputs. The amount of similarity across inputs indicates the potential for across-input and ahead-of-time optimization.

We first evaluate behavioral similarity for each input and the aggregate using the time spent in methods called from monomorphic call sites (single target call sites) in Figure 4. The Y-axis is execution time (approximated by method call and back-edge counts) spent in these calls normalized to the total time. The ratio is almost identical (3% variation) across inputs and the aggregate profiles for all applications except *pyparsing* and *w3af*.

In Figure 5, we perform the same evaluation but for time spent in methods called from monomorphic call sites that are single-typed (for which the argument types are invariant). As expected, there is less time spent in these methods than if we disregard variation in argument types. For example, approximately half of *pyparsing* call-sites are monomorphic, yet, for two inputs, only 10% are monomorphic to single-typed methods. On the contrary, the methods called from monomorphic call-sites in *drpython* are almost all single-typed. This figure shows higher variation across inputs. For

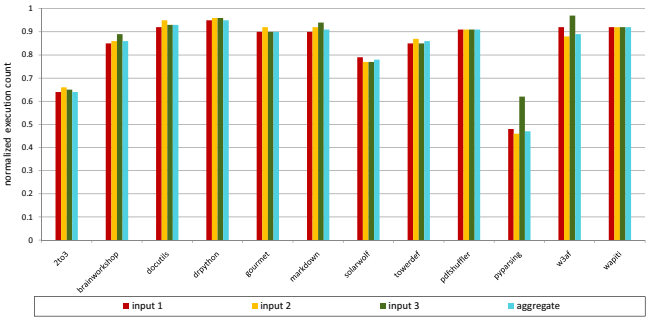

Figure 4. Effect of profile aggregation on the total execution count of monomorphic calls. Numbers are normalized to the workload total execution count. Higher is better.

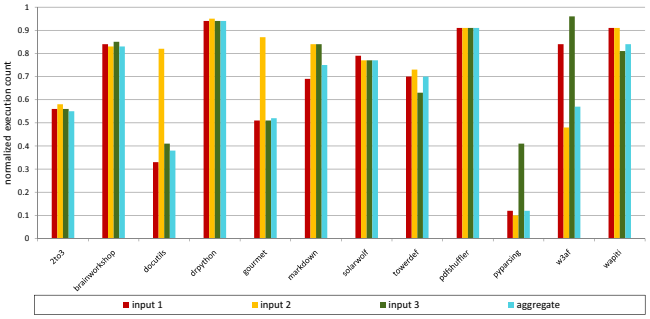

Figure 5. Effect of profile aggregation on the total execution count of monomorphic calls with invariant argument types. Numbers are normalized to the total execution count. Higher is better.

docutils, *gourmet*, *pyparsing* and *w3af*, one or two of the inputs have different ratio than the rest, and the aggregate profile tends to capture the behavior of the majority of inputs. For all other applications, the aggregate profile remains similar to that of the per-input profiles.

We next consider behavioral similarity across inputs at the Python bytecode level. Figure 6 shows the ratio of bytecodes executed that are single-typed (operate on one invariant type). With the exception of *gourmet* and *w3af*, the figure shows that single-typed bytecodes tend to remain that way across inputs. This suggests that by specializing based on the aggregate profile, we can benefit different inputs.

So far we have assessed how representative an aggregate profile is of per-input profiles in terms of the amount of static behavior it identifies across different execution of a program. The data indicates that our aggregate profiles are good representatives of individual inputs (i.e. they do not lose significant information). However, this result is only useful if an optimizing compiler specializes all methods in the code. Since we attempt to balance performance gains with memory footprint, we only optimize hot methods. To do so, the hot methods must also be the same (or similar) across inputs.

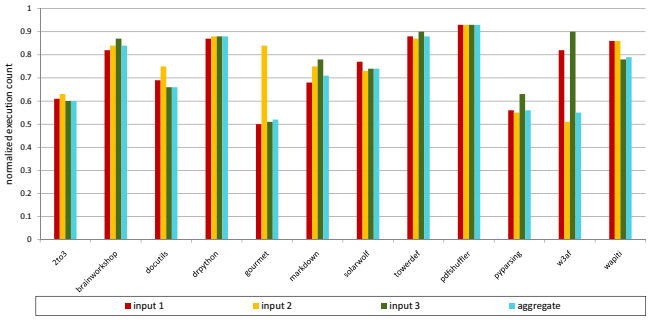

Figure 6. Effect of profile aggregation on the dynamic count of single-typed bytecodes normalized to the total bytecodes dynamic count. Higher is better.

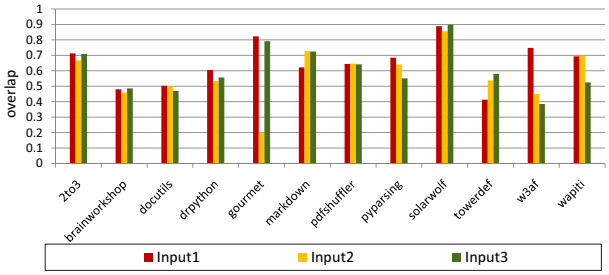

Figure 7. Overlap between the aggregate profile and each of the per-input profiles contributing to it. The graph shows the coverage over the per-input profiles when the top 50% execution count of the aggregate profile are optimized.

Figure 7 quantifies hot methods similarity of our programs. We measure the overlap of the methods that constitute the top 50% of approximated execution time between the aggregate and per-input profiles. In other words, if the compiler specializes the top 50% methods in the aggregate profile, we measure how much of the per-input profile gets covered by the specialization. For seven applications, the figure shows that the overlap is 50% or more for all inputs (up to 90% in *solarwolf*). For the remaining five programs, overlap ranges from 40% to 50%. *gourmet*'s second input exhibits a low overlap (20%). This is because this input exercises a different part of the application than the others.

7.2 Sampling Overhead and Accuracy

We next evaluate the overhead and accuracy of the RCF CCT sampling system. We first consider overhead.

Figure 8 shows the average runtime overhead for the command-line interface (CLI) applications with a randomized sampling rate between 5000 and 10000 events (calls or back-edges). On average, the overhead is less than 2%. Some applications show minor speedup. This is because the sampling overhead is so low that it falls within the margin

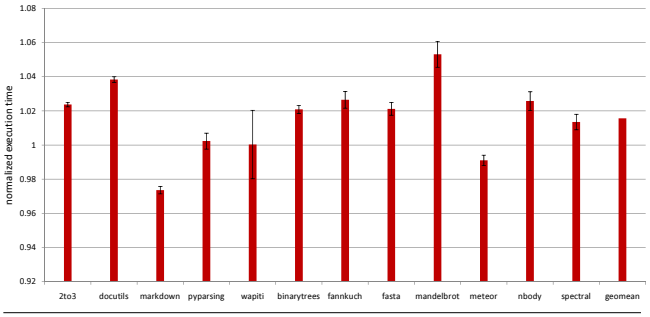

Figure 8. Runtime overhead of sampling calling context and bytecode types.

of noise in measurements[1]. *wapiti* has higher performance variation than the other programs. This is likely due to its network I/O as part of its implementation and not due to performance sampling.

In Figure 9, we investigate how fast a sampled aggregate profile converges to the complete aggregate one. That is, how many samples are needed before the sampled profile overlap with the complete profile is above a threshold. We sample evenly from three different inputs to build the sampled aggregate profile. On every hundred samples, we record a snapshot of the sampled profile and compute the overlap. Our overlap metric is how much of the complete profile is covered by the top 50% call-sites of the snapshot. That is, if we choose to specialize the top call-sites accounting for 50% of the execution count in the sampled snapshot, how much of the complete profile is covered. Hence, a reasonable overlap threshold for convergence is 50%.

Surprisingly, by only 5000 samples, we achieve an overlap of at least 50% for all applications. This indicates that, for the programs we consider, if we perform online sample collection (i.e. the user's runtime communicates samples to the optimization server while the program is executing), very few samples are needed from a small number of users to build a sampled profile that is representative enough to guide optimization. This is because client-side Python applications are typically of single-purpose and have highly peaked profiles.

7.3 Potential of Context-aware Specialization

We next investigate the feasibility and potential of context-aware specialization. We first measure the runtime overhead that is imposed by tracking the context hash (updating the context hash upon every method call). Figure 10 summarizes our findings. The average overhead is 0%, while the maximum is 4%. Similar to the sampling overhead, some applications get minor speedup. Next, we analyze the effect of context-awareness on extracting static execution behavior from programs. We measure the amount of time spent

[1] The performance of the CPython main dispatch loop is very sensitive to modification. Hence, by adding extra code, one can obtain minor speedups like the ones shown here.

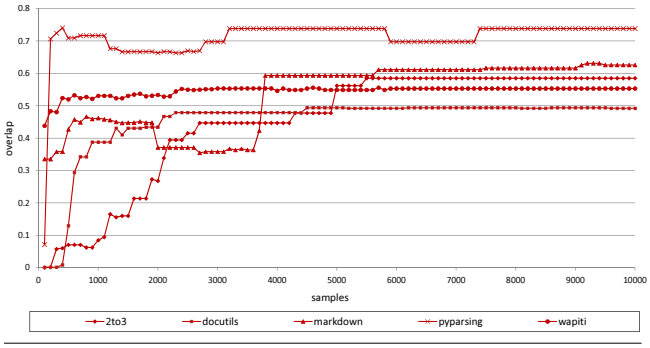

Figure 9. Convergence of the sampled profile. X-axis is the sample count and Y-axis is the overlap between the sampled and the complete profile for the top 50% of the execution count. Higher is better.

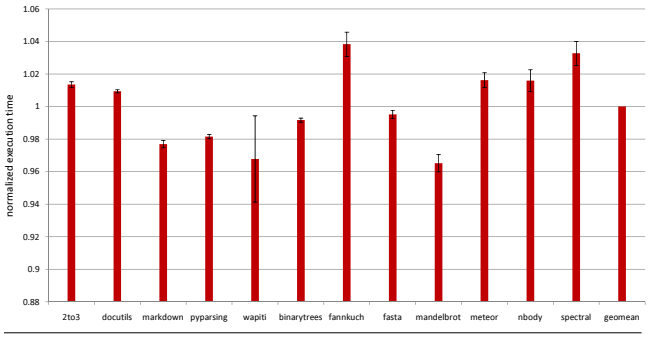

Figure 10. Runtime overhead of context hashing and tracking. Execution time is normalized to baseline. Lower is better.

in methods called from single-typed monomorphic call sites (Figure 11) when we distinguish call-sites by their contexts.

The data shows that using context information increases the time spent in these specializable sections of code for all applications. The increase is large for dynamic applications such as *pyparsing*, *docutils* and *w3af*, indicating potential for additional performance improvement over not using context-awareness. Other programs exhibit smaller improvements (*drpython*, *pdfshuffler* and *brainworkshop*). Programs that are developed in a way that does not use dynamic features extensively benefit less from context awareness. Such programs are arguably easier to optimize and as such, our context-aware profiling technique can benefit more dynamic programs – those that are currently challenging to optimize effectively.

7.4 Speedup

RCF applies the optimizations described in Section 4 to the hottest monomorphic callsites – those that account for 50% of the total execution count. We next evaluate the speedup that RCF achieves from these optimization for a subset of our applications and microbenchmarks.

Figure 12 and Figure 13 show the average speedup and error bars for six of the microbenchmarks and five applica-

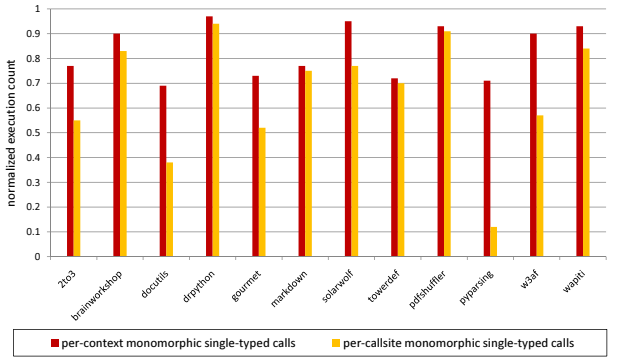

Figure 11. Effect of context-awareness on the execution count of monomorphic calls with fixed argument types. Numbers are normalized to the workload total execution count. Higher is better.

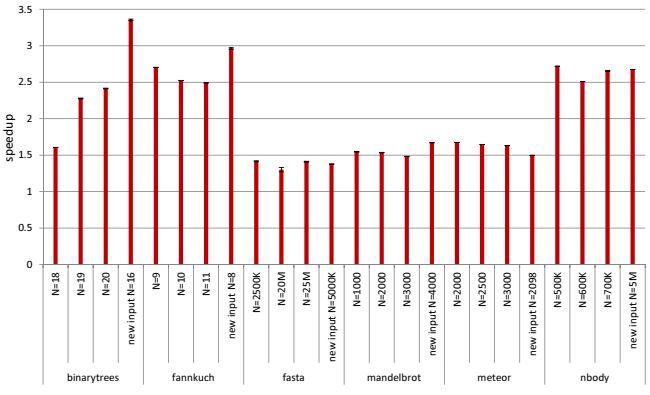

Figure 12. Speedup for six microbenchmarks using four different inputs. Three inputs are part of the aggregate profile and the fourth is a new input.

tions, respectively. We show four different inputs for each program. The first three are used to generate the aggregate profile. The fourth profile is a new input – not used as part of the aggregate profile. The speedup for the microbenchmarks ranges from 1.3× to 3.4×. The speedup for the applications ranges from 1.1× to 1.7×. The microbenchmarks benefit to a greater degree, as expected, because they have a small tight kernel which can be easily optimized with very few specialized clones. The applications have flatter profiles. This speedup for applications is significant, however, in that it is on par or better than that reported by other Python optimization efforts that employ complex dynamic compilation such as Google's (now defunct) Unladen Swallow project [34, 35].

The other interesting result is that the fourth (new) input shows nearly equal, and sometimes much higher, speedup for all programs. This means that RCF can enable performance benefits for users that employ arbitrary and new inputs for some programs. It also shows that for these pro-

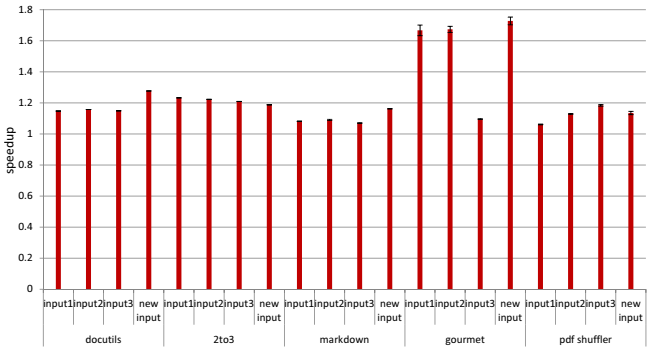

Figure 13. Speedup for real applications using four different inputs. Three inputs are part of the aggregate profile and the fourth is a new input.

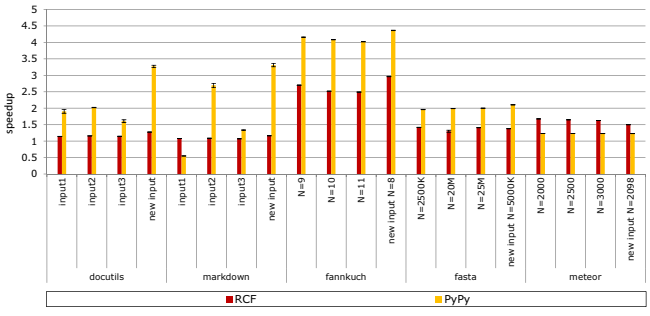

Figure 14. Speedup comparison of RCF to PyPy-1.6 without startup cost. We average the execution time of each workload over 10 runs after an initial warm-up run.

grams at least, a core subset of behavior can be captured by a small number of inputs.

7.5 PyPy Comparison

To compare these speedups against an on-going (client-side) dynamic compilation project, we next evaluate PyPy v1.6. PyPy compiles hot traces of the interpreter dispatch loop, and performs partial evaluation of methods with type and value specialization. We employ a different set of programs for this evaluation since PyPy was unable to execute most of our programs because it either does not support PyGTK or crashes during execution.

We compare RCF against the steady-state performance of PyPy by excluding the initial "warm-up" run in which PyPy identifies hot methods and optimizes them. We present the average and standard deviation of execution speedup (relative to the baseline – unoptimized, interpreted performance) across 10 runs that follow the warm-up run.

Figure 14 shows the steady-state performance comparison over four inputs. RCF outperforms PyPy on all inputs for *meteor*, and the first input for *markdown*, for which PyPy shows a 50% slowdown. RCF also shows approaching speedup to PyPy for *fasta*. PyPy achieves better performance for all other programs, specially *fannkuch*. PyPy's advantage

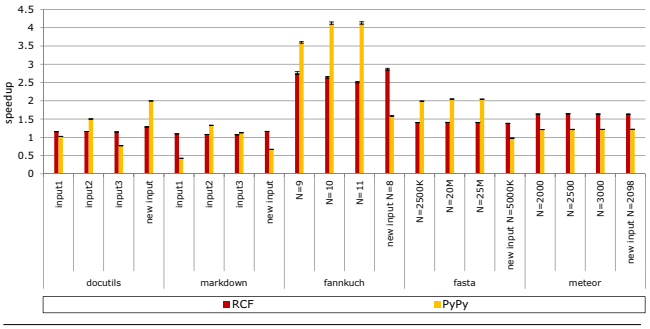

Figure 15. Speedup comparison of RCF to PyPy-1.6 for cold runs only. We average execution time over 10 cold runs to measure the effect of startup cost.

stem from (1) our exclusion of PyPy compilation overhead due to steady-state comparison in these results, (2) the use of value specialization by PyPy (which we do not employ in our RCF prototype), (3) specializations that target the behavior of the current input (as opposed to RCF's conservative application of optimizations using aggregate, multi-input profiles), and (4) compilation of a greater number of methods than RCF.

We next consider the overhead of compilation and optimization that PyPy imposes. Such overhead impacts startup time, user interactivity, as well as overall performance. We consider only the warmup run, which we execute 10 times without the harness, and present the average and error. Figure 15 shows the results. Except for *fannkuch*, for which the differences are now smaller, RCF and PyPy show similar speedup. In half of the cases, RCF outperforms PyPy. This result suggests that RCF may be more suitable for short running and user-interactive applications than dynamic compilation systems that are unable to quickly amortize their optimization overhead.

7.6 Memory Footprint

Since our goal with RCF is to achieve performance benefits without the memory footprint of dynamic compilation systems, we next investigate the impact of RCF on footprint versus that of PyPy. Figure 16 shows the memory footprint of CPython (solid), RCF (dashed), and PyPy (dotted). The X-axis is the percentage of execution time and the Y-axis is the memory in KiloBytes. We approximate memory footprint by measuring virtual memory resident set size via the Linux `ps` command. We query this value approximately every 0.1 seconds to generate this data.

RCF shows a similar memory usage pattern as CPython while PyPy footprint is significantly larger (and keeps increasing for all programs except *markdown*). The footprint of the dynamic compiler is larger due to the code objects generated by PyPy as well as the code required to implement the compilation system. The memory usage of PyPy is 2×-7× greater than that of baseline and RCF for the pro-

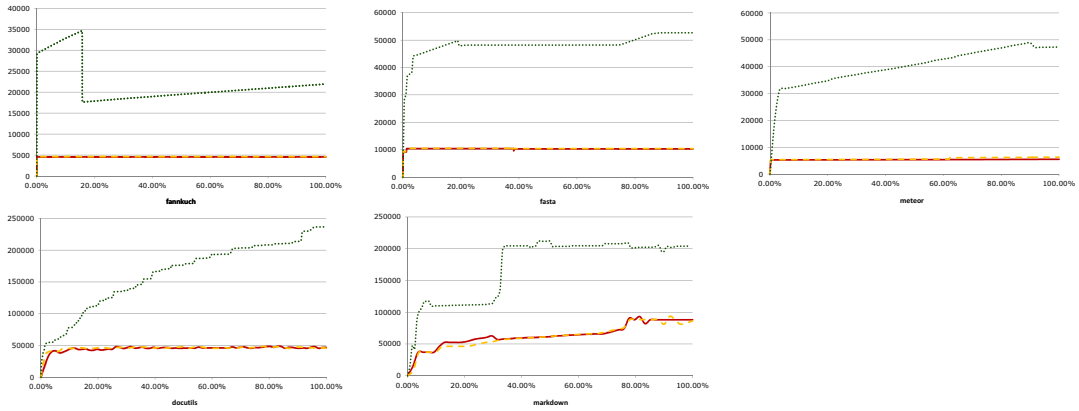

Figure 16. Memory footprint of CPython (solid), RCF (dashed), and PyPy-1.6 (dotted). The X-axis is the percentage of execution time and the Y-axis is the memory in KiloBytes

Name	Description	Value
CT	Overlap threshold to exceed for a CC to converge	0.4
CCD	Number of snapshots a CC must remain convergent before it is terminated	20
PT	Overlap threshold to fall under for a phase to be detected	0.3
PCD	Number of CCs a phase must remain convergent before it is considered stable	4
SD	Distance between two snapshots compared during a CC	10
SI	Number of samples between snapshot recordings	100

Table 3. RCF phase detection parameters. **CT:** Convergence Threshold, **CCD**: Convergence Cycle Duration, **PT**: Phase Threshold, **PCD**: Phase Convergence Duration, **SD**: Snapshot Distance, **SI**: Snapshot Interval. For each sensitivity experiment, one parameter is varied while other parameters maintain their default values.

grams we evaluated. This is on par with that reported on the Unladen Swallow project website [35].

7.7 Phase Detection

We next investigate how well RCF can react to large, complex Python programs that demonstrate phases in their behavior through its phase detection and recompilation technique. We first conduct a sensitivity analysis of the system to choose a suitable range for its parameters. We then evaluate the efficacy of the system, using the chosen parameters, on a multi-user experiment.

7.7.1 Sensitivity Analysis

RCF phase detection operation is controlled by a set of parameters that we presented earlier in Section 5. To understand the effect of those parameters on the system and the best range of values for each, we conducted a sensitivity analysis of the system. For each parameter, we vary it over a range of values while holding other parameters constant to a default value. Table 3 summarizes the system parameters and their default values. For each experiment, we observe the effect on

- **Number of false positive/negative recompilations:** False positive are recompilations that cannot be associated with a phase shift. A false negative is a phase shift that goes undetected with no recompilation triggered.
- **Average recompilation delay:** The delay, in samples, between the position where an actual phase shift occur and where RCF triggers a recompilation.

Evaluation of RCF phase detection demands a baseline for comparison. For that purpose, we devise a set of baseline workloads with known phase shift locations to which we can compare RCF. We call these *true phase shifts*. For each workload, we generate a trace that contains a mark for every true phase shift that occurred during execution. We then feed the trace to RCF phase detection module. We finally compare the *detected phase shifts* and recompilation points with the true phase shifts marks in the trace. This enables us to find the number of false recompilations, recompilation delays and other metrics.

To emulate real, complex, multi-component applications, we chain multiple applications from our evaluation suite (Table 1) together. Every application is homogeneous by itself and thus constitutes a single stable phase. A transition from one application to the next causes significant changes to the aggregate profile that lead to a phase shift. We run each workload to generate a trace of samples where points of transitions are marked. The workloads are in two groups: The first group consists of the applications *2to3* and *docutils* exercised back-to-back with three different inputs. We iterate over the execution 5 times to generate 10 phases. The second group consists of *pyparsing*, *wapiti* and *markdown*. They are also executed with three different inputs and iterated 5 times to generate 15 phases. Table 4 summarizes the workloads. In this section, we investigate the effect of all parameters.

Convergence Threshold (CT)

Every snapshot recorded during a CC is compared to a snapshot that is SD snapshots in the past, where SD is the *Snap-*

short name	# of samples	phases	# of phases
exp1_inp1	3,297,142	2to3, docutils	10
exp1_inp2	7,081,851	2to3, docutils	10
exp1_inp3	2,077,072	2to3, docutils	10
exp2_inp1	3,701,229	pyparsing, wapiti, markdown	15
exp2_inp2	2,521,970	pyparsing, wapiti, markdown	15
exp2_inp3	3,989,511	pyparsing, wapiti, markdown	15

Table 4. Attributes of the six experiments used for sensitivity analysis. The table shows, for each experiment, the number of samples it generates, the phases it consists of, and the total number of phases.

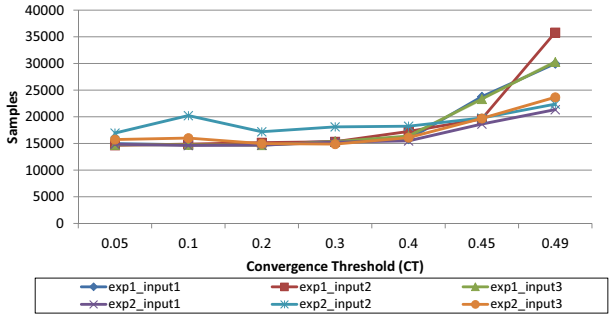

Figure 17. Effect of convergence threshold (*CT*) on average recompilation delay.

shot Distance. Only when the overlap exceeds CT and remains above it for a while is the CC terminated. Therefore, CT should control the CC length. Our results show that varying CT causes no false compilations. Although one might expect that increasing the CC length will reduce the system sensitivity, and hence lead to undetected phase shifts, this is not the case. In our experiments, the CC never gets long enough to hide a short phase. In other words, even with high CT, CCs always converge fast enough and thus have fine enough granularity to be able to detect all phases.

We investigate the effect on recompilation delay in Figure 17. As expected, the increase in CT causes an increase in recompilation delay as the CCs grow larger. The increase is significant for CT above 0.4 with the exception of *exp2_input2* at a CT of 0.1. At this point there is a spike in delay from 17,000 to 20,000 samples. This increase is due to how, by affecting the CC length, the CT also affects CC termination points and how they align with phase shifts. If a phase shift occurs very close to a CC end, the last snapshot (LS) of that CC will not reflect the shift and we have to wait for another CC before the shift is detected. We find that the average CC length for this experiment to be around 3000 samples which makes the spike amount (≈3000 samples) reasonable. This is an artifact of the randomness of the sampling process.

Convergence Cycle Duration (CCD)
CCD dictates how long the CC overlap must remain above CT before a CC terminates. It has a direct impact on the CC length. By elongating CCs, CCD also indirectly increases the phase convergence duration (PCD) which is the number of CCs RCF must wait before it declares a phase stable. Longer PCD makes the system less sensitive to phase shifts. In other words, a PCD can outgrow a phase length which will make the system consider a whole phase as a temporary transient behavior before an actual stable phase is reached. On the other end, a short CCD causes premature termination of CCs which leads to randomness in the last snapshot (LS) of CCs. Figure 18 (a) summarizes this effect. At CCD below 10, there is increase in false positives (false phase detections) due to randomness in CCs. Starting at 80, the number of false negatives (undetected phases) grows significantly. Figure 18 (b) shows the effect on recompilation delay. At a CCD of 60, the delay starts growing significantly. Although, for this experiment, a reasonable range for CCD is between 10 and 60, our experiment with larger number of users (Section 7.7.2) reveals that CCD must scale with the number of users to achieve accurate phase detection.

Phase Threshold (PT)
Phase threshold (PT) is the overlap threshold that the last snapshot (LS) of the current CC and the LS used in latest optimization must fall under for a phase shift to be detected. This parameter affects operation at the CC level, not snapshot level. It has a direct effect on false recompilations and recompilation delay. With low PT, the system becomes less sensitive to phase shifts which can lead to false negatives. With high PT, two contradicting effects take place. On one hand, the system is more sensitive which can cause false positives. On the other hand, increased sensitivity causes more interruptions during the PCD period and thus elongates it (refer to Figure 3). Elongated PCD can cause major phase shifts to be overlooked as transient fluctuation hence causing false negatives. Figure 19 (a) summarizes these effects. We see false negatives at very low PT values (0.001). No false recompilations happen in the range 0.01 until 0.3. Starting from 0.4 false recompilation, both positive and negative, arise. Figure 19 (b) shows the effect on recompilation delay. The delay remains fairly constant until a PT of 0.4, after which there is increase in delay due to the longer PCD. Reasonable PT value should be in the 0.01 to 0.3 range, depending on how sensitive we want the system to be.

Phase Convergence Duration (PCD)
PCD parameter tells RCF how many CCs the profile has to be stable before a stable phase is declared. It has the intuitive effect of controlling the length of the phase stability period and the system sensitivity to relatively shorter phases. It also causes the recompilation delay to be longer, since recompilations happen only after stable new phases are declared. Figure 20 (a) and (b) show the effects. At a PCD of 8, we start seeing false negatives. Nearly all experiments have false negatives at a PCD of 128. There is a consistent increase in recompilation delay for higher PCDs.

Snapshot Distance (SD)
The distance between every two snapshots compared dur-

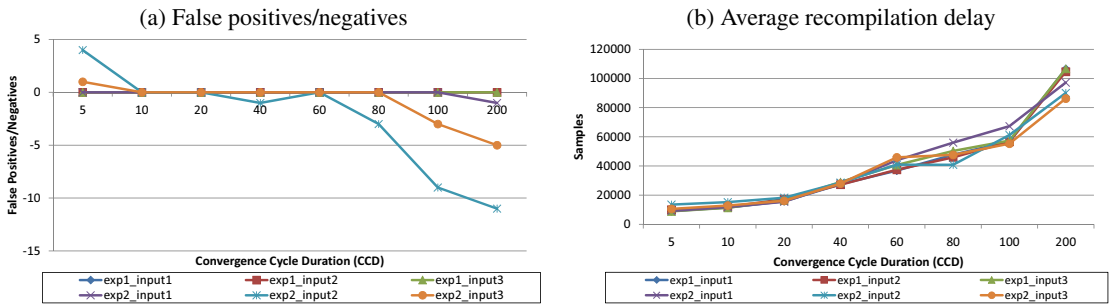

Figure 18. Effect of convergence cycle duration (*CCD*) (in snapshots) on false positives/negatives and average recompilation delay.

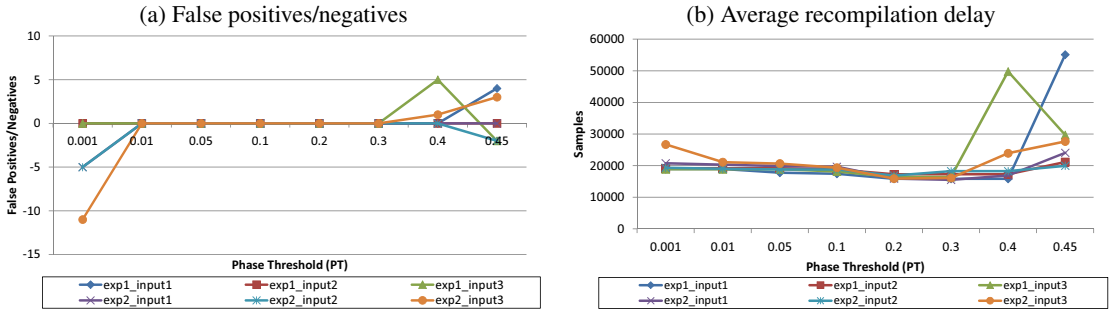

Figure 19. Effect of phase threshold (*PT*) on false positives/negatives and average recompilation delay.

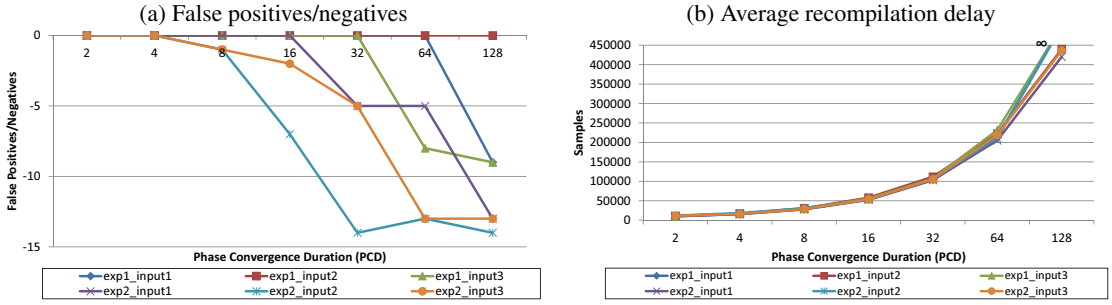

Figure 20. Effect of phase convergence duration (*PCD*) on false positives/negatives and average recompilation delay. Infinite recompilation delay means no recompilations took place.

ing a CC is the snapshot distance (SD). A low SD causes a CC to terminate early leading to randomness in CC endings which can possibly lead to false positives. A high SD expands the CC length, and consequently the PCD, and lead to false negatives. Figure 21 (a) shows false negatives at values larger than or equal to 60. At 100, RCF does not detect any phases for *exp2_input2* due to the long PCD. At 1000, all experiments, except *exp1_input2*, have no phases detected. A good range for this parameter is 10 to 40 which avoids false recompilations and has low recompilation delay (Figure 21(b)).

Snapshot Interval (SI)

Snapshot interval (SI) determines the frequency by which snapshots are recorded. Although snapshots collected at higher rate (low SI) provide finer granularity for profile comparison, they need to be accompanied by an increase in the CCD. Otherwise, the comparison of close (hence, similar) snapshots will yield a false indication of CC convergence leading to randomness in LSs and false positives. On the other extreme, higher SI will cause long CCs and consequently longer PCD and false negatives. Figure 22 (a) shows how SI can generate false compilations. On the high end, we see false negatives, which is explained by the undetected phase shifts due to a long PCD. On other low end, there is a mix of false positives and negatives. This is induced by the randomness in CCs which can either cause the overlap to drop below PT, hence leading to false phase detections (false positives), or overlap drops during PCD which can lead to excessively long PCD and hence undetected phases (false negatives). This randomness also causes increase in recompilation delay on both ends in Figure 22 (b). A reasonable choice of SI range from 100 to 250. The minimum recompi-

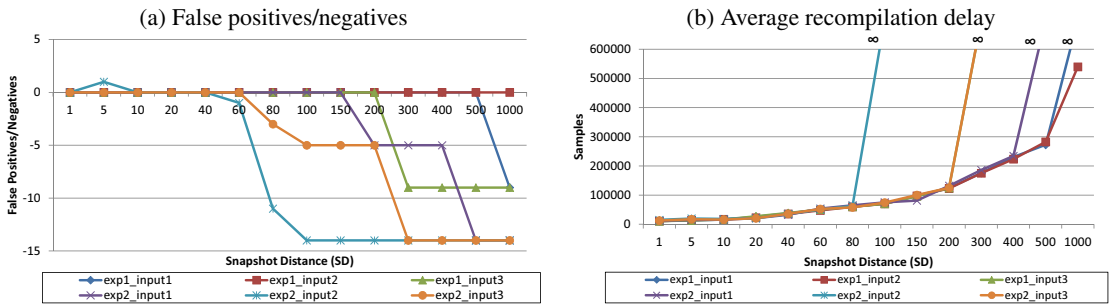

Figure 21. Effect of snapshot distance (*SD*) on false positives/negatives and average recompilation delay. Infinite recompilation delay means no recompilations took place.

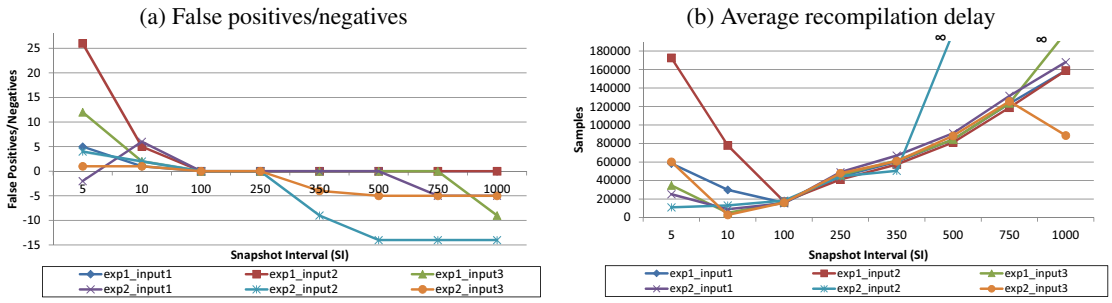

Figure 22. Effect of snapshot interval (*SI*) on false positives/negatives and average recompilation delay. Infinite recompilation delay means no recompilations took place.

lation delay for all experiments with no false recompilation happens at a SI=100.

7.7.2 Multi-user experiments

We next evaluate the efficacy of RCF phase detection for large multi-user experiments. We set the system parameters to the values shown in Table 3. For CCD, however, we set it to be equal to the number of users connected to the system. This is necessary to overcome the randomness in CCs with the increased number of users. We will explore the effect of CCD on accuracy later in this section.

To be able to assess RCF accuracy, we devised a workload consisting of five applications (components): *markdown*, *wapiti*, *pyparsing*, *docutils* and *2to3* executed in that order back-to-back. Each transition from one application to the next is a phase shift. We experiment with 1000 users and divide them into three equal groups, each running with a specific input. For every group of users, we collect a trace of samples that is fed into RCF.

Figure 23 illustrates the behavior of the workload and RCF response to it. For every input (bottom three bars), we show the phase shift locations at which the workload moves from one component to the next to form five phases for each. The phases are plotted against the snapshot count. Since the number of users in each group (input) is equal, a phase shift in one group always has an impact on the overall phase of the workload. The top bar shows RCF reaction to every phase

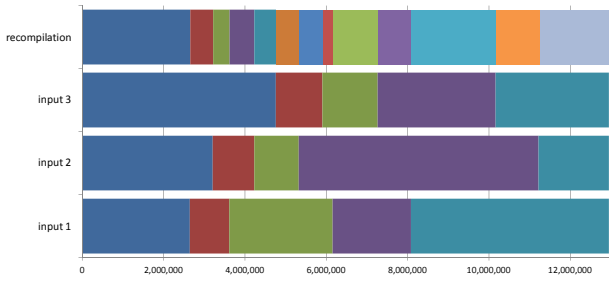

Figure 23. Illustration of phase shifts and recompilation points plotted against snapshots count for 1000 users.

# of users	100	500	1000
Samples	134,826,000	678,215,300	1,294,850,600
Avg CC length (samples)	14,799	61,556	259,033
CC count	9,110	11,017	10,402
Avg phase length (samples)	9,050,700	45,528,000	85,858,100
Avg recompilation delay (%)	3.88%	3.68%	2.08%

Table 5. Difference in behavior for the multi-user experiment when varying the number of users.

shift. There are 13 workload phase shifts, RCF detects all of them accurately with no false positives or negatives.

Table 5 shows the effect of increasing the number of users on the workload. The number of samples nearly scales linearly with the number of users to reach 1.3 billion samples at 1000 users. The average CC length increases with the number of users, this is partially because we scale CCD

CCD	100	500
Avg CC length (samples)	15114	61556
CC count	44872	11017
False positives	588	0

Table 6. Effect of decreasing CCD on number of false positives, average CC length and CC count

to be equal to the number of users and also because with more users comes more variation in the profile that it takes longer for a CC to stabilize. The table also shows average phase length, which is the average number of samples between consecutive phase shifts. The phase length scales linearly with the number of users. Finally, the average recompilation delay is shown which is the distance between when a phase shift occurs and when RCF reacts with recompilation. The number shown is a percentage of the new phase length. For all users, on average, recompilation happens before 4% of the phase samples are read. The delay declines with increasing number of users and longer phases. This indicates that although the phases are longer, only a small percentage is needed to recognize the shift.

Effect of Convergence Cycle Duration We mentioned earlier that we need to scale the CCD with the number of users. Because the CCD defines the length of a CC, if we keep the CCD constant then the number of samples received during each CC will be roughly the same. Now, if we increase the number of users connected to RCF while CCD is fixed, then, during one CC, RCF will receive less samples from each user on average. In other words, we will learn less about how each user is using the code since the number of users is increasing and the amount of samples collected per CC is the same. Since users are using the program differently, it is imperative that RCF knows enough about every use case. This is achieved by growing CCD with the number of users. If the CCD remains fixed, then the last snapshot (LS) of every CC may not be sufficiently representative of that CC leading to fluctuations in LSs and false phases detections.

While increasing CCD increases the number of samples per CC, it does not necessarily mean that the CCs will have longer time duration. With more users connected to the system, samples may be received at higher rate. Table 6 demonstrates the effect of lowering the CCD for the 500 users experiment to 100. There is an expected decrease in average CC length, increase in their count and many false recompilations.

7.8 Limitations

This work lays the groundwork for practical remote compilation for dynamic scripting languages and aims to understand its feasibility and potential. Although our prototype reveals many of the design issues involved and how to overcome them, there are still certain limitations that require further research.

First, RCF is a cross-user optimization framework that relies upon aggregate profiles to guide optimizations. Since aggregate profiles capture the way *most* of the users use the program, RCF optimizations will target this majority. A problem may arise if a relatively smaller groups of users use the program significantly different from the majority, and they do not contribute enough to the aggregate profile. RCF may then overlook these smaller groups, and they may not see any optimization benefit. One possible way of overcoming this to group users based on how they use the program and target each group with a different set of optimizations.

Second, RCF assumes that all users are running the same version of the program while profiling. However, released program updates, in addition to RCF optimization updates, affect the profile shape and program semantics. Aggregating samples gathered from mixed versions of the program will yield an inaccurate and misleading profile. A possible solution is to make RCF aware of what version each sample is coming from and route it to the correct aggregate profile.

8. Related Work

The idea of distributed remote profiling has been employed before for different purposes. Liblit et al. [15] propose an approach to isolate bugs in a program by remotely collecting assertions outcomes from large user community. Orso et al. [19] propose *GAMMA* system, a low-overhead software monitoring framework where program profiles are constructed by merging partial profiles gathered from many users connected to a network. Nagpurkar et al. [17, 18] present a phase-aware [27] profiling framework to collect accurate per-user profile with low-overhead. This approach is different from ours in that we are interested in gathering a *global* performance profile, not a per-user profile, that spans multiple users and runs of the program. This enables us to use very low sampling rates without sacrificing accuracy.

Tian et al. [33] use an approach similar to ours of cross-user profiling to gather program input information and loop trip counts with low overhead to enable input-aware optimizations. However, unlike RCF, the profiles analysis and optimizations happen locally on the user machine. The optimizations remain dynamic (happen at runtime), per-user and rely on a relatively shorter profiling period to identify the input features. Also, their work targets Java, which is a statically-typed language. In contrast, RCF is geared more towards DSL, performs AOT compilation and relies on cross-user profiling to devise optimizations for all users.

The idea of continuous program optimization has been proposed before where tuning/optimization plans are devised based on profiling across execution layers [37]. RCF employs the same concept by continuously monitoring programs running on the client-side and reacting to phase shifts via recompilation. A remote compilation strategy is used by Lee et al. [14] for Java to offload compilation tasks to a remote server resulting in lower compilation overhead and

footprint. Similar work by Palm et al. [20] propose a remote compilation service for hand-held devices in order to achieve better battery life. Sirer et al. [32] factor runtime services such as verification, security enforcement, compilation and optimization to a cluster of machines serving as a distributed virtual machine (DVM) for Java. Although our work bears similarity in concept, we do not do code replacement while the client application is running, instead we compile offline and send program updates back to the user. Furthermore, to the best of our knowledge, no remote compilation framework exists for dynamic scripting languages.

There has been several prior work on understanding overall program behavior and phases of execution [7, 16, 28–30, 36]. In [30], Sherwood et al. present a hardware-based phase detection and prediction scheme. Their technique relies on generating basic blocks execution frequency profiles at equal profiling intervals and classifying these intervals into unique phases. They use an encoding of history of phases and their durations to predict the next phase. In [36], Vijayn et al. reduce the hardware storage needed by identifying phases using branches footprint (sequence of branches addresses). Their technique also reduces the number of unique phases by collapsing phases with small variation in footprint. Nagpurkar et al. [16] devise an online software-based phase detector for virtual machines. Their solution is based on grouping profile elements into windows and doing a similarity check to detect a phase shift.

RCF phase detection scheme differs in several ways. First, RCF operates on a higher level profile of hot methods and their argument types which aim to guide a type-specializing compiler. Second, the profile samples are collected online from different instances of the application running with different input. Finally, although RCF still relies on the idea of a profiling interval (a profile snapshot in our case), we add second layer of adaptive profiling interval: the convergence cycle. A convergence cycle is terminated only when the aggregate profile is stable long enough. This dampens the randomness in the snapshots expected when sampling from multiple sources.

9. Conclusion

In this paper, we present the Remote Compilation Framework (RCF) for improving the performance of Python programs. RCF decouples compilation from interpretation so that each can be performed at a different location. The RCF model is one in which a large number of users execute a particular program using an interpreter extended with support for collecting samples of program execution behavior unobtrusively. The samples are communicated to an optimization server where they are aggregated into a calling-context-aware type profile that it then uses to guide phase detection and selective program specialization. RCF then periodically delivers optimized versions of the program (Python plus compiled and optimized libraries) to users as a part of a software update that they then use for future executions of the program using any unmodified interpreter.

By targeting the sweet spot between interpretation and dynamic compilation, RCF retains the benefits of both: simplicity and low footprint of user interpreter-based runtimes at the client side, and type-specialized performance of dynamic languages. Specifically, we evaluate RCF using community benchmarks and real applications and find that (i) RCF enables $1.1 \times -3.4\times$ performance gains, (ii) these gains are similar to those of achieved by the popular, yet complex, PyPy client-side dynamic compilation system for Python, (iii) introduces very low overhead for performance sampling ($< 2\%$), and (iv) that has a memory footprint that is $2.7 \times -7\times$ smaller than PyPy.

We also propose a phase detection mechanism for RCF that enables it to perform adaptive recompilation as needed. RCF continuously compares snapshots of the aggregate profile to detect phase shifts and triggers recompilation if the program enters a new phase. We evaluate our techniques through an extensive sensitivity analysis of the systems parameters and by a large multi-user experiment. RCF is able to accurately detect all phase shifts and recompiles with a delay of less than 4% on average.

References

[1] AMMONS, G., BALL, T., AND LARUS, J. R. Exploiting hardware performance counters with flow and context sensitive profiling. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation* (New York, NY, USA, 1997), ACM, pp. 85–96.

[2] ARNOLD, M., AND SWEENEY, P. F. Approximating the calling context tree via sampling. Tech. rep., IBM Research, July 2000.

[3] BOND, M. D., AND MCKINLEY, K. S. Probabilistic calling context. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications* (New York, NY, USA, 2007), ACM, pp. 97–112.

[4] CHOHAN, N., BUNCH, C., PANG, S., KRINTZ, C., MOSTAFA, N., SOMAN, S., AND WOLSKI, R. Appscale: Scalable and open appengine application development and deployment. In *International Conference on Cloud Computing* (2009).

[5] CPython. `http://www.python.org/`.

[6] Cython: C-Extensions for Python. `http://cython.org`.

[7] DHODAPKAR, A. S., AND SMITH, J. E. Comparing program phase detection techniques. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2003), MICRO 36, IEEE Computer Society, pp. 217–.

[8] ERTL, M. A., AND GREGG, D. The structure and performance of efficient interpreters. *The Journal of Instruction-Level Parallelism vol. 5* (November 2003).

[9] FELLER, P. T. Value profiling for instructions and memory locations. Master's thesis, University of California, San Diego,

April 1998.

[10] GAL, A., EICH, B., SHAVER, M., ANDERSON, D., MANDELIN, D., HAGHIGHAT, M. R., KAPLAN, B., HOARE, G., ZBARSKY, B., ORENDORFF, J., RUDERMAN, J., SMITH, E. W., REITMAIER, R., BEBENITA, M., CHANG, M., AND FRANZ, M. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation* (2009), PLDI '09, ACM, pp. 465–478.

[11] Google app engine. `http://code.google.com/appengine/`.

[12] IronPython. `http://ironpython.codeplex.com/`.

[13] Jython. `http://www.jython.org/`.

[14] LEE, H. B., DIWAN, A., AND MOSS, J. E. B. Design, implementation, and evaluation of a compilation server. *ACM Trans. Program. Lang. Syst. 29* (August 2007).

[15] LIBLIT, B., AIKEN, A., ZHENG, A., AND JORDAN, M. Bug Isolation via Remote Program Sampling. In *Conference on Programming Language Design and Implementation (PLDI)* (2003).

[16] NAGPURKAR, P., KRINTZ, C., HIND, M., SWEENEY, P. F., AND RAJAN, V. T. Online phase detection algorithms. In *Proceedings of the International Symposium on Code Generation and Optimization* (Washington, DC, USA, 2006), CGO '06, IEEE Computer Society, pp. 111–123.

[17] NAGPURKAR, P., KRINTZ, C., AND SHERWOOD, T. Phase-Aware Remote Profiling. In *International Symposium on Code Generation and Optimization (CGO)* (Mar. 2005).

[18] NAGPURKAR, P., MOUSA, H., KRINTZ, C., AND SHERWOOD, T. Efficient remote profiling for resource-constrained devices. *ACM Trans. Archit. Code Optim. (TACO) 3*, 1 (2006), 35–66.

[19] ORSO, A., LIANG, D., HARROLD, M., AND LIPTON, R. GAMMA System: Continous Evolution for Software After Deployment. In *International Symposium on Software Testing and Analysis* (2002), pp. 65–69.

[20] PALM, J., LEE, H., DIWAN, A., AND MOSS, E. When to use a compilation service? In *LCTES* (2002).

[21] Perl language. `http://www.perl.org`.

[22] PyPy virtual machine. `http://pypy.org/`.

[23] RIGO, A. Representation-based just-in-time specialization and the Psyco prototype for Python. In *ACM SIGPLAN 2004 Symposium on Partial Evaluation and Program Manipulation - PEPM'04* (Aug 2004).

[24] RIGO, A., AND PEDRONI, S. PyPy's approach to virtual machine construction. In *Proceedings of the Dynamic Languages Symposium* (Oct 2006).

[25] Ruby language. `http://www.ruby-lang.org/`.

[26] SERRANO, M., AND ZHUANG, X. Building approximate calling context from partial call traces. In *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization* (Washington, DC, USA, 2009), CGO '09, IEEE Computer Society, pp. 221–230.

[27] SHERWOOD, T., PERELMAN, E., AND CALDER, B. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *International Conference on Parallel Architectures and Compilation Techniques* (Sept. 2001).

[28] SHERWOOD, T., PERELMAN, E., AND CALDER, B. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques* (Washington, DC, USA, 2001), PACT '01, IEEE Computer Society, pp. 3–14.

[29] SHERWOOD, T., PERELMAN, E., HAMERLY, G., AND CALDER, B. Automatically characterizing large scale program behavior. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2002), ASPLOS-X, ACM, pp. 45–57.

[30] SHERWOOD, T., SAIR, S., AND CALDER, B. Phase tracking and prediction. In *Proceedings of the 30th annual international symposium on Computer architecture* (New York, NY, USA, 2003), ISCA '03, ACM, pp. 336–349.

[31] The computer language benchmarks. `http://shootout.alioth.debian.org/`.

[32] SIRER, E. G., GRIMM, R., GREGORY, A. J., AND BERSHAD, B. N. Design and implementation of a distributed virtual machine for networked computers. In *Proceedings of the seventeenth ACM symposium on Operating systems principles* (New York, NY, USA, 1999), SOSP '99, ACM, pp. 202–216.

[33] TIAN, K., ZHANG, E., AND SHEN, X. A step towards transparent integration of input-consciousness into dynamic program optimizations. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications* (New York, NY, USA, 2011), OOPSLA '11, ACM, pp. 445–462.

[34] Unladen-Swallow project. `http://code.google.com/p/unladen-swallow/wiki/Benchmarks`.

[35] Unladen-Swallow project results. `http://code.google.com/p/unladen-swallow/wiki/`.

[36] VIJAYN, B., AND PONOMAREV, D. V. Accurate and low-overhead dynamic detection and prediction of program phases using branch signatures. In *Proceedings of the 2008 20th International Symposium on Computer Architecture and High Performance Computing* (Washington, DC, USA, 2008), IEEE Computer Society, pp. 3–10.

[37] WISNIEWSKI, R. W., SWEENEY, P. F., SUDEEP, K., AND HAUSWIRTH, M. Performance and environment monitoring for whole-system characterization and optimization. In *In Proc. of the 2nd IBM Watson Conference on Interaction between Architecture, Circuits, and Compilers (PAC), Yorktown Heights* (2004), pp. 15–24.

[38] ZHUANG, X., SERRANO, M. J., CAIN, H. W., AND CHOI, J.-D. Accurate, efficient, and adaptive calling context profiling. *SIGPLAN Not. 41*, 6 (2006), 263–271.