Design, Implementation, and Evaluation of a Compilation Server

Technical Report CU-CS-978-04

HAN B. LEE
University of Colorado
AMER DIWAN
University of Colorado
and
J. ELIOT B. MOSS
University of Massachusetts

Modern JVM implementations interleave execution with compilation of "hot" methods to achieve reasonable performance. Since compilation overhead impacts the execution time of the application and induces run-time pauses, we explore offloading compilation onto a compilation server. In this paper, we present the design, implementation, and evaluation of compilation server which compiles and optimizes Java bytecodes on behalf of its clients.

We show that the compilation server provides the following benefits: (i) lower execution and pause times of the benchmark application due to reducing the overhead of optimization; (ii) lower memory consumption of the client by eliminating allocations due to optimizing compilation and footprint of the optimizing compiler; (iii) lower execution time of the application due to sharing of profile information across different runs of the same application and runs of different applications.

We implemented compilation server in Jikes RVM, and our results indicate that it can reduce run time by an average of 20.5%, pause times by an average of 81.0%, and dynamic memory allocation by 8.6% for our benchmark programs. Our simulation results indicate that our current implementation of compilation server is able to handle more than 50 concurrent clients while still allowing them to outperform best performing adaptive configuration.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Compilers*

General Terms: Design, Experimentation, Performance, Measurement, Languages

Additional Key Words and Phrases: Compilation server, Java Virtual Machine

1. INTRODUCTION

One of the key features behind the enormous success of the Java programming language is its network- and platform-independence: one can compile Java programs into platform-neutral bytecode instructions and ship those instruction across a network to their final destination where a Java Virtual Machine (JVM) executes them [Lindholm and Yellin 1996].

This material is based upon work supported by the National Science Foundation grant CCR-0085792 and CCR-0133457, and IBM. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the sponsors.

Java's "write once, run everywhere" paradigm is especially useful for mobile computing, where applications may be downloaded on demand.

There are at least two basic methods of executing Java bytecode instructions: (i) by interpreting the bytecode instructions; and (ii) by first compiling the bytecode instructions into native machine code instructions and then executing the resulting native code. The interpretation-only approach often results in poor performance because of interpretation overheads, while the compilation-only approach introduces additional run-time overhead because compilation occurs at run time. Thus, many JVMs use hybrid methods: combine interpretation with optimizing compilation of "hot" methods at run time or combine a cheap compiler with an optimizing compiler used only for hot methods. While these combined approaches result in more efficient execution of Java programs, they still incur the time, memory, and pause time overhead of optimizing compilation.

This optimizing compilation overhead is already quite significant in desktop systems, and the situation is worse in devices where one has to balance computing power with other attributes such as battery life. With the advent of mobile computing, there is an increasing number of networked (wired or wireless) devices with limited CPU speed, small memory size, and/or limited battery life, where the optimization overhead may be prohibitive.

We present here the design, implementation, and evaluation of *Compilation Server* (*CS*), which is a server-assist mechanism to eliminate or reduce the compilation overhead. *CS* can compile and optimize code on behalf of clients. This paper includes and greatly extends our earlier work [Palm et al. 2002], where, using a power model, we investigated the feasibility of using a compilation service in energy constrained devices. Our implementation of *CS* is based on Jikes RVM [Burke et al. 1999], and we present results for the SPECjvm98 benchmark suite, ipsixql, and pseudojbb, a variant of SPECjbb2000, benchmarks.

We show that *CS* provides the following benefits compared to the best performing *Adaptive* configuration: (i) lower execution by an average of 20.5% and pause times by an average of 81% for the benchmark applications, by reducing the overhead of optimization; (ii) lower memory management load on the client by an average of 8.6% by eliminating allocation coming from optimizing compilation and by reducing the footprint of the optimizing compiler; (iii) lower execution time of the application due to sharing of profile information across different runs of the same application and runs of different applications.

Our results also indicate that *CS* scales well and can support over 50 concurrent clients while still allowing them to outperform the *Adaptive* configuration. We also show that client performance is only slightly affected by slow network speeds, and that profile-driven optimizations are feasible in the context of *CS*.

We organize the remainder of the paper as follows. Section 2 describes the state-of-the-art execution model in JVMs and motivates *CS*. Section 3 presents design decisions behind our current *CS* implementation. Section 4 describes our experimental methodology, while Section 5 presents the results. Section 6 reviews prior work in the area. Finally, Section 7 concludes.

2. MOTIVATION

In Section 1, we introduced the concept of *CS*. In this section, we motivate the need for *CS* by discussing limitations of the execution model of current JVMs.

2.1 Execution Model of State-of-the-Art Java Virtual Machines

2.1.1 *Background.* Running Java programs normally consists of two steps: converting Java programs into bytecode instructions (i.e., compiling Java source to *.class* files), and executing the resulting class files [Gosling et al. 2000]. Because the compiled class files are network- and platform-neutral, one can easily ship them across a network to any number of diverse clients without having to recompile them.

JVMs then execute these class files, and to achieve reasonable performance, state-of-the-art JVMs, such as HotSpot [Paleczny et al. 2001] and Jikes RVM [Burke et al. 1999], also perform dynamic native code generation and optimization of selected methods. Instead of using an interpreter, Jikes RVM [Arnold et al. 2000] includes a *baseline* (non-optimizing) and an *optimizing* compiler. The baseline compiler is designed to be fast, and easy to implement correctly, while the optimizing compiler is designed to produce more efficient machine code by performing both traditional compiler optimizations (such as common-subexpression elimination) and modern optimizations designed for object-oriented programs (such as pre-existence-based inlining [Detlefs and Agesen 1999]). In addition to providing flags to control individual optimizations, Jikes RVM provides three optimization levels: O0, O1, and O2. The lower levels (O0 and O1) perform optimizations that are fast (usually linear time) and offer high payoff. For example, O0 performs inlining, which is considered to be one of the most important optimizations for object-oriented programs. O2 contains more expensive optimizations such as ones based on static single assignment (SSA) form [Cytron et al. 1991]. The optimizing compiler can introduce significant run-time overhead because compilation of Java programs happens during the execution of the program (i.e., at run time). Thus, even though the optimizing compiler produces more efficient machine code, it is not necessarily the best performing configuration when used alone.

In fact, the best performing configuration in Jikes RVM is the *Adaptive* system. This system tries to combine the best of both the baseline and the optimizing compiler by initially compiling all methods with the baseline compiler, and once it has identified "hot" methods, it recompiles them with the optimizing compiler. The goal of the adaptive system is to perform as well as the optimizing compiler-only approach without incurring much run-time compilation overhead.

2.1.2 *Implications.* We now examine the performance characteristics of the various optimization levels in Jikes RVM, and compare them to those of the Adaptive system.

The compilation overhead incurred by Jikes RVM at run time ranges widely depending on its selection of compilers and optimizations. We examine and compare the execution speed and compilation cost of different compilers and optimization levels in Jikes RVM. Figure 1 gives the execution time of the SPECjvm98 benchmarks [Standard Performance Evaluation Corporation (SPEC) 1998] (input size 100) on a Pentium 3 running at 500 MHz with 512 MB of physical memory. We obtained the data using the FastAdaptiveSemiSpace configuration (i.e., semi-space copying collector) of Jikes RVM version 2.2.2.

For each benchmark, Figure 1 has five bars, corresponding to different compilers and optimization levels, with Baseline performing no optimizations and O2 performing aggressive (including SSA-based) optimizations. The *Adaptive* bars use the adaptive system in Jikes RVM, and optimize methods only when they become *hot* (are seen to be executed more than a certain threshold). Each bar has two segments: *Running time* gives the execution time of the benchmark (i.e., with compilation time removed), and *Compilation time*

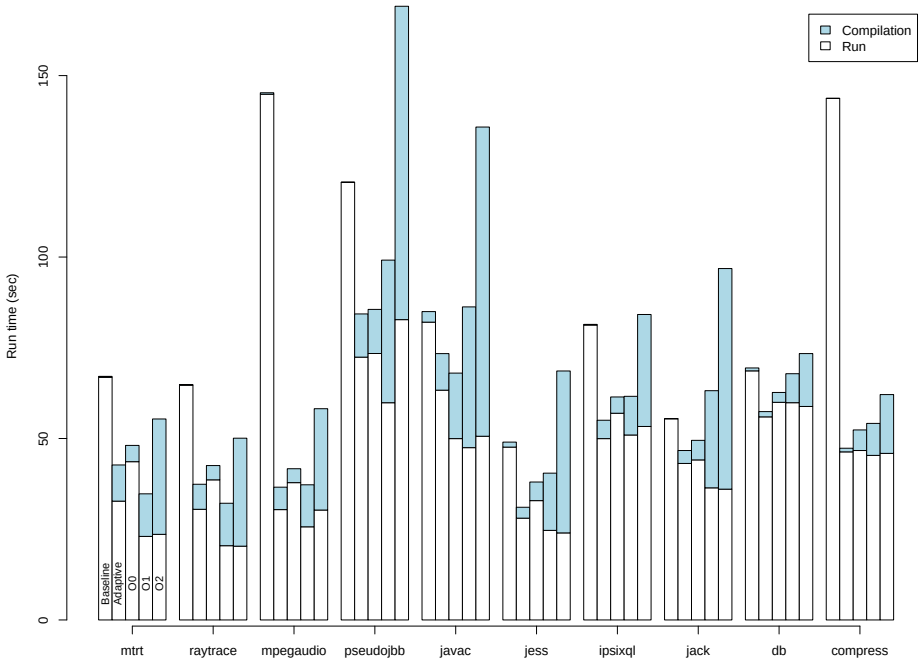

Fig. 1. Execution and compilation times (seconds). The bars, from left to right, are: Baseline, Adaptive, O0, O1, and O2

gives the time to run the optimizing compiler.

From Figure 1 we see that, for the most part, increasing the optimization level decreases the running time of the application. However, increasing the optimization level also increases the compilation time, and in many cases the increase in compilation time is greater than the reduction in running time of the application. The *Adaptive* configuration performs well, giving performance that is close to O1 and O2, but with lower compilation cost. However, even the adaptive compiler spends up to 23% of total execution time in the compiler. This suggests that it may be worthwhile to migrate compilation to a more capable server not only to reduce compilation cost but also to increase the number of methods benefiting from optimization.

2.2 Compilation Pause Times

2.2.1 *Background.* In addition to affecting overall running time of applications, dynamic compilation also affects their responsiveness because it induces pauses in program execution. The use of pause times as a performance metric is popular when evaluating garbage collection algorithms (e.g., [Cheng and Blelloch 2001]), and it should be equally important in evaluating dynamic compilation systems. For example, Hölzle and Ungar [Hölzle and Ungar 1994] uses the concept of absolute pause times to evaluate the responsiveness of the SELF programming system.

2.2.2 *Implications.* Figures 2 and 3 show pause times due to dynamic compilation by the optimizing compiler in the Adaptive system. The pauses due to baseline compilation are insignificant as they are very short. The x-axis shows the actual execution time for a particular benchmark, and thus its scale varies across different benchmarks. The y-axis represents pause times in seconds. Thus, the bar at execution time *x* with the height of *y* in Figures 2 and 3 says that *x* seconds into the benchmark run, there is a compilation pause of *y* seconds. There are two things we can notice from the figures. First, even in the Adaptive system, there are many methods that need to be compiled by the optimizing compiler in order to achieve reasonable performance. Second, the bars are often tall. In fact, some compilations take longer than 700 ms, which is unacceptable for interactive applications, and most compilations last for more than 100 ms. Migrating compilation to a server may alleviate the problem of long pause times since (i) the server, being a more powerful machine (and not involved in running the application program), would be able to compile methods faster; and (ii) the client may continue executing baseline compiled code while waiting for optimized machine code to come back from the server.

2.3 Memory Usage

2.3.1 *Background.* Performing code optimizations consumes memory and thus may degrade memory system performance. Since Java programs will be optimized at run time, there are two memory costs for optimizations: (i) the data space cost, i.e., the space required by the optimizer to run; and (ii) the instruction space cost, i.e., the footprint of the optimizer. (The final size of optimized code may be larger or smaller than unoptimized code, but this size effect is much smaller than the other two.)

2.3.2 *Implications.* Figure 4 shows the bytes allocated by the various configurations of Jikes RVM. We see that as we increase the level of optimization, the number of bytes allocated increases dramatically. For resource-limited systems, such as handheld devices, the memory costs alone may make aggressive optimizations infeasible. Since the *Adaptive* configuration optimizes only the hot methods, we see that it does not allocate many more bytes than the non-optimizing configuration (*Baseline*). However, even the *Adaptive* configuration will have the full instruction space cost of the normally optimizing configurations since it includes the code of the optimizing compiler. This footprint is approximately 14 MB. By migrating optimizing compilation onto a server, one would be able to reduce dynamic allocation due to compilation and also to eliminate the large footprint associated with the optimizing compiler.

In summary, by offloading compilation onto a compilation server, we should be able to improve end-to-end execution time, lower compilation pauses, and reduce memory usage of Java programs.

3. DESIGN AND IMPLEMENTATION OF COMPILATION SERVER

Our current implementation of the *CS* is designed to address those concerns presented in Section 2, and in this section, we discuss design trade-offs behind the current implementation.

The primary design goal of *CS* is to minimize client execution time. *CS* clients may include desktop PCs, laptops, and PDAs, and thus are likely to be limited in one form or another compared to *CS*, which would be equipped with plenty of memory, fast disk drives, and fast network connection(s). Therefore, it would be beneficial to allow the server to

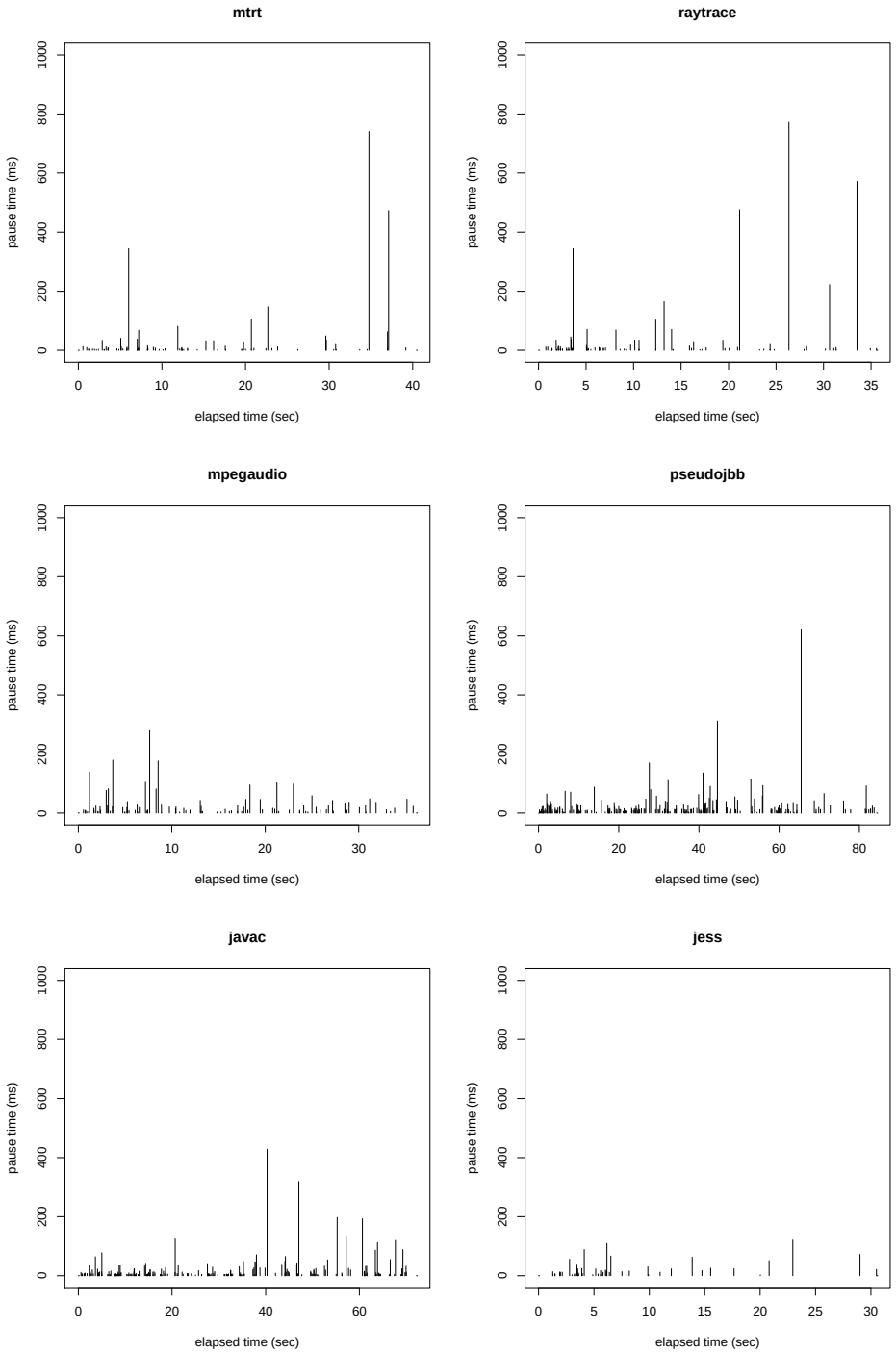

Fig. 2. Pause times due to compilation in the Adaptive system.

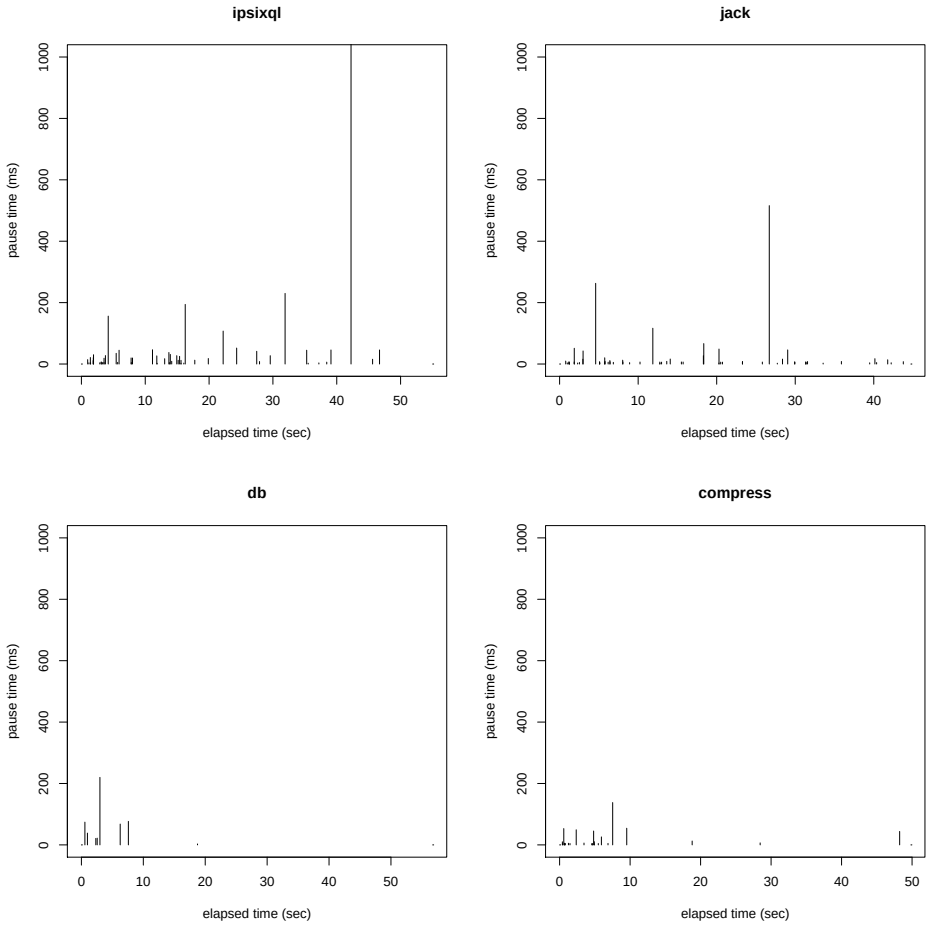

Fig. 3. Pause times due to compilation in the Adaptive system (continued).

perform clients' work to the extent possible. We try to adhere too this goal in choosing our design parameters. Another related approach to improve client performance is task migration, and we discuss some of the related work in this area in Section 6.2. However, we do not consider task migration other than compilation in this paper.

3.1 Overview

Figure 5 shows the overall client and *CS* architecture. Broadly speaking, there are three steps involved in using *CS*:

(1) Client requests download of class files via *CS*. In this respect, *CS* acts as a proxy server that relays client's request to the outside world. However, unlike a regular proxy server, *CS* may choose to annotate the class files with profile directives. These directives instruct the client as to which methods or which basic blocks within a method it

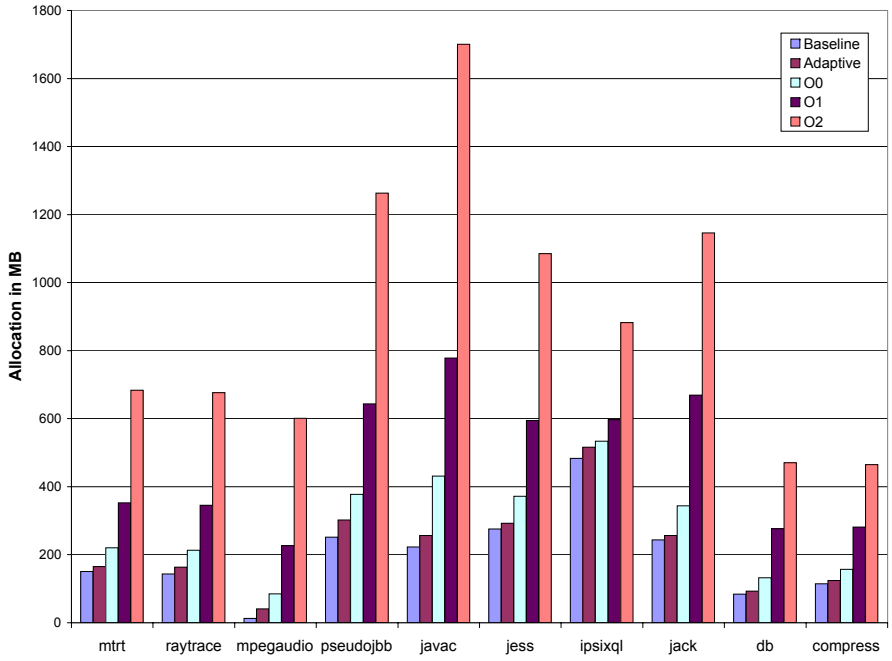

Fig. 4. Bytes allocated. The bars, from left to right, are: Baseline, Adaptive, O0, O1, and O2

should instrument and gather profile information for profile-driven optimizations. We discuss and evaluate profile-driven optimizations in more detail in Section 5.6.

(2) Client compiles and executes downloaded methods with the *baseline* compiler. The modified adaptive subsystem identifies "hot" methods and sends optimization requests to *CS*. These requests are accompanied by various information discussed in detail in Section 3.2.2.

(3) The front-end for *CS*, which we call the *CS* driver, stores individual client state data segregated by kind and described in Table I. The driver is also responsible for managing profile information, reusing optimized code, driving the optimizing compiler, and sending optimized machine code back to the clients. The installer threads on clients are responsible for receiving and installing optimized code.

In order to implement and evaluate *CS* in a reasonable time frame, we implemented both the client support and *CS* in Jikes RVM. Our current implementation is limited in that it requires that clients and *CS* run on the same architecture (IA32) and on identically built Jikes RVM boot images.

Since *CS* also acts as a proxy server as depicted in Figure 5, we make another simplifying assumption, namely that *CS* has access to all client class files. Therefore, client requests for method optimization do not need to send the bytecode instructions to *CS*. However, since the size of the bytecode instructions is usually small, and network speed does not affect client performance very much as shown in Section 5.3, this assumption is not likely to affect client performance.

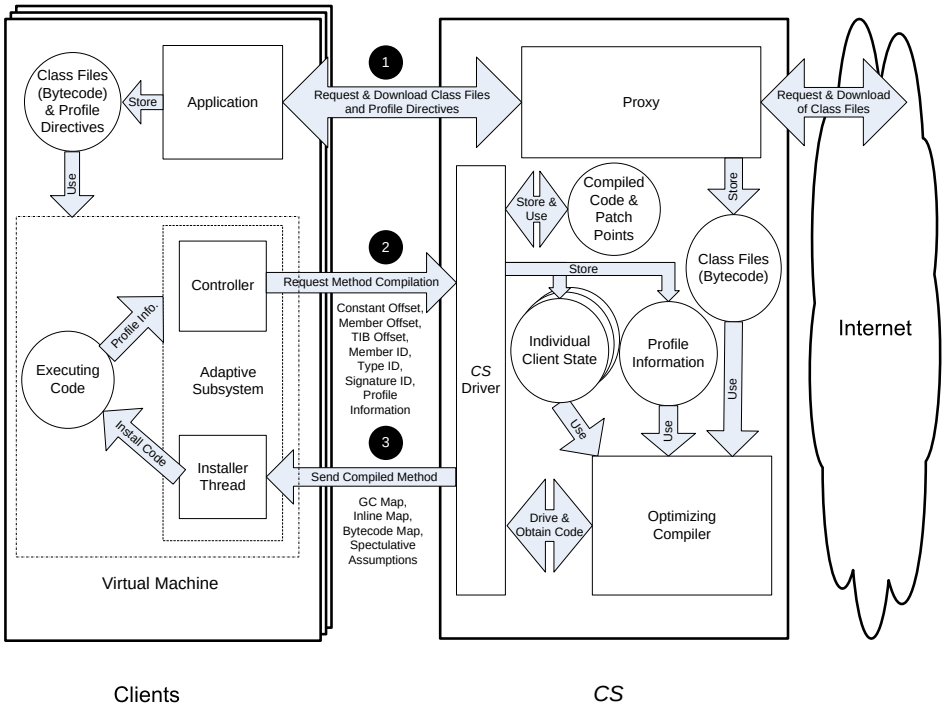

Fig. 5. Overview of *CS* and client interaction.

We discuss our design choices and our implementation of client support and *CS* in detail below.

3.2 Client Design and Implementation

3.2.1 *Method Selection.* One of the most important aspects that affects client performance is selecting which methods to optimize on *CS*. Although it would be less demanding on clients to have *CS* select which methods to optimize, having the server choose does not work well because *CS* does not have enough run-time information to make the best decision. Therefore, we delegate the task of method selection solely to clients.

While it is more efficient to optimize methods using *CS* rather than on the client, there still is some cost associated with using the server, such as collection of client state information, the overhead of communication, and installation of compiled methods. Optimizing too many methods, i.e., too eagerly, may increase this cost so significantly that it will offset any benefit gained by optimization. In any case it is impractical to optimize all methods on *CS* since it would overload the server. On the other hand, optimizing too few hot methods may eliminate any performance benefit.

Fortunately, the problem of method selection for recompilation is also present for the adaptive system and has already been addressed [Arnold et al. 2000]. Our client implementation is based on the adaptive subsystem of Jikes RVM, and we use the default runtime measurement subsystem, without any modification, to gather information about the

executing methods.

The default adaptive controller uses a cost-benefit analysis to decide whether it would be worthwhile to recompile a method using a more aggressive optimization level. The default controller assumes that a given method *m* will execute as long as it has executed thus far, and uses an offline benefit model to estimate how much time it can save. If the improved running time estimate plus the cost of optimizing *m* to a certain optimization level is less than the expected execution time of the current version of *m*, then the controller decides to recompile *m*.

The default adaptive controller's cost-benefit model is calibrated offline using a set of benchmarks. Since many variables, including network speed and server load, may fluctuate during client execution, we use a modified cost-benefit model that takes into account this variability. We express the benefit model as the speedup of our custom optimization level (Table II) over *baseline*, and calibrate it using an offline execution of our benchmarks. Thus the benefit model remains constant throughout client execution. The cost model is dynamic, and we derive it by measuring the time it takes for a given optimization request to complete. The cost is expressed in terms of the number of bytecodes compiled per unit time, and each individual client maintains their own cost model.

3.2.2 *Client State.* Once a method *m* is identified as being "hot" using our modified cost-benefit model, the adaptive controller issues a compilation request to *CS* (step 2 in Figure 5). In addition to sending a description of *m*, the adaptive controller also collects and sends *client state* corresponding to *m* and its callees to *CS*. The reason for sending state information of *m*'s callees to *CS* is because the optimizing compiler may decide to inline them into *m* and thus needs to know about their state information. The collection of client state relies on the baseline compiler, and thus if a callee has not yet been compiled by the baseline compiler, its state information is not sent to *CS*. This process is described in more detail in Section 3.3.3 where we describe inlining.

We want clients to perform minimal work, and to achieve that goal we designed *CS* to generate machine code that can be installed and executed without any modification by the client. To this end, *CS* needs to know some things about the state of the client VM. To keep it simple, client state information is sent as name (string), kind (byte), and value (integer) triples, except for profile information. This representation is not optimized for size but for simplicity. It is possible for *CS* and the client to agree upon a more compact representation such as an index into the constant pool table in a class file. However, as we will see in Section 5.3, the size of client state information is small and does not impact client performance much.

Table I summarizes the different kinds of client state information.

To avoid resending information, *CS* maintains per-client state information.

3.2.3 *Mode of Communication.* Another important design dimension that can affect client performance is the way it communicates with *CS*. Both the amount of transmitted data and the mode of communication (synchronous or asynchronous) can impact how clients perform.

Offloading compilation to a server means that clients can continue executing unoptimized machine code while they are waiting for optimized code to arrive from the server. It may be beneficial to wait for the server for long running methods since waiting and executing optimized code may be shorter than executing unoptimized code followed by optimized code. However, our experiments show that this happens rarely, and asynchronous

Table I. Different Kinds of Client State Information and Their Uses

Kind of client state	Uses
Jikes RVM Table of Contents (JTOC) offsets of float, double, long, and string constants	The optimizing compiler generates code that indexes into the JTOC (globals/statics area) to access these types of constants for various flavors of *ldc* bytecode instructions.
JTOC offsets of static fields and methods	The optimizing compiler uses these offsets to generate code for *getstatic*, *putstatic*, *invokestatic*, and sometimes *invokespecial* bytecode instructions.
JTOC offsets of type information blocks	The optimizing compiler uses these offsets, if present, to generate code for *invokespecial*, *new*, *newarray*, *anewarray*, *checkcast*, and *instanceof* bytecode instructions.
ID of unresolved fields and methods	The optimizing compiler uses these IDs to generate "unresolved" code for many different bytecode instructions.
ID of types	The optimizing compiler uses these IDs to generate code for *checkcast* and *instanceof* instructions. The optimizing compiler also uses these IDs to build exception tables.
ID of signatures	The optimizing compiler uses these IDs to generate code for *invokeinterface* bytecode instruction.
Profile information	This information may or may not be present depending on the server's profile directives. We currently support for only one type of profile information: edge counters. Different types of profile information will require different transmission formats.

communication with the server often offers better client performance. Furthermore, the problem with long running methods can also be addressed by performing *on-stack replacement* [Fink and Qian 2003] of compiled methods. Therefore, we use the asynchronous mode of communication in our current implementation of *CS*.

The use of asynchronous communication also has the benefit of reducing client pause times since a client needs to pause only to send the request and to receive and install compiled code. We explore the impact of using *CS* on pause times in detail in Section 5.2.

In addition to simple synchronous or asynchronous communication, we also investigated the idea of *split transactions*, i.e., to split an optimization request in two transactions as described below. The idea behind split transaction is that we could further reduce wait time for optimized code by sending the request as early as possible.

In the first transaction, client would identify "getting-warm" methods and send requests for these methods to *CS*. In the second transaction, the client would identify "actually-hot" methods and send requests for these methods to *CS*. Since the set of methods in "getting-warm" would include most, if not all, of the methods in "actually-hot", *CS* would simply send previously optimized code back to the client without additional delay.

We implemented the split transaction model in *CS*, and found its performance lacking for several reasons. If we set the threshold for identifying "getting-warm" methods too low, then we ended up sending too many requests to *CS*, which became overloaded and was slow at responding to the client's request for "actually-hot" methods. If we set the "getting-warm" threshold too high, then we were just delaying receipt of "actually-hot" methods by having *CS* wait to send optimized code until the second request came in.

Despite our discouraging initial results, there may be situations where split transactions

would perform well. For example, in a setting where *CS* is more capable and relatively idle, or in situations where optimization or analysis is much more expensive, split transactions may be effective at reducing client wait time.

3.2.4 *Security and Privacy.* Any client-server architecture is affected by security and privacy issues, and ours is no exception. Questions such as "How can clients trust *CS*?" or "What kinds of privacy policies should be implemented in *CS*?" are interesting but beyond the scope of our study. In our study, we assume that clients can trust *CS* to send correctly optimized code and that *CS* can trust its clients to send valid profile information. We also assume that other appropriate security and privacy policies are in place, which is a reasonable assumption to make if both client and server are behind a firewall, for example.

3.3 Server Design and Implementation

As shown in Figure 5, our *CS* implementation consists of a driver that mediates between clients and the optimizing compiler. The *CS* driver receives and stores per-client state information, segregated by kind so as to speed lookups by the optimizing compiler. It is also responsible for combining profile information received from its clients (Section 5.6). The *CS* driver also drives the optimizing compiler and relays optimized code back to the clients. In addition to machine code, it also sends GC maps, inline maps, bytecode maps, and the list of classes that must not be subclassed and methods that must not be overridden back to its clients (these have to do with speculative optimizations, discussed below).

3.3.1 *Optimization Selection.* The Adaptive system allows multi-level recompilation using the various optimization levels. For example, a method which has been recompiled using optimization level O0 may later be recompiled using a higher optimization level, if that is estimated to be beneficial. Since using the same multi-level recompilation strategy in *CS* may result in higher server load and increased network traffic, we limit the number of recompilations per method to one. However, instead of using a predefined optimization level such as O0, O1, O2, we use a custom optimization level based on our earlier study on the effectiveness of optimizations [Lee et al. 2004].

In that earlier work, we categorized an optimization as "should be implemented", "may be implemented", and "should not be implemented" by using application and kernel benchmarks that stress tested each single optimization as well as by evaluating optimizations individually and in the presence of other optimizations. These recommendations were based on the cost-benefit ratio of an optimization also taking into account its potential benefit. Our custom optimization level used in this study includes all optimizations in the "should be implemented" and "may be implemented" categories, and are listed and described in Table II. Note that Table II corresponds to Tables VI and VII from our earlier paper [Lee et al. 2004]. These optimizations have high potential for benefit with low to moderate compilation cost.

3.3.2 *Speculative Optimizations.* One may need to apply some speculative optimizations, such as guarded inlining, in order to generate best quality code. However, speculative optimizations make some assumptions about the state of the world, and these assumptions need to hold for correct execution of programs. The Adaptive system maintains and checks the validity of these assumptions, and when these assumptions are violated, it re-optimizes the method in question (as we previously alluded). The same approach is used in the *CS* setting: in order to accommodate speculative optimizations, the server sends a list of class

Table II. Optimizations included in our custom optimization level

Optimization	Description
inl_new	Inline all run-time system routines for allocating arrays and scalars. Occurs during the expansion of high-level IR into low-level IR.
inl	Inline statically resolvable calls. Its heuristics include: inline if callee's estimated size is less than 11 machine instructions, do not inline if callee's estimated size is greater than 23 machine instructions, do not inline more than a maximum inlining depth of 5, and do not inline if the absolute size of the caller due to inlining is greater than 4096 machine instructions. We use the default values and do not use inline plans to control inlining decisions.
preex_inl	Perform pre-existence based inlining when the single implementation assumption is met and the receiver object is extant, using invariant argument analysis [Detlefs and Agesen 1999].
guarded_inl	Perform guarded inlining of virtual calls using code patching (default on IA32) or class or method test (default on PPC). It uses the same size heuristics as inl.
guarded_inl_ifc	Speculatively inline interface calls when there is a unique implementation of the interface method, using guards as in guarded_inl. It uses the same size heuristics as inl.
scalar_repl_aggr	Perform scalar replacement of aggregates by treating fields of objects and elements of arrays that do not escape a method like normal scalar variables. It uses a flow insensitive escape analysis, which determines that the referent of a symbolic register (local or temporary variable) escapes a method when the register is stored in memory, assigned to another register, or returned from a method. The escape analysis assumes the worst case about calls.
monitor_removal	Remove unnecessary synchronizations of objects that do not escape a thread using a flow insensitive escape analysis. The escape analysis is similar to that for scalar_replace_aggregates except that rather than assuming that passing a register to a call causes the referent of the register to escape, it uses a summary of the callee. If the compiler does not know who the callee is (e.g., due to dynamic dispatching or dynamic class loading), then it assumes that the referent object escapes.
static_splitting	Split the control flow graph to create hot traces based on static heuristics (e.g., using hints left by guarded inlining) to avoid control flow merges that inhibit other optimizations.
gcse	Perform global common subexpression elimination on LIR (low-level machine independent IR) using the dominator tree to determine positions for operations.
loc_copy_prop	Perform local copy propagation (i.e., within basic blocks) using flow sensitive analysis.
loc_constant_prop	Perform local constant propagation (i.e., within basic blocks) using flow sensitive analysis.
loc_sr	Perform local scalar replacement of loads of fields (i.e., within basic blocks) using flow sensitive analysis. Call and synchronization instructions invalidate all memory locations.
loc_cse	Perform local common subexpression elimination (i.e., within basic blocks) using flow sensitive analysis.
loc_check	Eliminate redundant null, array bounds, and zero checks using a local (i.e., within basic blocks) flow sensitive analysis.

names and method descriptions back to the client that affect the validity of a compiled method. The client is then responsible for checking that these classes and methods are not subclassed and overridden, respectively. Even though *CS* has access to all the classes a client may execute, *CS* cannot perform these checks since it does not know which classes a client may load dynamically in the future. *CS* could assume that any class that a client has access to could potentially be subclassed and its methods overridden, but that would result in code that is too conservative. If a client finds that these assumptions are violated, the server sends non-speculatively optimized code to the client upon request. In Section 5.1.3, we presents results that show the effectiveness of implementing speculative optimizations in *CS*.

3.3.3 *Inlining.* We have found that one of the most effective optimizations is method inlining [Lee et al. 2004], and *CS* sometimes is not as aggressive as the Adaptive system in inlining, due to its limited knowledge about client states. In order for *CS* to inline a method *m1* into another method being compiled, *m2*, *CS* needs to have full information about *m1* (i.e., *m1*'s state information shown in Table I). Unless *m1* has been baseline compiled on the client already, it is not possible for *CS* to know about its state. Because clients ask only for "hot" methods to be compiled by *CS*, there is a high probability that most of the frequent method invocations contained within these "hot" methods have already been executed (and therefore compiled by the Baseline compiler) at least once, and thus all of the frequent call sites can be inlined by *CS*. The fact that *CS* cannot inline as aggressively as the optimizing compiler in *Opt* or *Adaptive* may seem like a disadvantage, but in fact, it is not, because the filtering of call sites that happens on clients works as a primitive feedback-directed adaptive inlining mechanism. Due to this filtering, *CS* does not bother to inline infrequently executed call sites; inlining them may result in worse instruction cache behavior.

3.3.4 *Linking.* While producing generic machine code that can be used on different clients has its benefits, it can be costly because each individual client must link the machine code before executing it. Therefore, in our current implementation, *CS* produces optimized machine code that clients can simply download, install, and execute without any modification. Because we are linking on *CS*, this requires that *CS* know some things about each client's state (Section 3.2.2). For example, if a static field is referenced in a method, the server will use the offset of the corresponding static field obtained from the client to generate code for that particular method.

3.3.5 *Compiled Code Reuse.* To reduce compilation time and server load even further, we also investigate the issue of code reuse. To reuse code, *CS* makes a scan of generated machine code to find patch points: patch points are name (string) and machine code offset (integer) pairs that describe which bytes have to be patched before compiled code can be delivered to another client. When a request for the same method comes in, *CS* simply walks over the list of patch points modifying the compiled code as necessary.

However, it is sometimes not enough to patch compiled code in this manner because there may be other differences between clients. For example, depending on whether a static field referenced in a method has been resolved on the client, the server may need to generate additional code to resolve the field before accessing it. In the case where clients' states differ, the server simply re-optimizes the method in question. Another approach would be to generate lowest common denominator code that could be used universally

Table III. Benchmark descriptions.

Name	Running time	Compilation	Lines	Description
mtrt	42.71 s	23.3%	3,751	Multi-threaded version of raytrace
raytrace	37.40 s	18.5%	3,751	Raytracer
mpegaudio	36.59 s	16.9%	not avail.	Decompresses audio files
pseudojbb	85.31 s	14.0%	26,587	SPECjbb2000 modified to perform fixed amount of work
javac	73.39 s	13.7%	25,211	Java compiler from the JDK 1.0.2
jess	31.05 s	9.7%	10,579	Java Expert Shell
ipsixql	55.91 s	9.1%	not avail.	Performs queries against persistent XML document
jack	46.69 s	7.6%	not avail.	Parser generator, earlier version of JavaCC
db	57.41 s	2.5%	1,028	Database operations on a memory resident DB
compress	47.32 s	2.1%	927	Modified Lempel-Ziv method

across different clients, but such code would perform relatively poorly due to additional checks.[1]

4. EXPERIMENTAL METHODOLOGY

To evaluate the concept of *CS*, we have implemented a server and corresponding client system in Jikes RVM. In this section, we describe our infrastructure, benchmarks, and measurement methodology.

4.1 Infrastructure

Our implementation of the server and client is based on Jikes RVM version 2.2.2 from IBM Research [Burke et al. 1999]. Jikes RVM is designed to support research into virtual machines and includes an aggressively optimizing just-in-time compiler and an adaptive compilation system [Arnold et al. 2000].

We base our client implementation on the Adaptive system with a copying garbage collector with the modifications outlined in Section 3.2.1. That is, the cost-benefit model is no longer static but is updated dynamically throughout client execution based on server load and network conditions (i.e., overall server response time). Instead of having a compilation thread, we implement a separate thread for downloading and installing optimized code from the server. The server is implemented as a stand-alone application that runs on top of Jikes RVM and acts as a driver to the Jikes RVM's optimizing compiler. The server is responsible for maintaining client state information segregated by kind, for loading and resolving classes as needed, for maintaining a list of compiled methods for reuse, and for patching compiled code as needed.

4.2 Benchmarks

We report results for the SPECjvm98 benchmarks [Standard Performance Evaluation Corporation (SPEC) 1998] (using input size 100), plus ipsixql, an XML database engine, and SPECjbb2000 [Standard Performance Evaluation Corporation (SPEC) 2000] modified to perform a fixed number of transactions, which we refer to as pseudojbb. Table III describes these benchmark programs. The column labeled "Running time" shows the running time of the benchmark programs in the *Adaptive* configuration, and the "Compilation" column shows the compilation time as a percentage of total execution time in that configuration. The table is sorted by the "Compilation" column in decreasing order. That is, the top entries in Table III are more likely to benefit from *CS* than the bottom ones since there is more compilation overhead to be removed.

4.3 Measurement Methodology

We conducted our experiments on a 500 MHz Pentium III processor with 512 MB of memory running as a client, and a 2.4 GHz Pentium 4 processor with 2 GB of memory running as a server. The speed difference between the two machines is about a factor of five. We use different network speeds between client and server ranging from 56 Kbps to 100 Mbps. Our initial networking configuration uses a 100 Mbps connection.

We perform each run-time measurement ten times and report the average running time along with its standard deviation. We conducted experiments using the default heap size of 64 MB for all benchmarks, except for pseudojbb which was run with a 128 MB heap size because it would otherwise throw an *OutofMemoryException* exception. Since clients using *CS* allocate less dynamic memory, they are likely to perform better with smaller heap size.

5. RESULTS

We now present and discuss detailed results evaluating the performance of both the client and the server in *CS*. Section 5.1 gives the overall client speedup when using *CS*. Section 5.2 presents improvements in pause times. Section 5.3 evaluates the effect of a slower network on client performance. Section 5.4 shows client memory savings gained by offloading compilation onto a server. Section 5.5 shows the scalability of *CS* and the effect of code reuse. Finally, Section 5.6 presents preliminary results that show the feasibility of performing profile based optimizations in the *CS* setting.

5.1 Overall Speedup

Figures 6 and 7 give the running time of different configurations for our benchmarks. The configurations we consider are:

—*Baseline*: This configuration uses the Baseline compiler exclusively and thus is the worst performing configuration for all benchmarks. The compilation cost is negligible in this configuration.

—*CS-NoSpec*: This configuration corresponds to a client using *CS*. The server performs no speculative optimization.

[1]One could also use a cached version if it makes assumptions consistent with those of the client now requesting the code but it may be costly.

Table IV. Summary of Speedup Data

Benchmark	Speedup over *Adaptive-OSR*	Compilation overhead
mtrt	44.8%	23.3%
raytrace	42.9%	18.5%
mpegaudio	29.5%	16.9%
pseudojbb	20.4%	14.0%
javac	32.3%	13.7%
jess	12.4%	9.7%
ipsixql	10.1%	9.1%
jack	18.3%	7.6%
db	-3.6%	2.5%
compress	-1.8%	2.1%
Mean	20.5%	11.7%

—*CS-Spec*: This configuration corresponds to a client using *CS*. The server performs all optimizations listed in Table II, including speculative optimizations.

—*Adaptive-NoOSR*: This is the Adaptive system of Jikes RVM without On-Stack Replacement.

—*Adaptive-OSR*: This is the default adaptive system of Jikes RVM *with* On-Stack Replacement.

—*Opt*: This configuration uses the optimizing compiler at optimization level O1. We use optimization level O1 since it outperforms O2 for most benchmarks. We do not include any compilation cost for this configuration, and thus this configuration is not realistic. However, we can gauge how good other configurations are performing by comparing them to this *Opt* configuration.

The height of each bar represent its running time in seconds, and the number within a given bar represents its running time as a ratio over the running time of the *Adaptive-OSR* configuration. We compare the run-time performance of *CS-Spec* with that of *Adaptive-OSR* since *Adaptive-OSR* is the default realistic configuration. The error bars are placed at one standard deviation away from the mean in all bars. Note that sometimes these error bars seem to be missing because the errors are too small to see. Running times of *CS-Spec* configurations are usually bounded by the *Baseline* and *Opt* configurations.

From Figures 6 and 7, we see that the effect of using *CS* ranges from speed improvements of up to 45% to a slowdown of up to 3.6%. Table IV summarizes the speedup data. The second column in Table IV, labeled "Speedup over *Adaptive-OSR*," shows the percentage improvement of *CS-Spec* over the *Adaptive-OSR* configuration. The third column, labeled "Compilation overhead," shows compilation time as a percentage of total execution time in the Adaptive system (also shown graphically in Figure 1).

Not only is *CS* effective at removing compilation cost (Sections 5.2 and 5.3), but it is also effective at improving client performance that goes beyond simply removing compilation cost. We also notice that *CS* sometimes degrades performance, and we explore both of these issues in detail below.

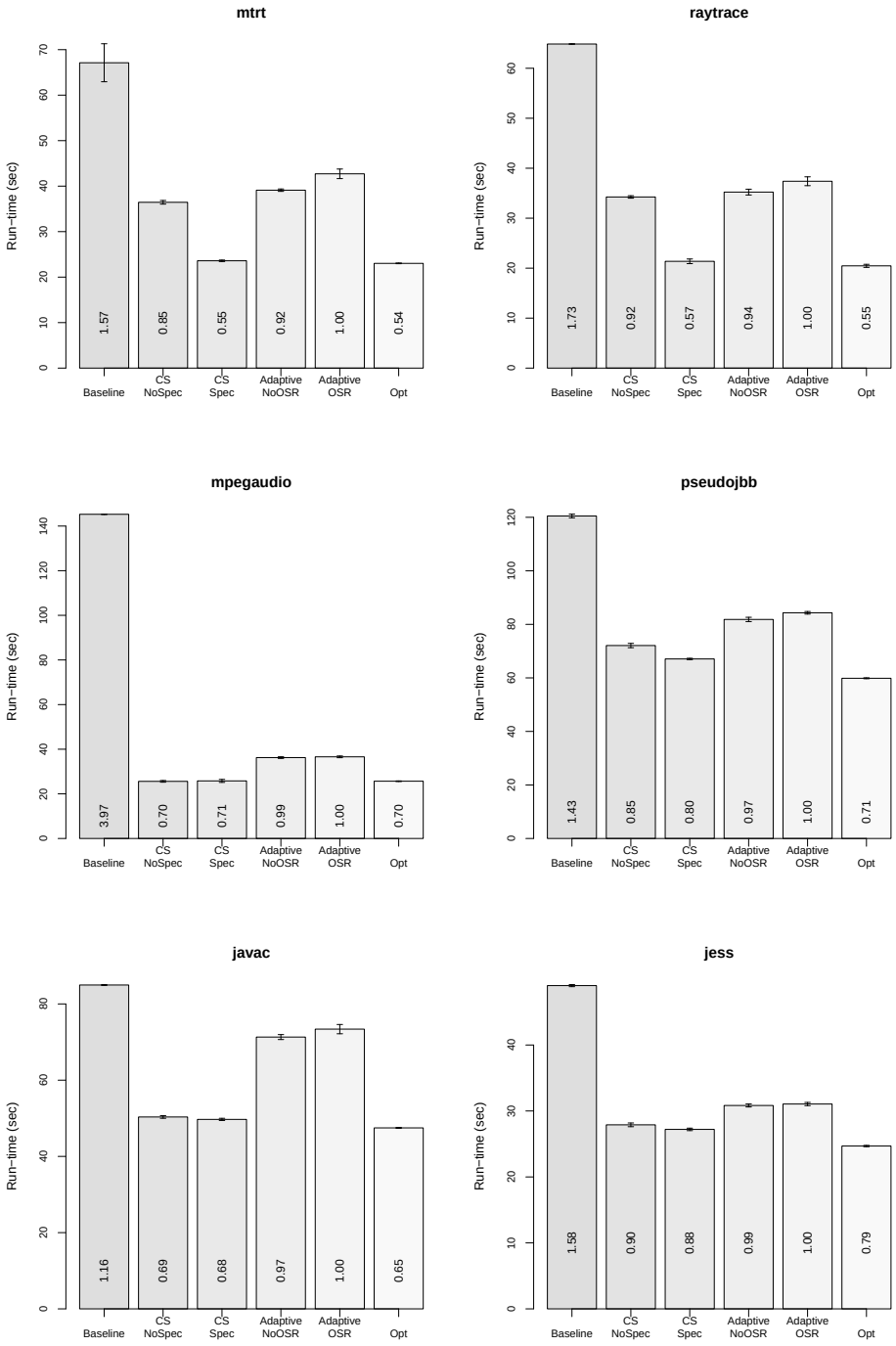

Fig. 6. Running time of different configurations.

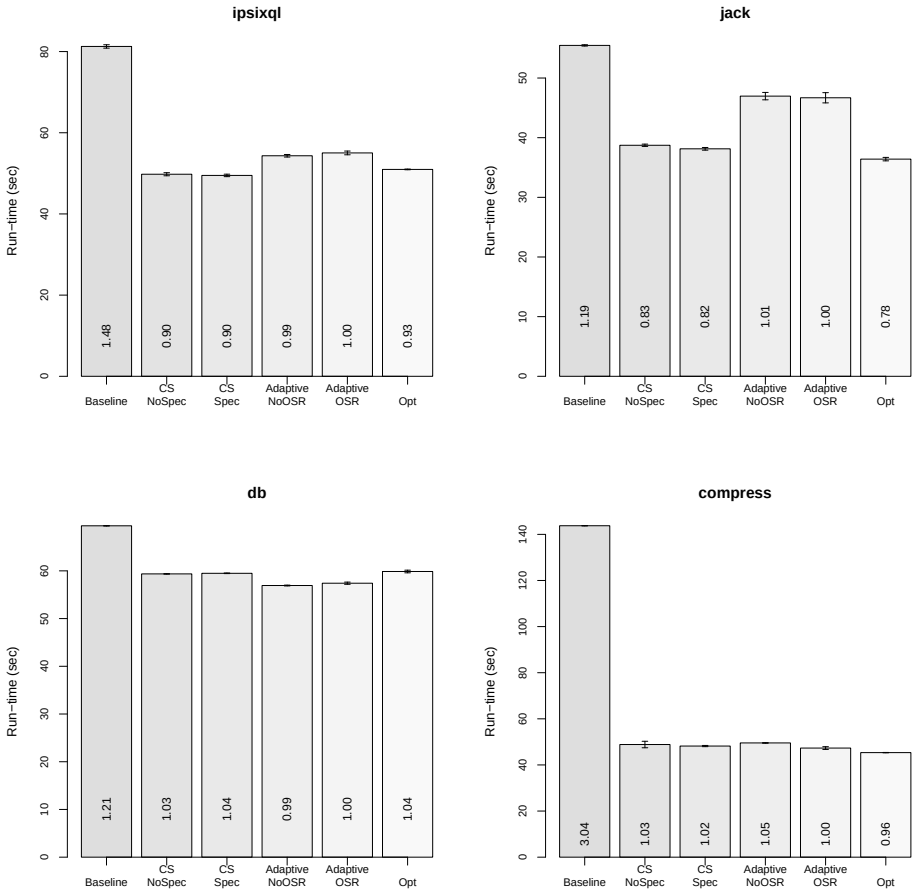

Fig. 7. Running time of different configurations (continued).

5.1.1 *Sources of Speedup.* Using *CS* results in client speedup that goes beyond removing compilation cost. This is explained by the fact that the cost-benefit model of *CS-Spec* is more aggressive (i.e., cost of compiling is lower) than that of *Adaptive-OSR* because we run the server on a more capable machine. The lower cost of *CS-Spec* means that there are more methods optimized in *CS-Spec* than in *Adaptive-OSR*. Furthermore, most methods in *Adaptive-OSR* are compiled to optimization level O0 because the cost in *Adaptive-OSR*'s cost-benefit model is too high to recompile methods at higher optimization levels. In contrast, *CS-Spec* compiles methods using optimizations listed in Table II, which includes many optimizations from optimization level O1 and one optimization from optimization level O2. The higher quality and number of optimized methods generated by *CS-Spec* result in performance improvement that goes beyond reducing compilation cost. Furthermore, clients using *CS* perform fewer dynamic allocations, which results in better memory

behavior (Section 5.4).

Another interesting observation we can make from Figures 6 and 7 is that for ipsixql, clients using *CS* actually outperform the *Opt* configuration. Recall that *Opt* is an ideal configuration where all methods are compiled by the optimizing compiler at optimization level O1 without incurring any compilation overhead. This rather surprising behavior can be explained by the difference in the quality of generated code.

First of all, the set of optimizations that *CS-Spec* uses is slightly different from that of *Opt*. In particular, we include one O2 level optimization (global common subexpression elimination) in our custom optimization level based on results from our earlier work [Lee et al. 2004], which results in slightly better optimized code. Second, due to the filtering that affects inlining decisions (as discussed in Section 3.3.3), *CS-Spec* generates code that may have better instruction cache behavior.

5.1.2 *Slowdown.* *CS* sometimes degrades client performance. This performance degradation can be explained by the overhead in using *CS*. This overhead includes the cost of collecting and sending client state, and the cost of receiving and installing compiled methods. Due to this overhead, not all of the savings in compilation time translate into performance improvement. This overhead is more readily seen when there is not much savings opportunity to begin with (i.e., compilation cost in *Adaptive-OSR* is low), such as in db and compress.

5.1.3 *Speculative Optimizations.* The initial implementation of *CS* did not include speculative optimizations because it required communicating the compile-time assumptions made by *CS* to clients, and invalidations discovered by clients back to *CS*. Furthermore, when there are invalidations, *CS* re-optimizes methods without speculative optimizations, and clients need to download them again, introducing further overhead.

The *CS-NoSpec* configurations in Figures 6 and 7 show the effect of omitting speculative optimizations in *CS*. There are two things we can notice from the height of the bars. The first is that, while speculative optimizations require further communication between clients and *CS*, the cost of that communication is fairly low. The reason for this is that there are relatively few invalidations in practice. For example, the number of invalidations for javac, which optimizes over 200 methods, is only 4. We can also notice that speculative optimizations work reasonably well, so that it is worthwhile to implement them in *CS* despite of their added complexity and communication cost.

5.1.4 *On-Stack Replacement.* On-stack replacement (OSR) [Chambers and Ungar 1991; Hölzle and Ungar 1994; Fink and Qian 2003] allows replacement of currently executing methods with newly compiled methods. One of the situations where OSR is useful is when there is a long running method that needs to be replaced with better optimized compiled code. Jikes RVM's Adaptive system implements OSR [Fink and Qian 2003], and to determine whether it would be worthwhile to implement OSR in *CS*, we examine its potential in Figures 6 and 7. The configuration labeled *Adaptive-NoOSR* in Figures 6 and 7 shows the effect of disabling OSR in the Adaptive system. There are only two benchmarks, compress and jack, where disabling OSR results in performance degradation (5% and 1%, respectively). For other benchmarks, disabling OSR does not make much difference and sometimes improves performance (by up to 8% in mtrt). While implementing OSR in *CS* may give more benefit than in the Adaptive system since optimized code would be available sooner than it would in the Adaptive system, results from Figures 6 and 7 do not seem

Table V. Summary of Pause Times

	CS client		*Adaptive-OSR*		*CS* client over *Adaptive-OSR*	
Benchmark	Mean (ms)	Maximum (ms)	Mean (ms)	Maximum (ms)	Mean (ratio)	Maximum (ratio)
mtrt	2.47	52.43	32.97	741.27	0.07	0.07
raytrace	3.61	55.50	39.49	772.06	0.09	0.07
mpegaudio	0.97	27.03	26.89	279.09	0.04	0.10
pseudojbb	2.05	83.09	23.98	620.81	0.09	0.13
javac	1.82	71.79	20.38	428.47	0.09	0.17
jess	1.53	48.41	22.70	120.96	0.07	0.40
ipsixql	2.96	45.29	53.54	1367.35	0.05	0.03
jack	1.39	41.22	24.50	515.93	0.06	0.08
db	0.57	1.23	52.82	219.93	0.11	0.01
compress	2.58	40.61	18.97	137.90	0.14	0.29
Mean	2.00	46.66	31.67	520.38	0.08	0.19

to justify the extra effort in implementing OSR in *CS*.

5.2 Pause Times

In addition to improvements to total running time, *CS* should also be effective in removing (or shortening) some of the run-time pauses due to dynamic compilation. Section 2.2 showed that there were a number of long pauses ($>$ 100 ms) in the Adaptive system. In this section, we present pause times in *CS* clients and compare them to those in the Adaptive system.

Figures 8 and 9 show the pause times due to dynamic compilation in *CS* clients, similar to Figures 2 and 3. The x-axis represents the actual execution time for a particular benchmark, and the y-axis show the pause times in seconds. Note that the scale of y-axis in Figures 8 and 9 is the same as the one used in Figures 2 and 3, and thus they can be compared directly.

One of the differences we can see between the pause times of *CS* clients and those of the adaptive system is the shorter height of the bars in *CS* clients. This is to be expected since compilation is being off-loaded to *CS*, and thus clients are pausing only to request compilation and to receive and install optimized code from the server. *CS* clients are able to perform useful work while waiting for server to finish compilation of requested methods.

Figure 10 shows the comparison graphically using box plots. The box plots show the median, and the lower and upper quartiles of pause times. Note that the y-axis has been clipped at 200 ms to make the charts more readable. We see from Figure 10 that, in addition to having drastically lower mean, the bars that correspond to *CS* clients also have much lower variability, not only for a given benchmark but across all benchmarks. The lower variability may be appealing for soft real-time systems. Table V summarizes pause time differences between *CS* clients and the Adaptive system, and it shows that we are able to reduce mean pause times by an average of 92% and maximum pause times by an average of 81% by using *CS*.

While knowing the mean and maximum pause times of a system is useful, they are

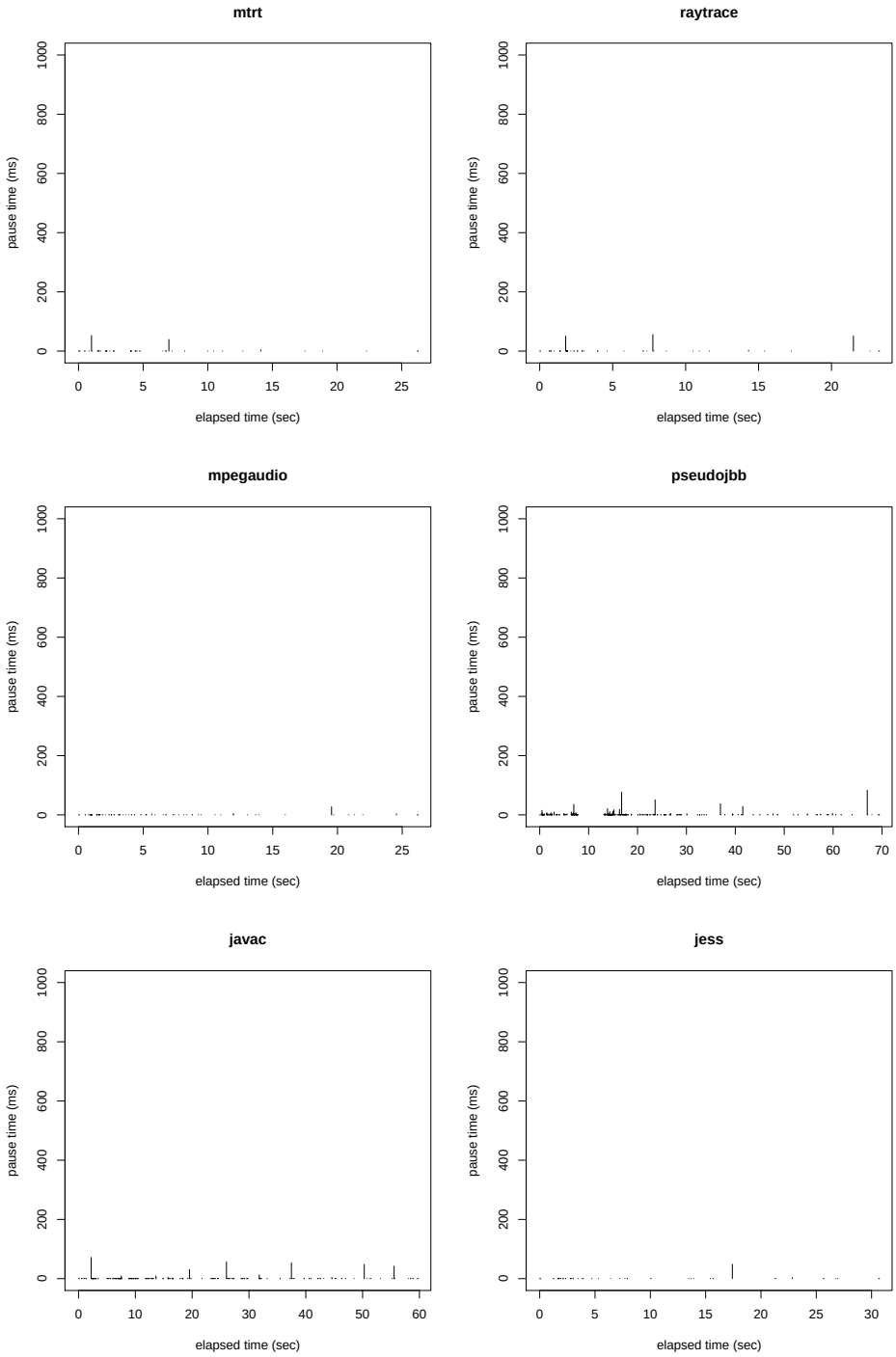

Fig. 8. Pause times due to compilation in *CS* clients.

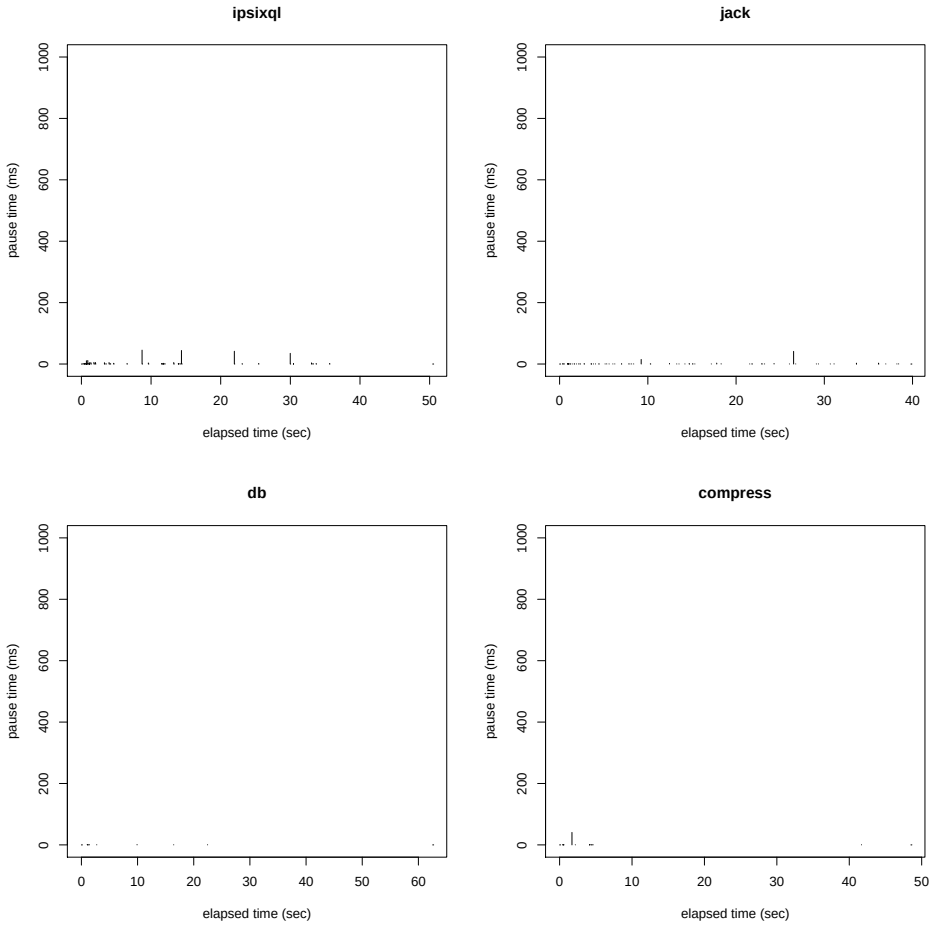

Fig. 9. Pause times due to compilation in *CS* clients (continued).

not sufficient to characterize the occurrences of the pauses because they do not say how frequent or clustered the pauses are. In order to address this limitation and to evaluate the real-time behavior of their collector, [Cheng and Blelloch 2001] introduced the concept of *minimum mutator utilization* (MMU). In their context, *utilization* is defined as the fraction of time that the mutator executes in a given time window, and minimum mutator utilization denotes the minimum amount of processor time available to the mutator for a given window size throughout the program execution. Since we are not evaluating garbage collectors, we use the term minimum utilization rate (MUR) to denote the fraction of processor time accessible to non-compilation threads for a given time window on a client, and use MUR to evaluate *CS* clients. We present the results in Figures 11 and 12.

The x-intercept shows the maximum pause time, because the MUR will be zero up until the time window gets bigger than the maximum pause time. Overall, we see that the MUR

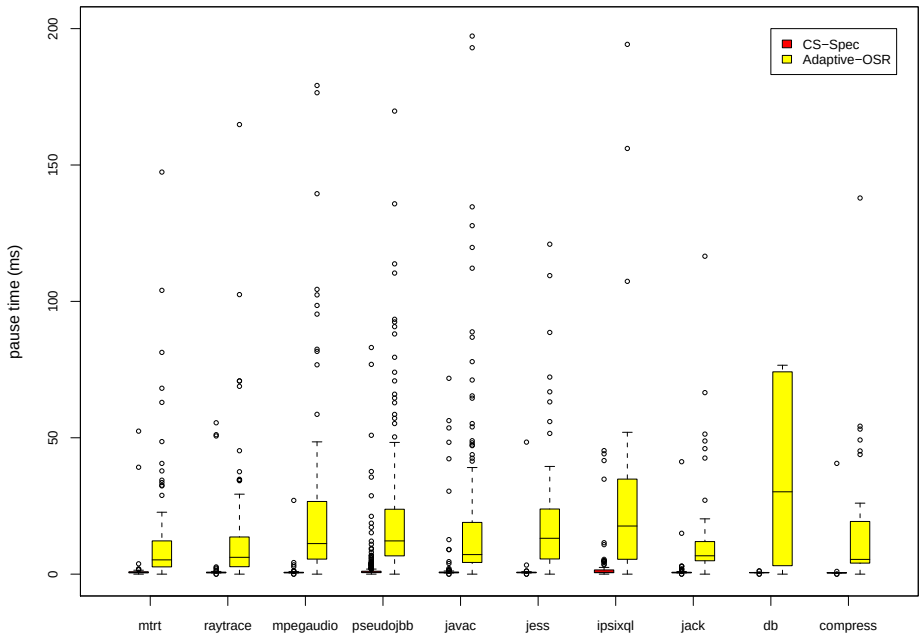

Fig. 10. Comparison of pause times between *CS* clients and the Adaptive system.

curves of *CS* clients have a smaller x-intercept value and rise much more quickly than those of the Adaptive system.

For time windows of 50 ms and 100 ms, the MUR of *CS* clients ranges from 0.00 to 0.96 and 0.16 to 0.98 respectively. The MUR of the Adaptive system remains at 0 for both 50 ms and 100 ms time windows. In other words, for a given time window size of 100 ms, clients using *CS* spend at least 16 ms to 98 ms in non-compilation threads. The higher MUR of *CS* clients may be desirable in interactive or real-time systems.

The asymptotic y-value of the MUR curves gives the overall fraction of time available for user computation over long time scales. The curves for *CS* clients show that they reach the asymptote faster than in the Adaptive system, and generally have 90% or more of the time available for user computation. In contrast, the Adaptive system runs require larger time scales to reach their asymptotic value, and this value appears to be lower than for *CS* runs (since the curves presented have not all attained the asymptotic value it is hard to tell; however, we know that *CS* generally improves performance over running the Adaptive system locally).

5.3 Network Speed

As stated in Section 4.3, so far we have assumed that the client and server are networked in a local area network using a 100 Mbps connection. In this section, we explore the effect of different network speeds on client performance.

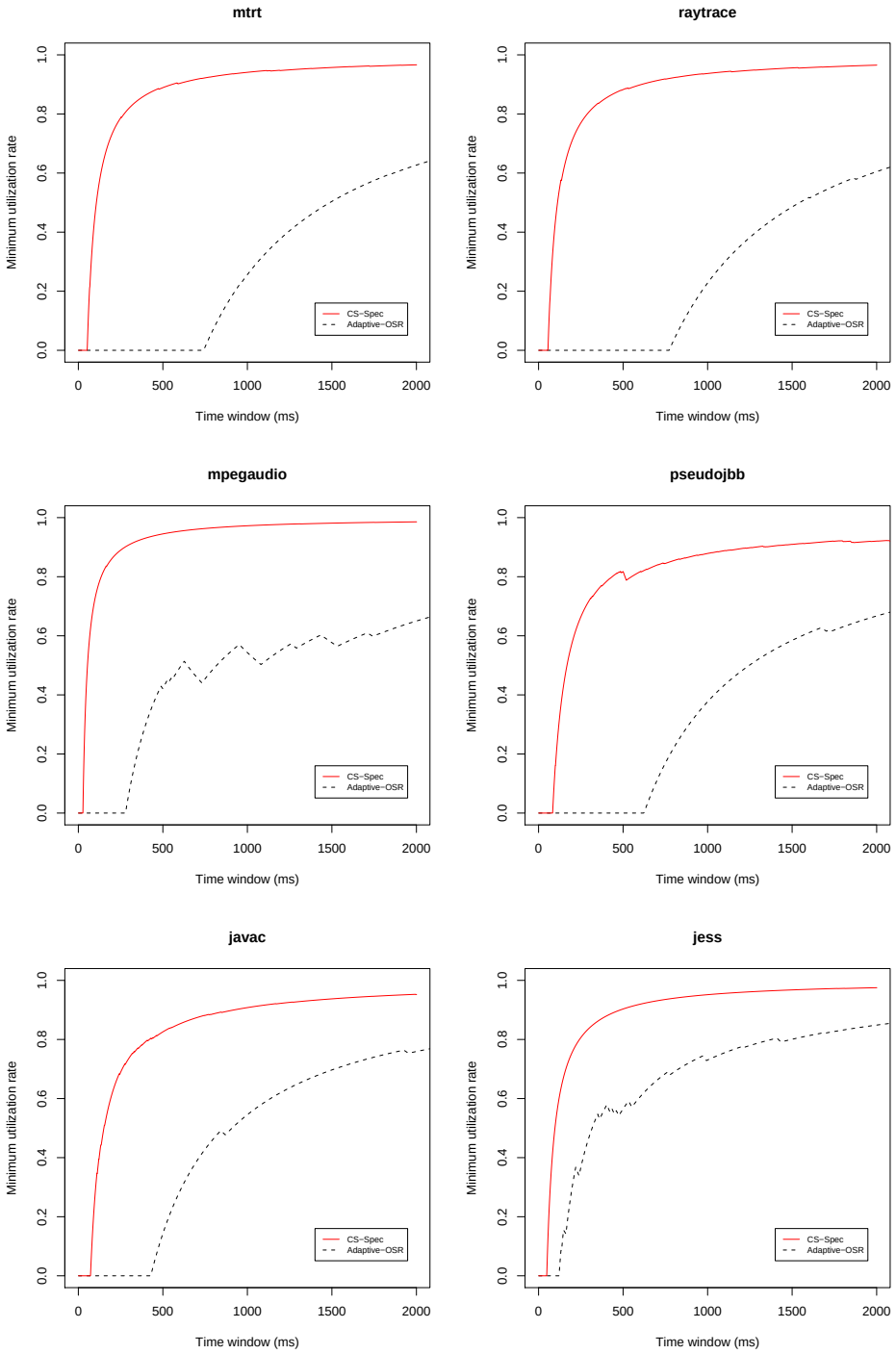

Fig. 11. Minimum utilization rate of *CS* clients and the Adaptive system.

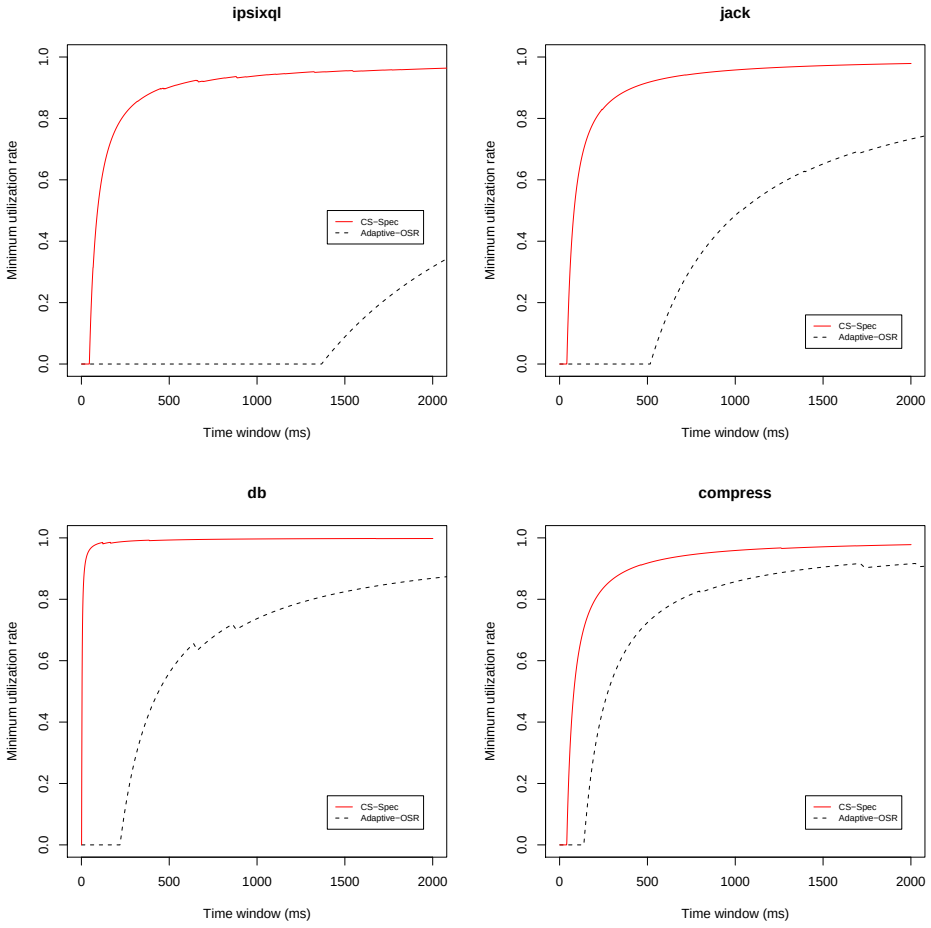

Fig. 12. Minimum utilization rate of *CS* clients and the Adaptive system (continued).

We use the *tc* tool from the *IPROUTE2* utility suite [Kuznetsov 1998] to manipulate kernel structures controlling the IP networking configuration. *tc* can be used for traffic control, and we use it to limit the network speed between client and server. Table VI shows the network speeds we consider.

We repeated our experiments with different network speeds and plot the results in Figure 13. The x-axis, which is on a logarithmic scale, represents different network speeds in decreasing order, and the y-axis presents running time in seconds. Each line represents a single benchmark, and each point shows the mean of ten measurements for a particular configuration. As before, we place error bars at one standard deviation from the mean in both directions. Most of the time the standard deviation is so small that it is difficult to see the error bars. Note that as we decrease the network speed, there is more variability between runs (i.e., longer error bars). This is to be expected since slightly different re-

Table VI. Different Network Speeds Considered

Speed	Description
100 Mbps	This is the network speed of Fast Ethernet, often used in local area networks.
10 Mbps	This is the network speed of Ethernet. It is a good approximation to the speed of the *802.11b* wireless protocol (11 Mbps).
1 Mbps	This is typical cable modem and DSL speed. It also represents the speed of the Home Phoneline Network Alliance (HomePNA) 1.0 standard as well as the speed of the *802.11* wireless protocol.
512 Kbps	This represents congested cable modem and DSL speed.
128 Kbps	This represents typical ISDN speed.
56 Kbps	This represents a good dialup modem speed.

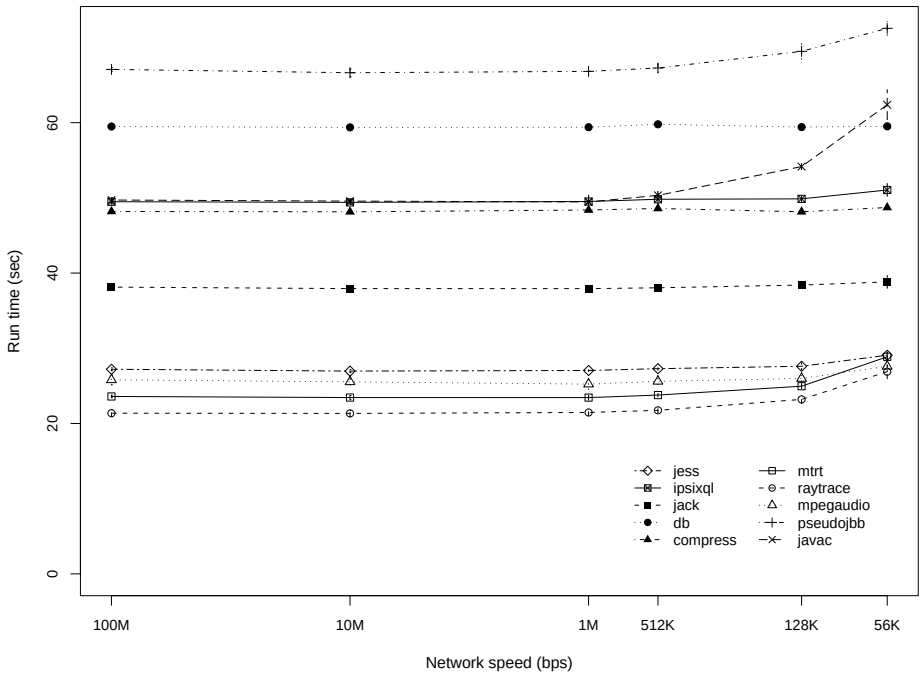

Fig. 13. Overview of effects of network speed on *CS* clients.

compilation decisions may result in rather large differences in the arrival time of compiled methods.

We can see from Figure 13 that network speed does not affect client performance much until it reaches the dialup modem speed of 56 Kbps. Even at 56 Kbps, the average run-time performance degradation compared to the 100 Mbps network is only 10%. The maximum

Table VII. Dynamic memory allocation.

Name	*CS* client (MB)	*Adaptive-OSR* (MB)	Savings (%)
mtrt	154.78	164.94	6.16
raytrace	152.35	163.55	6.85
mpegaudio	16.73	40.96	59.16
pseudojbb	255.28	302.25	15.54
javac	233.03	256.38	9.11
jess	276.89	292.78	5.43
ipsixql	484.65	515.98	6.07
jack	246.56	256.70	3.95
db	85.76	93.29	8.07
compress	116.06	124.16	6.53
Mean	202.21	221.10	8.54

performance degradation of 26% is observed for raytrace.

To put this performance degradation due to slower network speed in perspective, we plot running times of each individual benchmark separately along with running times of its *Baseline*, *Opt*, and *Adaptive-OSR* configurations in Figures 14 and 15. The raytrace benchmark, which suffers the most performance degradation at 56 Kbps compared to 100 Mbps, still outperforms the *Adaptive-OSR* configuration by 28%. In fact, most of the benchmarks still outperform *Adaptive-OSR* configuration even at the 56 Kbps connection speed. The only exceptions are db and compress, but neither outperformed the *Adaptive-OSR* configuration to begin with, even at 100 Mbps. Running times of db and compress at 56 Kbps degrade by only 0.1% and 1% respectively compared to those at 100 Mbps, since there is only a handful of methods that need to be transferred.

In summary, for those benchmarks that make heavy use of *CS*, the run-time degradation due to a slower network is relatively small compared to the benefits of using *CS*, and thus they are still able to outperform the *Adaptive-OSR* configuration. On the other hand, for those benchmarks that do not benefit much from *CS* to begin with (i.e., have little compilation cost in *Adaptive-OSR*), slowing down the network has insignificant effect on their running time because so little data is transferred between the client and the server.

5.4 Memory Usage

CS clients no longer need the optimizing compiler since *CS* performs all optimizing compilations for them. The absence of the optimizing compiler results in a memory footprint savings of about 14 MB compared to that of the Adaptive system. To measure the size of the optimizing compiler, we measured the size of Jikes RVM bootimage with and without the optimizing compiler and took the difference between the two bootimages. Considering that the size of the bootimage for *FastAdaptiveSemiSpace* is approximately 36 MB, this memory footprint reduction is significant: 39%.

In addition to the reduction in instruction space, *CS* clients also perform fewer dynamic allocations since they do not need to allocate the dynamic data needed by the optimizing compiler. We measured the number of dynamically allocated bytes in *CS* clients and in the *Adaptive-OSR* configuration and show the results in Table VII.

The reduction in the number of bytes allocated at run time ranges from 4% to 59% with a mean reduction of 8.5% over that of *Adaptive-OSR*. Programs may well have better cache

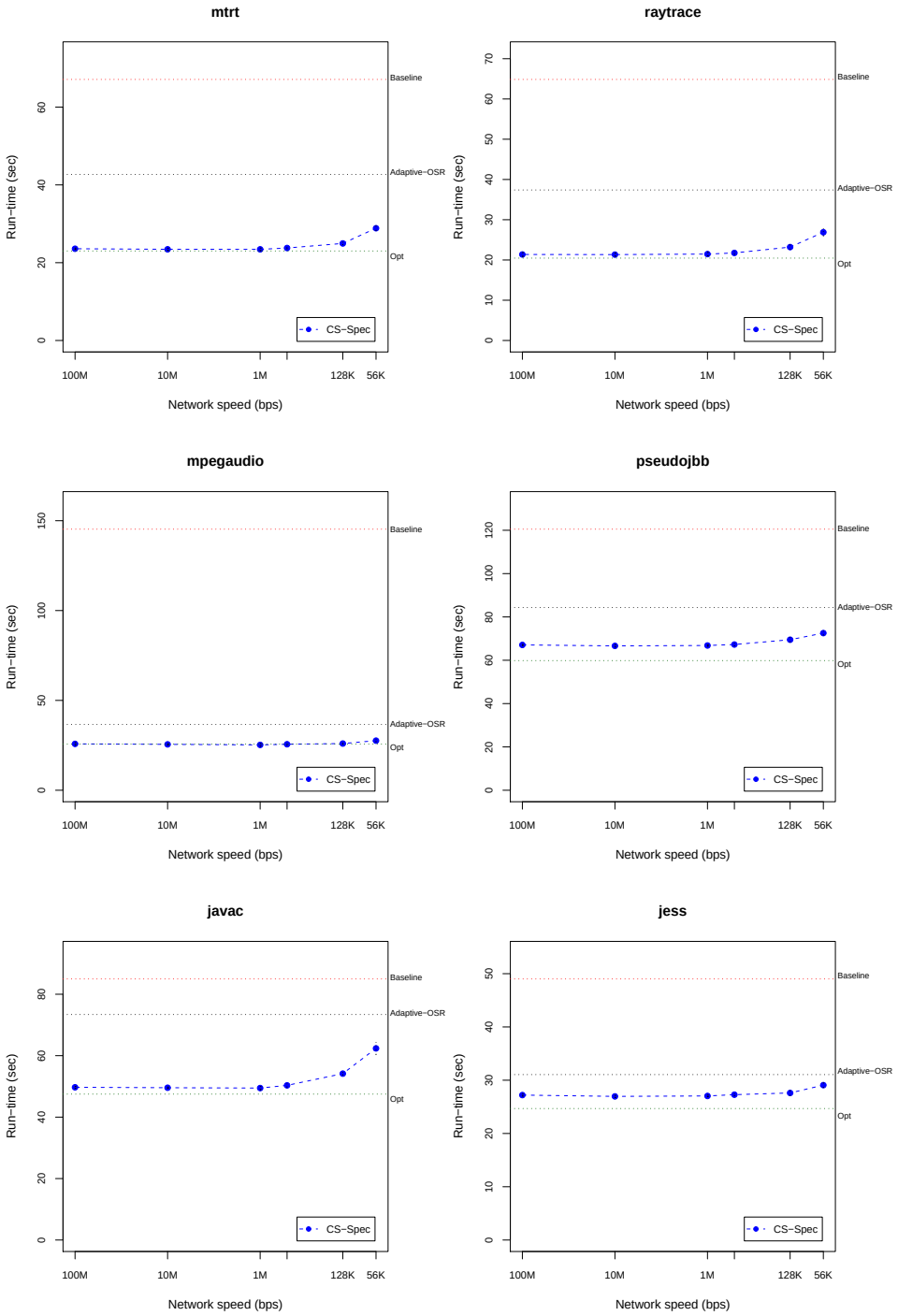

Fig. 14. Effects of network speed on *CS* clients.

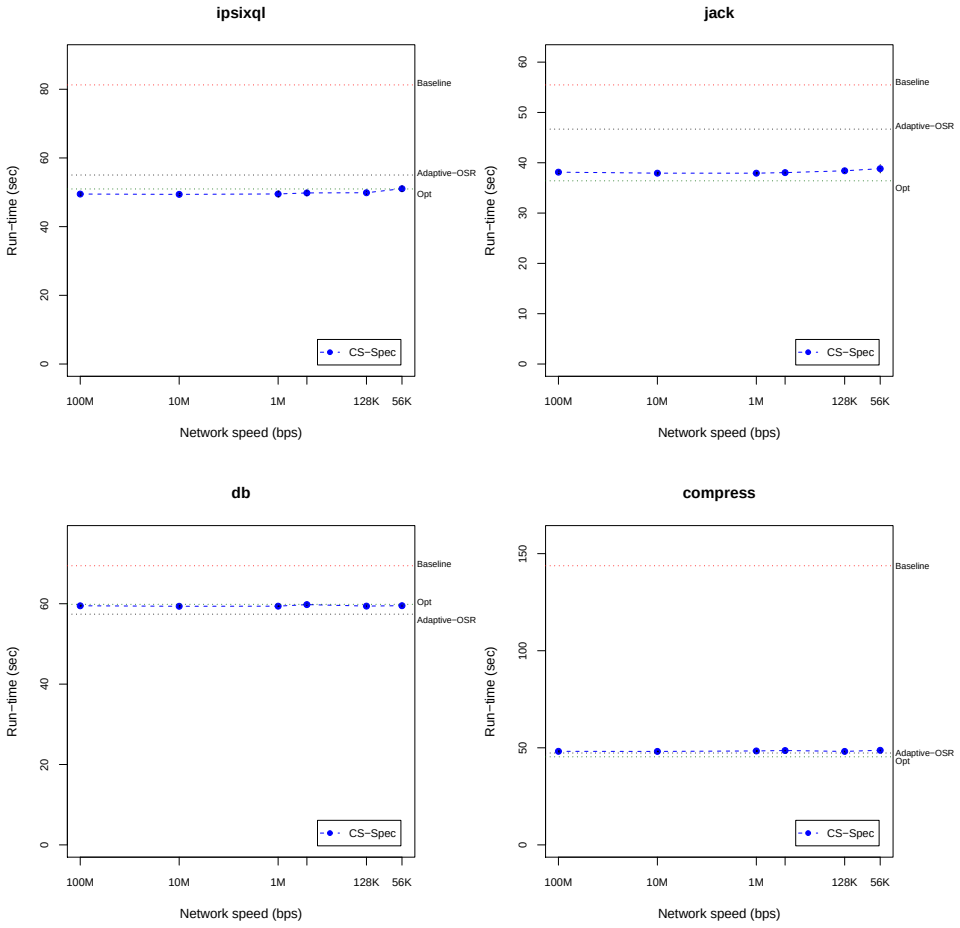

Fig. 15. Effects of network speed on *CS* clients (continued).

and TLB (translation look-aside buffer) behavior given the reduced volume of dynamic allocation.

5.5 Server Performance

Since a given *CS* server is likely to be used by multiple clients, it would be interesting to explore how well *CS* scales. Unfortunately, the optimizing compiler of Jikes RVM is not re-entrant, which means that our current implementation of *CS* is not re-entrant either. We could spawn a number of *CS* servers as separate processes, but this approach would not scale well. To investigate the scalability of *CS* in the absence of a re-entrant optimizing compiler, we gather traces from the current implementation and use them in simulated runs.

The trace information we gather consists of the inter-arrival times of compilation re-

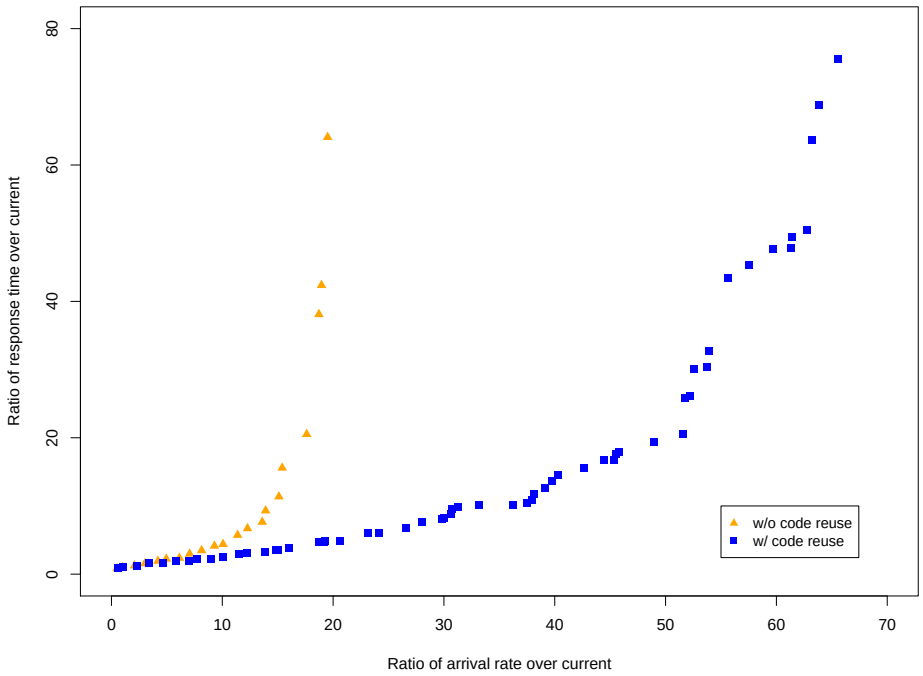

Fig. 16. Server response time as a function of arrival rate of compilation requests.

quests, and the service times of individual compilations, for *CS*. Our simulator then simulates a queuing system with one processor and a queue of infinite length, by randomly picking an inter-arrival time and a compilation service time from the traces. We vary the arrival rate of compilation requests and observe its effect on response times and server utilization. For each simulated data point, we perform 10,000 simulated compilation requests.

Figure 16 shows the simulated response time of *CS* as we vary the arrival rate of compilation requests. We define the response time as the elapsed time between request arrival and request completion; thus it includes the time a request remains in the queue and its service time. The x-axis represents the arrival rate of compilation requests as a ratio over the arrival rate in the traces from our benchmark suite, and thus approximates the number of concurrent clients using *CS*. The y-axis scale is the ratio of response time over current response time. Therefore, a point at (x,y) shows that response is slowed by a factor of y when the arrival rate of compilation requests is increased by a factor of x (i.e., x number of concurrent clients). Figure 16 shows two different series. One series corresponds to simulation results using traces with the code reuse enhancement whereas the other series is based on traces without reusing code. We discussed code reuse in Section 3.3.5.

Without code reuse, the response time of *CS* scales linearly for up to about ten clients and increases exponentially afterward, resulting in a response time ratio of about 20 for 18 concurrent clients and response time ratio of over 60 for 20 concurrent clients. We see that

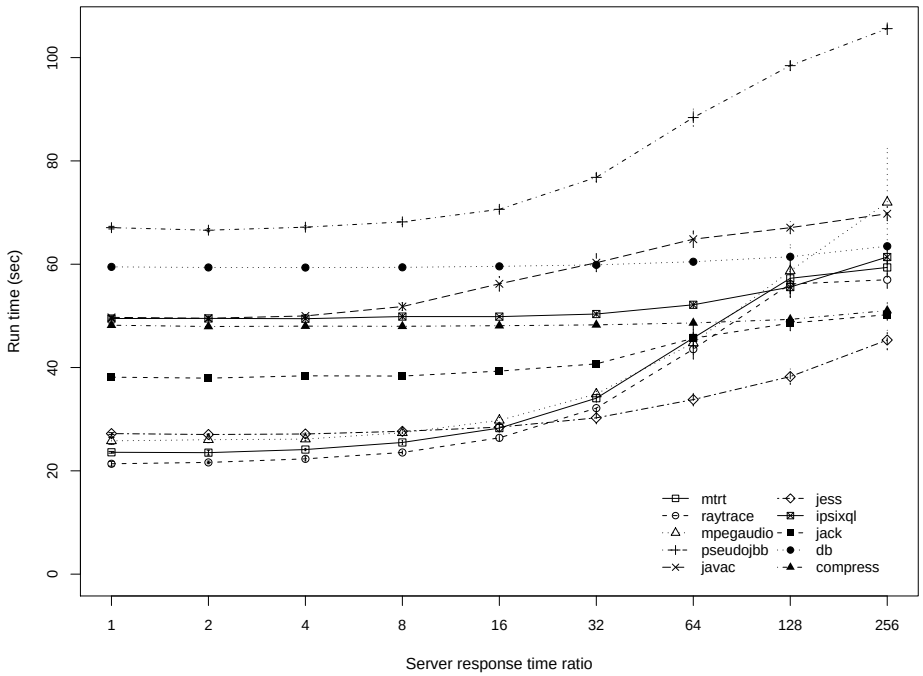

Fig. 17. Overview of effects of slow response time from *CS* on client performance.

code reuse is quite effective at improving the scalability of *CS*. With code reuse, *CS* scales linearly for up to about 40 clients, and its response time ratio is about 20 even for over 50 clients. While knowing about scalability of *CS* is useful, it would be more interesting to know how well clients perform given slowed responses from *CS*. For instance, if a slightly increased response time from *CS* is enough to cause clients to perform worse than the *Adaptive-OSR* configuration, then *CS* may not be very useful.

Figure 17 shows the effect of slow response times on client running time. The x-axis represents *CS* response time as a ratio over current (i.e., single load) response time. Note that the x-axis is to a logarithmic scale. The y-axis represents client running time in seconds. Each line represents a single benchmark, and each point shows the mean of ten measurements for a particular server response time ratio. As before, we place error bars at one standard deviation from the mean in both directions. Increasing response time results in increased variability of running times (i.e., larger error bars).

We see that client running times remain steady until the response time of *CS* increases by a factor of about 16. To place this slowdown in running time in perspective, Figures 18 and 19 show running times of individual benchmarks separately, along with running times for the *Baseline*, *Opt*, and *Adaptive-OSR* configurations.

We see from Figures 18 and 19 that all benchmarks except for db and compress outperform *Adaptive-OSR* even when the response time from the server increases by a factor

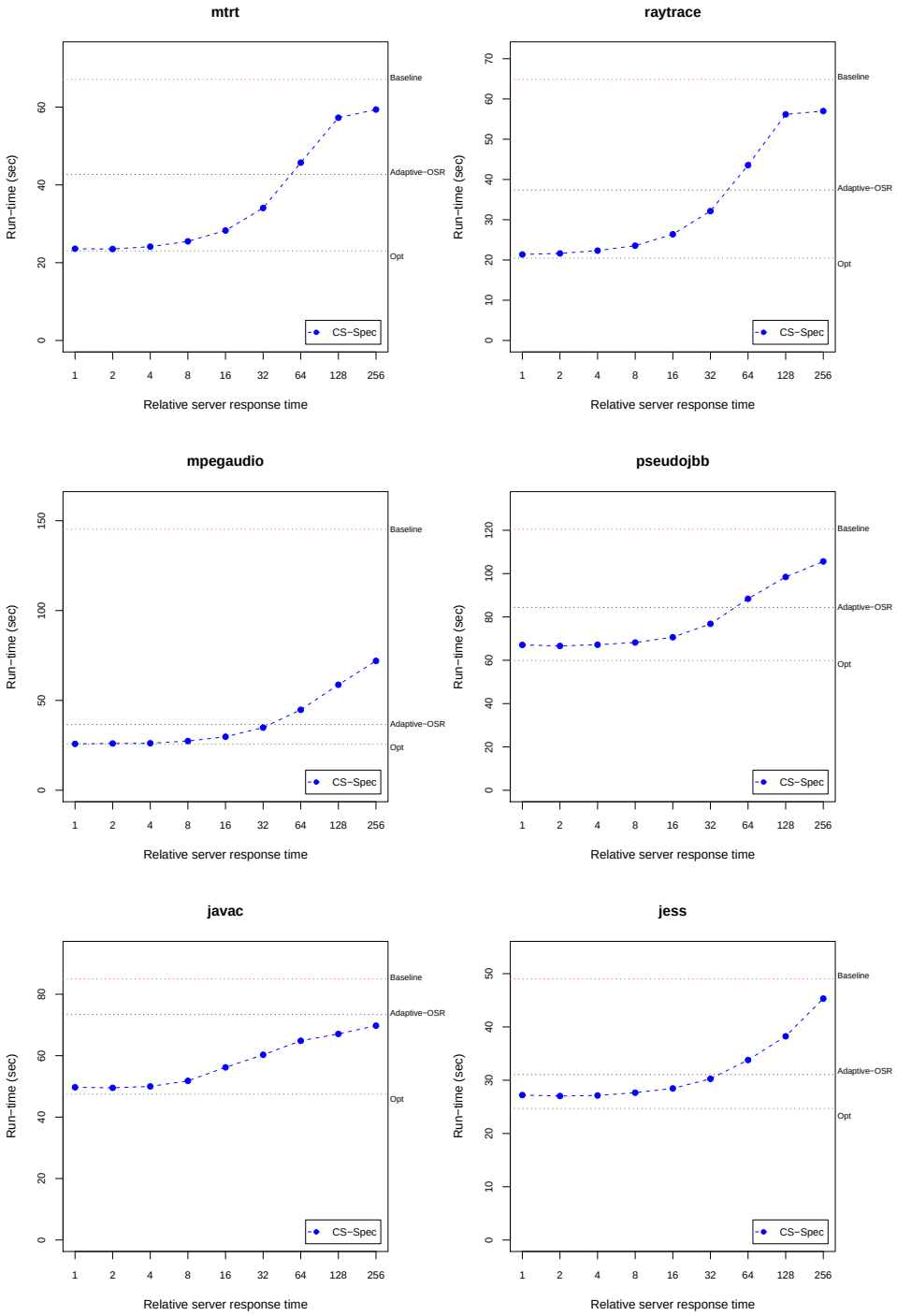

Fig. 18. Effects of slow response time from *CS* on client performance.

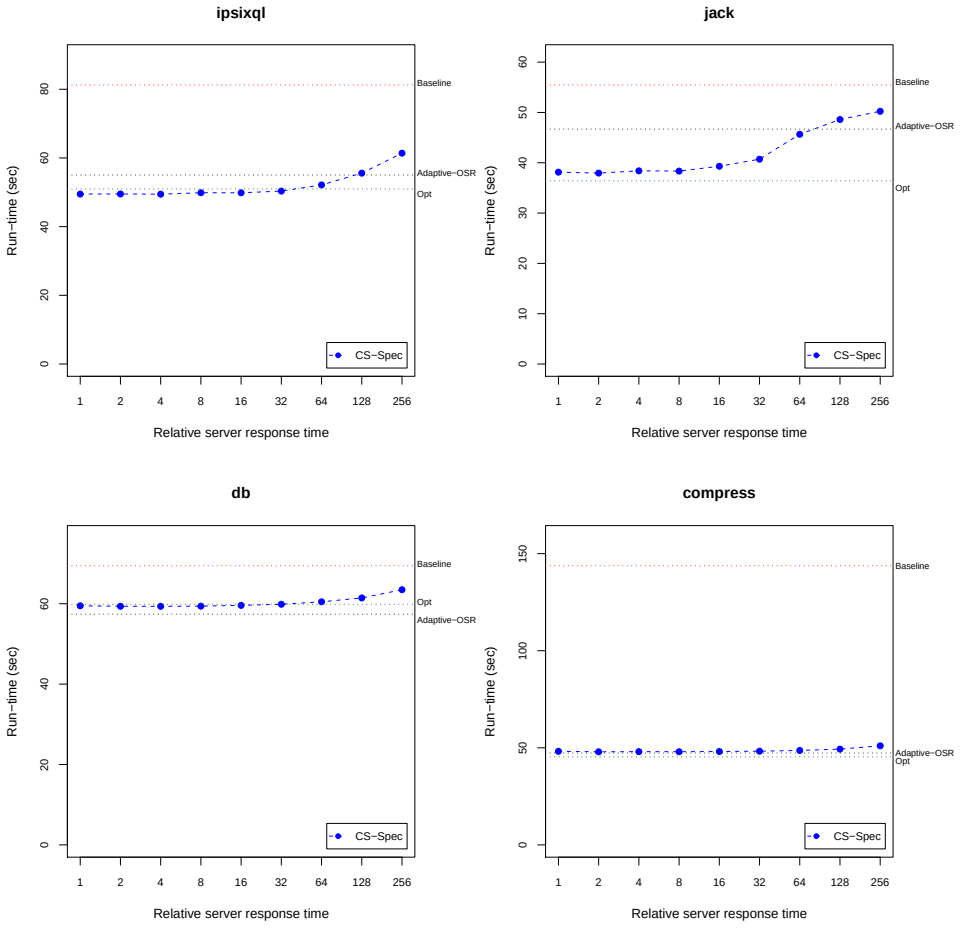

Fig. 19. Effects of slow response time from *CS* on client performance (continued).

of 32. As mentioned before in Section 5.3, db and compress were both slightly outperformed by *Adaptive-OSR* to begin with. In fact, because db and compress make such a light use of *CS*, their running times degrade by only 0.6% and 0.1% respectively, even when the response time from the server increases by a factor of 32. In other words, as with network speed, benchmarks that have much to gain by using *CS* are affected the most by slow response time, but they are still able to outperform *Adaptive-OSR* because they benefit so much from using *CS* already.

In summary our results indicate that:

—When there is no reuse across clients (which models a situation where different clients execute completely distinct code) a CS can support 18 clients while still outperforming *Adaptive-OSR*.

—When we restrict clients to one of our benchmark programs and there is reuse (e.g., two

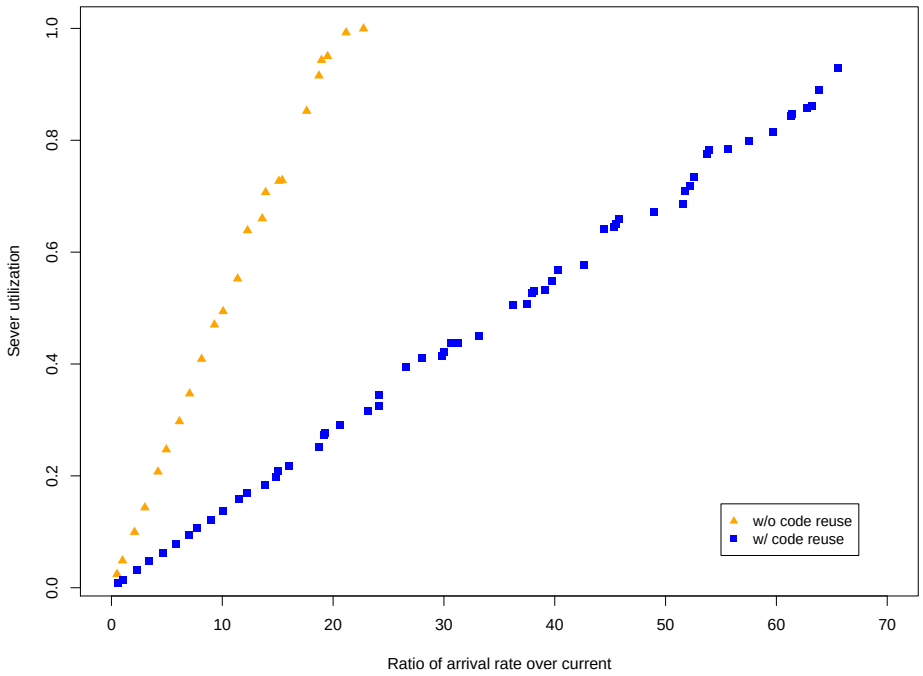

Fig. 20. Server utilization as a function of arrival rate of compilation requests.

clients execute the same benchmark or two clients execute different benchmarks that use common libraries), a CS can support 50 clients while still outperforming *Adaptive-OSR*.

It is worthwhile to mention that code reuse also improves client performance slightly since it reduces the amount of time a client has to wait before it can start executing optimized machine code. However, the benefits of code reuse on clients are small, averaging about 0.8% over all benchmarks.

Finally, Figure 20 shows server utilization as a function of the arrival rate of compilation requests. The x-axis remains the same as in Figure 16, and the y-axis represents the utilization of *CS*. Figure 20 contains two different series as before. We see that server utilization scales linearly for both series, albeit with different slope. This figure, along with Figure 16, confirms the common belief that servers' response time scales linearly until their utilization rate goes above 60% to 70%; *CS* is no exception.

5.6 Profile-driven Optimizations

Having a central server that compiles client code presents interesting and unique challenges. What if one can collect profile information from clients and use it in code generation (for the same or other clients) similarly to online feedback-directed optimizers [Arnold et al. 2002]? Even though each client may execute different applications, it is quite possible that they execute the same common library routines, and if that is the case, then we

could reap the benefits of feedback-directed optimizations even when applications do not run for "long" periods of time. Furthermore, if we could distribute profile collection to multiple clients, we may be able to gather more expensive profiles, such as path profiling [Ball and Larus 1996], that can be used to drive other optimizations without incurring too much slowdown on any single client.

Since *CS* already acts as a proxy server for its clients, we believe it is reasonable to assume the following. First, we assume that *CS* knows about the performance characteristics of each client it serves, and thus can put appropriate weight to profile data gathered by each client. This avoids the problem of profile data from faster clients dominating profile data from slower clients. Second, we assume that *CS* keeps track of profile directives it sends to its clients (step 1 in Figure 5) in order to distribute evenly the task of profile collection. This division of profile gathering may be at the granularity of a method or may even be at the granularity of basic blocks. For example, if *CS* determines that it had gathered enough profile data for all methods a client is requesting, *CS* will not require any profiling from this client.

In this section, we explore and present preliminary data that show the feasibility of gathering profile information on clients and using those data in later runs, using the edge profiling mechanism implemented in Jikes RVM. In adaptive mode, the Baseline compiler of Jikes RVM inserts instrumentation to collect edge profiles, which are then used by the optimizing compiler when the adaptive controller decides to recompile a particular method. The edge profile information is used in multiple places inside the optimizing compiler such as in deciding whether a loop is worthwhile to unroll, and in reordering code to improve instruction cache behavior and branch prediction. So far in our experiments, we disabled profiling of Baseline compiled code, and thus did not use profile information to drive the optimizing compiler of *CS*. For the following experiments, we implemented a mechanism for transmitting edge profile information gathered by Baseline compiled code on the client to *CS* along with each method recompilation request.

The second column, labeled "Instrumentation overhead," in Table VIII summarizes the overhead of collecting edge profile information by instrumenting Baseline compiled code and sending it to *CS*. The instrumentation overhead ranges from 0.3% to 4.2% with a mean of 2.1% over non-instrumented clients. This represents the overhead that a client would incur if *CS* were to decide to delegate instrumentation of all methods to a single client. Different kinds of profiles, of course, would require different instrumentation with different amounts of overhead. As an example, Ball and Larus report that their fast path profiling implementation has an average of 30.9% overhead for their benchmark programs [Ball and Larus 1996].

The last column, labeled "Improvement with profile," in Table VIII shows performance improvement of a client that benefits from previous edge profile data stored on *CS*. To measure this improvement, we ran the same benchmark twice. The first run provided edge profile data to *CS*, and the second run used profile-based optimized code without incurring any instrumentation overhead. This situation, therefore, represents the most optimistic case. The performance improvement ranges from 1.5% to 6.6% with a mean of 3.8%.

This result is by no means an exhaustive evaluation of the use of profiling in the context of *CS*, but it serves as a proof of concept that profiling can be used to improve performance of clients that use *CS*.

We also investigate cross-benchmark benefits of using edge profiling by running one benchmark after a different benchmark. We executed the first benchmark with edge pro-

Table VIII. Summary of instrumentation and profile data

Benchmark	Instrumentation overhead	Improvement with profile
mtrt	2.9%	3.3%
raytrace	3.3%	4.6%
mpegaudio	3.7%	6.6%
pseudojbb	0.3%	2.2%
javac	4.2%	6.5%
jess	1.6%	2.1%
ipsixql	2.1%	4.4%
jack	0.9%	4.9%
db	0.4%	1.5%
compress	1.3%	1.7%
Mean	2.1%	3.8%

filing enabled, which sent its profile data to *CS*, and the second benchmark used code optimized with edge profile information without incurring any instrumentation overhead. To emulate an environment where some of the libraries were shared between different applications, we ran our experiments using a bootimage without the Java library classes compiled in. This allows for profiling and optimized compiling of library methods when they are identified as "hot". We exclude the combination of the mtrt and raytrace benchmarks from this discussion since they use the same code base. Our results indicate that there is some degree of method sharing between the first and the second client, with the speed improvement due to edge profiling ranging from 0.0% to 2.6% with a mean of 0.8%. While this speed improvement is not big, it illustrates that clients do not have to be executing the same program to benefit from each other's profile information. Since *CS* will service many clients and run for a long time, we would argue that the likelihood of a client benefiting from prior profile information is higher than what we show here. Furthermore, the benefit will also depend on what kinds of profile information is collected and used by *CS*.

6. RELATED WORK

We investigated the idea of a compilation service as a means of reducing the energy consumption of mobile devices using power models in [Palm et al. 2002]. This work is an extension of that prior work, but instead of using power models to investigate energy consumption, we presented design, implementation, and evaluation of an actual implementation of a compilation server and clients.

6.1 Server-based Compilation

There is related work in the area of server-based compilation for Java as well as for static programming languages such as C and C++.

Delsart et al. [Delsart et al. 2002] describe a framework called JCOD designed to perform native compilation of Java classes similar to ours but with completely different design goals. Their design is tailored for embedded devices with very limited memory, and thus focuses on improving client performance with minimal increase in code size and memory requirements. In fact, their compile-server performs only a few optimizations that reduce

code size: they perform no method inlining or loop unrolling since those optimizations may increase code size. They are also concerned with producing code that is independent of the operating system and virtual machine, and to that end they implemented a generic object format that must be linked on the client, which results in high overhead. Our design philosophy is that any task that can be performed on the server should be done there since servers can be expected to be much more capable machines. In addition to these differences, we present thorough evaluation of the effects of network speed, server load, and profile-driven optimizations, and investigate the memory usage and pause times of clients using *CS* and the scalability of *CS*.

Newsome and Watson [Newsome and Watson 2002] describe a proxy compilation scheme called MoJo in which a server compiles Java class files to C source code and then to an object file to be used by a client using GNU gcc. MoJo handles only a subset of Java and does not allow recompilation of "hot" methods but rather compiles whole class files at once. In this sense, MoJo acts more like a way-ahead-of-time compiler that batch compiles for its clients. Client execution is halted until compiled code is received. Our work differs in that we consider optimization of "hot" methods only, interleaving execution with optimization request.

There has been some effort in distributing compilation of static programming languages such as C. The problem that these approaches is trying to solve is to reduce overall compilation wait-time and is much simpler to solve since everything can be compiled ahead-of-time. distcc [Pool 2003] allows distributed compilation of C, C++, Objective C, and Objective C++ code using a number of machines on a network. distcc ships individual compilation units (e.g., C files) across the network and links resulting object files together to generate the final executable file. nc [Ellis 2003] is another tool that works in a similar fashion.

6.2 Task Migration

The idea of offloading compilation onto a dedicated server can be considered as a specific instance of task migration.

Flinn et al. [Flinn et al. 2001] describe a framework that automatically downloads tasks to a wired server based on information provided by the application and past profiles. Their work also incorporates a notion of "fidelity". For example, their system may decide based on the environment to use either the full or a short vocabulary for a speech recognition system. Our work is in a sense a detailed evaluation of one application of their approach.

Kremer et al. [Kremer et al. 2000; 2001] describe a framework for migrating the execution of applications to a server. The server periodically (as determined by the compiler) sends checkpoints to the handheld; if the server dies or is disconnected, the handheld can continue execution from the last checkpoint. These ideas are useful in building a distributed compilation system.

Teodorescu and Pandey [Teodorescu and Pandey 2001] describe a Java system that is distributed across servers and resource-limited devices. The resource-limited devices run minimal kernels that download parts of the run-time system on demand. Like our work, the granularity of transferring code is a method. However, unlike our work, all compilation is done on the server.

Sirer et al. [Sirer et al. 1999] consider how to distribute the tasks of a Java virtual machine between a personal computer and a proxy server. By putting tasks such as bytecode verification on a server, they expect to enhance the reliability and stability of a system.

For example, a system administrator would need to install many security patches on only a small number of servers rather than on every computer in the organization. Since the proxy servers are behind firewalls and relatively secure, a client in the organization can trust the server. We follow the general model of Sirer at al. for our work. However, the motivations behind our work and theirs are different: they are more concerned with security while we are concerned with performance and reducing memory consumption and pauses. We also investigate the use of profile data on the server.

6.3 Just-in-time and Adaptive Systems

Our specific implementation of compilation clients is based on the adaptive optimization system in Jikes RVM [Arnold et al. 2000], but there have been a number of just-in-time and adaptive optimization systems built before [Cierniak et al. 2000; Plezbert and Cytron 1997; Radhakrishnan et al. 2000; Suganuma et al. 2001; Hölzle and Ungar 1996; Voss and Eigemann 2001; Auslander et al. 1996]. The work by Plezbert and Cytron [Plezbert and Cytron 1997] is particularly relevant to our work because one of the systems they consider is "continuous compilation" where they assume that there is a second idle processor available for performing compilation. One could consider the second idle processor as a compilation processor that can be used just like a compilation server. However, we believe that it is unreasonable to assume that a typical client (desktop or mobile device) would have multi-processor capabilities. Moreover, they consider a file-based level of granularity for compilation as opposed to our method-based level of granularity. They also claim that "continuous compilation" can eliminate pauses due to compilation but no concrete experimental data is presented.

6.4 Profile-driven Optimizations

Our client implementation is based on the Adaptive Optimization System (AOS) in Jikes RVM [Arnold et al. 2000]. Those authors recognize that optimization cost must be taken into consideration in an adaptive optimizer, and demonstrate (for instance) that for short running programs the most aggressively optimizing configurations perform the worst (because their compilation cost is not recovered). Thus, they use a cost-benefit model for deciding when and how to recompile a method based on many kinds of profiles. They consider four different optimization levels (Baseline, O0, O1, and O2), and any given method may be recompiled multiple times until it reaches level O2. They find that there is significant benefit to picking different optimization strategies for different methods, particularly for long-running benchmarks. In addition, AOS instruments the Baseline compiled methods to obtain edge profile information to be used in the optimization of the same methods. Our exploration into the feasibility of using profile information on *CS* is also based on this mechanism.

In an extension of the above work, Arnold et al. describe an online feedback-directed optimization system that performs four optimizations (splitting, inlining, code positioning, and loop unrolling) [Arnold et al. 2002]. They note that the profile data collected during the execution of Baseline compiled code are limited in that they capture only program startup behavior. Therefore, they use a low-overhead instrumentation sampling framework [Arnold and Ryder 2001] to instrument optimized code to capture steady-state behavior. They report performance improvements ranging from 2.8% to 20.5% with an average of 8.1% using these four feedback-directed optimizations. However, to achieve these improvements, one has to run the benchmark for several iterations to reach "steady-state".

Since many clients use a given *CS*, the combination of feedback-directed optimizations and *CS* is likely to yield "steady-state" performance without much delay.

However, the combination of profiles from different clients may also lead to optimizing for the average system behavior, and thus profile combination is an interesting problem that deserves its own investigation. There have been some efforts in finding similarities and combining profiles in a single application setting. Kistler and Franz [Kistler and Franz 1998] describe a technique for computing the similarity of profile data and how to detect program behavior changes to trigger program re-optimization. Savari and Young [Savari and Young 2000] present another technique based on information theory to compare and combine profile information. More recently, Krintz [Krintz 2003] describe a system based on Jikes RVM that uses both on-line and off-line profile information to improve program performance without incurring much run-time profiling overhead. Krintz combines both types of profile information in determining hot methods and hot call sites and shows that her system can improve overall program performance by 9%.

7. CONCLUSION

To achieve reasonable performance, many modern virtual machines resort to *Just-in-time* compilation. However, the cost of dynamic compilation even in state-of-the-art virtual machines is moderately high. In this paper, we presented *CS*, a *compilation server*, which compiles client code at the granularity of methods to reduce or eliminate the cost associated with dynamic compilation. We evaluated *CS* using a number of different criteria, and showed that *CS* is effective at reducing clients end-to-end execution times, pause times, and memory consumption, using ten benchmarks. In addition, we evaluated the scalability of *CS* and the effect of network speed on client performance. Finally, we showed the feasibility of performing profile-based optimizations in *CS*. We believe that being able to migrate compilation onto a remote server using our *CS* approach will have significant impact on the way virtual machines and optimizations are designed and implemented.

REFERENCES

ARNOLD, M., FINK, S., GROVE, D., HIND, M., AND SWEENEY, P. 2000. Adaptive optimization in the Jalapeño JVM. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, Minneapolis, MN, 47–65.

ARNOLD, M., HIND, M., AND RYDER, B. G. 2002. Online feedback-directed optimization of Java. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, Seattle, WA, 111–129.

ARNOLD, M. AND RYDER, B. G. 2001. A framework for reducing the cost of instrumented code. In *Proceedings of the ACM SIGPLAN'01 Conference on Programming Language Design and Implementation*. ACM Press, Snowbird, UT, 168–179.

AUSLANDER, J., PHILIPOSE, M., CHAMBERS, C., EGGERS, S. J., AND BERSHAD, B. N. 1996. Fast, effective dynamic compilation. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*. ACM Press, Philadelphia, PA, 149–159.

BALL, T. AND LARUS, J. R. 1996. Efficient path profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE Computer Society, Paris, France, 46–57.

BURKE, M., CHOI, J.-D., FINK, S., GROVE, D., HIND, M., SARKAR, V., SERRANO, M., SREEDHAR, V. C., AND SRINIVASAN, H. 1999. The Jalapeño dynamic optimizing compiler for Java. In *ACM Java Grande Conference*. ACM Press, San Francisco, CA, 129–141.

CHAMBERS, C. AND UNGAR, D. 1991. Making pure object-oriented languages practical. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, Phoenix, AZ, 1–15.

CHENG, P. AND BLELLOCH, G. E. 2001. A parallel, real-time garbage collector. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*. ACM Press, Snowbird, UT, 125–136.

CIERNIAK, M., LUEH, G.-Y., AND STICHNOTH, J. M. 2000. Practicing JUDO: Java under dynamic optimizations. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*. ACM Press, Vancouver, BC, 13–26.

CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS) 13*, 4, 451–490.

DELSART, B., JOLOBOFF, V., AND PAIRE, E. 2002. JCOD: A lightweight modular compilation technology for embedded Java. In *Proceedings of the Second International Conference on Embedded Software*. Springer-Verlag, Grenoble, France, 197–212.

DETLEFS, D. AND AGESEN, O. 1999. Inlining of virtual methods. In *Proceedings of the 13th European Conference on Object-Oriented Programming*. Springer-Verlag, Lisbon, Portugal, 258–278.

ELLIS, S. 2003. The 'nc' network compilation tool. Available at http://www.brouhaha.com/ ellis/software/index.html.

FINK, S. J. AND QIAN, F. 2003. Design, implementation, and evaluation of adaptive recompilation with on-stack replacement. In *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society, San Francisco, California, 241–252.

FLINN, J., NARAYANAN, D., AND SATYANARAYANAN, M. 2001. Self-tuned remote execution for pervasive computing. In *8th Workshop on Hot Topics in Operating Systems (HotOS-VIII*. IEEE Computer Society Press, Schloss Elmau, Oberbayern, Germany.

GOSLING, J., JOY, B., STEELE, G., AND BRACHA, G. 2000. *Java Language Specification, Second Edition: The Java Series*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA.

HÖLZLE, U. AND UNGAR, D. 1994. A third-generation SELF implementation: reconciling responsiveness with performance. In *Proceedings of the Ninth Annual Conference on Object-oriented Programming Systems, Language, and Applications*. ACM Press, Portland, OR, 229–243.

HÖLZLE, U. AND UNGAR, D. 1996. Reconciling responsiveness with performance in pure object-oriented languages. *ACM Transactions on Programming Languages and Systems (TOPLAS) 18*, 4, 355–400.

KISTLER, T. AND FRANZ, M. 1998. Computing the similarity of profiling data: Heuristics for guiding adaptive compilation. In *Workshop on Profile and Feedback-Directed Compilation*. Springer Verlag, Paris, France.

KREMER, U., HICKS, J., AND REHG, J. M. 2000. Compiler-directed remote task execution for power management. In *Workshop on Compilers and Operating Systems for Low Power (COLP'00)*. Philadelphia, PA.

KREMER, U., HICKS, J., AND REHG, J. M. 2001. A compilation framework for power and energy management on mobile computers. Tech. Rep. DCS-TR-446, Rutgers University.

KRINTZ, C. 2003. Coupling on-line and off-line profile information to improve program performance. In *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society, San Francisco, California, 69–78.

KUZNETSOV, A. 1998. IPROUTE2 utility suite howto. Available at http://www.linuxgrill.com/iproute2-toc.html.

LEE, H. B., VON DINCKLAGE, D., DIWAN, A., AND MOSS, J. E. B. 2004. Understanding the behavior of compiler optimizations. Tech. Rep. CU-CS-972-04, University of Colorado, Dept. of Computer Science, CB 430, Boulder, CO 80309-0430. Mar.

LINDHOLM, T. AND YELLIN, F. 1996. *The Java Virtual Machine Specification*. Addison-Wesley, Boston, MA.

NEWSOME, M. AND WATSON, D. 2002. Proxy compilation of dynamically loaded Java classes with MoJo. In *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems*. ACM Press, Berlin, Germany, 204–212.

PALECZNY, M., VICK, C., AND CLICK, C. 2001. The Java HotSpot(TM) server compiler. In *Java Virtual Machine Research and Technology Symposium*. The Usenix Association, Monterey, CA.

PALM, J., LEE, H., DIWAN, A., AND MOSS, J. E. B. 2002. When to use a compilation service? In *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems*. ACM Press, Berlin, Germany, 194–203.

PLEZBERT, M. P. AND CYTRON, R. K. 1997. Does "just in time" = "better late than never"? In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, Paris, France, 120–131.

POOL, M. 2003. distcc, a fast free distributed compiler. Available at http://distcc.samba.org/doc/lca2004/distcc-lca-2004.html. White Paper.

RADHAKRISHNAN, R., VIJAYKRISHNAN, N., JOHN, L. K., AND SIVASUBRAMANIAM, A. 2000. Architectural issues in Java runtime systems. In *Proceedings of the 6th International Symposium on High Performance Computer Architecture (HPCA-6)*. 387–398.

SAVARI, S. AND YOUNG, C. 2000. Comparing and combining profiles. *2*.

SIRER, E. G., GRIMM, R., GREGORY, A. J., AND BERSHAD, B. N. 1999. Design and implementation of a distributed virtual machine for networked computers. In *17th ACM Symposium on Operating System Principles (SOSP '99)*. ACM Press, Kiawah Island, SC, 202–216.

STANDARD PERFORMANCE EVALUATION CORPORATION (SPEC). 1998. SPECjvm98 benchmarks. http://www.specbench.org/osg/jvm98.

STANDARD PERFORMANCE EVALUATION CORPORATION (SPEC). 2000. SPECjbb2000 benchmark. http://www.specbench.org/jbb2000/.

SUGANUMA, T., YASUE, T., KAWAHITO, M., KOMATSU, H., AND NAKATANI, T. 2001. A dynamic optimization framework for a Java just-in-time compiler. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, Tampa Bay, FL, USA, 180–195.

TEODORESCU, R. AND PANDEY, R. 2001. Using JIT compilation and configurable runtime systems for efficient deployment of Java programs on ubiquitous devices. In *Ubiquitous Computing 2001, LNCS 2201*. Springer Verlag, Atlanta, GA, 76–95.

VOSS, M. J. AND EIGEMANN, R. 2001. High-level adaptive program optimization with ADAPT. In *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*. ACM Press, Snowbird, UT, 93–102.