© 2009 Nicholas J. Riley

EXPLICIT SOFTWARE SPECULATION
FOR DYNAMIC LANGUAGE RUNTIMES

BY

NICHOLAS J. RILEY

B.A., Brandeis University, 1999

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2009

Urbana, Illinois

Doctoral Committee:

Assistant Professor Craig Zilles, Chair
Associate Professor Vikram Adve
Assistant Professor Matthew I. Frank
Research Associate Professor Ralph Johnson
Associate Professor Samuel Kamin

Abstract

While dynamic languages are now mainstream choices for application development, most popular dynamic languages are primarily executed by interpreters whose performance and capabilities restrict their wider application. The only successful remedy for these limitations has been dynamic optimization infrastructures developed specifically for each language. Building such infrastructures is a substantial undertaking.

This dissertation presents explicit software speculation, which encloses common or expected case code in atomic regions. A best-effort hardware transactional memory executes a region speculatively when specified assumptions, or correctness constraints, permit. When an assumption becomes invalid, the hardware and runtime software transition to corresponding nonspeculative, fallback code which executes correctly, albeit more slowly, under all circumstances.

I demonstrate that explicit speculation improves the performance of dynamic language implementations on existing managed runtimes by speculatively executing dynamic language code with a common case interpretation of the language semantics. I implement a variety of optimizations at a high level, maintaining correct execution without requiring the sophisticated static analysis, language-specific runtime profiling infrastructure or potentially intricate, low-level recovery code that would be otherwise necessary. In addition, I explore how explicit speculation can guarantee speculative optimization correctness in an unmanaged dynamic language runtime by utilizing additional hardware support for fine-grain memory protection.

In memory of my mother, Dr Julia H Riley.

Acknowledgments

If there is a single reason I made it to the other side of my Ph.D., it is the patient guidance of my advisor, Craig Zilles. Virtually everything I know about the process of research I have learned by his example. Like all good teachers, he gives others the capability to learn independently, with the desire for self-improvement to last a lifetime.

This work would not have been practical without the dynamic language implementations upon which it builds. Thanks to the Jython developers Jim Baker, Tobias Ivarsson, Philip Jenvey and Frank Wierzbicki—and to the PyPy and Psyco developers, especially Samuele Pedroni and Armin Rigo, for sharing their knowledge. Hadley Wickham saved me a great deal of time in the last few months with `ggplot2`, his capable plotting software that is a joy to use.

I would like to thank my committee for their input. Ralph Johnson volunteered to read several drafts of this document, helped me put my work in context, encouraged collaboration and offered useful advice. Sam Kamin's comments significantly improved the clarity of what follows. Vikram Adve asked many good questions, investigations of which have unquestionably benefited my work.

The staff of the Medical Scholars Program have been, since my first interview on campus, unfailingly decent, generous, caring and supportive. Working with the MSP, I began to understand how truly hard they try to ensure their students' success. The mission statement in their email signatures is no empty promise.

My fellow students in Prof. Zilles's research group, particularly Naveen Neelakan-

tam, Lee Baugh and Pierre Salverda, have offered assistance, companionship and a sounding board for my ideas. The University of Illinois student chapter of the Association for Computing Machinery has given me a second home and years of wonderful memories. My friends, who long since stopped asking when I was going to graduate, have nevertheless supported me through the process—thank you, Jon Roma, Dan Sachs, Ben Staffin, David Stipp and Nicholas Straker.

Finally, I would not be where I am without my parents, with whom I have celebrated successes and whose love, counsel and belief in my ability have helped me through hard times. I only hope I can give back as generously as I've received.

Table of Contents

Chapter 1 Introduction **1**
1.1 An example 2
1.2 Overview 4
1.3 Summary 6

Chapter 2 Background **9**
2.1 Hardware atomic region execution 9
2.2 Speculation and specialization 11
2.3 Dynamic languages and managed runtimes 13

Chapter 3 Explicit speculation **18**
3.1 Atomic regions and runtime interaction 18
3.2 Implementing speculative optimizations 21
3.2.1 Characterizing the common case 21
3.2.2 Writing speculative code 24
3.2.3 Managing speculative data 25
3.2.4 Exceptions and control flow 27
3.2.5 Assumption checks 30

Chapter 4 Explicit speculation on managed runtimes **34**
4.1 Speculative optimizations in Jython 34
4.1.1 Experimental method 36
4.1.2 Dictionary (HashMap) synchronization 38
4.1.3 Global caching 40
4.1.4 Eliminating exception metadata 42
4.1.5 Joni 43
4.1.6 Direct local variable access (unframe_locals) 44
4.1.7 Direct dispatch 45
4.2 Atomic region usage 48

Chapter 5 Explicit speculation on unmanaged runtimes **55**
5.1 Fine-grain memory protection hardware 56
5.2 Psyco and specialization-by-need 58
5.3 Ensuring correct speculation in Psyco 60
5.3.1 Class attribute caching and dictionary watching 60

5.3.2 Class changing and recovery 64
5.3.3 UFO in atomic regions 67
5.4 Other assumptions 68
5.4.1 Class attributes with multiple inheritance 68
5.4.2 Changing __bases__ 68
5.4.3 Builtins 69
5.4.4 Changing tp_getattro and __getattribute__ 69
5.4.5 Runaway operations 70
5.5 Results 70

Chapter 6 Conclusion 75
6.1 Explicit speculation for dynamic languages 77
6.2 Future directions 78

References 80

Author's Biography 84

Chapter 1

Introduction

Dynamic languages such as Python and Ruby are increasingly popular choices for general-purpose application development. Programmers value their unique features, pragmatic design and rapid evolution. However, their interpreted implementations restrict their applicability. Attempts to build faster, more sophisticated implementations [19, 30, 36] have been frustrated by the need to preserve compatibility with the languages' flexible semantics.

This dissertation introduces **explicit software speculation** as a technique for building optimized yet fully compatible dynamic language runtime systems. A system implementing explicit speculation makes speculation accessible to the high-level language programmer as an optimization technique. It differs in this way from existing hardware, compiler and runtime-managed speculation methods in which speculation occurs transparently to the application programmer.

I evaluate the effectiveness of speculatively executing Python programs, where possible, by assuming common case interpretations of language semantics. With hardware support for atomic region execution, the resulting simpler, thus easier to optimize, speculative code can execute at full speed. If an executing program encounters a situation outside the expected common case, explicit speculation performs hardware-assisted rollback and recovery, then re-executes along an alternate path which provides full semantic fidelity. Overall, explicit software speculation facilitates improved dynamic language performance with minimal added implementation complexity when compared with software-only approaches.

1.1 An example

A user of explicit speculation begins with an existing, correct region of code that could be simplified and made faster by assuming a common case scenario holds true for the duration of its execution. Consider the following Python code:

```
1  global g, x
2
3  def g(i):
4      return i
5
6  x = 5
7  y = g(x)
```

Jython, an implementation of Python on the Java virtual machine (discussed further in Section 4.1), would translate line 7 into Java as:[1]

```
1  x = frame.getglobal("x") // → Python integer object
2  g = frame.getglobal("g") // → Python function object
3  y = g.__call__(x) // → Python integer object
4    g.func_code.call(x, globals, closure)
5      f = new PyFrame(g.func_code, globals) // stack frame for g
6      // add parameter x to Python frame object
7      g.func_code.call(f, closure)
8        Py.getThreadState() // → Python thread state object
9        // finish setting up frame and closure
10       g.funcs.call_function(g.func_id, f)
11         g$1(f) // [function body]
12 frame.setlocal(1, y) // write to local variable
```

Because the Java virtual machine's semantics for variable lookup, method invocation and local variable access are insufficiently flexible to implement their Python counterparts, Jython employs a series of wrapper objects and operations. For a Python function invocation, Jython performs several hash table lookups and constructs an object representing the stack frame before invoking the Java method implementing the desired Python function.

[1] For clarity, in this and subsequent examples I approximate Jython's output in Java, though the Jython 2.5 compiler produces bytecode which has no direct Java source equivalent.

In most cases, the Java semantics for these operations are sufficient; the additional information encoded in Jython's wrapper objects are never used. However, the Java virtual machine usually can't eliminate the code which populates these data structures because it is unable to determine they won't be used. With explicit speculation, the user expresses a set of common case assumptions about the environment under which the call to g is executed and modifies the Jython compiler to output speculative code of the form:

```
1 | local$y = global$g_function(global$x)
```

The common case assumptions, or their converse—the cases in which the speculative code would no longer be valid, include:

1. Redefinition of g. The Java virtual machine semantics support neither arbitrary method replacement of the called function nor replacement of the caller or call site if it is currently executing (though this is changing [39]).

2. Evaluation with an arbitrary object implementing the Python mapping interface representing the global or local namespace. Java provides no equivalent method for executing arbitrary code when a variable is accessed.

3. An invocation of the locals() or globals() function. These functions return mapping objects which reflect the bindings of each name in scope at the call site. If Java's scoping rules were used speculatively instead, the corresponding variables may no longer be accessible.

While the above-described situations are uncommon, it is not reasonable to assume they will never occur. For example, while locals() and globals() are mainly intended as debugging aids, some Python programs use locals() as a data source for string formatting, such as:

```
1 | x = 5
2 | y = 7
```

```
3
4   print 'x is %(x)d, y is %(y)d' % locals()
```

which produces the result `x is 5, y is 7`.

The explicit speculation runtime uses either explicit tests in code or hardware conflict detection to detect when an assumption is no longer valid. It initially reacts to the invalidation of a common case assumption by ensuring that any dependent speculative code does not and will not execute. For example, if `g` is redefined in another executing thread, any speculative optimizations that depend on the now-invalid assumption need to be reverted. Wrapper objects which were speculatively eliminated may need to be restored.

Without explicit speculation, the greater a speculative optimization's scope, the more complex it becomes to recover the state necessary to honor the full generality of language semantics on demand. By buffering speculative state in hardware, explicit speculation eliminates much of the complexity of recovering from misspeculation.

1.2 Overview

A program that uses explicit speculation consists of three components, shown in Figure 1.1. First are regions of **common case speculative** code. Second, corresponding general-purpose, nonspeculative **fallback** code correctly handles the uncommon case as well. Fallback code may be written conservatively or even inefficiently, as it ordinarily will not execute. Last, a set of **assumptions** make explicit the conditions under which the common case speculative code is correct.

Speculation assumptions consist of assertions about data or control flow in the executing program. Assumptions can be invalidated at any time; the runtime system evaluates these assumptions and dispatches to the appropriate version of the code. If an assumption is violated while an affected speculative region is not currently exe-

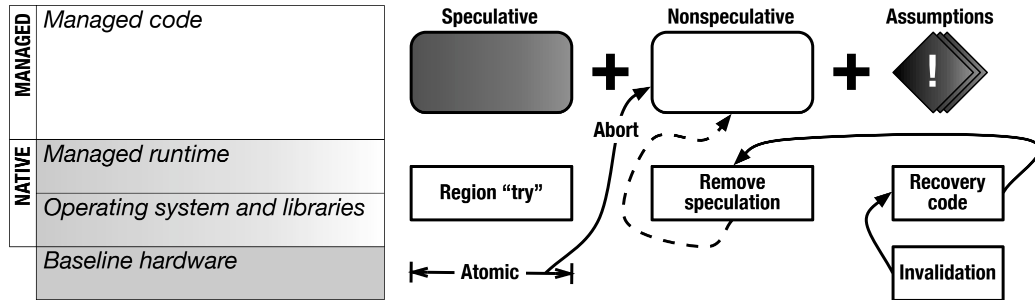

Figure 1.1: Implementing explicit speculation on a managed runtime.

cuting, the region is modified such that nonspeculative code executes unconditionally in the future, potentially pending generation of replacement speculative code that handles the new situation.

Speculative code is placed in *atomic regions*, such that each region's execution is logically buffered and isolated pending its completion. Potential region boundaries may be delineated in advance of execution: the application doesn't need to generate and process runtime feedback to discover them. If common case speculative code is executing at the time one of its assumptions is invalidated, the speculative region must atomically undo, or roll back, any changes it made. The runtime then redirects control flow to the corresponding nonspeculative code as if speculation had never been attempted.

Guaranteed assumption enforcement and atomic rollback simplify the programming model for explicit speculation. The programmer doesn't need to handle the mechanics of speculation; instead, application-provided information guides the runtime and hardware to speculate where advantageous. Explicit speculation resembles exception handling: a region of common case speculative code corresponds to the *try*; equivalent general purpose, nonspeculative fallback code inhabits the body of the *catch*. When the common case code can't handle a situation, it may "give up" by *throwing* an exception (thus expressing a simple control flow assumption). Unlike when handling an exception, the application need not include explicit *compensation*

code to recover from an assumption violation or other exceptional condition encountered during execution of a speculative region.

Even a sophisticated implementation of explicit speculation requires no systemic modifications to the managed runtime or lower layers. A managed runtime's compiler, just as the programmer, may treat explicit speculation as a special case of exception handling. By employing best-effort atomic region execution hardware (proposed in contexts including transactional memory and speculative compiler optimizations [6, 10, 28] and implemented in Azul Systems' compute appliances and Sun's forthcoming Rock processor [9, 15]), exclusively speculative execution incurs no performance penalty in the common case. An efficient implementation can optimize assumption validation and choose the version of code to execute based on execution feedback. For example, if a speculative region is likely to never be valid, it is a waste of time to try.

Hardware atomic execution, when compared with software-only methods, is particularly beneficial for explicit speculation. Most importantly, the runtime need not painstakingly track execution state for recovery on assumption invalidation. A given assumption needs to be checked at most once per region, assuming no code within the region itself may violate it. The runtime can move these assumption checks early in the region and coalesce adjoining regions with identical assumptions to eliminate redundant checks.

1.3 Summary

The principal objective of my work is to investigate the utility of applying previously proposed mechanisms for hardware atomic region execution to a novel area: dynamic language runtime implementation. In doing so, I explore the structure and mechanisms of explicit speculation in both managed and unmanaged runtime environments.

My primary focus is improving the performance of managed runtime dynamic language implementations, by providing Java virtual machines' sophisticated optimizers with simpler and easier-to-optimize common case code and data structures. I also demonstrate the use of additional hardware for fine-grain memory protection in improving unmanaged runtimes' safety.

Explicit speculation addresses a significant outstanding problem facing most dynamic language implementers. In particular, it simplifies the implementations of several optimizations which would otherwise require sophisticated static analysis or runtime feedback systems.

The concepts of explicit speculation simply and logically extend existing and near-future managed runtime abstractions and facilities. Much as hardware exception handling and synchronization primitives were first made available in high-level languages as library functions and later incorporated directly as programming language constructs, the methods of explicit software speculation expose hardware atomic region execution to language developers and users.

Ad hoc optimizations implemented using explicit speculation require little support infrastructure and may be written in a high-level language running on a managed runtime with little if any coupling to the particular runtime and hardware platform. With experience, abstractions for handling speculative code, data structures and assumptions can enable more sophisticated optimizations.

The primary issue, I discovered, limiting explicit speculation's applicability is that of coordinating hardware atomic region size with optimization scope. For some applications, reduced scope translates into reduced performance. My results suggest that certain control flow structures would be better served by a more sophisticated region selection strategy than the simple inlining-like aggregation of regions with predetermined boundaries I performed.

The remainder of my dissertation is organized as follows. Chapter 2 presents

background on the hardware and software infrastructure which underlies explicit speculation. Chapter 3 describes the process of writing optimizations with explicit speculation. Chapter 4 applies explicit speculation to speculative performance optimizations in Jython, a managed runtime dynamic language implementation. Chapter 5 extends explicit speculation to ensuring correct execution of existing speculative optimizations in Psyco, an unmanaged dynamic language implementation. Finally, Chapter 6 discusses potential future development directions for the hardware and software implementing explicit speculation.

Chapter 2

Background

My work has focused on the use of existing and proposed hardware mechanisms to improve dynamic language performance while maintaining correctness. The following sections present background material on one of these mechanisms—hardware atomic region execution (the other is discussed in Section 5.1)—as well as speculation and specialization, dynamic languages and managed runtimes.

2.1 Hardware atomic region execution

With the recent proliferation of multicore processors, the research community has refocused its attention on primitives for concurrency control, to address the well-known shortcomings of locks [44]. Two mechanisms that facilitate writing correct, high-concurrency programs are Transactional Memory (TM) (*e.g.*, [3, 8, 17, 27, 34] and Speculative Lock Elision (SLE) [32, 33]. These techniques permit potentially-conflicting critical sections to be (optimistically) executed concurrently and rolled back if a conflict occurs. Azul Systems' compute appliances and Sun's forthcoming Rock processor [9, 15]) implement hardware support for SLE and TM. Given that current microprocessor vendor roadmaps project an exponential growth in the number of cores over time, it is widely anticipated that additional mainstream microprocessors will include hardware support for one or both of these techniques in the future.

While SLE and TM provide different interfaces to the programmer, the hardware necessary to support them is rather similar. SLE is designed to permit optimistic

execution of existing lock-synchronized code. As such, SLE is merely a performance optimization, since it can fall back on locks to protect critical sections whose needs outstrip the available hardware buffering capability. In contrast, TM is a new programming model in which a programmer annotates critical sections as needing to execute atomically. As a result, TM systems need to provide support for transactions of arbitrary size, although many proposed hardware TM[1] (HTM) systems provide high performance for transactions that fit in processor caches, with somewhat lower performance for larger transactions.

The hardware necessary to support both SLE and in-cache transactions consists of three components: 1) support for checkpointing the register file, 2) support for speculative buffering and atomic commit of memory stores, and 3) support for detecting read-write or write-write conflicts with other threads. The first mechanism is straightforward; the others are generally built by overloading an existing cache coherence protocol. Typically, two bits are added for each cache line for tracking which cache lines have been transactionally read and which written. If an invalidation (or downgrade for written lines) is received from the coherence protocol for a cache line with bits set, the atomicity of the transaction could potentially be violated, so an abort is signalled (invalidating transactionally written lines and clearing the bits). To ensure that data is not lost on an abort, the cache must write back a dirty line before transactionally writing to it, so that there is copy of the most recent non-transactional data somewhere in the system. To commit a transaction, the bits are cleared making the writes visible to other threads. For transactions that fit in the cache, it is expected that these mechanisms can be implemented with little or no overhead relative to non-transactional execution.

The specific implementation I consider in this work is a best-effort HTM similar

[1]In this work, I focus on hardware TM systems. While software TM (STM) systems can likely provide the necessary semantics, it is still an open question whether the overhead of STM systems preclude their use in performance optimization.

to that used in previous work [6]. It can be viewed as either a subset of the functionality in most HTM proposals, or like SLE without the ability to automatically detect critical sections. I model atomic region execution hardware which is exposed to software with a standard TM interface and permits speculative execution of working sets that fit in the first-level cache. Overflowing the cache results in the transaction being aborted. Processor interrupts or exceptions received during transactional execution also cause the transaction to abort.

The model includes support for explicit aborts and redirecting control flow upon an abort. The specific hardware primitives I use are:

- **Begin atomic region (abort PC).** Take a checkpoint and begin associating register and memory accesses with the region. The abort PC argument is saved to be used as a jump target if the region aborts.
- **Commit region.** End the region and atomically commit changes.
- **Abort region.** Discard changes by reverting to the saved checkpoint, then jump to the abort PC.

A special-purpose **atomic region state** register provides a means to determine whether code is currently executing within an atomic region and to query the cause of the most recent region abort.

2.2 Speculation and specialization

Speculation improves performance by predicting that software will repeatedly use the same subset of its functionality. Hardware predictors and software profiling guide speculation by identifying this common-case subset, then determining and enforcing the conditions permitting its use. Applications' direct input into speculation may only consist of "hints" which, if incorrect in practice, can adversely affect performance. If

no existing speculation mechanism can identify and exploit common case behavior, the opportunity is lost.

Existing speculation mechanisms are designed to operate on particular classes of conditions, in order to assure low-overhead data collection and efficient speculative execution which permits recovery from misspeculation. Hardware-only mechanisms such as branch and value predictors are self-contained and guided by data collected without modifying the executable. Software-only mechanisms trigger selective recompilation based upon profile results and correlating execution behavior with a high-level program representation. Recently proposed hardware-assisted software speculation mechanisms [28, 42] are more general-purpose, requiring less intricate knowledge of execution behavior, but still act with no application assistance; it is on this class of mechanism that my work builds.

General-purpose hardware-assisted software speculation requires an explicit interface for expressing speculation's scope and preconditions to the hardware and runtime systems which implement it. Directly exposing hardware and/or runtime speculation facilities in a high-level language is a bad idea, because it ties the application directly to the limits of the particular hardware and runtime. But adding explicit speculation constructs to high-level languages and their runtime interfaces in an implementation-agnostic fashion is useful because they allow the programmer to explicitly define common case behavior. Mainstream managed runtimes make these facilities possible to implement efficiently, augmenting rather than potentially conflicting with existing speculation mechanisms. Rather than mapping speculation constructs directly to hardware, a runtime can tailor the programmer-specified speculation opportunities to the capabilities of the available hardware.

Explicit speculation's primary function is to broaden the applicability of specialization. *Specialization* is an optimization in which code is transformed by assuming particular input values [7]. When the values can be proven to be constant (or are

guaranteed by the programmer not to change), specialization is equivalent to partial evaluation. Where values cannot be guaranteed constant, a specializer makes *optimistic assumptions* about the values it expects to see and generates code using those assumptions. In order to select assumptions, a specializer typically profiles executed code paths and records values that it observes to be constant (or near constant) during computation.

Speculation constructs make common case scenarios explicit in order to facilitate compiler and runtime optimization. However, explicit speculation should be applied reactively, as it is only worthwhile where existing techniques cannot already discover and exploit these optimization opportunities. Among these techniques, automated runtime specialization is an active research area, with several recent proposals that exploit language semantics [40] and hardware support for atomic execution [28, 42]. This work is thus positioned as a complement to—or, in fact, as an enabler of—automated specialization and compiler-supported speculation, by focusing on optimization opportunities they miss.

2.3 Dynamic languages and managed runtimes

Over the past decade, dynamic languages such as Perl, Python and Ruby have become a common workload. Dynamic language use has grown past traditional "scripting" applications to encompass mainstream scientific and commercial development. Often, users of these languages initially adopt them in order to glue components together or perform ad hoc analyses. As users succeed with dynamic languages on small-scale projects, the languages' concise, pragmatic syntax, ease of use, extensibility and embeddability encourage broader application. Dynamic languages' spread is aided by the many high-quality, community-developed libraries and frameworks available and made practical by the availability of hardware which executes dynamic language

code quickly enough for many purposes.

The Perl, Python and Ruby languages, among others, were developed in concert with their original and still most commonly used implementations—AST or bytecode interpreters written in C. These dynamic languages differ from older dynamic languages such as Smalltalk and Lisp in that much of their standard libraries are also implemented in C, rather than in the language itself. This design was borne out of necessity—processing large amounts of data in an interpreted language is impractical.

Interpreted implementations of these languages remain dominant because they have relatively low resource requirements and start up quickly. They are extremely portable as they perform no native code generation and present a low barrier of entry to people wishing to extend or modify the implementation. However, dynamic languages currently face challenges which cannot be met by these implementations. To continue expanding their reach, dynamic languages must improve their performance and safety on current and forthcoming hardware without sacrificing compatibility—the loss of which can dissuade current users from keeping up with language and runtime evolution.

Several alternative dynamic language runtimes have attempted to address the C-based interpreters' deficiencies. Figure 2.1 plots some representative runtimes along axes of implementation complexity and execution speed.

Figure 2.1a represents the class of dynamic language-specific just-in-time (JIT) compilers, which are currently the best-performing runtimes for their languages. Each targets a specific dynamic language and directly generates native code. Some of these compilers use novel techniques such as just-in-time specialization (Psyco [35]) and trace compilation (TraceMonkey, LuaJIT [13]), while others apply more traditional dynamic language optimization (V8 [5] resembles Self [18]). Many of them target interactive applications, such as Web browsers and embedded devices, in which fast startup, short compilation time and low memory usage for both the compiler and

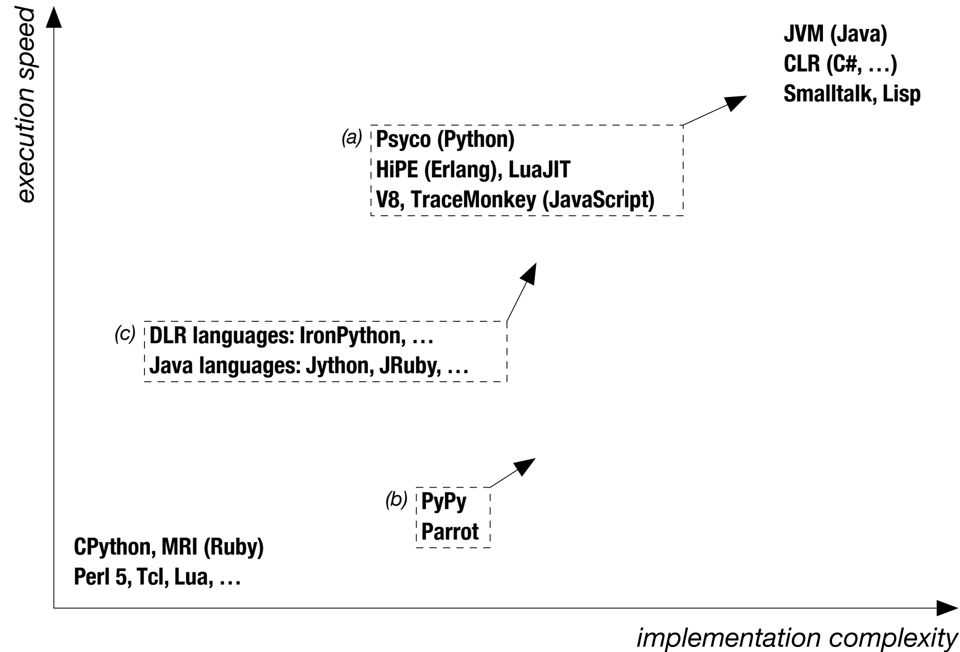

Figure 2.1: Dynamic language implementation points. Arrows indicate the trend of future development for each implementation class.

generated code are essential. As these compilers have generally grown out of existing interpreted dynamic language runtimes–in fact, some fall back to interpretation for faster startup or when the JIT cannot handle a situation—they retain C/C++-based libraries. (Rubinius [31] is an exception; it includes a reimplementation of the Ruby standard library written in Ruby itself.)

The box of Figure 2.1b contains PyPy and Parrot, two dynamic language "runtime construction kits" currently under development. While these systems grew out of the Python and Perl 6 communities, respectively, both intend to support execution of many different dynamic languages, though neither has yet produced a viable replacement for its primary language's traditional interpreter. Each includes much more than a single runtime for the languages it supports—PyPy, for example, can automatically generate a CPython-like interpreter, Psyco-like just-in-time compiler, compilers emitting .NET or Java bytecode, or program analysis tools, all by transforming a description of Python written in a restricted subset of Python itself. At

least in the case of PyPy, however, the cost of this flexibility is an extremely complex and difficult-to-learn infrastructure.

Figure 2.1c includes some examples of dynamic language implementations built on managed runtimes. Here, dynamic language code and its implementation execute on the same runtime, so dynamic optimization can span both domains. Dynamic language implementations can be written entirely in high-level languages such as Java or C#, thereby leveraging the tremendous development effort invested in robust, capable, high-performance Java and .NET managed runtimes, while remaining relatively simple, understandable and "hackable". This implementation strategy preserves managed runtime advantages such as safety, seamless cross-language interoperability, debugging and rich tool support.

While the best managed runtime dynamic language implementations, such as JRuby, outperform the classic dynamic language interpreters, they are not yet competitive with the language-specific VMs, despite the underlying runtimes' greater sophistication. Managed runtimes are capable of much better performance when executing code written in a language designed specifically for them—a language whose semantics match the runtime's. In contrast, dynamic languages' semantics differ from typical managed runtime semantics for such basic operations as function invocation. "Wrapping" runtime functionality with language-specific semantics is common in managed runtime dynamic language implementations. Because of its global scope, this wrapper code is difficult to optimize away.

In practice, most dynamic language code doesn't exploit the full flexibility of its language's semantics. Existing managed runtime primitives can directly encode this common case behavior, resulting in high performance execution. However, it is difficult or impossible to identify the subset of the program that can be so encoded without executing it. Dynamic language implementations cannot therefore unconditionally generate fast code for the common case. In high-performance dynamic

language-specific runtimes, this lack of foreknowledge is remedied by sophisticated, equally language-specific execution feedback mechanisms.

Managed runtime facilities for dynamic language implementation are actively being developed, currently emphasizing rudimentary support for common dynamic primitives. For example, the Microsoft Common Language Runtime (CLR) includes a few dynamic language-friendly primitives such as lightweight code generation, a method "hot-swap" capability and a "delegate" construct akin to a type-safe function pointer. A Dynamic Language Runtime (DLR) layer builds on the CLR, providing an additional, dynamic language-specific type system which overlays yet does not fully interoperate with the CLR type system. The Da Vinci Machine project for Java provides similar primitives, including invokedynamic, a flexible method dispatch mechanism which maps well to the needs of explicit speculation, as I discuss in Section 6.2. In each case, closing the performance gap remains largely the responsibility of individual dynamic language implementers.

Chapter 3

Explicit speculation

This chapter discusses the basis of explicit speculation independent of a particular application. Section 3.1 describes the means by which software and hardware interact and a simple method for selecting atomic region boundaries. In Section 3.2, I characterize aspects of explicit speculation from the standpoint of a user developing an optimization.

3.1 Atomic regions and runtime interaction

Unlike previous work which uses profile feedback to determine regions for speculation [28], I adopt a simpler region selection approach which does not require profile feedback. During initial code generation, potential region boundaries are placed surrounding function calls and loop iterations. At runtime, region execution is aggregated through *flat nesting*, in which a hardware counter is incremented at the beginning of a region and decremented at its end. Speculative state is committed when the counter returns to zero. Because the individual nested regions are not independently identified in hardware, an abort rolls back the changes made by all regions and jumps to the abort PC of the outermost region.

Naïvely applying this method of region selection, a single region would initially surround the entire program's execution. This is clearly impractical, as for all but the most trivial programs, executing this region would exhaust the hardware's ability to buffer speculative state. Execution with explicit speculation thus typically begins

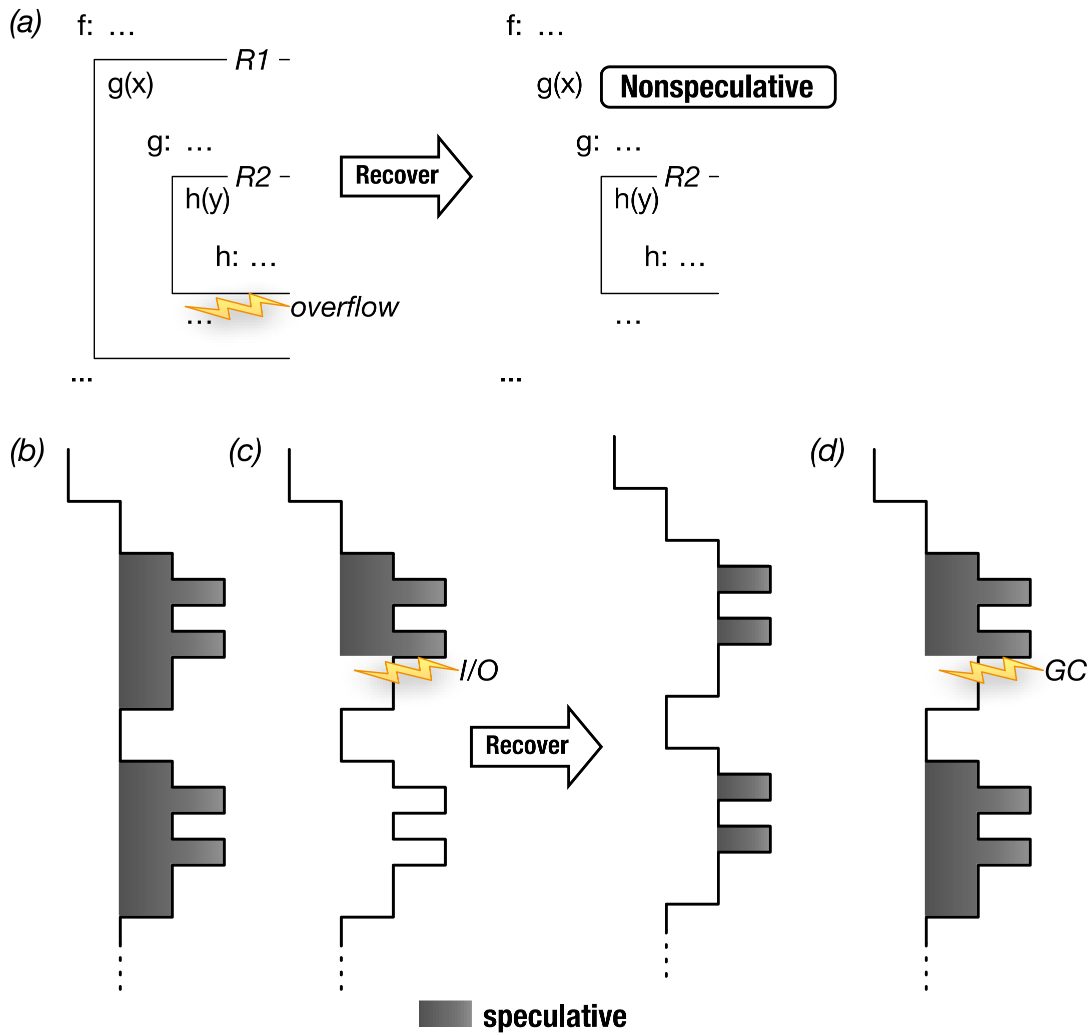

Figure 3.1: Basic speculative region aggregation and abort behavior.

with an aborted attempt to speculatively execute a program-sized region. The runtime feedback generated by this and other *capacity* aborts are then used to adjust nesting behavior.

Consider the example Python execution of Figure 3.1a. On the left, speculative execution begins with region R1, as the function f invokes the function g. g invokes h in turn, beginning the nested region R2. While R2 completes successfully, R1 overflows the storage available for speculative state in the remainder of g. The overflow triggers a capacity abort in R1, which reverts R2's speculative data along with its own.

The recovery process triggered by R1's abort varies depending on the abort reason read from a hardware register akin to Rock's checkpoint status (`cps`) register [9]. On a capacity abort, the outermost aborting region (R1 in this case) is disabled, its entry replaced by an unconditional branch to the corresponding nonspeculative version. However, this replacement does not affect regions nested inside that region, as shown on the right side of Figure 3.1a. For the current and subsequent executions, R2 continues to execute speculatively. Over the first few executions of a piece of code, the attempted atomic region nesting thus "shrinks to fit" the capabilities of the hardware.

In general, a region abort may be handled in one of three ways: re-executing the speculative region, executing the corresponding nonspeculative version *once*, or disabling the region *permanently*. The runtime must keep track of re-execution attempts as the region may repeatedly abort, impeding forward progress. Either of the latter two options guarantees forward progress; the recovery code chooses to permanently disable the region if subsequent speculative executions are also likely to fail.

Figure 3.1b depicts two successful speculative executions of a loop body, consisting of an atomic region containing two nested regions. If the outer region were to perform I/O which cannot be buffered within the speculative region, the recovery code permanently disables the region, leaving the nested regions intact (Figure 3.1c).

Aborts from interrupts, such as those triggered by garbage collection pauses or operating system context switches, do not permanently disable the region as it may execute successfully in future, such as in the next iteration of the loop (Figure 3.1d).

The final abort reasons to address are *explicit* and *conflict aborts*. Both are typically the result of an assumption invalidation. Failed assertions within the atomic region trigger explicit aborts; conflict aborts are generated by conflicting memory accesses from other threads. In these cases, depending on the frequency of observed aborts and the assumption being invalidated, the runtime may choose to disable the speculative region, generate a replacement region less likely to abort, or retry the region.

3.2 Implementing speculative optimizations

I have previously characterized the components of explicit speculation from the standpoint of the system implementing them. This section examines explicit speculation from the perspective of a user attempting to solve a performance problem, by exploring the decisions the user makes in the process of writing an optimization.

3.2.1 Characterizing the common case

Before generating code that exploits common case behavior, the user must determine what the common case is. Explicit speculative code is analogous to a cache or binding for the common case. Efficiently embedding the common case reduces per-execution lookup overhead and improves performance.

A common case value to be used by speculative code, or even its data type, may not be statically available. While explicit speculation offers performance benefits even if the common case is sometimes misidentified, it may be advantageous to wait until the common case can be more reliably identified before generating speculative

code. It is also important not to push common case characterization too early at the expense of implementation complexity—it may be tempting to build a static analysis infrastructure in order to compute the common case from the source code when a simple runtime evaluation would produce the same results with much less effort.

The common case may be determined in any of several ways:

Static speculation. In the simplest case, the common case is available at the time explicit speculative code is first written or generated. Speculative operations and data can be referenced directly.

Runtime and partial evaluation. When the common case is not easily accessible during initial compilation, it can be determined at runtime, perhaps after the non-speculative code has been loaded into memory, after program initialization or at a steady state during execution. Common case behavior can be computed by partially evaluating the nonspeculative code in its runtime context. This technique works well when the common case is not expected to change over a substantial fraction of the program's runtime.

This marks a shift from *declarative* to *imperative* specification of the common case. Instead of generating code which populates a cache of common case values at runtime, the values can be stored as the common case code is generated, and simply referenced by that code.

Execution feedback and instrumentation. Characterizing common case behavior may require observation of the program over time. To accumulate execution feedback, any of the existing techniques developed for dynamic optimization systems may be applied to the nonspeculative variant of the code; often, the managed runtime may already be collecting the necessary information, such as path profiles, though it may not be able to expose this information to the managed code.

Alternately, and most usefully when explicit speculative code can support more than one common case, profiling can be inserted as an inline assumption in the speculative variant. The assumption is invalidated when the speculative code encounters a situation it cannot handle. The recovery code could then reconstruct the unhandled common case observed and produce a replacement speculative region which is handle the newly observed case as well as previously observed ones.

However, by the time the recovery code has been triggered by the invalidation, the atomic region's outstanding speculative state has rolled back. Little remains of the context where the abort occurred—just the abort reason and perhaps the address of the offending instruction. Thus, while recovery code may track speculation failures that occur on region aborts, it isn't necessarily able to identify which assumption failed.

For assumptions about external, mostly unchanging data, the recovery code can simply check the assumptions in turn. If the assumption depends on data computed *inside* the region, such as a type of an expression, the speculative region was unable to proceed because it saw a scenario it hasn't been specialized to handle, and which isn't immediately visible to the recovery code. The soon-to-be-aborted region needs to be able to communicate with the recovery code about to execute. One or more of the following techniques could be used:

- A profiling version of nonspeculative code—or the nonspeculative code itself—could evaluate the assumptions under which the corresponding speculative region failed.

- Speculative regions could be organized such that they can commit even when particular assumptions fail, by placing all the assumption checks at the beginning of the atomic hardware region. Since assumption checks do not modify data, it is safe to commit the region regardless of the value of these checks.

This technique cannot be used in general because it is only applicable to the outermost hardware atomic region.

- A hardware mechanism could be designed to permit a limited amount of information to escape, or be exported, from an aborted atomic region. This could include a subset of memory writes, or or more registers could be explicitly excluded from hardware checkpointing and rollback, thus making them usable for profiling information. (If the address of an explicit abort instruction is accessible in the abort handler, it could itself act as a source of profile information.)

For extremely frequently executed regions, it may be advantageous for the managed runtime to assist with aggregating observed abort information rather than delegating it to high-level code, only invoking code generation after a certain amount of profile data has accumulated.

3.2.2 Writing speculative code

A speculative code region may perform one or more of the following operations, often in order. The following section discusses *transition code,* which performs speculative-nonspeculative conversion, in more detail; assumption checks are described in Section 3.2.5.

- Conversion of data from a nonspeculative to speculative representation before it is read within the region.
- Verification of speculation assumptions, triggering an explicit or implicit abort if an assumption is invalid.
- Computation with speculative as well as nonspeculative code and data structures.

- Conversation of data from a speculative to a nonspeculative representation after it is written within the region.

The overhead of transition code and/or assumption checking code can negate the performance benefit of any "real work" performed in the speculative region. The user should therefore ensure that transition and assumption checking code is efficient and tailored to the specific dynamic instance of that region being executed. For example, only one path through a region may read from a particular speculative data structure; only executions on that path should convert that data structure from its nonspeculative version. Similarly, it may not be necessary to convert an entire data structure to its speculative version if only parts of that structure are referenced.

3.2.3 Managing speculative data

Current and near-future atomic region execution hardware is primarily limited by its capacity to buffer speculative data [11]. This limitation affects explicit speculation in two ways.

First, the memory footprint and locality of speculatively optimized code directly affect its ability to be used. In most cases, speculative versions of data structures are smaller than their nonspeculative counterparts. However, transition code touches nonspeculative data in order to convert it, and assumption checking code may also expand the region's footprint. To maintain consistency between speculative and nonspeculative versions of data structures, enough of the nonspeculative data structure to guarantee exclusive access must be "touched" (read from), such that any parallel nonspeculative write causes the atomic region to abort.

Second, the explicit speculation runtime must consider nested atomic regions in code, aggregated into single hardware regions, which overflow the hardware's capacity. Not only can too-large regions that always encounter capacity limitations

be removed, but adjacent too-small regions can be merged, potentially eliminating redundant assumption checks and transition code.

Footprint tuning provides an added benefit in trimming the number of potential combinations of successfully executed speculative regions for which fallback code is generated. Trimming these fallback paths reduces the overall code size overhead of explicit speculation.

A useful optimization, in some cases, is to store the speculative and nonspeculative versions of a particular item in the same object. This object would expose both a nonspeculative and speculative interface, the latter invoked only from within an atomic region.

Optimizations implemented with explicit speculation can be classified according to their use of speculative data:

No speculative data. These optimizations are the easiest to write and are trivial to aggregate.

Speculative data inside hardware region. Transition code produces speculative data for use within the region and/or nonspeculative data for use after the region commits. Other hardware regions wishing to use the same speculative data must include equivalent transition code. To eliminate allocation within a region, speculative storage may be shared between dynamic instances of a region. In this case, the existing speculative data is assumed to be suspect and never read before it is overwritten.

Speculative data spanning hardware regions. Speculative representations may persist after a region of explicit speculation commits its state. The intervening nonspeculative code cooperates to ensure speculative representations remain valid. Depending on the data's expected access pattern, it may either update the speculative version of data with its nonspeculative equivalent, or convert the speculative data

to and from a nonspeculative version. In the former case, the nonspeculative code may use hardware atomic region functionality to reduce synchronization overhead necessary to make atomic updates to the speculative and nonspeculative versions.

Code executing inside a region of explicit speculation may not have been written or generated with any awareness of speculative data or its scope. If speculative data is passed to this code, it may retain a reference to this data and inadvertently "leak" speculative representations outside the executing atomic region. Thus, care must be taken to limit the exposure of references to speculative data representations. In the context of a dynamic language, this means that speculative representations may be used in invoking other dynamic language user code or known-safe dynamic language library code, but not other code, such as that in the Java standard library, unless it has been analyzed to ensure it does not retain the data passed to it.

3.2.4 Exceptions and control flow

Exceptions are both a part of the mechanism of explicit speculation (as exposed to the managed runtime) as well as a feature used by user code. Left unchecked, an exception in user code may cause control to prematurely exit an atomic region, breaking the region nesting discipline discussed in Section 3.1. Consider the following pseudocode (the nonspeculative alternatives have been removed for simplicity's sake):

```
1  begin_atomic_region(); // R1
2  try {
3    if (...) {
4      begin_atomic_region(); // R2
5      ...
6      if (...) {
7        begin_atomic_region(); // R3
8        throw new SomeException(...);
9        end_atomic_region(); // R3 (never executed if exception thrown)
10     }
11     end_atomic_region(); // R2 (never executed if exception thrown)
12   }
```

```
13 | } catch (SomeException e) { ... }
14 | end_atomic_region(); // R1 (ends R3 if exception thrown)
```

In the case where the code throws SomeException, the end_atomic_region invocations for regions R2 and R3 are never executed. R1's end_atomic_region instead closes R3. Since the two "leaked" regions never end, atomic region execution continues without bound, which eventually results in a capacity or conflict abort. This behavior does result in correct (if inefficient) execution, though if the code being executed does truly use exceptions only in exceptional conditions, infrequent re-execution overhead may be acceptable.

Much real-world code uses exceptions in the course of normal execution, however. For example, neither Java nor Python includes a **goto** operation. Both instead provide exception handling mechanisms which may be used for their control flow effects in non-exceptional cases. An example is a Python idiom for testing whether a name is defined:

```
1 try:
2     some_name
3 except NameError:
4     # name is undefined
5 else:
6     # name is defined
```

If some_name is not defined, the **except** block executes, otherwise execution falls through to the exception handler's **else** clause.

To efficiently support exceptions used for control flow, a Java virtual machine can convert an exception **throw**/**catch** pair into a **goto** if it is able to analyze the exception block's scope. Execution with explicit speculation should ideally maintain this optimization and be able to support local exceptions within a hardware atomic region.

Explicit speculation thus supports the following roles for exceptions:

Nonspeculative fallback code. A special exception class (SpeculationFailure in my implementation) marks an exception handler as containing fallback code to be executed on an atomic region abort. An explicit abort, as on a failed inline assumption, may also map to a thrown exception.

Efficient assumption checking. A null pointer dereference generates an exception in Java. Because null pointer checks must in principle occur on every method invocation, Java virtual machines perform them efficiently, usually by trapping the hardware exception generated on a null pointer dereference. This behavior can be exploited to implement an efficient check for some assumptions, where a null pointer dereference occurs on an assumption invalidation. In some atomic region implementations, invoking a hardware exception handler from within a region would abort the region; otherwise, the exception handler for NullPointerException would trigger an explicit abort. ClassCastException may be intercepted similarly when a method is specialized for a certain data type. By exposing existing processor features, arithmetic exceptions (such as overflow) could be similarly trapped.

User exceptions. To support the common case in which a **try** block is contained entirely within a hardware atomic region, the regions nested inside the handler can be ended before the exception is thrown. The example at the beginning of this section would thus become:

```
1  begin_atomic_region(); // R1
2  try {
3    if (...) {
4      begin_atomic_region(); // R2
5      ...
6      if (...) {
7        begin_atomic_region(); // R3
8        /* R3 and R2 transition code (if any) */
9        end_nested_region_or_abort(); // R3
10       end_nested_region_or_abort(); // R2
```

```
11            throw new SomeException(...);
12         ...
13      } catch (SomeException e) { ... }
14      end_atomic_region(); // R1
```

Before throwing SomeException the code ends R2 and R3. However, given the exception object may reference speculative data, if R1 is not being executed in hardware, ending R2 or R3 will cause a hardware atomic region commit. Therefore I define end_nested_region_or_abort, which ends a nested atomic region (decrementing the nesting counter) unless the current region is outermost, in which case it aborts instead of committing in hardware, to avoid potential leakage of speculative state. While the need to perform one or more region ends slightly increases the overhead of throwing an exception, it does not affect the Java virtual machine's ability to optimize exception handling.

3.2.5 Assumption checks

An atomic region must never commit its speculative state if any of its speculation assumptions are invalid. The runtime is responsible for aborting and/or invalidating the regions which share a invalidated assumption before the assumption-invalidating action takes effect. Currently, assumption checks must be embedded into speculative and/or nonspeculative code by a user of explicit speculation. An **assumption registry** stores a mapping between assumptions and the corresponding regions to invalidate.

Assumption checks may take one of three forms: **implicit checks** within an atomic region (discussed in Section 3.2.4), which rely on managed runtime exceptions, potentially triggered by corresponding hardware exceptions, generated by existing runtime behavior; **explicit checks** or assertions within an atomic region, which verify aspects of, or consistency between, speculative and nonspeculative data; and **assumption traps**, which precede assumption-invalidating actions in both nonspec-

ulative and speculative execution.

Because they occur within a speculative atomic region, implicit and explicit assumption checks may both be categorized as **inline checks**. Inline assumption checks are the only choice for data whose scope does not escape a speculative atomic region.

Inline assumption checks and assumption traps represent opposite sides of an efficiency tradeoff. Inline checks add overhead to the common case path; assumption traps also impact nonspeculative execution. However, assumption traps are usually more efficient because they are seldom executed in the common case.

An assumption trap consists of the following actions, which are inserted before the action which invalidates the assumption.

1. Abort the current region, if any.

2. Retrieve the set of regions R corresponding to the to-be-invalidated assumption in the assumption registry.

3. Disable each region in R, typically by replacing the beginning of the region's code with an unconditional branch to the nonspeculative version.

4. Write to a memory location in the speculative footprint of each region in R, so that if a region is currently executing, a conflict abort occurs before it becomes invalid. Even if the recovery code attempts to re-execute the region, it will have already been disabled by step 3 and the nonspeculative version will execute instead.

   This location may form part of a data structure which is the target of the assumption, or simply be a sentinel location read by the region to identify the assumption or region (as used for lock-transaction coexistence in my work with PyPy [37]).

5. Perform the assumption-invalidating action.

6. (Optional) Tag the invalidated speculative regions for regeneration.

Note that assumption traps may be encountered *within* a speculative atomic region. In this case, it is most efficient to immediately abort the region as in step 1 above, as the assumption trap will be encountered at the corresponding location in the fallback code (unless the assumption becomes valid in the interim).

Assumption traps and implicit checks are simple to implement and have little or no overhead in the common case. By comparison, explicit checks, while used exclusively in compiler-based atomic region speculation systems (*e.g.*, [28]), must be carefully applied to explicit speculation.

Explicit assumption checks include those for which a invalidation involves a conflict or explicit abort. A common form of explicit check is established by transition code, which by reading from nonspeculative data protects against the speculative version becoming invalid if its nonspeculative counterpart is written by another concurrent speculative region or nonspeculative thread of execution. However, this does not protect against the nonspeculative data being written *inside* the atomic region or a nested region. It is thus necessary to analyze the scope of references to the nonspeculative data by any code reachable from the region.

Similarly, analysis may permit elimination of redundant explicit checks of the same assumption within an atomic region. These checks may be collapsed to a single check if the code within the region can be proven not to invalidate the assumption itself.

When designing conflict-based explicit checks, it is important to ensure that data is touched in an order such that the conflict occurs at the same time the data becomes visible, to avoid race conditions with atomic regions beginning or ending at the same time as the modification. Even in otherwise nonspeculative code, a speculative region may be used to ensure atomic modification of the data structure.

As they must consider issues of memory ordering outside and scope within (po-

tentially nested) atomic regions, explicit checks are difficult to reconcile with the high-level language orientation of explicit speculation. It may be too much to ask the potential general audience for explicit speculation to design correct explicit checks; they could instead be reserved as a type of system programming interface, or exposed in a limited fashion with compiler support for validation.

In cases where *recovery code* (as opposed to an assumption trap) disables one or more atomic regions, this disabling may only be treated as a performance optimization. If a region maps to more than one explicit assumption, the recovery code can test each assumption in turn, then disable the region and related regions if the assumption is expected to fail on subsequent executions. This is typically the case if the assumption was not established by transition code. If correctness were to depend on a conflict-generated assumption causing regions to be disabled, a race condition would exist between the time the abort occurred and the time the assumption was re-tested.

Chapter 4

Explicit speculation on managed runtimes

In this chapter, I discuss applying explicit speculation to a managed runtime, the HotSpot Java virtual machine (JVM), and Jython, a JVM-based dynamic language implementation. Section 4.1 explores the performance and ease of implementation for each of the six Jython optimizations I developed with explicit speculation, given "perfect" region selection. Section 4.2 then explores the atomic region usage of Jython code and the performance limitations which result from a simple atomic region selection mechanism.

4.1 Speculative optimizations in Jython

The Jython compiler processes Python source code into an AST and emits Java bytecode from that AST, with one class per Python source file. Most of Jython's functionality is implemented in Java library routines rather than in the emitted bytecode. Explicit speculation opportunities thus exist at the Java level in the Jython standard library, as well as in the compiler itself.

Jython application execution is dominated by Java code that implements Python's dynamic lookup and dispatch logic for every function or invocation, variable reference or access. This code's performance under Jython is approximately that of the equivalent code in CPython (Figure 4.1).

I evaluated three Python benchmarks, `pystone`, `richards` and `pyparsing`, on versions of Jython and the HotSpot JVM extended with support for explicit spec-

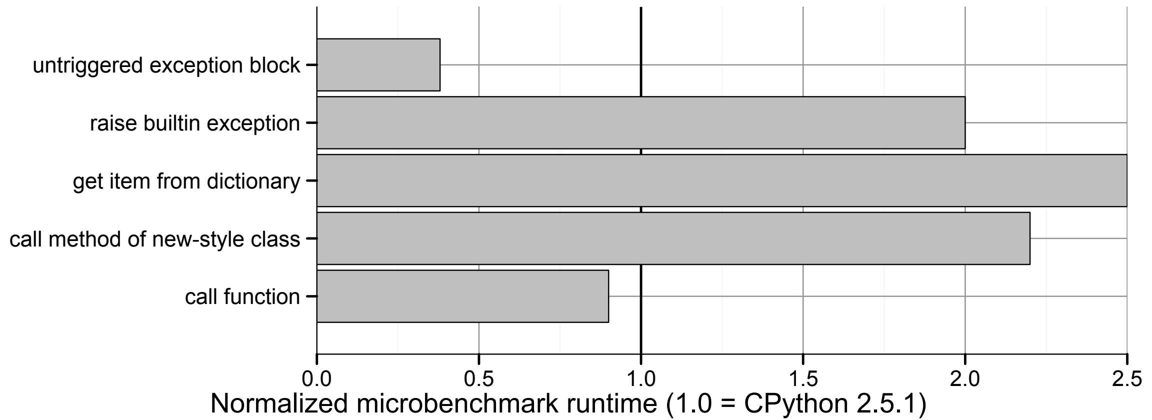

Figure 4.1: Jython and CPython performance on a subset of the PyPy microbenchmark suite [1].

ulation. The first two of these benchmarks are commonly used to compare Python implementations: `pystone` [2] consists of integer array computation written in a procedural style; `richards` [47] is an object-oriented simulation of an operating system. Both are straightforwardly written but non-idiomatic Python, ported from Ada and Java, respectively; I expected them to uniformly exhibit common case behavior. In contrast, `pyparsing` [25] uses Python-specific features to implement a domain-specific language for recursive descent parsers; I chose it as an example of code that is both potentially performance-critical and exploits the language's semantics.

With explicit speculation, Jython can optimistically substitute the equivalent native Java operations. The majority of the optimizations I implemented follow this pattern. For example, I addressed mostly-unnecessary synchronization in lookup (involving the use of the Java HashMap data structure; Section 4.1.2) and eliminated 1-2 hash table lookups from each global variable access (Section 4.1.3). The results improved Jython execution of `pystone` from 40% faster to over twice the speed of CPython; `richards` went from 25% slower to 3% faster. The effects on `pyparsing` were considerably less, even after these optimizations.

Next, I implemented two optimizations which addressed dominant aspects of `pyparsing` execution: exception handling and regular expression evaluation (Sec-

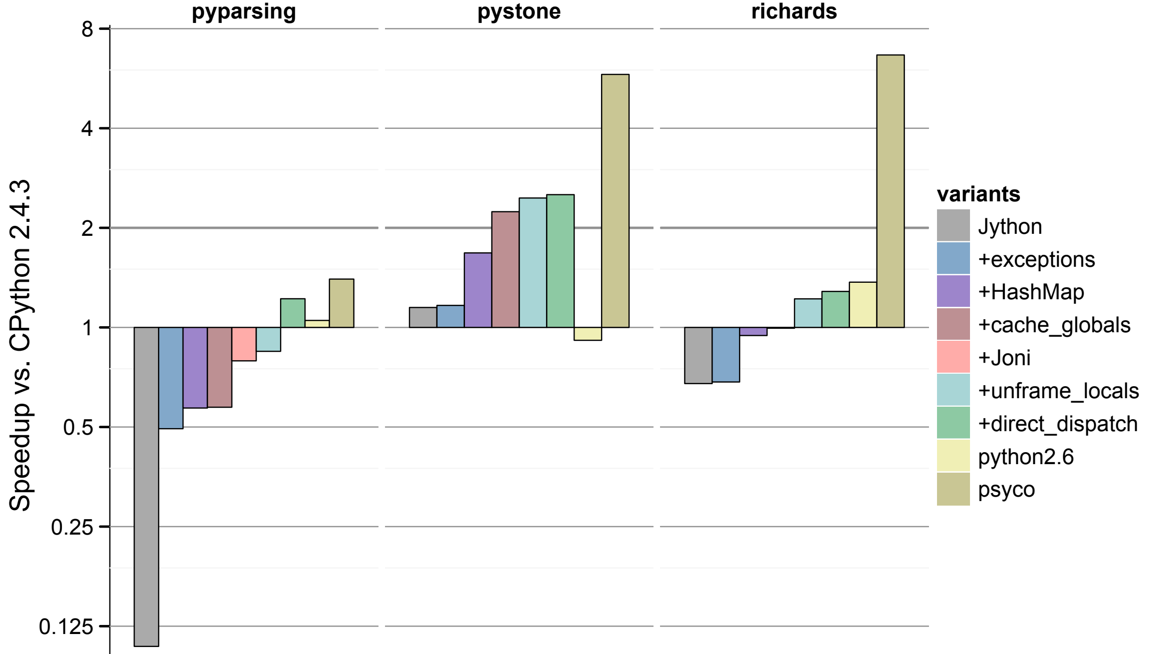

Figure 4.2: Jython and Psyco speedups over CPython on three Python benchmarks, given unlimited buffering capacity for speculative state.

tions 4.1.4 and 4.1.5). Together, these optimizations improve `pyparsing`'s performance by over 600%.

Finally, two more advanced optimizations apply to all the benchmarks, reducing the overhead of Python local variable access, function invocation and object self-reference.

4.1.1 Experimental method

Because the atomic regions in the benchmarks I evaluated divide cleanly into "always abort" and "never abort" groups, I can approximate the steady state performance—*i.e.*, performance after no more invalidations occur—of a single-threaded workload on a HTM-enabled JVM with a real machine without HTM, akin to measuring timings after JIT warmup.

The full simulated execution process I use to converge on a steady state execution is as follows. I employed the Pin dynamic instrumentation system [23] to collect

instruction counts, memory footprint and nesting information for executed atomic regions, the results of which appear in Section 4.2.

1. Run the entire benchmark in Pin with atomic regions surrounding *nonspeculative* execution.

2. Merge those regions whose maximum observed footprints are smaller than 2 KB. Merging small regions eliminates overhead caused by redundant transition and assumption verification code.

3. Execute the remaining speculative regions. As these real-machine experiments provide no capability to roll back execution, I configured the JVM exception handlers associated with atomic regions to abort the process if triggered, recording the identifier of the region being aborted. With an actual HTM, the exception handlers would execute recovery code instead; subsequent executions would take the nonspeculative fallback path.

4. Disable the region aborted by the exception handler.

5. Repeat steps 2, 3 and 4 until execution completes successfully.

6. Time the execution. The inactive recovery and nonspeculative code remains in the running system, such that any performance loss attributable to code size growth should be accounted for by these results.

7. (Optionally) Measure region footprints for speculative execution with Pin. Disable those regions whose maximum observed footprints were larger than 16, 32 and 64 KB, approximating typical first level caches, then reexecute to collect timing data.

Figure 4.2 plots Jython performance with and without speculative optimizations relative to interpreted execution with CPython 2.4.3. As the most commonly used

Python implementation, CPython's performance is a logical baseline for Python users. The leftmost bar in each group plots the performance of unmodified Jython; each subsequent bar to the right includes the effect of an additional optimization. The rightmost two bars of each group represent CPython 2.6 and CPython 2.6 with the Psyco 1.6 dynamic specializer, respectively. Note that Psyco is commonly benchmarked with `pystone` and `richards`, such that the speedups obtained with these two programs are atypically large. `pyparsing`'s behavior is more representative of Python programs in general.

Explicit speculation's effectiveness is largely a function of its ability to generate atomic regions capable of being executed in hardware. The results of Figure 4.2 approximate the performance of explicit speculative Jython on hardware with unbounded, zero-overhead storage for speculative data. As discussed in Section 3.2.3, while the assumption of zero overhead is realistic, the capacities of present-day atomic region execution hardware are considerably more limited. Section 4.2 includes results with finite speculative buffering capacities.

The following sections discuss each of these optimizations in turn.

4.1.2 Dictionary (HashMap) synchronization

The Python dictionary data structure provides an unordered mapping between keys and values. Dictionaries are frequently accessed directly by user code and as part of the implementation of the Python language. In particular, dictionaries are the default storage mechanism for object attributes (*e.g.*, instance variables) and the basis of Python namespaces in which both code and global data reside. Fast dictionary lookups are essential to Python performance, as nearly every Python function call, method invocation and global variable reference involves one or more dictionary lookups.

Both CPython [21] and Jython's dictionary implementations include specializa-

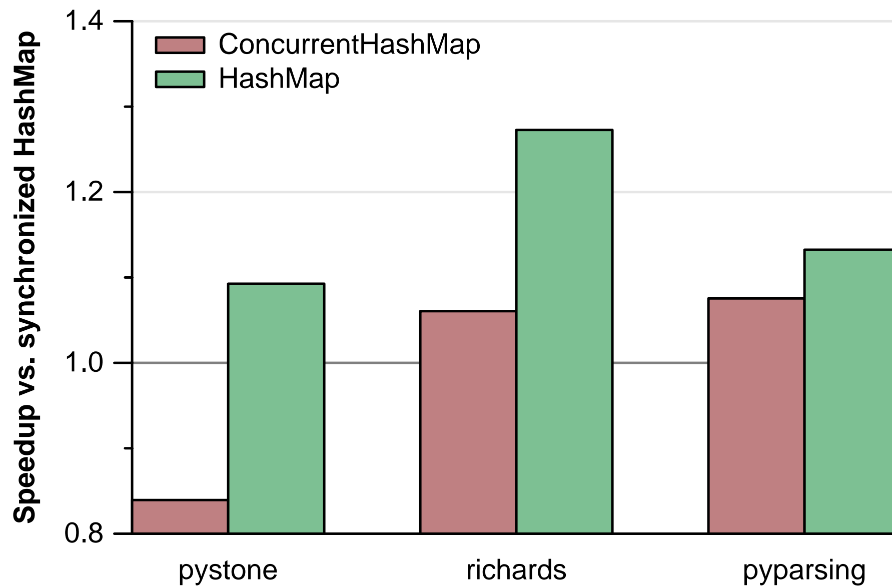

Figure 4.3: Relaxing correctness and synchronization constraints on Python dictionaries.

tions tailored to particular use cases. For example, the keys of object dictionaries (object.__dict__) are nearly always strings representing attribute names. Jython's PyStringMap optimizes string-keyed lookup by requiring that the keys be interned when stored, so a string hash computation need not be performed during lookup, and permitting Java strings to be directly used as keys, rather than being wrapped in an adapter implementing Python string semantics.

Python dictionary operations must be performed atomically. Jython enforces this constraint by using a ConcurrentHashMap [22], a hash table-based data structure designed to permit concurrent multithreaded access, for both general-purpose PyDictionary and special-purpose PyStringMap dictionaries.

Previous Jython versions wrapped a simple HashMap object with Java synchronization. While replacing the synchronized HashMap with a ConcurrentHashMap improved performance overall, the implementation has two drawbacks. First, even on a single-core machine it is slower: on `pystone`, the naïve single-lock model performs

better (Figure 4.3). Second and more seriously, ConcurrentHashMap does not conform to Python semantics; in particular, while a Python dictionary's contents remain unmodified, repeated attempts to iterate through it must return key-value pairs in the same order. With ConcurrentHashMap, the iteration order may change under some access patterns even though the dictionary contents do not.

By using hardware atomic regions for speculative lock elision (SLE) [32], uncontended access to a synchronized HashMap can perform as well as the unsynchronized version. Thus, I can productively replace the various Jython dictionary implementations with unsynchronized versions and wrap their invocations in atomic regions. Synchronization overhead can be eliminated entirely when accesses are subsumed by an outer atomic region, as will usually be the case.

4.1.3 Global caching

I next chose to address the overhead associated with accessing Python module-level globals (typically defined at the top level of a source file) and "builtins" (basic type names, constants, functions and exceptions such as int and None, which would be keywords in many other languages). While local variable references are resolved at compilation time (typically as an array access), global and builtin name references are looked up in a dictionary at runtime [45].[1]

Jython compiles each module (usually corresponding to a source file) to a Java class. It converts the Python code,

```
1 def f():
2     g(x)
```

representing an invocation of a module function g from another function f with a module global variable x as a parameter, into Java as:

```
1 public PyObject f$1(PyFrame frame) {
```

[1]Since globals shadow builtins, a builtin resolution involves one unsuccessful (global) and one successful (builtin) hash probe.

```
2    frame.getglobal("g").__call__(frame.getglobal("x"));
3  }
```

While the externally visible Python namespace representation must remain a dictionary for compatibility, its internal representation can be speculatively optimized for performance by exploiting the characteristic access pattern of dictionaries used for instance variable and namespace storage. Early in their lifetime, these dictionaries are populated with a set of name-value pairs. Thereafter, the values may change but the set of names is unlikely to do so.

To take advantage of this behavior, I modified the Jython compiler to keep track of global and builtin references as they are compiled, emit corresponding Java variable declarations, and cache the global and builtin values in these static variables during module loading.[2] With this optimization, the code becomes:

```
1  public PyObject g$g, g$x;
2  public PyObject f$1(PyFrame frame) {
3    try {
4      begin_atomic_region();
5      g$g.__call__(g$x);
6      end_atomic_region();
7    } catch (SpeculationFailure e) {
8      frame.getglobal("g").__call__(frame.getglobal("x"));
9    }
10 }
```

I additionally subclass PyStringMap with a version that redirects reads and writes to the module dictionary's g and x keys to the corresponding static fields. This does slow down access through this dictionary by unspecialized code (specialized code uses the fields directly), but since such accesses are both infrequent and dominated by a hash table lookup, this is a reasonable tradeoff.

Attempts to delete x have no direct Java analog. I map the semantics of Python

[2]Note that the variables are instance variables rather than static variables. Jython compiles each Python module to a Java class. A module may be imported in different contexts, each with its own set of globals and builtins. Jython represents each context by a different instantiation of the module object.

deletion to the Java null value to represent a deleted variable. I then preserve the Python semantics of raising an exception when a deleted value is referenced, by ensure that an assumption check takes place before any access to x. In some cases, this is a truly implicit check involving no extra code: dereferencing a null g$g would generate a NullPointerException. In others, an explicit check must be included: for example, g's __call__ implementation converts a Java null into a Python None object, which would not trigger an exception when dereferenced. This behavior was further discussed in Section 3.2.4.

4.1.4 Eliminating exception metadata

The pyparsing benchmark implements a parser framework which uses exceptions extensively for backtracking. While parsing a 7416-byte Verilog file, pyparsing raises 90 different ParseExceptions a total of 11691 times. Not only are raising and trapping of Python exceptions very expensive in Jython—Python-specific exception objects must be constructed, and Python-specific exception matching and decoding performed—but in my original implementation of explicit speculation in Java [38] any Java exception raised in an atomic region was converted into an explicit region abort. To support speculative optimization with exceptions used for non-exceptional control flow, a (Java) exception should be able to be thrown within an atomic region. Only throwing of specific exception types should cause the region to explicitly abort.

The most effective Jython optimization I evaluated on pyparsing was eliminating Java stack collection during a Python exception throw. This process, implemented in the JVM by the native method java.lang.Throwable.fillInStackTrace, can defeat runtime optimizations because it must reconstruct ordinarily unneeded execution state. The exception's Java stack trace is only accessed when Java code attempts to introspect a Python exception, which typically only occurs during debugging; it is an ideal candidate for speculative removal. Removing unused Python exception state and its

extraction code also sped up execution, though much less dramatically.

Given support for exceptions inside atomic regions as described in Section 3.2.4, this optimization was relatively simple to implement. Inside speculative regions, I modified explicit raises of Python-specific exceptions to instead use a PyException subclass with an empty fillInStackTrace method (bypassing the expensive native method in Throwable) and methods which include assumption traps triggering explicit aborts on any metadata access from Python or Java. Jython-generated exception handlers, excepting those I added to implement assumption checks for explicit speculation, always catch all exceptions in order to bridge the Python and Java exception models. A Java class is not created for each Python exception class; the Python exception class is stored in the type instance variable of every PyException, so the Python-side exception matching behavior does not need to be modified.

A logical next step in Jython exception optimization would be the speculative elimination of Python-specific exception raising and matching machinery entirely when the control flow occurs entirely within an atomic region, though an experiment in which I eliminated this code in all cases sped up `pyparsing` on Jython by only 1–2%, so I did not pursue this optimization opportunity further.

4.1.5 Joni

`pyparsing` performs many steps of parsing with regular expression evaluation, but uses few regular expression features. The Joni optimization speculatively replaces Jython's sre regular expression implementation with Joni [26], a port of the C-based Oniguruma library [20] developed for use with JRuby (another Java dynamic language implementation). Unlike sre, Joni operates on byte strings (with its own Unicode-aware encoding handling) and is optimized for use with HotSpot. Python strings still need to be converted when entering Joni, but Joni's more compact target representation and more efficient implementation combine to offer a significant improvement. In

its current form, Joni only supports a subset of sre's capabilities. I placed assumption traps in the sre-to-Joni adapter which will trigger the a rollback and reversion to the fallback sre code if an unsupported operation is requested on a regular expression or match object. More efficient VM-level storage of strings [49] could eliminate the need for string conversion entirely, adding to this benefit.

This optimization is a simple example of adapting a more efficient common case interface subset while retaining the general functionality. The primary consideration here is determining the expected lifetime of the speculative representations. In this case, with the regular expression match object, the lifetime is relatively short, the speculative and nonspeculative representations can be independent. The compiled regular expressions last nearly the lifetime of the program, such that both Joni and sre representations are maintained and used as appropriate.

4.1.6 Direct local variable access (unframe_locals)

Jython local variable and parameter lookup is faster than global variable lookup, but neither maps directly to Java local variables. For example, the following code:

```
1  def f():
2    x = 5
3    return x
```

is compiled by Jython as:

```
1  public PyObject f$1(PyFrame pyframe) {
2    pyframe.setlocal(0, _1);
3    return pyframe.getlocal(0);
4  }
5  static final PyInteger _1 = Py.newInteger(5);
```

The PyFrame object uses an array to represent the method's local variables; in this case, x maps to index 0 in the array.[3]

[3]The array, or fastlocals representation, in which local variables are accessed by index rather than by name, is shared by CPython. In a small percentage of cases, such as at the top level of a module where the local and global namespaces are identical, local variables are stored in a dictionary instead.

In the common case, an explicit frame representation is unnecessary. The common case is already being computed statically by Jython, as evidenced by its use of an index rather than a name to look up x. With explicit speculation, the code above becomes:

```
public PyObject f$1(PyFrame pyframe) {
  try {
    begin_atomic_region();
    PyObject x = _1;
    end_atomic_region();
    return x;
  } catch (SpeculationFailure e) {
    pyframe.setlocal(0, _1);
    return pyframe.getlocal(0);
  }
}
static final PyInteger _1 = Py.newInteger(5);
```

If the entire method body does not fit in a single hardware atomic region, transition code must copy the Java local variables back into the PyFrame's f_fastlocals array for the surrounding nonspeculative code. An assertion is inserted in the implementations of locals() (which returns the local variables and parameters as a Python dictionary), or if an explicit representation of the frame is requested, such as during debugging or tracing.

4.1.7 Direct dispatch

A module is *imported*, or loaded into Jython in several steps. The module's source code is first parsed into a Python abstract syntax tree (AST), from which a Java class is generated. Jython then constructs an instance of the class, thereby evaluating the Python code at the top level of the module and populating runtime data structures with information about the module.

During import, Jython *binds* many of the module's names, such as those that reference variables, functions and other modules. In most cases, these names retain

Method	Time
PyTableCode.call(PyFrame,PyObject)	33%
PyBaseCode.call(PyObject[],String[],PyObject,PyObject[],PyObject)	20%
pyparsing$py.call_function(int,PyFrame)	16%
PyBaseCode.call(PyObject,PyObject[],String[],PyObject,PyObject[],PyObject)	4%
PyObject.__call__(PyObject,PyObject)	1%
PyObject.__call__(PyObject,PyObject,PyObject)	1%
PyObject.__call__(PyObject,PyObject,PyObject,PyObject)	1%

Table 4.1: `pyparsing` dispatch overhead without direct dispatch. Methods occupying less than 1% of runtime are omitted.

their bindings for the remainder of execution. By assuming initial bindings do not change, the direct dispatch optimizations described in this section eliminate repetitive lookup and dispatch overhead.

First, consider the Python function call example of Section 4.1.3:

```
1  def f():
2      g(x)
```

rendered into Java as:

```
1  public PyObject f$1(PyFrame frame) {
2    frame.getglobal("g").__call__(frame.getglobal("x"));
3  }
```

Note the lookup step performed by frame.getglobal and the dispatch step by the __call__ method on the resulting PyFunction object. While the lookup step is effectively bypassed by global caching, the more expensive dispatch is not. In `pyparsing`, for example, after applying the optimizations discussed this far, Jython methods implementing function dispatch consume 73% of runtime (Table 4.1).

A common case subset of function invocation maps directly to Java functionality:

- The called function is either a Java method or a Python function or method with a finite number of positional arguments.
- No implicit type conversion (*e.g.*, from objects of class java.lang.String to those of Python's unicode type) occurs when calling from Python to Java.

A direct dispatch optimization replaces the complex and general Jython function call path with an atomic region specialized for the above cases. The speculative caller code becomes:

```
1  begin_atomic_region();
2  g$g(g$x); // note no __call__
3  end_atomic_region();
```

For Python call targets, replacement callee code is generated, in which the single PyFrame parameters are replaced by the Python function's individual positional parameters.

Only the caller code needs to be invalidated if debugging or tracing is enabled; the speculative callee code will then be unreachable. Jython's existing response upon nonspeculative function entry to an active debugger or execution tracer is amended to disable speculative regions within the function during execution, in order to ensure debugger control or trace events. Dictionary watchers (such as those used in the global caching optimization) monitor nonspeculative writes to the names' bindings. Regions which speculatively write to the corresponding cached globals are disabled in advance, as simply writing to a Java variable cannot trigger region invalidation.

Similarly, I applied direct dispatch to calls from one method to another of the same class, and access to an object's attributes from methods of its class. In addition to the above assumptions, it is also necessary to verify the method or data attribute has not been overridden by a subclass or by an individual object instance, or assigned a custom data descriptor. Consider the following example:

```
1  class C(object):
2    def f(self):
3        return self.g()
4    def g(self):
5        return 1
```

Some of the ways in which direct dispatch of C.f can be invalidated include:

```
1  C.g = lambda self: 5 # assign to class dictionary
2
```

```
3   c = C()
4   c.g = lambda: 5 # override instance dictionary
5
6   class D(C): # subclass C
7       def g(self):
8           return 5
9
10  class E(object):
11      def g(self):
12          return 5
13  C.f.im_func(E()) # bypass class identity check, invoke on instance of other class
```

I was able to implement all the related assumption checks but the class identity check with assumption traps without affecting performance on the speculative path. Python does not permit overriding `__call__` on instance method objects.

These direct dispatch optimizations benefited `pyparsing` considerably more than `pystone`, with relatively few function calls, or `richards`, with a large number of polymorphic dispatch sites that have no single common case target.

4.2 Atomic region usage

This section includes a characterization of the atomic region usage patterns of explicit speculative code, as well as a simulation of explicit speculation's performance with limited buffer capacity for speculative data (Figure 4.4). `pyparsing` uses recursion extensively, such that many of the inner regions are disabled on the basis of their maximum footprints, despite their manageable actual footprints at inner nesting levels. As a result, its performance is significantly affected by smaller buffer capacities.

Figures 4.5, 4.7 and 4.9 plot the maximum observed instruction counts, and Figures 4.6, 4.8 and 4.10 the maximum speculative data footprint (in 32-byte blocks), of each potential static atomic region. The maximum nesting depth is the maximum dynamic depth of regions nested inside a static region (including the region itself). For example, if a region is always innermost, its maximum nesting depth is 1.

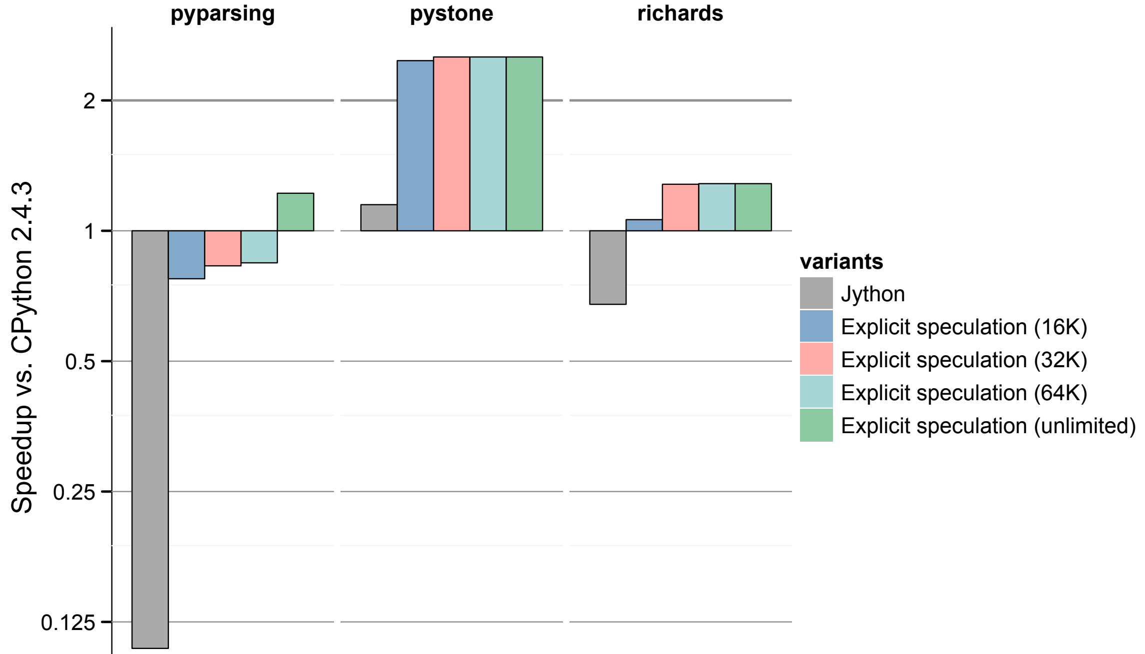

Figure 4.4: Jython speedups (all optimizations applied) over CPython on three Python benchmarks, with varied buffering capacity for speculative state.

Static region information conveys the range of observed behavior in the benchmarks, but not its frequency in execution. Figures 4.11, 4.12 and 4.13 thus weight the regions by their dynamic instruction counts (excluding nested regions).

Most explicit speculative optimizations I implemented did not measurably change the atomic regions' speculative data footprint. An exception was the Joni optimization to `pyparsing`, which eliminated string expansion from 2 bytes to 4 bytes per character. Between Figures 4.13, which plots the memory use of unoptimized `pyparsing` and 4.14, which includes the Joni optimization, the footprints of regions which perform regular expression matches (those clustered around 128 KB) were nearly halved.

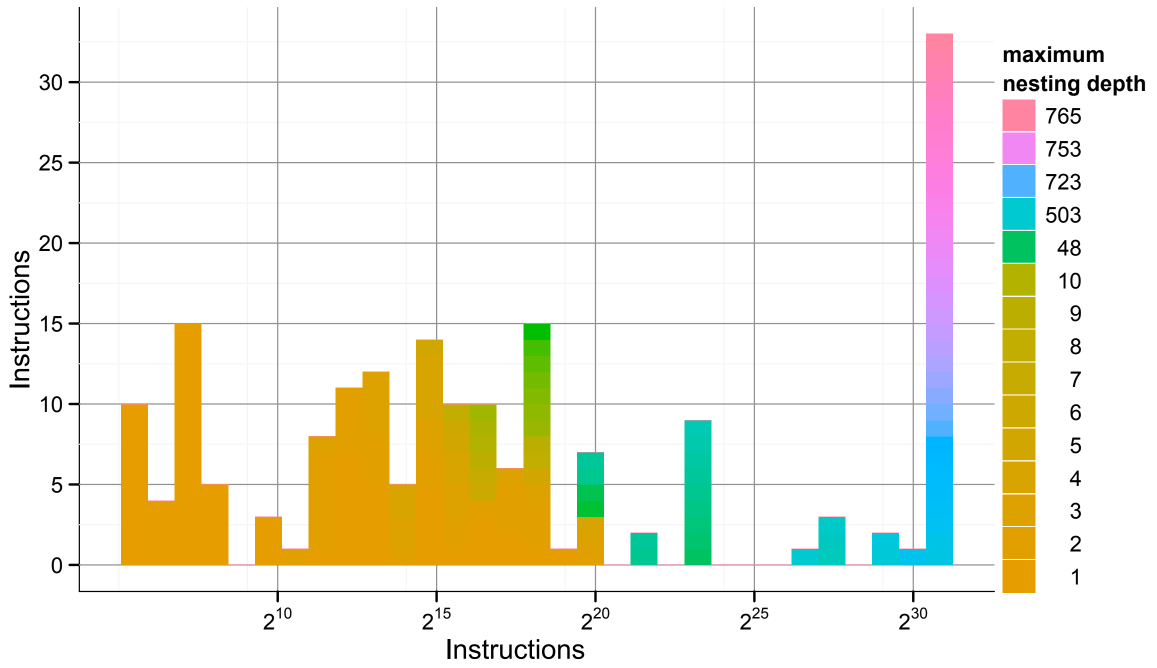

Figure 4.5: Instructions in static regions of `pystone`.

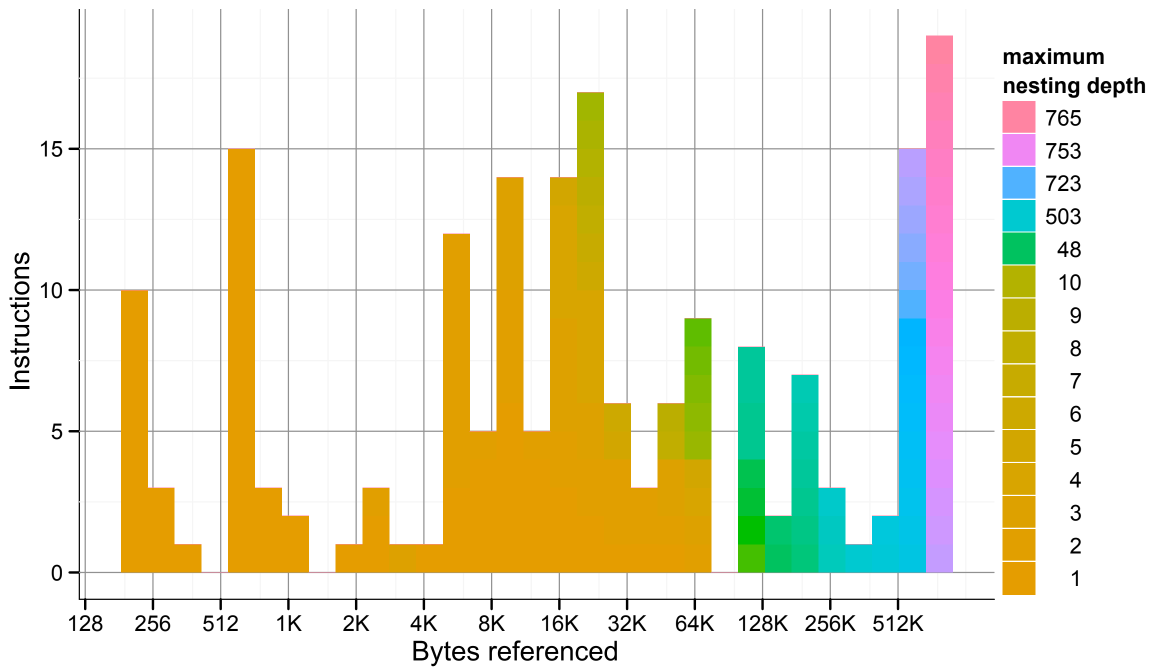

Figure 4.6: Memory use of static regions in `pystone`.

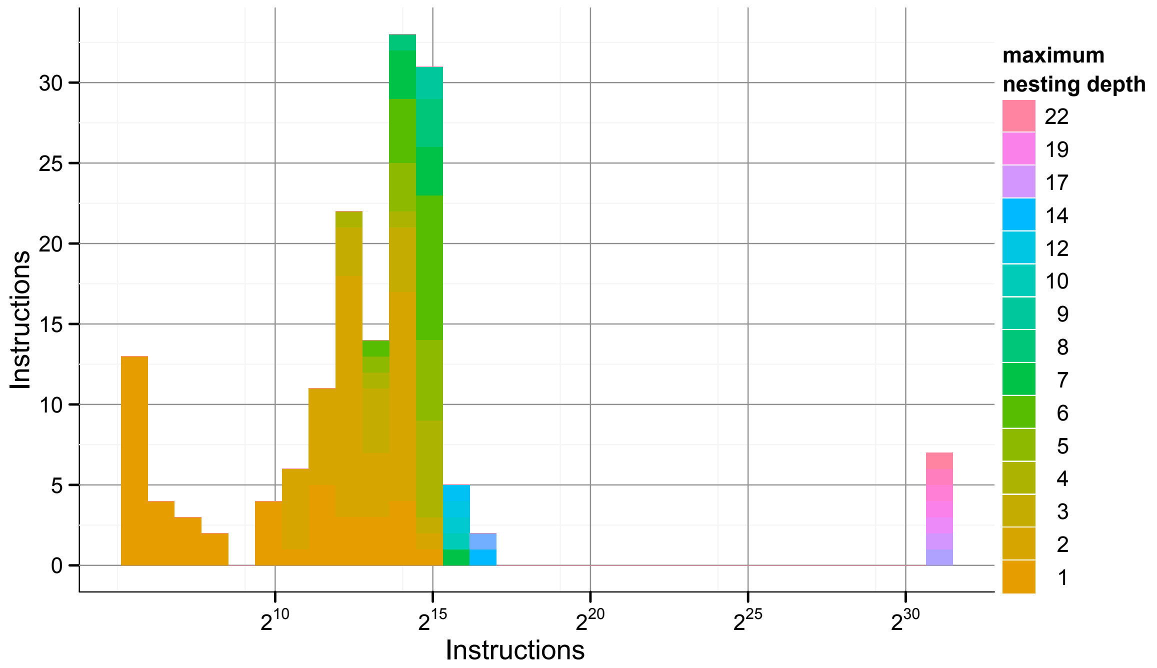

Figure 4.7: Instructions in static regions of `richards`.

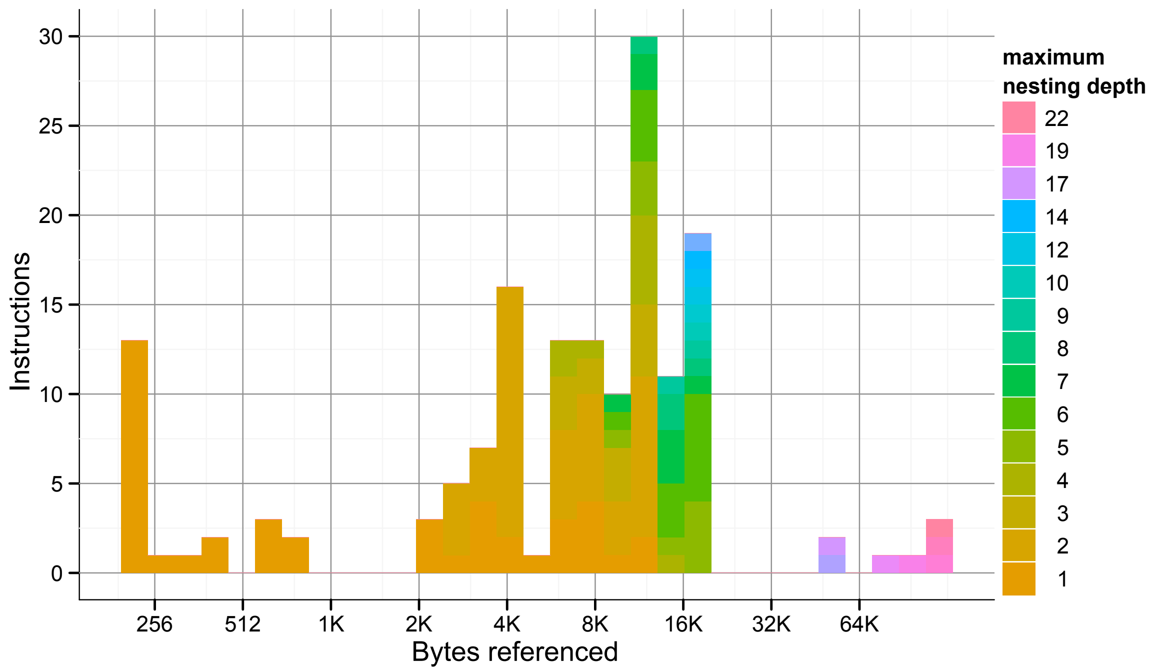

Figure 4.8: Memory use of static regions in `richards`.

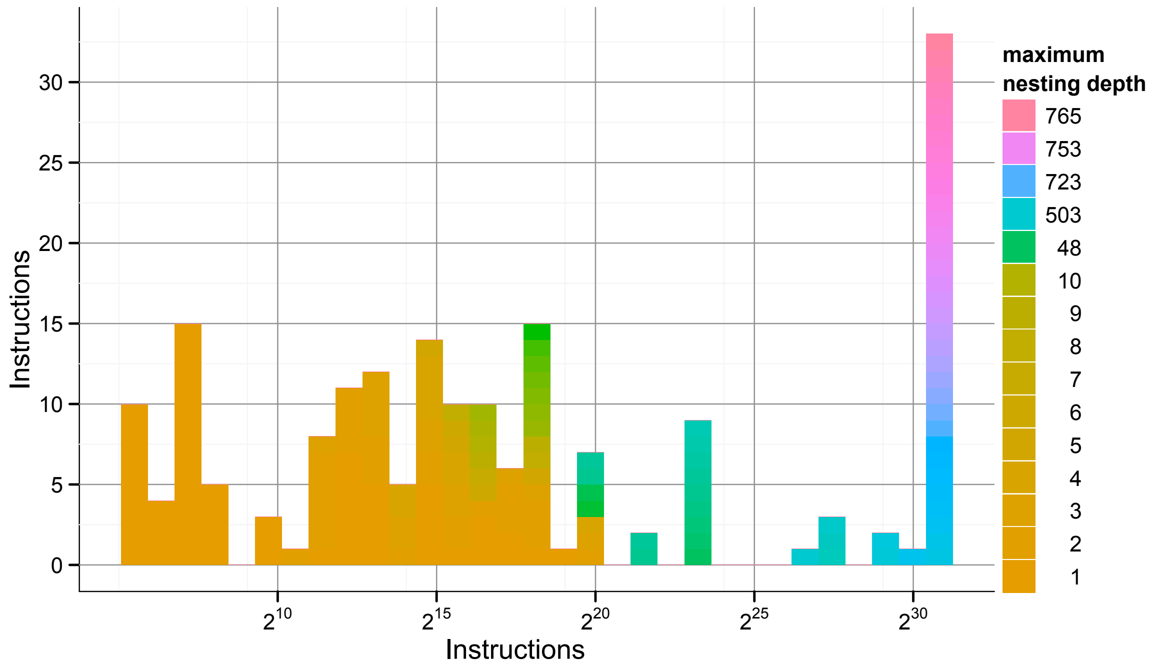

Figure 4.9: Instructions in static regions of `pyparsing`.

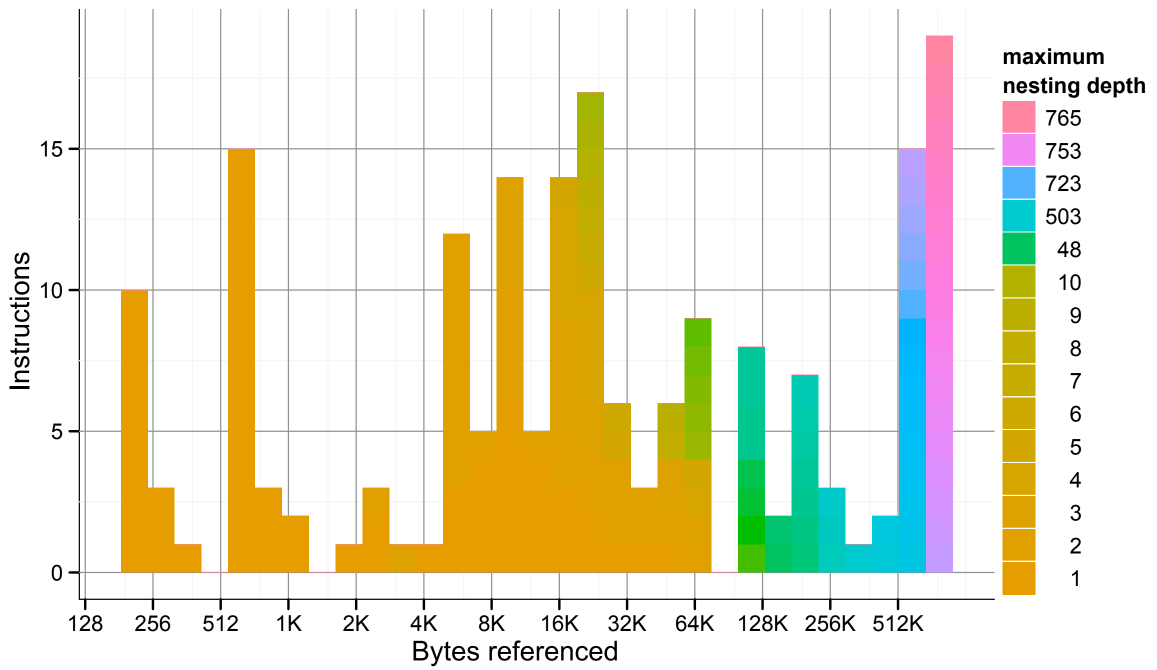

Figure 4.10: Memory use of static regions in `pyparsing`.

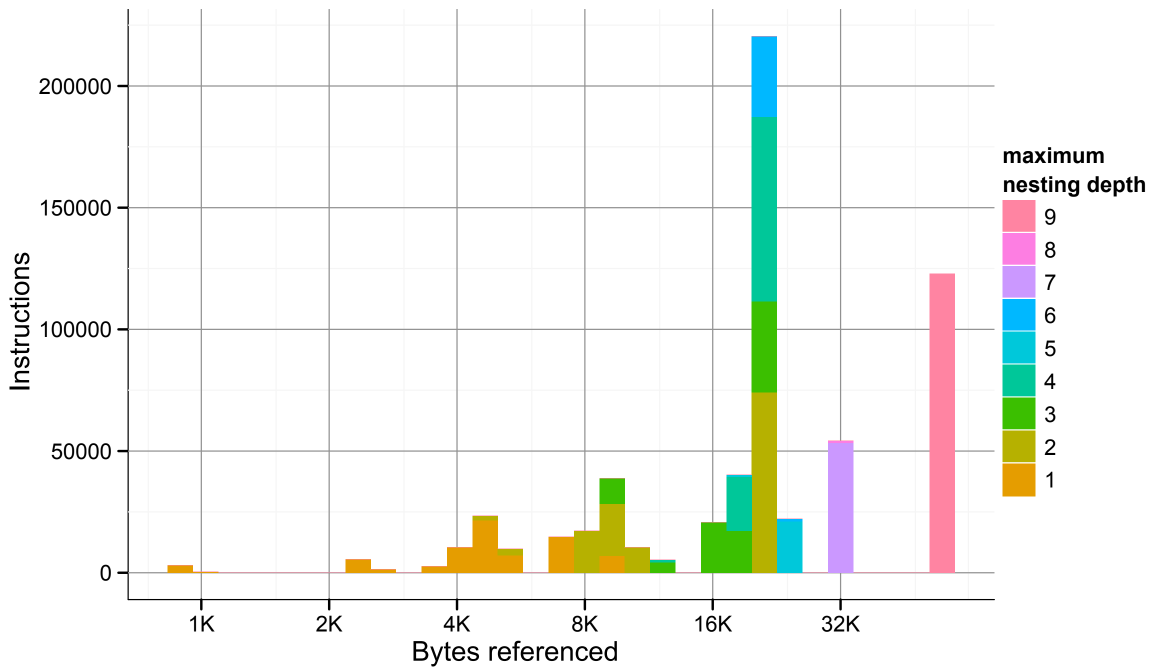

Figure 4.11: Memory use versus dynamic instruction count of atomic regions in unoptimized `pystone`.

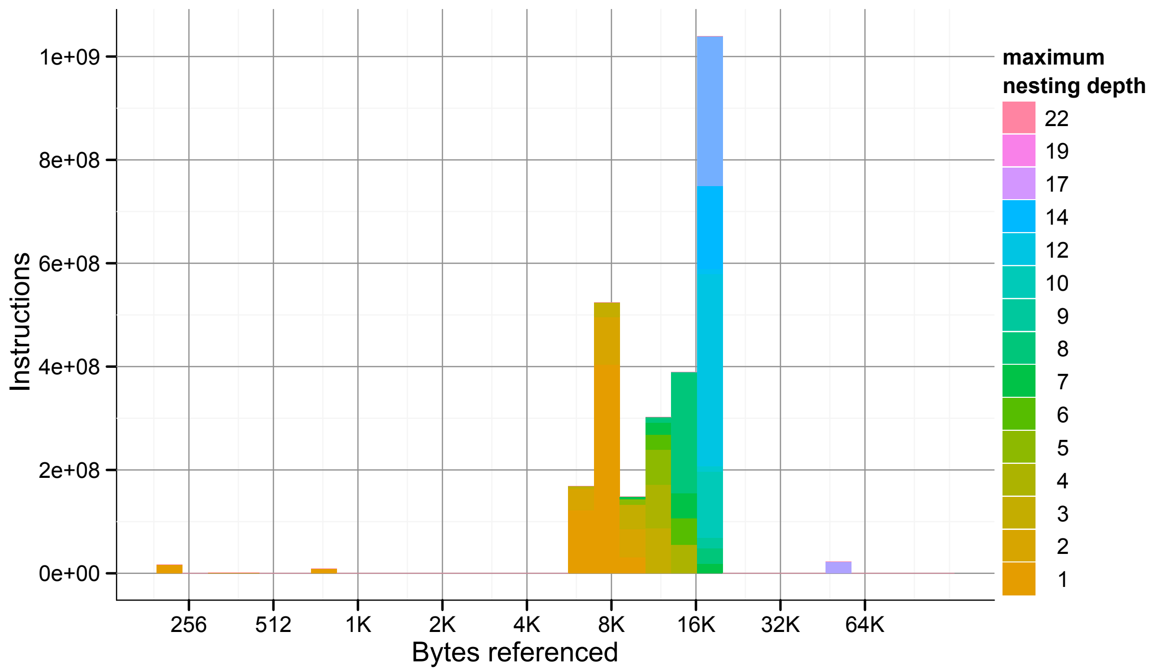

Figure 4.12: Memory use versus dynamic instruction count of atomic regions in unoptimized `richards`.

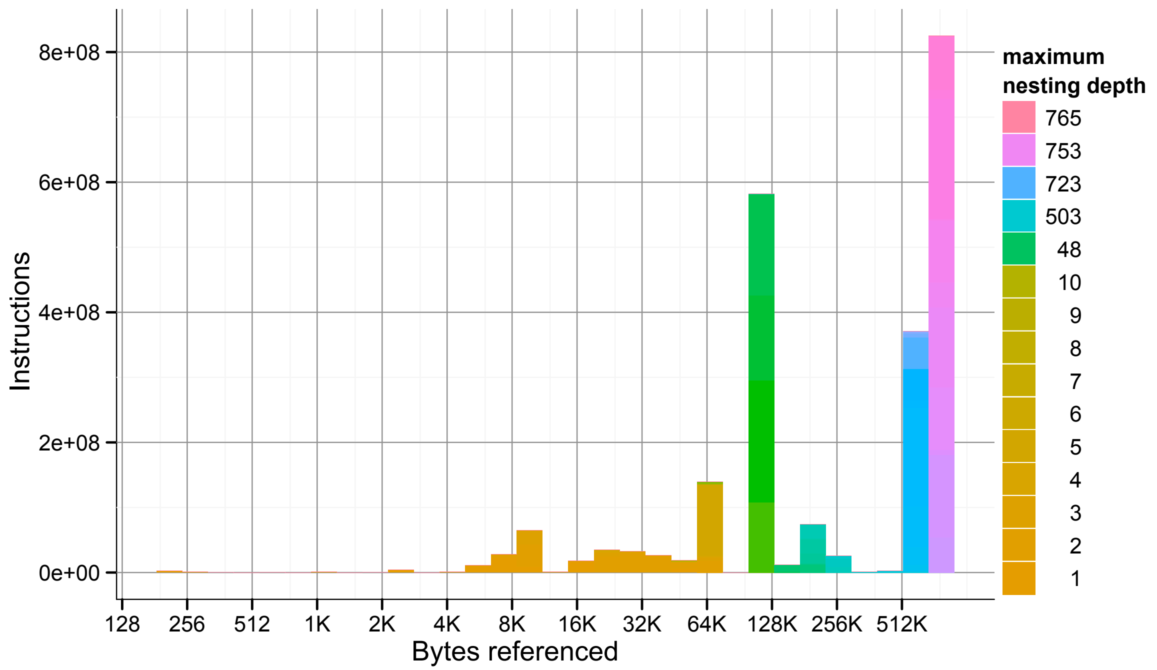

Figure 4.13: Memory use versus dynamic instruction count of atomic regions in unoptimized `pyparsing`.

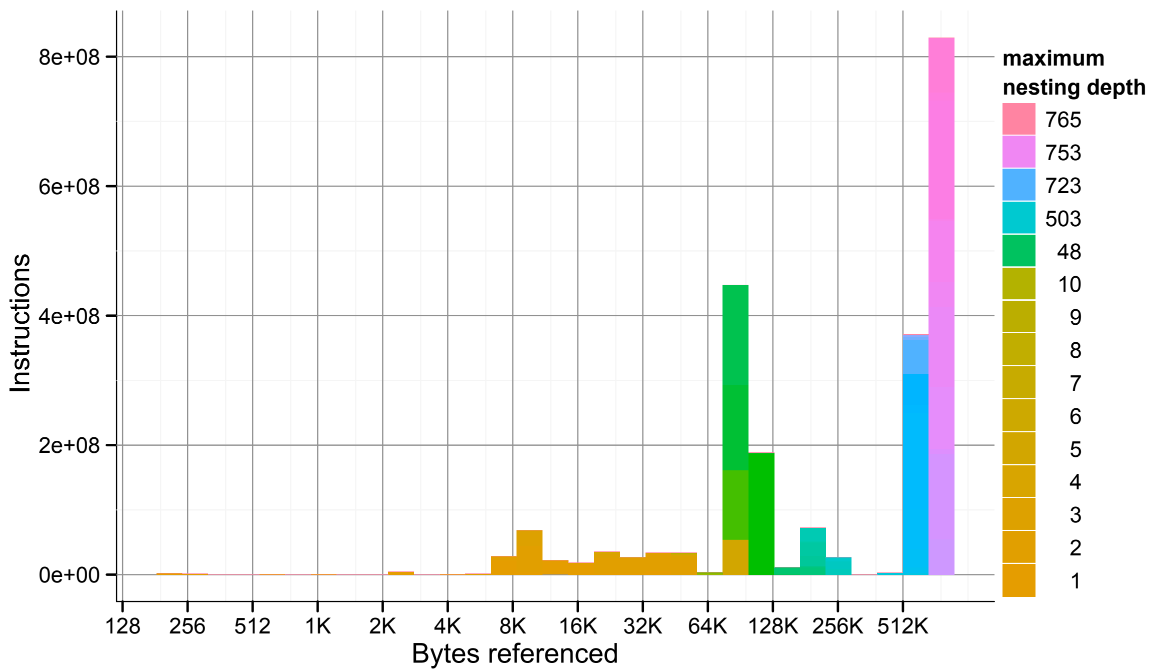

Figure 4.14: Memory use versus dynamic instruction count of atomic regions in `pyparsing` with explicit speculation.

Chapter 5

Explicit speculation on unmanaged runtimes

Explicit speculation may be adapted to serve the needs of unmanaged dynamic language runtimes. While this involves lower-level, less abstract expressions of atomic regions and implicit assumptions, the most significant change from managed runtime explicit speculation is a new type of assumption check—one which directly protects a region of memory.

I once again work with an implementation of the Python language, the CPython interpreter, and its dynamic specializer Psyco. Psyco generates native x86 code for common operations on basic Python types, such as iteration over lists and function invocation, and uses a more efficient "unboxed" representation of some object types (*e.g.*, integers, strings, and lists) if the objects live entirely within Psyco generated code regions. Psyco-specialized code usually runs faster than the equivalent interpreted execution, and can be an order of magnitude faster for certain applications, such as those involving collection operations or string manipulation [35].

To be sure, Psyco's performance could be improved further by more sophisticated optimizations, but that is not the focus of this chapter. Instead, I am concerned with implementing the optimizations in a way that preserves Python's semantics as a dynamic language. This is a concern because Psyco does not precisely emulate the semantics of the Python language. As clearly stated in Psyco's documentation, "Psyco makes assumptions that may be wrong (and will cause damage if they turn out to be)".

Examples of such assumptions are that class methods are not redefined, that

object instances do not have their class changed, global variables are not added or removed, builtin functions are not modified after module initialization, and no global variables are defined that override builtins.[1] Psyco makes these assumptions for performance, because explicitly checking them incurs overheads of a magnitude comparable to the benefit of specialization (as I show in Section 5.5). While many of the assumptions that Psyco makes hold for most Python programs, they prevent Psyco from being applied automatically. Instead, programmers have to specifically invoke Psyco to have their code specialized.

In this respect, Psyco's implementation serves as an upper bound of the performance for the optimizations it applies: a specializer with the same optimizations but that implements Python semantics perfectly should not outperform Psyco. Thus I seek to determine how close I can come to Psyco's performance while eliminating unchecked assumptions, such that Psyco can be applied automatically to all code in the presence of supporting hardware.

5.1 Fine-grain memory protection hardware

Recent architecture research has included a number of proposals for fine-grain memory protection techniques [6, 46, 48] that allow specifying protection granularities down to a single cache block, a word, or even a single byte, depending on the mechanism. These techniques have been proposed for flexible interprocess sharing (to avoid copying) [46] and software debugging [48]. Each of these techniques provide an interface for specifying access permissions for an arbitrary region of memory, such as the following:

```
void protect(addr_t start_addr, size_t size, access_type_t atype);
```

[1] Another class of assumptions Psyco makes consists of cases in which Psyco trades efficiency of execution in general for perfect emulation of CPython's behavior. For example, Psyco does not provide CPython's protection from stack overflow or properly handle recursive data structures.

Following the standard POSIX signal interface, each technique provides the ability to register a *callback function* that will be invoked when a thread touches a protected memory region. This callback provides the address of the faulting memory access, along with the type of access attempted and the context of the faulting thread:

```
void (*hndlr)(addr_t fault_addr, access_type_t atype, void *ucontext);
```

When an assumption has been violated, it will trigger a callback, which should then invalidate the specialized methods whose assumptions have been violated and remove the memory protection region, before returning to the application thread where the faulting access will be retried and succeed. An assumption registry tracks the relationship between assumptions, here represented by protected memory addresses, and the related specialized code.

While all of the above-referenced fine-grain memory protection proposals offer similar functionality to software, their hardware implementations differ dramatically. For these experiments, I selected the User Fault-On (UFO) bits system [6] for its low overhead and hardware simplicity.

The UFO proposal frees up a few of the error checking bits in ECC memory by encoding ECC at a larger granularity (*e.g.*, 128b vs. 64b), the same technique used to provide storage for the Alpha 21364's directory [16]. These bits are used to store one "fault-on" read and one "fault-on" write permission bit for each block of memory. When a memory block is brought on-chip, its UFO state accompanies it as part of the ECC state. In the processor cache, the UFO state is stored with the cache tag; in a system with 64-byte cache blocks, this represents a 0.4% storage overhead. Writes to the UFO bits are permitted in User mode, and are treated as writes to the cache block, forcing the processor to acquire *exclusive* coherence permission to the block and ensuring the coherence of the UFO bits. By piggybacking on the existing cache coherence protocol in this way, UFO avoids the TLB shootdown overhead associated with standard page protection.

Because UFO permissions are stored with the cache blocks they protect, UFO incurs no overhead when permissions are not violated. When a cache access is performed, the UFO bits are read along with the rest of the tag and compared with the type of access performed; when a protection-violating instruction attempts to retire, an exception is raised. Our implementation uses the standard x86 exception model and the Linux kernel's signal interface to communicate the faulting access back to the user thread. This signal invocation incurs significant overhead, but has little performance impact in practice because it is rarely invoked.

5.2 Psyco and specialization-by-need

Unlike other specializers, which operate on whole functions or program traces, Psyco's specializer tightly intersperses code generation with execution at a bytecode level until it has generated code for all the unique cases it encounters. This process, called *specialization-by-need*, allows Psyco to tolerate the lack of type information without having to collect value and/or path profiles ahead of time.

The process works by incrementally emitting small, specialized native *code buffers* linked by branches. Each code buffer uses the state of the execution up to that point (*e.g.*, the types of the objects that lead to this execution path) to generate code for the next set of bytecodes. At any point where Psyco cannot generate code because doing so would require a piece of runtime information, it stops code generation and leaves a stub, called a *respawn* code buffer,[2] that holds the state of the compilation and reinvokes Psyco if it gets executed. When a respawn point is reached, Psyco collects the run-time information it needs to continue code generation, emits a new code buffer, and continues execution at that code buffer. This process is best illustrated by an example.

[2]Psyco creates similar regions known as "code pause" and "changed global". I use "respawn" as a generic term in this paper to reference all points at which Psyco restarts its code generator.

Attribute lookup is a very common Python operation which takes two Python objects, an object and an attribute name, and returns the value the object has associated with that name, or an exception if the attribute isn't found. In CPython, like in Jython, its implementation involves a series of hash table lookups and pointer dereferences; it's also possible to interpose Python or native code in several places in the attribute lookup process. As the full flexibility of Python attribute lookup is seldom used in practice, it is an ideal candidate for specialization.

Psyco specializes attribute lookup for two common cases, but Psyco does not know while initially generating code which path(s) will be necessary at a given site.

- **The attribute's value is in the instance's dictionary.** Per-instance data is commonly stored here.
- **The attribute's value is in the dictionary of the instance's class or a superclass.** Methods are commonly stored here.

Presuming that this is the first access to the object, Psyco looks at the object's type when this code is being generated (call it a `C`), generates a guard that permits devirtualization of the object in the code downstream, and constructs a respawn point to handle the case when the guard fails. Then it generates code that checks whether a per-instance dictionary is available and, if so, retrieves the value from the dictionary if present. The cases where no dictionary is present or the dictionary lookup fail are left as respawn points (Figure 5.1a). If later, an instance is encountered that does not have a dictionary, the respawn point will be used to construct a code buffer that performs the dictionary lookup in the class's dictionary (Figure 5.1b). If the attribute is always found in the instance, the code to look in the class is never generated at all.

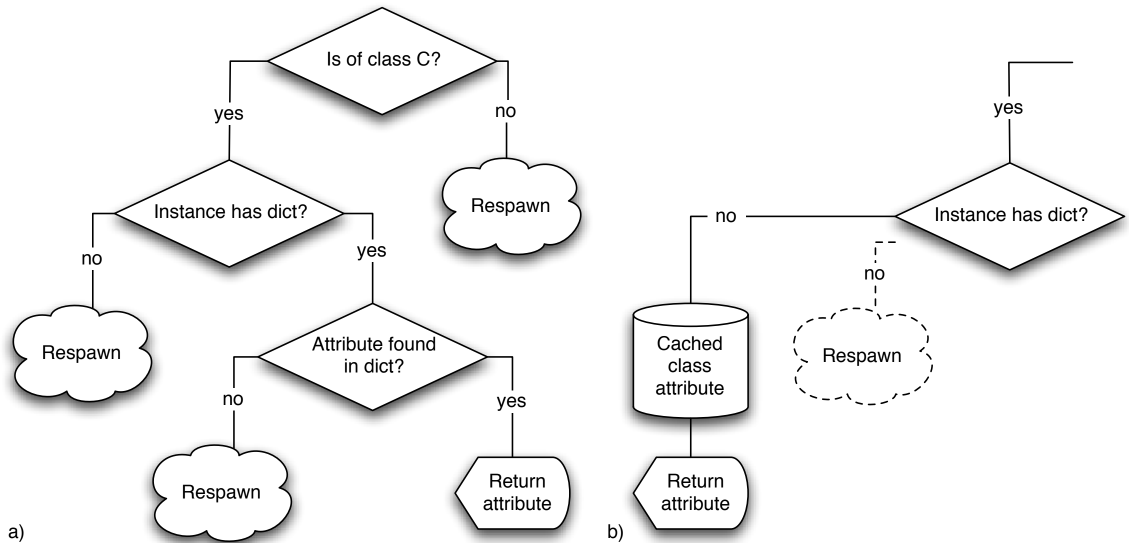

Figure 5.1: **Psyco's specialization of attribute lookup.** (a) If the instance always contains a dictionary with the requested attribute, no further code is generated. (b) If, instead, Psyco is reinvoked, it performs attribute lookup in the class, and if the attribute is present, it replaces the respawn buffer with code that returns a cached copy of the class attribute's value. Subsequent executions invoke the cache-returning branch directly, without involving Psyco's code generator at all.

5.3 Ensuring correct speculation in Psyco

So far, I have introduced the hardware mechanisms I apply to checking specialization assumptions and recovering from their violation, and the language and dynamic specializer to which I apply them. The techniques of fine-grain memory protection and checkpoint/rollback are orthogonal: they can be used individually, but they complement one another well. In this section, I explore several case studies: an assumption that can be checked using fine-grain memory protection alone, then one that exercises the checkpointing mechanism, and finally discuss how the two mechanisms interact.

5.3.1 Class attribute caching and dictionary watching

The example in Figure 5.1 demonstrates one of Psyco's unchecked assumptions: once it caches a class attribute, it never checks if the attribute has changed. This can result

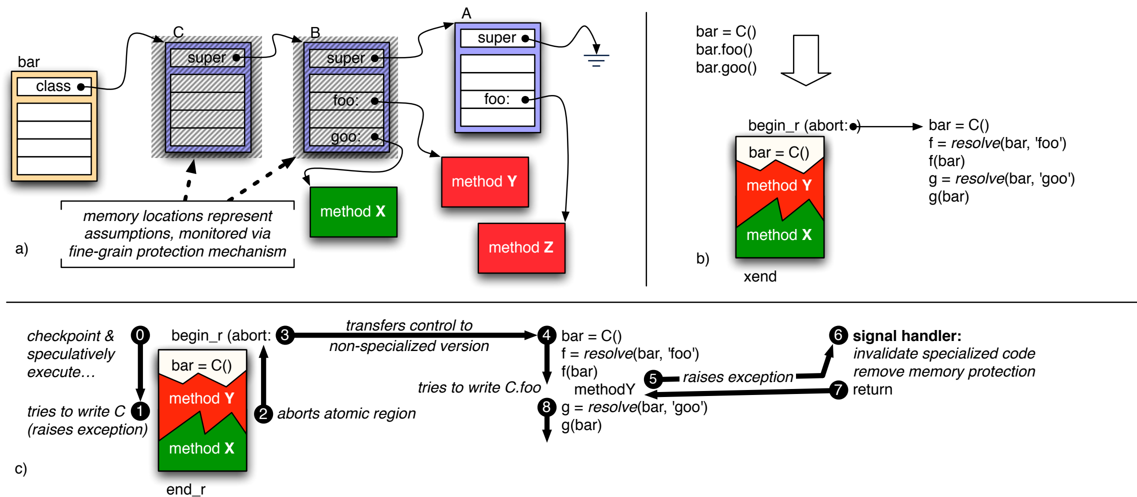

Figure 5.2: **Applying fine-grain memory protection and checkpointing/rollback support to optimistic specialization.** (a) During specialization, the assumed values' addresses are write-protected with fine-grain memory protection, (b) then the specialized code is emitted, wrapped in an atomic region, and a second non-specialized version of the code is generated to handle any cases when the region should fail. (c) An example invalidation scenario: a processor state checkpoint is taken, and the specialized code is speculatively executed (0), until method **Y** tries to redefine C.goo. Because C's dictionary is write protected, an exception is raised (1) before the method is redefined. This exception causes the atomic region to roll back to the checkpoint (2), dropping the raised exception. Hardware then transfers control to the non-specialized version (3), which begins execution (4). Again, an exception is raised, when method **Y** tries to write C's dictionary. (5) This time it succeeds, invoking the language runtime's signal handler. The signal handler uses the memory address of the violation to look up which specialized code fragments need to be fixed up or invalidated (6), before the fine-grain write protection is turned off. The signal handler returns (7), at which point the write to C's dictionary can finally complete. Execution in the un-specialized code continues (8) and because the standard method resolution is performed, method **W** is correctly invoked after method **Y**.

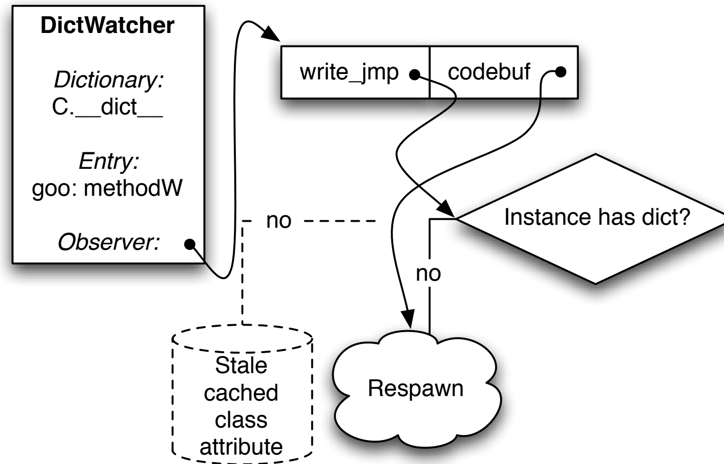

Figure 5.3: **A dictionary watcher in action.** The dictionary watcher notifies its observers when the dictionary's hash table adds, removes, or changes an entry with the specified key. In this case, it is watching class C's dictionary for changes to the goo key. The observer stores information—in this case, pointers to the address of a conditional branch instruction and its new target—which it uses to carry out an invalidation when notified of a change.

in a stale value being returned, or even a crash, if the cached object is deallocated.

To allow this specialization and maintain correct Python semantics, can apply the technique depicted in Figure 5.2a. We use UFO's fine-grain memory protection to *watch* the class dictionaries for changes. For every specialized code region that makes assumptions about a given dictionary, I allocate a software data structure that holds pointers to where the specialized code is linked in and to the respawn point[3] that the specialized code will replace (as shown in Figure 5.3). If the cached attribute later becomes invalid, these data structures are walked and, at each branch to a now invalid specialized code buffer, the respawn point is relinked, effectively unspecializing the code. Once an invalidation occurs, the related dictionary watcher can deregister itself from UFO, as the assumption no longer needs to be monitored. The next time Psyco reaches that place in the code, it re-respawns, installs a new dictionary watcher, and caches the new (correct) attribute. Thus, if a watched value changes multiple times

[3]As these respawn points are typically deallocated when they are replaced by code buffers, we have to modify Psyco to not deallocate them.

before the related specialized code is reinvoked, it only effects a single invalidation attempt.

Generally, Psyco makes assumptions about dictionaries relating to a specific key. There are three logical operations that can potentially invalidate an assumption: an entry with the key can be added, deleted, or changed. To ease the management of tracking these events, we implemented a standalone component called a *dictionary watcher*. While this dictionary watcher behaves much like an assumption trap in a managed environment, there is one important distinction. Rather than tracking *logical* operations on a dictionary like addition, deletion and modification, the watcher tracks *physical* modifications to the hash table's implementation. This has two implications: first, it needs to respond to UFO faults due to physical modifications that have no logical counterparts (*e.g.*, resizing a dictionary without changing any of its contents), and second, it needs to discern from the sequence of physical operations what logical operations are being performed on the dictionary. The first leads to the potential for unnecessary execution overhead, as the UFO protections will need to be updated on resizing. The second requires the dictionary watching code to implement a finite state machine which maps physical events to logical operations.

Because UFO faults are expensive (our current implementation uses the Linux kernel's signal mechanism) the viability of this approach relies on them being infrequent. This necessitates a low frequency of false positives—events that trigger UFO faults yet do not lead to invalidations of specialized code. Two potential sources of false positives must be considered: first, dictionaries that are being watched must be rarely modified, as the physical actions involved in most dictionary modifications will trigger UFO events even if for keys other than the ones being watched, and, second, if the fine-grain protection mechanism has a minimum granularity (64-byte blocks for our UFO implementation) then values located "nearby" watched items should be rarely modified. In practice, I have not found false positives to be a problem.

5.3.2 Class changing and recovery

The previous section's attribute changing only required use of our proposed hardware to watch memory for changes that invalidate Psyco's assumptions. Existing Psyco software mechanisms recovered from this invalidation. However, not all of Psyco's unchecked assumptions are as simple to recover from. Where necessary, Psyco can recover with the assistance of our atomic region execution hardware's checkpoint/rollback mechanism.

The primary benefit of adding explicit speculation to Psyco is *arbitrary recovery with low overhead*. Our modifications enable Psyco to recover from an invalidation at any point during execution without incurring additional overheads of code size, memory use or compiler complexity in reconstructing the circumstances of the invalidation.

For example, a Python object's class is just another of its attributes that can be assigned to (with a few restrictions). Psyco assumes that, as long as it's using an object in a single code buffer, the object's class remains constant. If this assumption is violated, Psyco may for example invoke the wrong class's methods on the object, typically yielding incorrect results. Psyco must therefore stop executing specialized code immediately upon a class change, a task for which a hardware rollback mechanism is ideally suited.

Unlike with a class attribute change, which invokes the commonly used functions for Python dictionary access, there exists a location to insert an assumption trap before the class of an object is changed: CPython's object_set_class function. Conveniently, this function is not called when an object's class is set as it is created, only when it is modified thereafter. I use a write barrier to react to a class change by wrapping this function with a version that performs an explicit abort if it is executing within an atomic region.

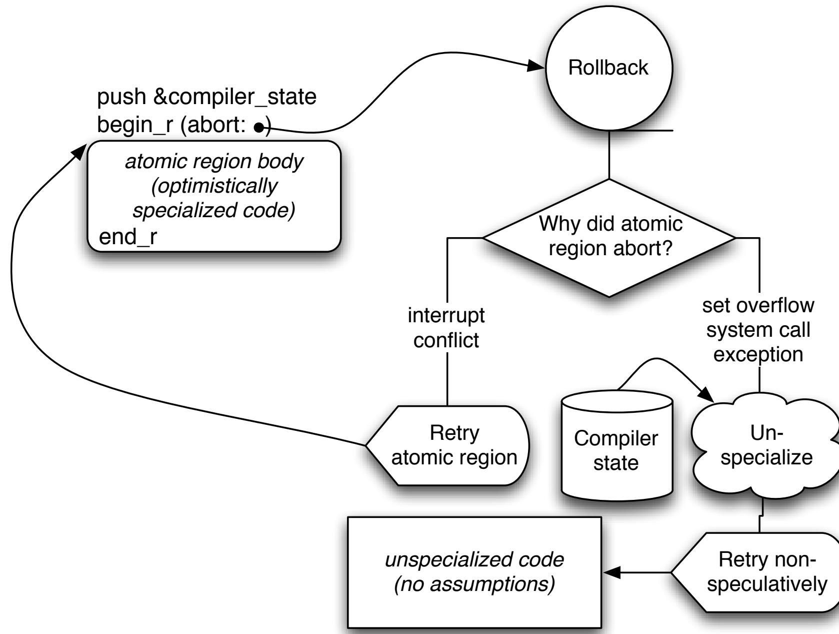

Figure 5.4: Recovery in Psyco.

An atomic region may abort for a number of reasons, which Psyco can query in an abort handler—a piece of code at an address to which the hardware jumps after rolling back to the checkpoint, as shown in Figure 5.4. A few reasons, namely interrupts and false sharing-induced conflicts, are expected to be transient and don't affect correctness, so the atomic region can retry as is. When an atomic region aborts because its speculative state overflows the processor cache, or a system call, exception, or explicit abort occurs, the modified Psyco discards all its assumptions and re-executes the code nonspeculatively along a slow, but correct, path, a process that closely resembles respawning.

The other "input" to the abort handler, other than the abort reason, is Psyco's compiler state (a pointer to which is pushed on the stack before the atomic region begin). Respawn buffers and similar structures in Psyco provide sufficient context for its compiler to create a generic, assumption-free version of the aborted region's code. In keeping with Psyco's philosophy of specialization-by-need, this safe version of the

code does not need to be generated until it is actually needed following an abort. Speculative execution resumes when the newly constructed code buffer exits. While temporarily ignoring all assumptions is over-conservative, it is trivial to implement in Psyco, requires no additional compiler state be retained beyond that which Psyco already requires, and is infrequently required.

With appropriate behavior for a region abort established, I placed atomic region boundaries in Psyco at appropriate locations to avoid degrading performance. Minimizing performance impact requires moderate atomic region sizes. As with Jython, if atomic regions are too small, the overhead associated with beginning and ending regions may not be negligible. If too large, they will potentially cause aborts due to cache overflows, and the occasional abort resulting from a stray interrupt will lead to a significant amount of work being lost.

I place atomic region boundaries in three sets of locations to generate modestly-sized regions while ensuring that Psyco can provide the "compiler state" necessary for recovery upon an abort. (Note that Psyco, unlike Jython, does not nest regions.) The first are function entry points and loop headers in specialized code, which prevent most forms of repeated execution inside a region that could lead to cache overflow. The second are entries to, and exits from, the Psyco compiler. (Psyco's compiler does not need to be speculatively executed: CPython and Psyco use a cooperative threading model, and there are no yield points in the Psyco compiler, so Python code can't execute and invalidate assumptions during compilation.) The third excludes external native function calls in specialized code from atomic regions, by ending a region before the call and beginning after it returns. Before invoking an external function, Psyco must relinquish control: the runtime state must be consistent enough that the existing exception-handling code in the callee can handle all potential conditions. Excluding external functions also eliminates from atomic execution many circumstances in which hardware would be unable to buffer speculative state—for example,

file I/O. Following an external call, Psyco includes a respawn point in case the callee generates an exception. I use the compiler state derived from this respawn point and its immediately preceding snapshot to revert to safe nonspeculative execution if the following region aborts.

5.3.3 UFO in atomic regions

There are two kinds of invalidations requiring atomic region rollback: *local invalidations*, which affect the currently executing code buffer, and *global invalidations*, which affect other code. Changing an object's class is an example of a local invalidation, because the scope of the class assumption extends only as far as the current code buffer; Psyco already dispatches based upon an object's class when entering or reentering other code buffers. Changing a Python builtin such as `abs` is an example of a global invalidation, because all code with a reference to the set of builtins or module globals containing `abs` is now incorrect.

Because global invalidations should be immediately visible, they cannot execute within an atomic region. Therefore, an attempt to invoke a UFO handler inside an atomic region causes the region to abort. When the hardware re-executes nonspeculatively, the same protected address is accessed, but this time, we execute the UFO handler and perform global invalidation. This sequence is depicted in Figure 5.2c.

Where local invalidation is triggered by a memory access, it is implemented by watching a piece of memory with no handler attached. This still causes the atomic region to abort and safely retry nonspeculatively. Outside an atomic region, the UFO hardware intercepts such an access, but software reacts by doing nothing.

5.4 Other assumptions

For completeness, I briefly present the methods by which I detect and react to the invalidation of the other unchecked Psyco assumptions. Some utilize hardware watchers, some involve an atomic region abort, and some do both.

5.4.1 Class attributes with multiple inheritance

The discussion of class attribute caching in Section 5.3.1 omits one piece of the puzzle: Python supports multiple inheritance. If an attribute isn't present in an instance's class, lookup continues with each of the class's superclasses in a deterministic sequence called the *method resolution order* (MRO). When watching an attribute for changes, Psyco in fact needs to monitor the attribute dictionary of the class where the attribute is defined, and the dictionaries of all subclasses in the MRO *preceding* that class. For example, if class D's MRO is (D,C,B,object) and an access to the attribute B.a for an instance of D is cached, Psyco must watch for the creation of D.a and C.a as well as for the change or deletion of B.a. Similarly, invalidating D.a also effects caches of a for the class D and its subclasses which do not define a themselves.

A similar issue could arise with Python's dynamic super type, used for explicit references to superclass attributes according to a class's MRO.

5.4.2 Changing __bases__

With some constraints, it's possible to change the __bases__ attribute of a Python class. __bases__ contains a list of classes from which a class inherits, and the order of the items in the list influences the class's method resolution order. There is a similar function to the __class__ setting one, type_set_bases, but this time, I insert an assumption trap as a wrapper function which performs a global invalidation. If executing within an atomic region, the wrapper function aborts the region explicitly;

otherwise, it invalidates all the code buffers which cache attributes for the class and its subclasses.

5.4.3 Builtins

I modified Psyco to associate each Python builtin with the Psyco-compiled functions which reference that builtin. Then, I use dictionary watchers to perform a global invalidation when a builtin is redefined or a same-named global created. Because most builtins have no "exceptional condition" to test for, with an associated respawn point, they can't invalidate in place by swapping a pointer as I do with class attributes. Instead, I invalidate the entire compiled function and remove the builtin from the list of Psyco-recognized builtins for the remainder of the execution session.

5.4.4 Changing tp_getattro and __getattribute__

It's possible not only to redefine a class's attributes, but override the very process of looking up attributes in an instance, by changing the tp_getattro "slot" in a PyTypeObject (from C) or a class's __getattribute__ attribute (from Python). There is a location in which to insert an assumption trap for such far-reaching slot changes, CPython's update_one_slot function, but it has several other uses, so it's easiest to simply watch the tp_getattro slot for changes. On a change, Psyco aborts the current region as usual, and for each place in the generated code which performs an attribute lookup on objects this class, I replace the conditional branch to the respawn buffer (which handles the case in which the attribute isn't present in the instance) with an unconditional branch to a new respawn buffer which, when entered, will recognize the nonstandard tp_getattro function and generate code to use the user's attribute lookup mechanism. This thereby bypasses both Psyco's instance and class attribute lookup specializations.

5.4.5 Runaway operations

Psyco does not check for stack overflow in user code, nor does it support comparing circular data structures without infinitely recursing. A stack overflow in Psyco as a result of either of these operations causes the runtime to crash. With recovery enabled, these operations cause a set overflow instead, and the safe version executed after the atomic region abort does not include the Psyco optimizations which do not check for adequate stack space.

5.5 Results

I performed an evaluation of these techniques with two intended goals: first, to demonstrate that these techniques do in fact enable the implementation of precise Python semantics and, second, to measure the overhead of their introduction. I found that the overhead was negligible.

To emulate a system where the proposed hardware mechanisms were available, I implemented the UFO and atomic region execution hardware as extensions to the x86 version of the Virtutech Simics full-system simulator [24] and the Linux 2.6.15.4 kernel. These experiments were performed on a simulated uniprocessor Intel Pentium 4-based machine.

To verify the functionality of my changes I implemented Python "microbenchmarks" that tested each of the language features discussed in Section 5.4; each of these demonstrated correct Python semantics. In addition, I identified two larger Python applications with which to test correctness: an application of the Django Web framework's templating system (`django`) and a Verilog file format parser constructed with the `pyparsing` toolkit (as discussed in the previous chapter). Both benefit from Psyco's specialization, but employing Psyco careless results in each executing incorrectly with the unmodified Psyco. In particular, Django changes class

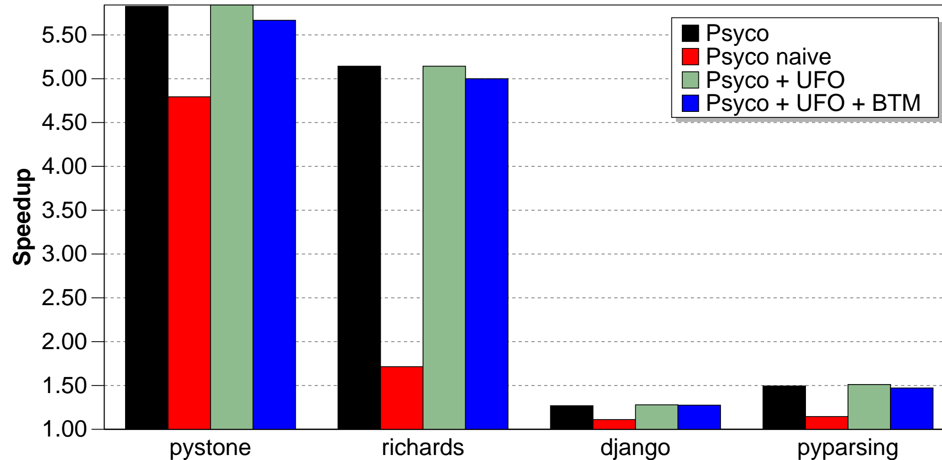

Figure 5.5: Speedups on a real machine, indicative of a lower bound on overhead for the Psyco modifications.

attributes, and `pyparsing` modifies objects' classes, though neither performs these operations frequently. Nevertheless, the modified Psyco correctly executes them.

As with Jython, I also collected performance estimates on real machines, since the UFO and atomic region execution mechanisms should only introduce overhead when a fault is detected or when a region is aborted. As a result, a real machine should serve as a good estimate for performance of the modified Psyco when specialization assumptions are not violated, the cases that require the proposed hardware and expose its overhead. Note that these experiments do account for the overhead of the additional instructions that were introduced to set up dictionary watchers for UFO as well as start and stop regions, check whether a region is in progress, and to keep Psyco compiler state on the stack where necessary for recovery. This code is implemented using normal x86 instruction sequences that are recognized by the modified version of Simics.

Figure 5.5 presents tests on an Intel Core 2 Duo E6600 (running in 32-bit mode) with Linux kernel 2.6.9-42.0.8ELsmp. On this system I ran an unmodified CPython (version 2.4.3) with four different versions of Psyco 1.5.1.[4] "Psyco" represents an un-

[4]The Psyco results in the previous chapter employed CPython 2.6 with Psyco 1.6. The modi-

modified Psyco. “Psyco (naïve)” is a version of Psyco where the attribute lookup specialization has been explicitly disabled, which executes correctly but at a considerable performance penalty indicative of the benefit of this specialization. “Psyco + UFO” is a version extended to register dictionary watchers, and “Psyco + UFO + BTM” is the same further extended to execute specialized code speculatively. Speedups are shown relative to the CPython interpreter for `django` and `pyparsing`, as well as two standard benchmarks commonly used to measure the speed of Python implementations: `pystone`, computationally intensive procedural code, and `richards`, which is heavily object-oriented.

Only `pystone`, which shows little object orientation, sees a modest impact from disabling the attribute specialization; in all of the other applications it nearly wipes out Psyco's benefits. In contrast, registering dictionary watchers (“Psyco + UFO”) adds negligible overhead, as this occurs rather infrequently (only during code generation). Actually, the “Psyco + UFO” case slightly outperforms unmodified Psyco; while this result was repeatable, I believe it to be a second-order effect resulting from a different memory layout, not actual differences in execution. The “Psyco + UFO + BTM” case incurred a slight slowdown with respect to the unmodified Psyco, but in the worst case this was only 2%.

While the functional simulator does not provide real performance numbers it can provide dynamic instruction counts, which further support the assertion that the benefit of the proposed hardware support can be exploited with little impact on the code's performance. In all but one case, the instruction count overhead with UFO and BTM-enabled assumption checking is negligible (shown in Figure 5.6). `pystone` is the benchmark with the highest proportion of Psyco-generated code to total code executed, which magnifies the region setup overhead.

---

fications discussed in this chapter were performed on the then-current versions of CPython and of Psyco.

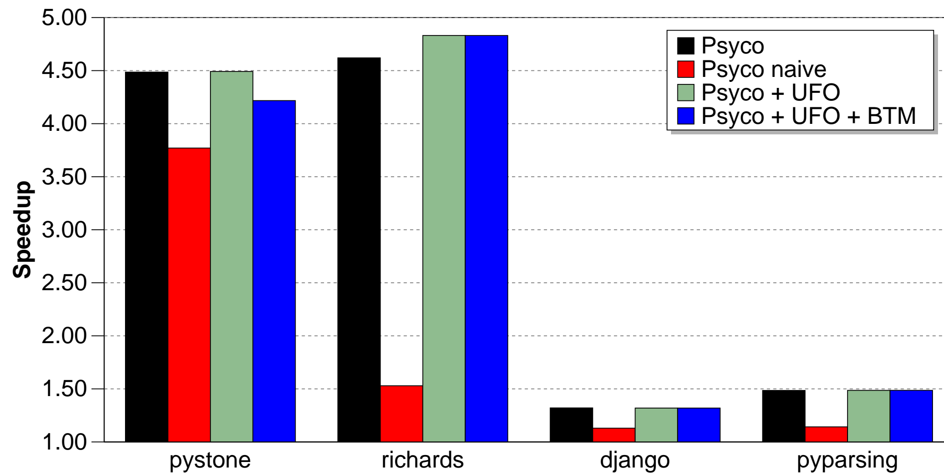

Figure 5.6: Instruction count-based “speedups” relative to interpreted CPython execution on a simulated system.

Finally, Table 5.1 presents a characterization of atomic region behavior for two benchmarks, in particular identifying aborted regions by their reason for aborting. Most aborts in simulation occurred due to atomic regions that were interrupted by INTERRUPTs; even at 0.1% these abort rates are pessimistically high because Simics has an unrealistically high interrupt rate to enable interactive use. The small number of SET_OVERFLOW aborts indicate that these heuristics successfully pick well-sized regions. That a small proportion of Psyco instructions leads to an approximately 30% speedup indicates that Psyco is effective at specializing where it can, but its coverage of Python operations is incomplete.

	`django`	`pystone`
Regions		
Total	1398	2512
Committed	1394	2509
EXCEPTION	0	0
INTERRUPT	1	3
SYSCALL	1	0
SET_OVERFLOW	2	0
EXPLICIT_ABORT	0	0
Instructions		
Total	2407753	1147561
Transactional	103930	70015
Committed	99524	70005

Table 5.1: Atomic region behavior.

Chapter 6

Conclusion

Explicit software speculation is a practical approach to improving the performance of dynamic languages whose common case semantics directly map to the capabilities of an existing managed runtime. With the first hardware supporting atomic region execution soon to be generally available, explicit speculation should not only deliver real performance benefits for widely used dynamic language code on managed runtimes, but the ability to test outside of a simulated environment will facilitate the further development of managed runtime and hardware support to make explicit speculation more effective and simpler to employ.

The set of optimizations possible with explicit speculation largely mirror those available by traditional software-only means. Only the mechanism of implementation differs. Explicit speculation gives a dynamic language implementer the ability to express speculative optimizations at a high level, without full knowledge of runtime behavior, yet maintain correctness and experience minimal runtime performance penalty for misspeculation in long-running applications. Its approach to optimization is simply "write efficient code for the common case, then codify the assumptions that make it correct." Explicit speculation constructs, such as assumptions and speculative atomic regions, are simple to reason about for the programmer and map cleanly to existing managed runtime facilities.

With correctly specified assumptions, explicit speculation can turn an incomplete static analysis or approximate runtime measurement into an effective speculative optimization with little effort. In implementing several optimizations for Python, I

showed that common case behavior of dynamic constructs can be simply established at compile time or via partial evaluation at load time. Both techniques require minimal infrastructure when compared with a profile feedback system.

While I have discussed in depth the tradeoffs involved in developing optimizations with explicit speculation, two concerns dominate in practice. First, can speculative state's scope be easily bounded and contained within an atomic region? At worst, global speculative state may need to be maintained by otherwise nonspeculative code (with an attendant slowdown). In this case, explicit speculation may offer no additional benefit over software-only approaches than elimination of synchronization overhead (*e.g.* in global caching, Section 4.1.3). Second, does an optimization's scope fit within the hardware's ability to buffer its speculative state? Certain control flow constructs such as recursion, as well as references to large amounts of data, can defeat a simple region selection algorithm. Mating explicit speculation to existing, more sophisticated techiques for region selection (developed for compiler-based speculative atomic regions [28]) should resolve many practical instances of this issue.

Selecting speculative optimizations with these considerations in mind delivers a fast implementation cycle from performance problem to solution. This encourages the use of explicit speculation to attack hot spots in individual applications (*e.g.*, regular expression evaluation as discussed in Section 4.1.5) as well as in dynamic language implementations as a whole. My experience suggests that the interfaces for explicit speculation and assumption verification I developed will be applicable in areas beyond dynamic language implementation.

In the following section, I place explicit speculation in the context of its alternatives for dynamic language implementation. In Section 6.2, I then suggest future hardware and software developments with the potential to broaden explicit speculation's applicability.

6.1 Explicit speculation for dynamic languages

Explicit speculation is simpler to implement than existing alternatives for improving dynamic language performance.

- One potential alternative is to extend a managed runtime's semantics to accommodate those of an additional language (*e.g.*, MagLev [14]). Repeating this process—extending a runtime's semantics to accommodate those of several additional languages—would require many systemic changes and result in a complex, unwieldy system.

- Building a language-specific runtime generating native code (*e.g.*, V8 [5]) is the only approach that has been reliably shown to work, but involves a nearly top-to-bottom reimplementation effort, requiring resources not available in many dynamic language communities.

- Static analysis-based approaches for dynamic language code either have extremely limited applicability (*e.g.*, ShedSkin [12]), or require a large implementation effort yet often require source code be modified for compatibility, through annotations or the use of dynamic language subsets or variants (*e.g.*, RPython [4], JSCompiler [41]).

- Parallel dynamic optimization systems constructed atop a managed runtime have been somewhat successful (*e.g.*, JRuby [30]), but are limited—in performance, correctness, or both—by the cost of supporting recovery at a high level. Such systems perform a limited amount of profile-guided speculative code generation without the managed runtime's explicit knowledge. They must embed into the speculative code functionality necessary to support potential future recovery from misspeculation.

In this last case, the intermixing of speculative code with “bread crumbs” to support later recovery leaves the resulting code less idiomatic, thus less predictably optimizable by the managed runtime [29]. Faced with inconsistent performance, the implementer then typically tunes aspects of the generated managed code to cater to a single managed runtime implementation. Modifications to the dynamic language code, upgrading or switching managed runtime implementations may then still result in seemingly counterintuitive performance characteristics. By comparison, in primarily eliminating recovery support from speculative execution (save for explicit assumption checks and transition code), explicit speculative code is easier for the managed runtime to optimize and exhibits more predictable behavior.

6.2 Future directions

Explicit speculation's needs map quite well to the capabilities of near-future atomic region execution hardware, making particular use of the ability to examine the reason a region aborted. Aside from incremental improvements in speculative state buffering capacity and fine-grain memory protection to extend explicit speculation to unmanaged code, hardware additions likely to benefit explicit speculation would expose the ability to perform nonspeculative memory accesses inside an otherwise-atomic region.

Nonspeculative *reads* inside an atomic region would permit access to shared, guaranteed read-only data, such as reference tables, reducing the footprint of a speculative region. Nonspeculative *writes* could be used by inline assumption checking code to export information regarding an inline assumption violation, eliminating redundant checks in recovery code. Both reads and writes could reference speculative state whose lifetime is guaranteed not to extend beyond an atomic region.

To implement region regeneration—that is, generation of a speculative replacement region to replace an existing atomic region—an explicit speculation runtime

must support the ability to exchange code implementations. Psyco already includes such a mechanism which I exploited; bytecode instrumentation provided as part of Java's debugging and profiling interfaces offers a partial solution (methods can be modified but not added [43]). In future, explicit speculation could take advantage of the facilities of the the Da Vinci Machine (MLVM) project for Java [39]. MLVM provides a *method handle* primitive and customizable dispatch via the invokedynamic mechanism, such that one or more alternative, compatible implementations and a fallback *bootstrap* implementation may be swapped into any call site. Call sites' bindings to method handles may be invalidated individually or in bulk. The MLVM functionality maps well to the needs of assumption invalidation; however, it would require that replaceable speculative regions be enclosed within their own methods.

In order to facilitate more sophisticated atomic region selection methods, the ability to generate transition code and evaluate assumptions could be partially delegated to the managed runtime, by providing standard interfaces for inline assumption expression and conversion between speculative and nonspeculative versions of objects. With this information, the managed runtime could adjust region boundaries by moving or duplicating the necessary transition and assumption enforcement code. Regions could thus be split as well as aggregated in order to adapt to the hardware's capabilities.

References

[1] PyPy microbenchmark suite. URL http://codespeak.net/pypy/dist/pypy/translator/microbench/.

[2] Pystone benchmark. URL http://svn.python.org/projects/python/trunk/Lib/test/pystone.py.

[3] C. S. Ananian, K. Asanović, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded Transactional Memory. In *Proceedings of the Eleventh IEEE Symposium on High-Performance Computer Architecture*, Feb. 2005.

[4] D. Ancona, M. Ancona, A. Cuni, and N. Matsakis. RPython: a step towards reconciling dynamically and statically typed OO languages. *Proceedings of the 2007 symposium on Dynamic languages*, pages 53–64, 2007.

[5] L. Bak et al. V8 JavaScript Engine. URL http://code.google.com/p/v8/.

[6] L. Baugh, N. Neelakantam, and C. Zilles. Using hardware memory protection to build a high-performance, strongly-atomic hybrid transactional memory. *ISCA'08. 35th International Symposium on Computer Architecture*, pages 115–126, 2008.

[7] C. Consel and F. Noël. A general approach for run-time specialization and its application to C. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 145–156, 1996.

[8] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. *SIGOPS Oper. Syst. Rev.*, 40(5):336–346, 2006. ISSN 0163-5980.

[9] D. Dice, M. Herlihy, D. Lea, Y. Lev, V. Luchangco, W. Mesard, M. Moir, K. Moore, and D. Nussbaum. The Adaptive Transactional Memory Test Platform: A tool for experimenting with transactional code for Rock. In *Proceedings of the the third annual ACM SIGPLAN Workshop on Transactional Computing*, 2008.

[10] D. Dice, M. Herlihy, D. Lea, Y. Lev, V. Luchangco, W. Mesard, M. Moir, K. Moore, and D. Nussbaum. Applications of the Adaptive Transactional Memory Test Platform. In *Proceedings of the the third annual ACM SIGPLAN Workshop on Transactional Computing*, 2008.

[11] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. Technical report TR-2009-180, Sun Microsystems Laboratories, 2009.

[12] M. Dufour. Shed Skin: an experimental (restricted) Python-to-C++ compiler. URL `http://code.google.com/p/shedskin/`.

[13] A. Gal, M. Bebenita, and M. Franz. One method at a time is quite a waste of time. In *Proceedings of the Second ECOOP Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS 2007)*, pages 11–16, July 2007.

[14] GemStone Systems. MagLev: a fast, stable, Ruby implementation with integrated object persistence and distributed shared cache. URL `http://maglev.gemstone.com/`.

[15] B. Goetz. Optimistic thread concurrency: Breaking the scale barrier. Technical report, Azul Systems, Inc., 2006. URL `http://www.azulsystems.com/products/whitepaper/wp_otc.pdf`.

[16] L. Gwennap. Alpha 21364 to ease memory bottleneck. *MICROREPORT*, Oct. 1998.

[17] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, May 1993.

[18] U. Hölzle. *Adaptive optimization for Self: reconciling high performance with exploratory programming*. PhD thesis, Stanford University, Department of Computer Science, 1994.

[19] J. Hugunin et al. IronPython: a fast Python implementation for .NET and Mono. URL `http://www.codeplex.com/IronPython`.

[20] K. Kosako. Oniguruma. URL `http://www.geocities.jp/kosako3/oniguruma/`.

[21] A. Kuchling. *Beautiful Code*, chapter 18, Python's Dictionary Implementation: Being All Things to All People. O'Reilly, 2007.

[22] D. Lea et al. JSR 166: Concurrency Utilities. URL `http://jcp.org/en/jsr/detail?id=166`.

[23] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 190–200, 2005.

[24] P. S. Magnussen et al. Simics: A full system simulation platform. *IEEE COMPUTER*, 35(2):50–58, February 2002.

[25] P. McGuire. Pyparsing: a general parsing module for Python. URL `http://pyparsing.wikispaces.com/`.

[26] M. Mielczynski. Joni. URL `http://svn.codehaus.org/jruby/joni/`.

[27] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based Transactional Memory. In *Proceedings of the Twelfth IEEE Symposium on High-Performance Computer Architecture*, Feb. 2006.

[28] N. Neelakantam, C. Zilles, R. Rajwar, S. Srinivas, and U. Srinivasan. Hardware Atomicity: An Effective Abstraction for Reliable Software Speculation. *IEEE Micro*, pages 21–31, 2008.

[29] C. Nutter. Exploring JRuby performance on HotSpot?, April 2009. URL `http://permalink.gmane.org/gmane.comp.java.openjdk.hotspot.compiler.devel/1052`.

[30] C. Nutter et al. JRuby: a Java powered Ruby implementation. URL `http://jruby.codehaus.org/`.

[31] E. Phoenix et al. Rubinius: The Ruby virtual machine. URL `http://rubini.us/`.

[32] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2001.

[33] R. Rajwar and J. R. Goodman. Transactional Lock-Free Execution of Lock-Based Programs. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[34] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing Transactional Memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, June 2005.

[35] A. Rigo. Representation-based just-in-time specialization and the Psyco prototype for Python. In *PEPM'04*, August 2004.

[36] A. Rigo and S. Pedroni. PyPy's approach to virtual machine construction. In *OOPSLA Dynamic Languages Symposium*, Portland, Oregon, October 2006.

[37] N. Riley and C. Zilles. Hardware transactional memory support for lightweight dynamic language evolution. In *OOPSLA Dynamic Languages Symposium*, Portland, Oregon, October 2006.

[38] N. Riley and C. Zilles. Transactional runtime extensions for dynamic language performance. In *EPHAM 2008: Workshop on Exploiting Parallelism with Transactional Memory and Other Hardware Assisted Methods*, April 2008.

[39] J. Rose et al. JSR 292: Supporting Dynamically Typed Languages on the Java™ Platform. URL `http://jcp.org/en/jsr/detail?id=292`.

[40] A. Shankar, S. S. Sastry, R. Bodík, and J. E. Smith. Runtime specialization with optimistic heap analysis. *SIGPLAN Not.*, 40(10):327–343, 2005.

[41] J. Spolsky, J. Atwood, and S. Yegge. The StackOverflow Podcast, episode 50. URL `https://stackoverflow.fogbugz.com/default.asp?W29043`.

[42] L. Su and M. Lipasti. Speculative optimization using hardware-monitored guarded regions for Java virtual machines. In *Proceedings of the Third International ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, June 2007.

[43] Sun Microsystems. Java Virtual Machine Tool Interface (JVM TI). URL `http://java.sun.com/javase/6/docs/technotes/guides/jvmti/`.

[44] H. Sutter and J. Larus. Software and the Concurrency Revolution. *ACM Queue*, 3(7):54–62, Sept. 2005.

[45] G. van Rossum et al. Python reference manual: Naming and binding. URL `http://docs.python.org/ref/naming.html`.

[46] E. Witchel et al. Mondrian memory protection. In *Proceedings of ASPLOS-X*, Oct 2002.

[47] M. Wolczko. Benchmarking Java with Richards and DeltaBlue. URL `http://research.sun.com/people/mario/java_benchmarking/`.

[48] P. Zhou et al. iWatcher: Efficient architectural support for software debugging. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1994.

[49] C. Zilles. Accordion arrays: Selective compression of Unicode arrays in Java. *International Symposium on Memory Management*, 2007.

Author's Biography

Nicholas Riley was born on March 26, 1979, in Des Moines, Iowa. After graduating from Buckingham Browne and Nichols School in Cambridge, Massachusetts, he attended Brandeis University, from which he earned a Bachelor of Arts degree with Honors in Computer Science and Biology in 1999. The same year, he entered the Medical Scholars (M.D./Ph.D.) Program at the University of Illinois at Urbana-Champaign. In 2009, he completed his Ph.D. in Computer Science under the direction of his advisor, Craig Zilles.