Transactional Runtime Extensions for Dynamic Language Performance

Nicholas Riley
njriley@uiuc.edu

Craig Zilles
zilles@uiuc.edu

Department of Computer Science
University of Illinois at Urbana-Champaign

ABSTRACT

We propose exposing best-effort atomic execution, as provided by a simple hardware transactional memory (HTM), in a managed runtime's bytecode interface. Dynamic language implementations built on such a runtime can generate more efficient, code, using speculation to eliminate the overhead and obstructions to optimization incurred by code needed to preserve rarely used language semantics. In this initial work, we demonstrate the applicability of this framework with two optimizations in Jython, a Python implementation for the Java virtual machine, that would yield speedups between 13% and 38% in the presence of HTM. Three additional optimizations, which we applied by hand to Jython-generated code, provide an additional 60% speedup for one benchmark.

1. INTRODUCTION

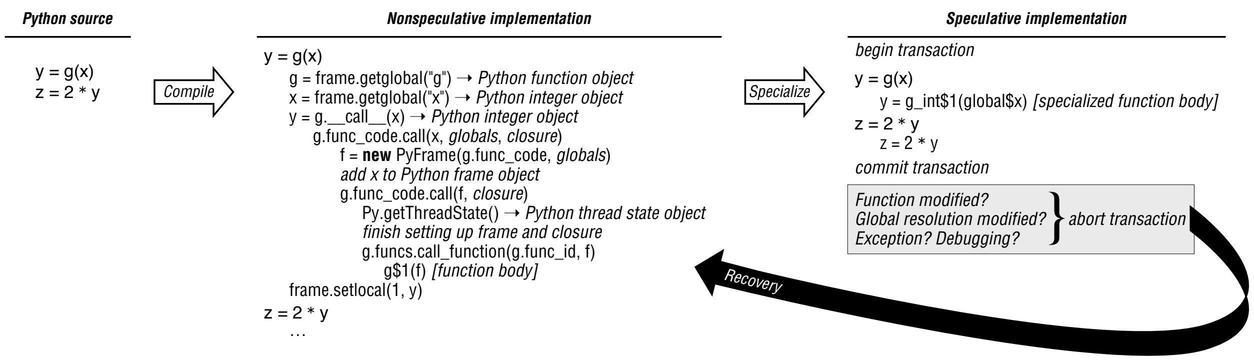
Dynamic languages such as Python and Ruby are increasingly popular choices for general-purpose application development. Programmers value the languages' unique features, flexibility, pragmatic design and continued evolution; they are eager to apply them to a widening range of tasks. However, performance can be limiting: most programs written in these languages are executed by interpreters. Attempts to build more faster, more sophisticated implementations (*e.g.*, [5, 11, 15]) have been frustrated by the need to preserve compatibility with the languages' flexible semantics.

We evaluate the effectiveness of speculatively executing Python programs, where possible, with a common case *subset* of these semantics which correctly handles most Python code. It it difficult to efficiently perform this speculation in software alone, but with hardware support for transactional execution, easier-to-optimize speculative code can execute at full speed. When behavior unsupported by the common case semantics is needed, our implementation initiates hardware-assisted rollback, recovery and nontransactional reexecution with full semantic fidelity. Overall, our approach facilitates improved dynamic language performance with minimal cost in implementation complexity when compared with software-only approaches.

Our framework consists of three components. First, an efficient hardware checkpoint/rollback mechanism provides a set of transactional memory (TM) primitives to a Java virtual machine (JVM). Second, the JVM exposes the TM's capabilities as an interface for explicit speculation. Finally, dynamic language runtimes written in Java employ these extensions at the implementation language level in order to speculatively execute code with common case semantics. With hardware and JVM support for speculation, the dynamic language implementation doesn't need to know a priori when the common case applies: it can simply try it at runtime.

We demonstrate this approach by modifying a JVM dynamic language runtime—Jython [2], a Python implementation—to generate two copies of code (*e.g.*, Figure 1). First, a speculatively specialized version implements a commonly used subset of Python semantics in Java, which by simplifying or eliminating unused data structures and control flow, exposes additional optimization opportunities to the JVM. Second, a nonspeculative version consists of the original implementation, which provides correct execution in all cases. Execution begins with the faster, speculative version. If speculative code encounters an unsupported condition, an explicit transactional abort transfers control to the nonspeculative version. As a result, the JVM does not need dynamic language-specific adaptations, and high-performance dynamic language implementations can be written entirely in languages such as Java and C#, maintaining the properties of safety, security and ease of debugging which distinguish them from traditional interpreters written in C. Even without execution feedback to the dynamic language implementation, transactional speculation leads to significant performance gains in single-threaded code. In three Python benchmarks run on Jython, a few simple speculation techniques improved performance between 13 and 38%.

The remainder of this paper is organized as follows. Section 2 introduces the challenges of dynamic language runtime performance. Section 3.3 describes the components of our framework and the optimizations we implemented in Jython, section 4 presents our experimental method and section 5 discusses the performance results.



```
y = g(x)
z = 2 * y
```

```
y = g(x)
    g = frame.getglobal("g") → Python function object
    x = frame.getglobal("x") → Python integer object
    y = g.__call__(x) → Python integer object
        g.func_code.call(x, globals, closure)
            f = new PyFrame(g.func_code, globals)
            add x to Python frame object
            g.func_code.call(f, closure)
                Py.getThreadState() → Python thread state object
                finish setting up frame and closure
                g.funcs.call_function(g.func_id, f)
                    g$1(f) [function body]
    frame.setlocal(1, y)
z = 2 * y
    ...
```

```
begin transaction
y = g(x)
    y = g_int$1(global$x) [specialized function body]
z = 2 * y
    z = 2 * y
commit transaction
Function modified?
Global resolution modified?  } abort transaction
Exception? Debugging?
```

Figure 1: Potential transactional memory-aided speculative specialization of Python code.

2. BACKGROUND AND RELATED WORK

Python, Ruby, Perl and similar dynamic languages differ substantially from historically interpreted counterparts such as Java and Smalltalk in that large portions of their basic data structures and libraries are written in the runtime's implementation language, typically C, rather than in the dynamic languages themselves. This choice substantially improves the performance and practicality of these languages even as they remain interpreted.

However, to get the best performance out of interpreted dynamic language implementations, programmers must ensure that as much work as possible takes place in native code. As a result, interpreter implementation details can inspire less-than-optimal programming practices motivated by performance. For example, in the CPython interpreter, function calls are relatively expensive, so performance-critical inner loops are typically written to minimize them. A more sophisticated runtime could perform inlining and specialization. Or, to sort a list by an arbitrary key, Perl and Python have adopted an idiom known as "decorate-sort-undecorate" in which each list item is transformed to and from a sublist of the form (sort key, item), so the list can be sorted entirely from compiled code instead of repeatedly invoking an interpreted comparator.

To reduce the need for such workarounds and improve execution performance in general, an obvious next step in dynamic language evolution is uniting the language and its implementation on a single platform which can aggressively optimize both. New dynamic language platforms do this in one of two ways. Some write the language implementation and/or its libraries in a common language, and build a dynamic language-specific runtime to execute it: for example, Rubinius [12] executes Ruby on a Smalltalk-style VM; PyPy [15] implements a translation infrastructure that can transform a Python implementation written in a subset of Python into an interpreter or naïve JIT, which is then compiled for one of several platforms. While these systems are written targeting dynamic language semantics, they must also incorporate dynamic optimization techniques and native code interfaces, which represents a nontrivial duplication of effort.

The other option is building on an existing, mature platform: the Microsoft Common Language Runtime (CLR) or JVM. On these platforms, dynamic language code is translated into CLR/JVM bytecode, which references implementation components written in a platform-native language (C#/Java). Dynamic language implementations must generate code and maintain data structures to express the languages' rich semantics in terms of more limited CLR/JVM functionality, even for such basic operations as function invocation. This semantic adaptation layer acts as a barrier to optimization by the CLR/JVM, hindering performance.

One example of Python's flexible semantics is method dispatch, which in Jython involves call frame object instantiation and one or more hash table lookups (**new** PyFrame(...) and frame.getglobal calls, respectively, as shown in Figure 1). Dispatch must also simulate Python's multiple inheritance of classes on Java's single inheritance model [18]. Each of these operations is significantly more costly than dynamic dispatch in Java.

The CLR/JVM platforms could be extended with dynamic language-specific primitives to reduce the cost of adaptation. But selecting the right set of primitives is challenging, as each dynamic language has its own unique semantics. We have observed that the majority of a typical dynamic language program doesn't exploit its language's full semantic flexibility. In fact, existing CLR/JVM platform primitives can directly encode common uses of most dynamic language features, significantly improving performance. However, without a way to identify which operations can be so encoded without observing them during execution—which would require hooks into the CLR/JVM internals—dynamic language implementations cannot unconditionally generate fast code for the common case.

While adapting the CLR/JVM to accommodate individual dynamic languages' semantics would be inadvisable, some general-purpose VM additions can simplify dynamic language implementation. For example, some CLR-based dynamic languages are hosted on a layer called the Dynamic Language Runtime (DLR) [4]. The DLR implements a common dynamic language type system overlayed upon, but not interoperable with, the underlying CLR type system, and code generation machinery facilitating a primitive form of runtime specialization. A DynamicSite facility [8] builds runtime code replacement on two CLR features: delegates (akin to function pointers) and Lightweight Code Generation (low-overhead runtime code generation).

Such micro-scale specialization effectively reduces dynamic dispatch overhead but does not substantially eliminate the need for a dynamic language-specific feedback-directed optimization infrastructure. In contrast, our framework for optimistic specialization *exposes optimization opportunity* to the VM, without requiring that the dynamic language implementation react to execution feedback. The dynamic language-targeted facilities of the CLR and corresponding forthcoming JVM features (*e.g.*, invokedynamic, anonymous classes and method hotswapping [16, 17]) are in fact largely complementary to our work, allowing it to be extended to permit runtime adaptation to misspeculation.

3. A FRAMEWORK FOR COMMON CASE SPECULATION

We now present our framework for common case speculative optimization and two specific Jython optimizations we implemented on this framework.

3.1 Transactional memory support

Our framework employs a relatively simple, best-effort HTM implementation with support for explicit aborts and redirecting control flow upon an abort. We assume a collapsed transaction nesting model, where beginning a transaction essentially increments a counter, ending a transaction decrements it, and transaction state commits when the counter reaches zero.

The TM system provides software with a guarantee of atomicity and isolation. An atomic region either commits successfully or rolls back, undoing any changes made since the region began. The HTM makes its best effort to execute an atomic region given constraints such as fixed buffer sizes. If outstanding transactional data exceeds the capacity of the hardware to buffer it, or in other unsupported circumstances (*e.g.*, I/O and system calls), the atomic region is aborted [10]. The specific HTM primitives we use are:

Begin transaction (abort PC). Take a checkpoint and begin associating register and memory accesses with the transaction. The abort PC argument is saved to be used when redirecting control flow in the event the region aborts.

Commit transaction. End the region and atomically commit changes.

Abort transaction. Discard changes by reverting to the saved checkpoint, then jump to the abort PC.

3.2 JVM speculation interface

Our proposed JVM support for explicit speculation constructs resembles existing structured exception mechanisms: the speculative code is analagous to the *try* block, assumption violation to *throwing* an exception, and nonspeculative code to an exception *catch*. As a result, we expose an interface for explicit speculation in Java source and JVM bytecode with a simple extension to the Java exception model, as shown in Figure 2. The changes do not affect the Java bytecode interface or compiler, which simplifies the implementation and maintains compatibility with existing tools. Instead, changes were confined to the Java class library (to add a new exception class, methods for introspecting abort state and region invalidation) and the JVM itself (to output instructions recognized by the HTM and implement atomic region invalidation).

A speculative atomic region (1) and its fallback code (3), which encompasses both the recovery code and original, unspecialized code, are wrapped in a standard JVM exception block and a new exception class. Throwing an exception of this class triggers the JVM to emit a HTM abort instruction (2). In fact, exceptions of any class raised within an atomic region will cause the region to abort. Transforming any thrown exception into an abort is advantageous not only because it reduces code size by allowing exception handlers to be eliminated from speculative regions, but because existing JVM exception behavior (*e.g.*, converting a processor exception on a null pointer dereference into a Java exception) can be a more efficient substitute for explicitly checked assertions, eliminating the corresponding checks from the speculative path.

A critical advantage provided by speculative atomic regions is in expanding the scope with which the JVM can perform optimizations. Not only can the specialized code employ simpler data structures, repetitive control flow checking for a particular condition can be replaced by a single guard. Just as with JVM-delimited regions [10], the JVM can eliminate such redundant checks in regions of explicit speculation.

Speculative regions may be generated under the assumption that certain global state remains unchanged (*e.g.*, that a function is not redefined). The speculation client code must keep track of these regions with their corresponding assumptions, and emit code which invalidates these regions when necessary.[1] It would of course be possible for the code to check inside each speculative region whether any of the global state upon which it depends has changed, but it is more efficient to use a code bottleneck (a) which can invalidate, or unspecialize, a region while it is not currently executing. To perform unspecialization, the beginning of the **try** block is replaced with an unconditional branch to nonspeculative code, potentially pending generation of replacement speculative code that handles the new situation. In a **catch** block which is executing because the region has been unspecialized, the exception's value is null and the recovery code does not execute.

3.3 Dynamic language specialization

With our framework, speculative regions are delineated in advance of execution—the language implementation doesn't need to collect and process runtime feedback to discover where speculation will be successful. The dynamic language compiler also need not include explicit *compensation code* which can recover from an exceptional condition encountered at any point in execution of a speculative region. When the atomic region aborts, it reverts any changes it made and redirects control flow to a recovery code block, which reacts to the reason for the abort before resuming with the cor-

[1]Our implementation exposes references to atomic regions as an index into a method's exception table, though we plan to implement a lighter-weight, opaque identifier in the future which is better suited for lightweight runtime code generation and the JVM security model.

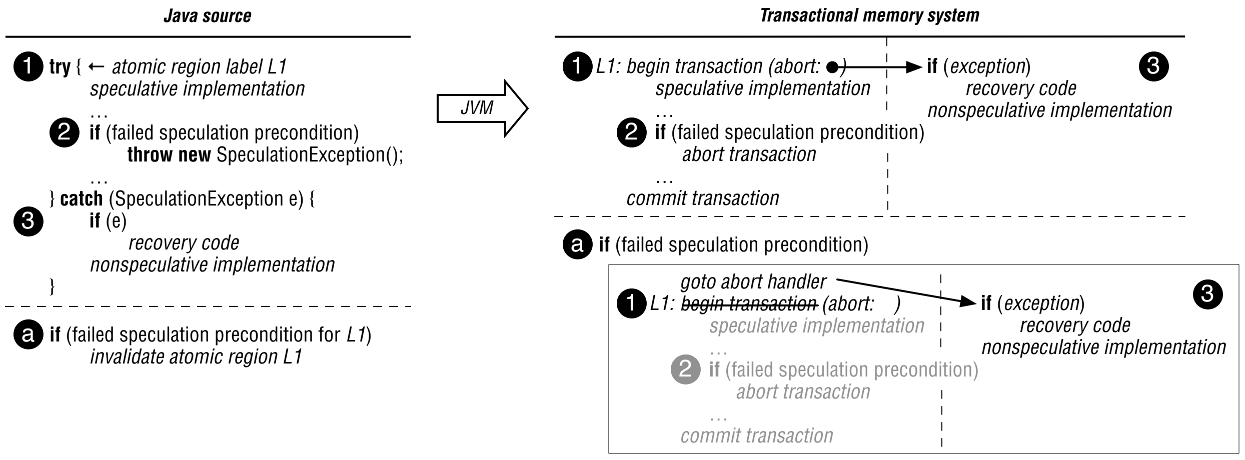

Java source
1 try { ← atomic region label L1
speculative implementation
…
2 if (failed speculation precondition)
throw new SpeculationException();
…
} catch (SpeculationException e) {
3 if (e)
recovery code
nonspeculative implementation
}
a if (failed speculation precondition for L1)
invalidate atomic region L1
JVM
Transactional memory system
1 L1: begin transaction (abort: ●)
speculative implementation
…
2 if (failed speculation precondition)
abort transaction
…
commit transaction
if (exception) 3
recovery code
nonspeculative implementation
a if (failed speculation precondition)
goto abort handler
1 L1: ~~begin transaction~~ (abort:)
speculative implementation
…
2 if (failed speculation precondition)
abort transaction
…
commit transaction
if (exception) 3
recovery code
nonspeculative implementation

Figure 2: Java language support for HTM-supported speculative specialization.

responding nonspeculative code as if speculation had never been attempted. If an atomic region aborts because of a condition that should result in a dynamic language exception, the nonspeculative code executed afterward accumulates the necessary exception state.

Our Jython modifications consist of three parts: a general-purpose mechanism which forks code generation into nonspeculative and speculative versions, a mechanism for keeping track of assumptions of global state (as discussed in the previous section) and the individual optimizations themselves. The optimizations' general form is shown in Figure 3. Common case semantics are statically determined, speculative regions are created by extracting the corresponding implementation, assertions are inserted to ensure the common case scenario is indeed in effect and recovery code is constructed by testing for and reacting to the violation of each assertion in the aborted atomic region, then propagating the violation to perform unspecialization of other affected regions if needed.

One of our optimizations involves a simple modification to the Jython library code; the other includes both compiler and library changes.

3.3.1 *Dictionary (*HashMap*) synchronization*

The Python dictionary data structure provides an unordered mapping between keys and values. Dictionaries are frequently accessed in user code, the default storage 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 [6] and Jython's dictionary implementations include specializations 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

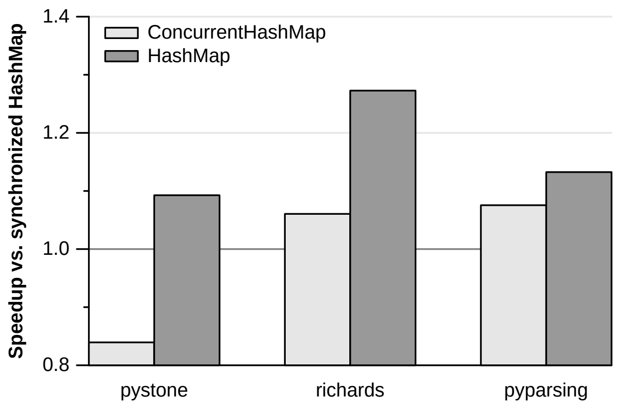


Figure 4: Relaxing correctness and synchronization constraints on Python dictionaries. Results obtained with HotSpot; consult section 4 for configuration details.

Java strings to be directly used as keys, rather than being wrapped in an adapter implementing Python string semantics.

Python dictionary accesses must be performed atomically. Jython enforces this constraint by using a synchronized HashMap (hash table) data structure for both general-purpose PyDictionary and special-purpose PyStringMap dictionaries. A single lock protects against simultaneous access at a cost of 8–27% in overall runtime (Figure 4).

Java 1.5 introduced a ConcurrentHashMap implementation [7] which optimizes lock usage. While replacing the synchronized HashMap with a ConcurrentHashMap improves 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. 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

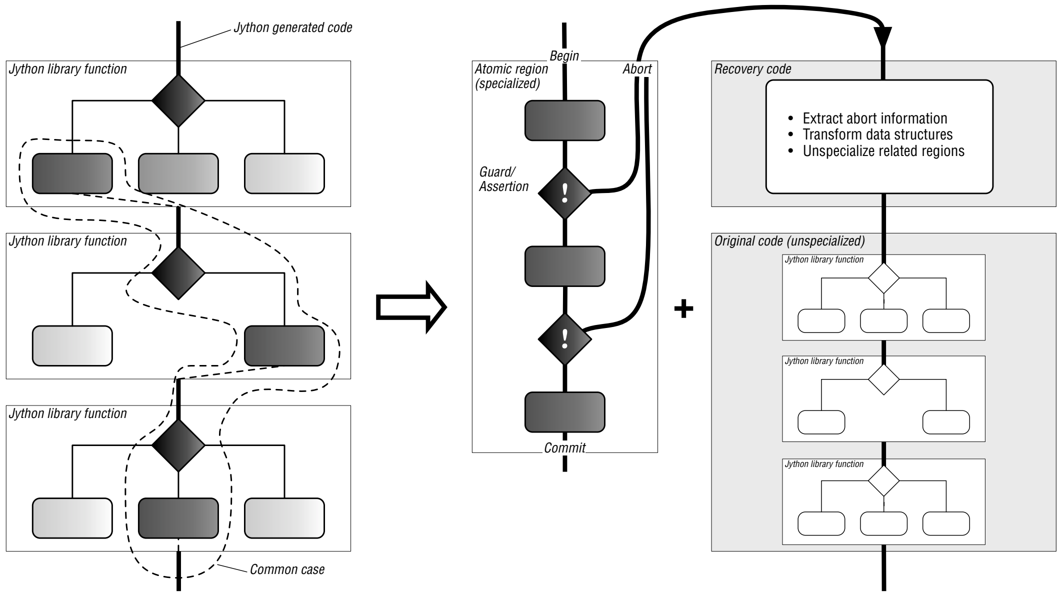

Jython generated code
Jython library function
Common case
Begin
Atomic region (specialized)
Abort
Guard/ Assertion
Commit
Recovery code
• Extract abort information
• Transform data structures
• Unspecialize related regions
Original code (unspecialized)
+

Figure 3: Transactional memory-aided speculative specialization of Jython code.

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) [13], uncontended access to a synchronized HashMap can perform as well as the unsynchronized version. Synchronization overhead can be eliminated entirely when accesses are already subsumed by a larger atomic region, as will usually be the case.[2]

3.3.2 Caching globals

We 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 [19].[3]

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

```
def f():
    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:

```
private static PyObject f$1(PyFrame frame) {
  frame.getglobal("g").__call__(frame.getglobal("x"));
}
```

While the externally visible Python namespace representation must remain a dictionary for compatibility, its internal representation can be speculatively optimized for performance by exploting 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, we modified the Jython compiler to keep track of global and builtin references as they are compiled, emit corresponding static variable declarations, and cache the global and builtin values in these static variables during module loading. With this optimization, the code becomes:

```
private static PyObject global$g, global$x;
private static PyObject f$1(PyFrame frame) {
  try {
    global$g.__call__(global$x);
  } catch (SpeculationException e) {
    frame.getglobal("g").__call__(frame.getglobal("x"));
  }
}
```

We 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.

Similarly, attempts to delete x must be trapped. In this case,

[2]The results in Figure 4 were obtained on a single-core machine; the speedup obtained using SLE is even greater on a multicore machine, where the JVM's uniprocessor optimizations do not apply.

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

we do not simply redirect the deletion but invalidate the atomic region in f$1, because there's no way to represent the semantics of Python deletion with a Java static field. If, for example, we used the Java null value to represent a deleted variable, by dereferencing a null global$g would generate a NullPointerException, but passing null to g would convert it into a Python None object instead of producing the expected Python NameError as required by Python semantics when an undefined variable is accessed.

3.3.3 Frame specialization

Python allows extensive introspection of executing program state, primarily for the purposes of debugging and exception handling [20]. While these facilities differ little from those of languages, including Java itself, for which dynamic compilation replaced interpretation, Jython cannot rely on the JVM to reconstruct Python execution state when it is requested, so it must maintain the state itself during execution (VM extensions may offer alternate approaches, *e.g.* [3]). This entails a significant amount of wasted work in the common case, when this state is never accessed.

Jython maintains the Python execution stack as a linked list of frame (PyFrame) objects, one of which is passed to the Java method generated from each Python function or method. Frame objects expose local variables including parameters, global variables, exception information and a trace hook for debuggers [21]; a simplified version of the PyFrame operations performed on a Python function call appears in Figure 1. An additional attribute represents the current Python bytecode index, though this is infrequently accessed and Jython does not maintain it.

While we have not yet implemented any automatic frame specializations in the Jython compiler, some of our early results from frame elimination are presented in section 5.1.

4. EXPERIMENTAL METHOD

Our goal in these experiments was to quickly explore the performance potential of dynamic language runtime optimizations enabled by transactional execution. We executed three Python benchmarks, pystone, richards and pyparsing, on a real machine while simulating the effects of a HTM.

The first two of these benchmarks are commonly used to compare Python implementations: pystone [1] consists of integer array computation written in a procedural style; richards [22] is an object-oriented simulation of an operating system. Both are straightforwardly written but non-idiomatic Python, ported from Ada and Java, respectively; we expected them to uniformly exhibit common case behavior. In contrast, pyparsing [9] uses Python-specific features to implement a domain-specific language for recursive descent parsers; we chose it as an example of code which is both potentially performance-critical and exploits the language's semantics.

Speedups gained from common case speculation rely on invalidation being an infrequent occurrence. While none of the the benchmarks' code violated any specialization preconditions, pyparsing uses exceptions as a form of control flow; Python exceptions inside specialized code cause the containing atomic region to abort. Since pystone and richards did not raise exceptions, no conflict-related transactional aborts would occur. Aborts would instead stem from context switches, I/O and system calls. Of the benchmarks, only pyparsing performs any I/O from Python code, and the functions containing I/O already executed unspecialized because they indirectly raise exceptions.

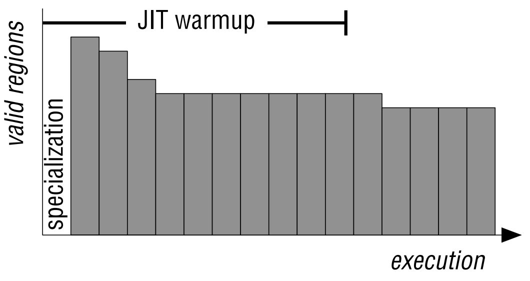


Figure 5: Typical atomic region invalidation profile.

Because the atomic regions in our benchmarks thus divide cleanly into "always abort" and "never abort" groups, we 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. We accomplished this by modifying the compiler's output as if the "always abort" regions had been permanently invalidated, so that the original code always executed.

As we had no capability to roll back execution in these real-machine experiments, we configured the JVM exception handlers associated with atomic regions to abort the process if triggered. With an actual HTM, the exception handlers would execute recovery code instead; subsequent executions would take the nontransactional fallback path. Recovery and nonspeculative code was still present in the running system, such that any performance loss attributable to code size growth should be accounted for by these results.

For the optimizations presented thus far, specialization is performed once, in the Jython compiler. As code which violates specialization assumptions is encountered during execution, the corresponding regions are permanently invalidated. Many performance-critical dynamic language applications are long-running servers, whose invalidation profiles should resemble Figure 5. Most invalidations would occur early in execution as code is first encountered and JIT is still actively compiling.

Just as "server" JITs' aggressive compilation strategies slow startup, early invalidation-related slowdowns are likely to be an acceptable tradeoff for higher sustained performance. In practice, HTM rollback latency would be unlikely to matter: the hardware rollback time would be overshadowed by recovery and fallback code execution. It would be possible to construct pathological cases in which region invalidation occurred frequently throughout execution, thus making the recovery path performance-critical, however for ordinary programs (and our benchmarks) this is extremely unlikely.

We selected two JVMs which execute Jython the fastest:

BEA JRockit (build R27.3.1-1-85830-1.6.0_01-20070716-1248-linux-ia32) and Sun HotSpot Server (build 1.6.0_02-b05). The JVM extensions we propose are not specific to any particular implementation; with the exception of honoring invalidation requests in recovery code, the JVM is permitted to execute any combination of specialized and unspecialized code for optimal performance.

Experiments were run on an Intel Core 2 Duo E6700 with Linux kernel 2.6.9-55.0.12.EL, using a single core in 32-bit mode. We used Jython's "newcompiler" and modern branch (r4025). To account for JVM warmup and execution variation, Jython speedups are based on the minimum runtime of ten benchmark executions per session over five JVM execution sessions.

For comparison, we include results from the CPython 2.4.3 interpreter and Psyco 1.5.1. Psyco, a just-in-time specializer for CPython [14], executes these benchmarks faster than any other Python implementation, at the expense of imperfect language compatibility and potentially unbounded memory use. As Psyco uses a simple template-based code generator, the Psyco results represent a lower bound on the performance of a Python-specific just-in-time compiler.

One important and unexplored question regarding these optimizations is whether atomic region boundaries, representing explicit speculation's scope, can be established during code generation—for example, at all function boundaries and loop back edges—or must be adjusted at runtime for optimal performance. In order to answer this question, we plan to measure transaction footprint and abort behavior with a full-system HTM simulation. Our previous work with dynamic language runtimes in simulated HTM environments, including a transactional Psyco executing the same benchmarks we present here, has not identified transactional cache overflows as a common issue. While it is possible that Java's memory access patterns may differ enough to affect the transaction footprint, existing work using hardware atomic regions for specialization in Java suggests that careful tuning of region sizes is not essential [10].

5. RESULTS

We express Jython performance with and without speculative optimizations as speedups relative to interpreted CPython execution, as this is the baseline which would make the most sense to Python users. Performance differs substantially between the tested JVMs, whose just-in-time compilers were independently implemented; HotSpot always outperforms JRockit except in pyparsing.

Figure 6 includes the automated speculative optimizations we implemented in Jython, as described in section 3.3: speculative lock elision for dictionary access (HashMap) and elimination of hash table lookups in global and builtin access (cache_globals). Performance improvement varies according to the nature of each benchmark, which we now discuss in turn.

pystone contains primarily computation and function dispatch. Jython's baseline pystone performance already exceeds CPython's because numerical operations are encoded efficiently as their Java equivalents. Python function calls' flexible semantics are costly to implement in Jython, as they are in CPython, but because Python functions are declared as module-level globals, caching these values eliminates hash table lookups from each function invocation.

richards benefits comparatively less from global caching because most of its dispatch overhead is in the form of instance method invocations, which involve dictionary access. In addition, richards uses the dictionary storage mechanism for the instance variables of its primary data type. Elimination of synchronization overhead from these frequent dictionary accesses is responsible for the majority of the performance improvement. With a few more simple optimizations Jython can even outperform Psyco on the richards benchmark, as discussed in the following section.

The pyparsing results demonstrate the potentially large baseline performance gap between Jython and interpreted execution. While pyparsing includes both object-oriented dispatch and frequent dictionary access, execution is dominated by complex control flow. In particular, pyparsing uses Python exceptions to implement backtracking. Python exceptions in Jython incur a double penalty: in stock Jython, the complex Python exception state is re-created at the beginning of a Jython exception handler, regardless of whether or not it is used. pyparsing's exception handlers do not examine the exception state, thus this behavior causes needless object allocation. The majority of the methods in pyparsing do indeed either raise an exception or invoke a method which does, so speculative versions are rarely executed and little benefit is achieved.

5.1 More richards optimizations

To explore the performance potential of Jython with more aggressive speculative optimization, we manually applied several additional transformations to the richards benchmark, in addition to the HashMap and cache_globals optimizations discussed in the previous section. These changes target the primary bottleneck in richards: accesses to objects of class TaskState (representing processes in an OS scheduler simulation) and its subclasses. Figure 7 graphs the performance improvement.

Jython does not generate Java classes for user-defined Python classes such as TaskState. Instead, instance methods of these classes are defined as static methods of a class corresponding to the module in which the class is defined. For example, TaskState.isTaskHolding(), which returns its task_holding attribute, compiles to:

```
public class richards {
  private static PyObject isTaskHolding$22(PyFrame frame) {
    return frame.getlocal(0).__getattr__("task_holding");
  }
  ...
```

task-class. Optimize access to TaskState attributes by storing them in Java instance variables rather than in a PyStringMap. Attempts to access the dictionary directly are redirected to the instance variables, as with the cache_globals optimization (Section 3.3.2); recovery is only necessary when the default attribute access behavior is overridden. If the cast to TaskState fails, a Java exception is raised which triggers a

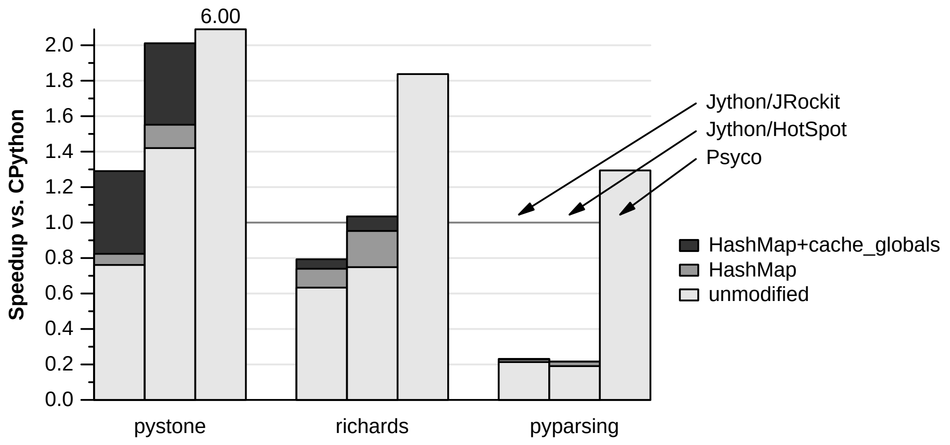


Figure 6: Jython and Psyco speedups over CPython on three Python benchmarks.

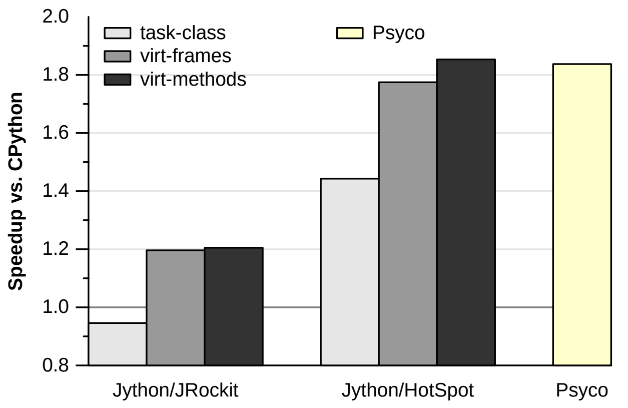


Figure 7: Hand-optimized Jython and Psyco speedups over CPython on the richards benchmark.

rollback; the original code will then generate the Pythonically correct TypeError.

```
public class TaskState extends PyObjectDerived {
  PyObject task_holding;
  /* rest of class */
}
private static PyObject isTaskHolding$22(PyFrame frame) {
  try {
    return ((TaskState)frame.getlocal(0)).task_holding;
  } catch (SpeculationException e) {
    /* recovery/nonspeculative code */
  }
}
```

virt-frames. For TaskState methods implemented as static methods, eliminate the intermediate PyFrame argument used to encapsulate the method's parameters and local variables. An atomic region surrounds the call site rather than the callee. Recovery is needed if a method is redefined; any atomic region containing code which directly references the static method must be invalidated. The PyFrame version of the method remains for compatibility with separately compiled code.

```
private static PyObject isTaskHolding$22(TaskState t) {
  return t.task_holding;
}
private static PyObject isTaskHolding$22(PyFrame frame) {
  /* nonspeculative code */
}
```

virt-methods. Move the TaskState method implementations into the TaskState class. While one might expect invoking a static method with a class's instance as its first argument would be as fast as an instance method on that class, the latter was faster with both tested JVMs.

```
public class TaskState extends PyObjectDerived {
  /* rest of class */
  public PyObject isTaskHolding() {
      return task_holding;
  }
}
```

6. CONCLUSION

In this paper, we have demonstrated that hardware transactional memory systems can facilitate the development of high-performance dynamic language implementations by exposing a simple interface for expressing common case speculation. A low-overhead hardware checkpoint/rollback mechanism can offer a largely transparent form of runtime adaptation, while the simplicity of common case code is an effective target for existing dynamic optimization systems.

7. ACKNOWLEDGMENTS

The authors wish to thank Paul T. McGuire for providing the Pyparsing Verilog parser benchmark and Jim Baker, Philip Jenvey and Tobias Ivarsson for assistance with Jython. Pierre Salverda and the anonymous reviewers offered valuable suggestions.

References

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

[2] J. Baker et al. The Jython Project. URL http://jython.org/.

[3] J. Clements. *Portable and high-level access to the stack with Continuation Marks.* PhD thesis, Northeastern University, 2006.

[4] J. Hugunin. A Dynamic Language Runtime (DLR). URL http://blogs.msdn.com/hugunin/archive/2007/04/30/a-dynamic-language-runtime-dlr.aspx.

[5] J. Hugunin et al. IronPython: a fast Python implementation for .NET and ECMA CLI. URL http://www.codeplex.com/IronPython.

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

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

[8] M. Maly. Building a DLR Language - Dynamic Behaviors 2. URL http://blogs.msdn.com/mmaly/archive/2008/01/19/building-a-dlr-language-dynamic-behaviors-2.aspx.

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

[10] N. Neelakantam, R. Rajwar, S. Srinivas, U. Srinivasan, and C. Zilles. Hardware atomicity for reliable software speculation. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, June 2007.

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

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

[13] 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.

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

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

[16] J. Rose. Anonymous classes in the VM. URL http://blogs.sun.com/jrose/entry/anonymous_classes_in_the_vm.

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

[18] G. van Rossum. Unifying types and classes in Python 2.2, April 2002. URL http://www.python.org/download/releases/2.2.3/descrintro/.

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

[20] G. van Rossum et al. Python library reference: The interpreter stack. URL http://docs.python.org/lib/inspect-stack.html.

[21] G. van Rossum et al. Python reference manual: The standard type hierarchy. URL http://docs.python.org/ref/types.html.

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