Efficient VM-Independent Runtime Checks for Parallel Programming

Michael Faes  
Department of Computer Science  
ETH Zurich  
Switzerland  
michael.faes@inf.ethz.ch

Thomas R. Gross  
Department of Computer Science  
ETH Zurich  
Switzerland  
trg@inf.ethz.ch

Abstract
Many concurrent or parallel programming languages rely on runtime checking to ensure safety. To implement such a language on a virtual machine (vm), such runtime checks are often implemented in a vm-independent way, using source-to-source translation or bytecode instrumentation. This approach avoids modifying complex vm components like the just-in-time (jit) compiler and offers great portability. However, obtaining good performance is challenging, as the approach cannot profit from custom jit optimizations to eliminate redundant checks.

In this paper, we present and evaluate two techniques to make the vm-independent approach efficient, using the example of a parallel programming language called Rolez. To guarantee that concurrent threads do not interfere, Rolez relies heavily on runtime checks: for every field access, the runtime system checks that the state of the target object currently permits this operation (unless the check is optimized away). The Rolez compiler we present here generates standard Java source code and the runtime system is implemented as a Java library. Nevertheless, many Rolez programs deliver performance roughly on par with manually synchronized Java implementations, which is achieved using these two techniques: 1) code-managed runtime data, which improves runtime check efficiency by passing performance-critical information from method to method, and 2) an interprocedural but modular concurrency analysis, which eliminates many runtime checks that are actually redundant.

1 Introduction
Many programming languages provide little or no safety for concurrent or parallel programming (pp). For example, in Java, it is entirely the responsibility of the programmer to ensure that objects that are shared among multiple threads are only accessed with proper synchronization, e.g., using the synchronized construct. Insufficient or incorrect synchronization can lead to bugs like data races or deadlock. Other languages aim to prevent such concurrency bugs by providing higher-level constructs, e.g., atomic blocks, or even by guaranteeing properties like data race freedom for the entire program. Such guarantees are attractive for parallel programmers, but they are difficult to implement efficiently, as they usually make heavy use of runtime checking. For example, atomic blocks are often implemented using transactional memory [6, 15], which requires the runtime system to buffer all modifications to objects and, in case of an interfering access from a concurrent thread, to roll them back. As another example, the recently presented Rolez language [5] prevents interference more eagerly and without speculation, but relies heavily on runtime checks as well. In this language, every object plays a role in every concurrent task and these roles determine which object operations are legal in which tasks. Since the roles of an object can change dynamically, a runtime check is required for every operation on an object (e.g., for every field access), unless optimized away.

Virtual machines are naturally suited to host such languages, but implementing pp runtime checks efficiently is still challenging. On the one hand, vms usually provide memory safety, which can simplify the implementation of pp runtime checks substantially. In addition, vm implementations
typically include a JIT compiler, which compiles programs at runtime. This dynamic compilation approach is a good fit for languages with heavy runtime checking, as it can exploit precise runtime information to eliminate redundant checks. On the other hand, implementing the checks directly in the VM has several drawbacks: First, VM components like the JIT compiler or the garbage collector are complex and often difficult to extend and maintain. Second, if PP runtime checks are built directly into the VM, they are tied to specific VM or even the VM implementation and cannot be easily ported to other platforms. These drawbacks hinder the development of new PP languages or constructs, especially for small teams of language designers or for researchers.

Thus, many PP languages or constructs are implemented in a VM-independent way, usually using source-to-source translation or bytecode instrumentation. With a VM-independent implementation, we mean one that does not rely on any specific VM feature or even modifies a specific VM implementation. A VM-independent implementation avoids the drawbacks mentioned above, but since it cannot rely on custom JIT optimizations, achieving efficiency (which is paramount for parallel programming) is much more difficult.

In this paper, we present two techniques for implementing PP runtime checks both VM-independently and efficiently. These techniques were developed for an implementation of Rolez, but they are applicable also to other concurrent or parallel languages. The Rolez compiler presented here transforms Rolez source code into standard Java source code. The generated code contains calls to the Rolez runtime, which maintains and checks the roles of objects, and is implemented as a Java library. This approach has a number of advantages:

First, it allows compatibility with other languages that run on the Java Virtual Machine (JVM). Since Rolez classes are transformed into Java classes, it is possible to write the parallel parts of a program in Rolez, and implement the rest in, e.g., Java or Scala. Second, the approach enables the use of existing software tools; for example, Rolez code can be tested using frameworks like JUnit or TestNG and analyzed using Java debuggers or code coverage tools. Finally, the approach results in great portability. Because the implementation does not require any modifications to a JVM component, such as the JIT compiler, Rolez programs that are compiled to Java can be executed on any standard JVM.

By using the following techniques, the implementation provides efficient runtime checking, even though it cannot profit from custom JIT optimizations to eliminate checks:

1. Code-managed Runtime Data. Because Rolez runtime checks depend on the currently executing task, every check needs to look up the “current” task to determine if an operation is permitted. To avoid this lookup, we co-designed the runtime system together with the code generator, such that this performance-critical information is managed by the generated code instead of the runtime library. By generating code that passes the current task from method to method and by relying on off-the-shelf JIT optimizations of modern JVMs, the checks become very efficient (Section 4).

2. Concurrency Analysis. In Rolez, every object operation (e.g., every field access) conceptually requires a runtime check; but these checks are only really required if there are any concurrent tasks that can potentially cause interference. We designed a static concurrency analysis that is used to remove runtime checks from all points in a program where this is not the case. The analysis is both interprocedural and modular, which is again achieved using a co-design with the code generator. The compiler generates two versions of each method, one with checks and one without. Then, when compiling a method call, it generates a call to the version that matches the information computed by an intraprocedural analysis, effectively propagating the information from method to method (Section 5).

These techniques deliver high performance for many Rolez programs, as we show in a detailed performance evaluation (Section 6). We implemented 6 programs with a range of parallel patterns, both in Rolez and in Java. Even though Rolez provides much stronger safety guarantees than Java, almost all Rolez programs achieve substantial parallel speedups, most of them being on par with the Java implementations.

2 Background

First, we give an overview of the Rolez language [5], focusing on the runtime checks that it requires.

Rolez is based on a PP model called Parallel Roles, which identifies three main entities: 1) objects; 2) tasks, which are similar to threads, but need to explicitly declare their parameters; and 3) roles, which represent the relationship between objects and tasks. Most object-oriented languages model objects as a collection of fields plus a collection of methods. Here, every object has a third component: its roles. Every object plays a role for every task in the program, as illustrated in Figure 1.

The roles of an object determine which operations are legal in which tasks and what happens in case of an illegal operation. In Rolez, there are three simple roles: READWRITE, READEONLY, and PURE. The READWRITE role permits both read and write operations, while READEONLY permits only read operations. PURE permits neither (except reading final fields, which is always permitted). The roles an object plays may change whenever a task that the object is shared with starts or finishes. There are role transition rules that define how the

<table>
<thead>
<tr>
<th>accNo</th>
<th>Account</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;23-1298-76&quot;</td>
<td>25000.00</td>
</tr>
</tbody>
</table>

getAccountNo() <code>
getBalance() <code>
withdraw() <code>

<table>
<thead>
<tr>
<th>roles</th>
<th>RW</th>
<th>PU</th>
<th>…</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>t_{max}</td>
<td>t_1</td>
<td>…</td>
</tr>
</tbody>
</table>

Figure 1. Object components.
There are two possible sharing patterns: either an object plays the declared role in a task, and the pure role in all others, or it plays the readonly or the pure role in multiple tasks. In both cases, tasks cannot interfere; and because tasks with conflicting accesses execute in a deterministic order, Rolez programs are deterministic by default [2].

To control what role an object plays in a certain task, the programmer provides a role declaration for that object, which can be thought of as an annotation for the task parameter that corresponds to that object. Once a task is started, all the objects that are shared with that task perform a role transition, such that their roles in the new task match the ones declared by the programmer. However, not only tasks, but also methods, fields, and local variables have role declarations. These are part of a role-based type system that statically ensures that no operation is performed that is illegal with respect to an object’s declared role in a task.

Because of this type system, no runtime checks are required to check that the declared role of an object is respected. Instead, the runtime checks in Rolez are required for a different reason. A role transition may also change the role an object plays in the parent task (the task that starts the new task). For example, this is the case if the new task declares the readonly role for an object. In that case, the object becomes readonly in the new task, but also in the parent task, even though it may be declared as readwrite in the parent task. Hence, while an object is guaranteed to play the declared role at the beginning of a task, a role declaration does not mean that the object plays this role for the whole duration of the task; it only means that the declared role is the most permissive role that this object may play in the task. And because objects may play a less permissive role than the declared one, some operations may become temporarily illegal, despite being legal under the declared role for a task.

This is the reason why runtime check are required in Rolez: to prevent errors that would result from temporarily illegal operations. Such a runtime check is called a guard and checks if an operation is permitted by the role the target object currently plays in the task that attempts to perform the operation. Otherwise, the guard delays the execution of that operation until the object plays the declared role again. This way, two tasks, a parent and its child task, can execute in parallel until the parent attempts to perform an interfering operation, at which point the parent is blocked until the child finishes (and another set of role transitions takes place).

**Example** Role transitions and guarding are illustrated in Figure 2. The top part shows two Rolez tasks. The first, which we assume is being executed in a \( t_{\text{parent}} \) task, has an Account parameter that is declared as readwrite. Thus, the object that is passed to that task initially plays the readwrite role in \( t_{\text{parent}} \), as shown on the bottom, at Number 1. Executing the deposit method on Line 2 is therefore legal, both with respect to the declared and to the current role of the object. Then, \( t_{\text{parent}} \) starts an instance of the computeInterest task, which we call \( t_{\text{child}} \), and shares the Account object with it (Line 3). The computeInterest task also has an Account parameter, but declared as readonly (Line 8). When starting this task, a role transition takes place, after which the shared Account object plays the readonly role in both tasks (Number 2). While \( t_{\text{child}} \) executes the code in computeInterest, \( t_{\text{parent}} \) continues to execute the code after the start statement (Lines 4–). To execute printBalance, \( t_{\text{parent}} \) only needs to read from the object, which is permitted by the readonly role that the object now plays in \( t_{\text{parent}} \). Therefore, printBalance is executed in parallel with computeInterest. On the other hand, the deposit method writes to the object and thus requires the object to play the readwrite role. This is permitted by the declared readwrite role of the object, but, depending on the progress of the child task, not by the object’s current role. Thus, a guard in the deposit method checks if the object already plays the readwrite role, blocking the execution if this is not the case. Once \( t_{\text{child}} \) finishes, the object performs another role transition and becomes again readwrite for \( t_{\text{parent}} \), at which point the deposit method may continue to execute (Number 3).

Note that Figure 2 shows only one guard, the single one that is required to execute this program correctly. However, also the operations in the first invocation of deposit and in printBalance must be guarded, unless the compiler determines that they are redundant. This issue is addressed by the concurrency analysis (Section 5).
3 Rolez Implementation Overview

Before we go into the details of the techniques, we provide an overview of the Rolez compiler and runtime system. The compiler is implemented using the Xtext framework [4] and takes as in input a set of Rolez source files which it transforms into a set of Java source files. The runtime system is implemented as a Java library.

Classes The primary code construct in Rolez is the class, which can contain fields and methods (and tasks, discussed below.) For every Rolez class, the compiler generates a corresponding Java class, with the same fields and methods. However, because Rolez objects also contain roles, the generated Java classes all extend a particular class from the Rolez runtime library. This class is called rolez.lang.Guarded and manages all the state and functionality that is required to keep track of the roles of an object.

As Rolez is similar to Java, the body of a Rolez method can be transformed into Java almost one-to-one, with one exception: guarding. Ignoring for now the techniques presented later, the Rolez compiler inserts a guard for every read or write access to an object. Because all generated classes extend Guarded, all objects are instances of that class and can be used as targets for the methods of that class. Thus, guards are implemented simply as calls to Guarded methods, with the same target as that of the guarded operation.

Figure 3 illustrates these points using a simple Account class. The original Rolez class is shown on the left side and the generated Java class on the right. Note that the generated class is simplified, leaving away various aspects that are discussed in later sections. In both methods, one guard is inserted. Two kinds of guards exist: guardReadWrite() for operations that require the READWRITE role and guardReadOnly() for operations that require the READONLY role. (Pure never requires guarding, as it is the least permissive role possible.)

![Figure 3. Rolez and generated Java class.](image)

1 The Rolez code in this and all following examples is also simplified, as we do not include the annotations that are required by the role-based type system. Since guarding is independent of the type system, as explained in Section 2, these annotations would only clutter the examples.
Figure 4 illustrates these points. The top part shows the declaration of a task and a method that starts it, while the bottom part shows the generated Java code. Note that the Task constructor that is invoked on Line 2 takes two arrays as arguments. The first contains all arguments declared as `readwrite`, while the second contains those declared as `readonly`. In this example, the first array is empty and the second contains the acc argument. Also note that tasks are started using a method in a `TaskSystem` class (Line 11). This class is also part of the Rolez runtime library and can be used to execute a task in a new thread or in a thread from a thread pool. A task is executed by calling the `runRolez` method, which contains the generated code of the task (Line 9). As soon as the `runRolez` method has finished, its return value is stored in the Task instance and can be retrieved using the `get` method, as shown on Line 13.

4 Code-managed Runtime Data

To achieve high performance with Rolez, efficient guarding is paramount, because a guard is conceptually required for every single read or write access to an object. And even though many redundant guards can be eliminated using concurrency analysis (Section 5), some guards may remain.

Guarding Overview Guarding is implemented in a class called `Guarded`, which is part of the Rolez runtime library. It contains fields and methods for maintaining and checking the roles of an object. As all Rolez objects are instances of a subclass of `Guarded`, they inherit these fields and methods. The following snippet shows how the `Guarded` class represents the roles of an object, using two fields. Note that the actual implementation is more complex, but this simplified version is more suitable for illustration purposes.

```java
public abstract class Guarded {
    private volatile int ownerId;
    private AtomicInteger readers;
    // ...
}
```

First, the `ownerId` field stores the ID of the current owner task of the object. This is the single task in which the object currently plays the `readwrite` role or, if the object was shared as `readonly`, the last task in which it played the `readwrite` role. Second, `readers` stores the number of tasks in which the object currently plays the `readonly` role. This information is sufficient to determine all roles of an object.

Besides the fields that store an object’s roles, the `Guarded` class contains the methods that implement role transitions and, most importantly, guarding. A guard operation checks whether the role of the target object in the current task permits the given operation and, if not, blocks the execution of the current task. Using the above representation of roles, guarding for `readwrite` can be implemented as follows:

```java
public final void guardReadWrite() { 
    while(!isReadWrite()) { LockSupport.park(); } 
}
private boolean isReadWrite() { 
    long taskId = /* get ID of current task */;
    return this.ownerId == taskId && this.readers.get() == 0;
}
```

The `isReadWrite` method checks if “this” object plays the `readwrite` role by comparing the ID of the current task with that in the object’s `ownerId` field. In addition, it checks that the object is not currently shared as `readonly`, in which case it would currently also play the `readonly` role in the owner task. The `guardReadWrite` method simply parks the currently executing thread, using the `LockSupport.park()` method from the Java standard library, until `isReadWrite()` returns `true`. Whenever a role transition happens, parked threads are unparked so that they can check the role again. The `guardReadOnly` method is implemented similarly.

Code Generation for the Current Task The challenging part in the guarding implementation is to efficiently retrieve the “current” task, i.e., the task in which the given code is being executed. This information could be stored in a thread-local variable, e.g., using the `ThreadLocal` class from the Java standard library. However, we found that reading such a thread-local variable is relatively slow compared to a normal heap read or write. Thus, reading a thread-local variable for every guarded operation would be inefficient.

Instead of solving this issue purely in the runtime system, we leverage a code generation technique: To each Java method generated from a Rolez method, the compiler adds an additional parameter for the ID of the current task. Then, each generated method invocation passes the current task in the enclosing method on to the invoked method. Finally, when a guard is inserted, the current task in the enclosing method is passed on to the `guard*()` method.

Figure 5 shows how this mechanism works using two example classes written in Rolez and the corresponding generated Java classes. Lines 11 and 16 show how an additional `$task` parameter is added to methods, while Lines 6 and 7 show how these methods are called. The actual ID of the currently executing task is retrieved exactly once, at the beginning of a task (Line 5). Finally, Line 17 shows how the ID is ultimately passed to `guardReadOnly`, which guards the read operation in the `getBalance` method. The `guardReadOnly` method simply passes the ID on to `isReadOnly`, where it is compared to the owner of the object:

```java
public final void guardReadOnly(int $task) { 
    while(!isReadOnly($task)) { LockSupport.park(); } 
}
```
class Banking {
    task computeInterest(acc: readonly Account): int {
        val balance = acc.getBalance();
        return complexComputation(balance);
    }
    def complexComputation(balance: int): int { /*...*/ }
}
class Account {
    var balance: int
    // ...
    def getBalance(): int { return this.balance; }
}

1 class Banking {
2     Task<Integer> computeInterest(final Account acc) {
3         return new Task<>((new Object[][], new Object[][])(acc)) {
4             protected Integer runRolez() {
5                 int $task = this.id();
6                 int balance = acc.getBalance($task);
7                 return complexComputation(balance, $task);
8             }
9         }
10     }
11 }
12 class Account {
13     int balance;
14     // ...
15     int getBalance(int $task) {
16         this.guardReadOnly($task);
17         return this.balance;
18     }
19 }
20 }

Figure 5. Code generation for passing the current task from method to method. On the top are two Rolez classes and on the bottom the corresponding generated Java code.

private boolean isReadOnly(int $task) {
    return this.ownerId == $task
    || this.readers.get() > 0;
}

Adding a method parameter to every method in the program may seem inefficient as well. In addition, the code snippets may imply that guarding could be implemented using, e.g., the ID of the current native Java thread, which is typically efficient to retrieve. However, the actual Guarded implementation is more complex and needs to store not only the number of readers, but the set of readers. To query this set efficiently, it is implemented as a bit set, relying on specific properties of Rolez task IDs. Since Rolez code could be executed in arbitrary threads, using the thread ID would not work. In addition, we have conducted experiments on the JVM that showed that adding a parameter to all methods of a program has no significant performance impact and that this approach is at least as efficient as using the thread ID.

Once inlined into the code that performs the guarded operation, a guard consists only of at most three field reads and a few comparisons. And because the ownerID and readers fields are in the same object as the field that is accessed by the guarded operation, they are likely to be stored closely in memory and to profit from CPU caching. Thus, guards result in very little overhead, despite being implemented without the explicit help of a JIT compiler.

5 Concurrency Analysis

Even though guards can be implemented efficiently, actually guarding every object access would still result in poor performance. Fortunately, this is not required, as many guards are redundant and can be eliminated, using concurrency analysis.

As explained in Section 2, guards prevent temporarily illegal operations. Such operations are illegal because the target object has been shared with a child task and currently plays a less permissive role than its declared one. The main insight behind concurrency analysis is that an operation in some task t can only be temporarily illegal if t has started a child task before and that child task has not yet finished. Or conversely, if there exists no child task, all objects are guaranteed to play the role that was declared for them. Concurrency analysis statically determines if there possibly exists any child task, for every point in the program. Where this is not the case, the compiler does not emit any guards.

**Modular Interprocedural Analysis** To be useful, concurrency analysis needs to be interprocedural, as a sound interprocedural analysis would have to assume that there already exists a child task at the beginning of a method’s execution. However, interprocedural analysis is usually expensive and precludes modular compilation, a standard feature for Java-like languages. The concurrency analysis is interprocedural and modular, offering the best of both worlds. This is achieved again using a co-design with code generation.

The key insight to make concurrency analysis both interprocedural and modular is that the analysis computes only a single bit of information per program point: whether there is at least one child task or not. This boolean information can be propagated through the program without actually analyzing the whole program at once. Instead, every method is analyzed using an intraprocedural version of the analysis, but twice: once under the assumption that there are child tasks at the beginning of the method, and once under the opposite assumption. Propagating the information through the program is done in two steps: First, the compiler generates two versions of every method, one for each assumption. And second, for every method call, the compiler generates a call to the version that matches the information computed for that program point by the intraprocedural analysis.
Because two different versions are generated for every method, the analysis is not only interprocedural, but also context-sensitive. When a method is called from a context with child tasks, the version with guarding is used, and when called from contexts without child tasks, the unguarded version. This is important for programs written in languages similar to Java, because these languages typically come with an extensive standard library that includes classes and methods that are used in many different parts of a program.

Example We illustrate concurrency analysis using the program in Figure 6. On the left, the figure shows the results of the intraprocedural version of the concurrency analysis. The $ symbol means that there possibly exist child tasks and thus guarding is necessary, while the $ symbol means the opposite. There are two columns: (1) shows the result of the analysis under the assumption that there are no child tasks at the beginning of a method, and (2) shows the results for the opposite assumption. Column (2) is empty for tasks; because it is impossible for new tasks to have any child tasks, there is no need to analyze a task under this assumption.

When the program begins execution in the main task, there are no child tasks, so no guarding is necessary on Line 2. The process method is analyzed twice. Assuming no child tasks exist initially, no guarding is necessary on Line 5, as shown in Column (1). However, since tasks are started inside the loop (Line 7), guarding is necessary for the whole loop and subsequently on Line 8. Under the opposite assumption, that child tasks initially exist, guarding is necessary throughout the whole method, as shown in Column (2). The printTotal method is also analyzed twice. Since no tasks are started inside the method, guarding is necessary or unnecessary, respectively, throughout the whole method, depending on the initial assumption. Finally, depositInterest is again analyzed only once, because it is a task. Even though there may be other tasks in the program by the time a depositInterest task is started, every instance of the task is guaranteed to have no child tasks when it starts execution. And because depositInterest does not start any tasks itself, no guarding is required throughout the whole task body.

The resulting generated code for this program is shown in Figure 7. Two versions are generated for each method, an $Unguarded and a $Guarded one, while there is only one version for tasks. Whenever a method is called, the compiler generates a call to the version that matches the results of the analysis shown in Figure 6. For example, in process$Unguarded on Line 10, the printTotal$Unguarded is called, because, for this version of the process method, the analysis assumed that there are no child tasks at the beginning of the method. However, after process has started some tasks, the guarded version of printTotal is called (Line 13), as the roles of objects may now differ from their declared ones. On the other hand, in process$Guarded, both calls to printTotal use the guarded version (Lines 16 and 19), because the analysis assumed that there exist child tasks at the beginning of this version of the process method.

Finally, the two generated versions of printTotal illustrate how the analysis actually eliminates guards. While the guarded version contains a guard that protects the access to the Account.balance field (Lines 30 and 31), the unguarded version accesses the field without a guard (Line 24), because for this version there cannot be any child tasks. Hence, the information that there are no child tasks is propagated all the way from the main task to this field access, even though every method is analyzed and compiled in isolation.

Dataflow Analysis The intraprocedural analysis that is illustrated in Figure 6 can be expressed as a forward dataflow analysis [9], which is performed for every method (twice) and every task (once). The analysis tracks the program state of interest, i.e., whether there are child tasks, along the edges of the control flow graph. To deal with loops, the dataflow analysis follows an iterative fixed-point algorithm.

To present the analysis concisely, we focus on a very small subset of the Rolez language. This subset includes only tasks and methods, and as statements just task starts and method invocations, as shown in Figure 8. Statements that affect control flow, like if-else statements or loops are not included explicitly, but are handled generally by combining program states when control flow joins. All other kinds of

\[\text{(1) (2)}\]

\begin{verbatim}
1 (1) (2) task main(): {
2     process(/.../);
3 }
4  def process(accounts: Array[Account]): {
5     printTotal(accounts);
6     for (var i = 0; i < accounts.length; i++)
7         start depositInterest(accounts[i]);
8     printTotal(accounts);
9 }
10 def printTotal(accounts: Array[Account]): {
11     var total = 0;
12     for (var i = 0; i < accounts.length; i++)
13         total += accounts[i].balance;
14     println(total);
15 }
16 task depositInterest(acc: readwrite Account): {
17     val balance = acc.balance;
18     acc.deposit(/.../);
19 }
\end{verbatim}

Figure 6. Concurrency analysis example.

\[^{3}\text{This code is simplified, leaving away the Task parameters introduced in Section 4 and simplifying the handling of arrays: the compiler wraps native Java arrays in instances of the GuardedArray class (a subclass of Guarded) and inserts guards where necessary.}\]

\[^{4}\text{Note that the process$Guarded method is not actually used anywhere in the program, but is still generated in case the process method is used in another Rolez program, where it may be called from a guarded context.}\]
Task<Void> main() {
    return new Task<Void>(new Object[0], new Object[0]) {
        protected Void runRolez() {
            process$Unguarded(/.../);
            return null;
        }
    }
}

Task<Void> depositInterest(Account acc) { /* ... */ }
System.out.println(total);

for (int i = 0; i < accounts.length; i++)
    TaskSystem.start(depositInterest(accounts[i]));
System.out.println(total);

for (int i = 0; i < accounts.length; i++)
    total += accounts[i].balance;
for (int i = 0; i < accounts.length; i++)
    accounts[i].guardReadOnly();

void process$Unguarded(Account[] accounts) {
    printTotal$Unguarded(accounts);
}

void process$Guarded(Account[] accounts) {
    printTotal$Guarded(accounts);
    TaskSystem.start(depositInterest(accounts[i]));
    printTotal$Guarded(accounts);
}

void printTotal$Guarded(Account[] accounts) {
    int total = 0;
    for (int i = 0; i < accounts.length; i++)
        total += accounts[i].balance;
    System.out.println(total);
}

void printTotal$Unguarded(Account[] accounts) {
    int total = 0;
    for (int i = 0; i < accounts.length; i++)
        accounts[i].guardReadOnly();
    total += accounts[i].balance;
    System.out.println(total);
}

Task<Void> main() {
    return new Task<Void>(new Object[]{}, new Object[]{}) {
        protected Void runRolez() {
            process$Unguarded(/.../);
            return null;
        }
    }
}

Task<Void> depositInterest(Account acc) { /* ... */ }
System.out.println(total);

for (int i = 0; i < accounts.length; i++)
    TaskSystem.start(depositInterest(accounts[i]));
System.out.println(total);

for (int i = 0; i < accounts.length; i++)
    total += accounts[i].balance;
for (int i = 0; i < accounts.length; i++)
    accounts[i].guardReadOnly();

void process$Unguarded(Account[] accounts) {
    printTotal$Unguarded(accounts);
}

void process$Guarded(Account[] accounts) {
    printTotal$Guarded(accounts);
    TaskSystem.start(depositInterest(accounts[i]));
    printTotal$Guarded(accounts);
}

void printTotal$Guarded(Account[] accounts) {
    int total = 0;
    for (int i = 0; i < accounts.length; i++)
        total += accounts[i].balance;
    System.out.println(total);
}

void printTotal$Unguarded(Account[] accounts) {
    int total = 0;
    for (int i = 0; i < accounts.length; i++)
        accounts[i].guardReadOnly();
    total += accounts[i].balance;
    System.out.println(total);
}

Task<Void> main() {
    return new Task<Void>(new Object[]{}, new Object[]{}) {
        protected Void runRolez() {
            process$Unguarded(/.../);
            return null;
        }
    }
}

Task<Void> depositInterest(Account acc) { /* ... */ }
System.out.println(total);

for (int i = 0; i < accounts.length; i++)
    TaskSystem.start(depositInterest(accounts[i]));
System.out.println(total);

for (int i = 0; i < accounts.length; i++)
    total += accounts[i].balance;
for (int i = 0; i < accounts.length; i++)
    accounts[i].guardReadOnly();

void process$Unguarded(Account[] accounts) {
    printTotal$Unguarded(accounts);
}

void process$Guarded(Account[] accounts) {
    printTotal$Guarded(accounts);
    TaskSystem.start(depositInterest(accounts[i]));
    printTotal$Guarded(accounts);
}

void printTotal$Guarded(Account[] accounts) {
    int total = 0;
    for (int i = 0; i < accounts.length; i++)
        total += accounts[i].balance;
    System.out.println(total);
}

void printTotal$Unguarded(Account[] accounts) {
    int total = 0;
    for (int i = 0; i < accounts.length; i++)
        accounts[i].guardReadOnly();
    total += accounts[i].balance;
    System.out.println(total);
}

Figure 7. Generated code for the program in Figure 6.

Figure 8. Program representation for the dataflow analysis.

statements that Rolez supports have no effect on the program state computed by the analysis and are omitted here.

As described in the literature [9], the dataflow analysis computes for each statement in the control flow graph (cfg) an in-state and an out-state, which represent the state of the program before and after a statement, respectively. We use $s \in \{\top, \bot\}$ to denote the program state, as shown in Figure 9. If $s = \top$, then there are possibly some child tasks; if $s = \bot$, there are none. A set of dataflow equations defines how a statement transforms the state of the program, using a set of transfer functions $f_{\text{stmt}}$. The out-state of a statement is the result of applying the transfer function to the in-state.

For a start statement, the program state changes to $\top$, as there is now at least one child task. For a method invocation, the program state depends on the declaration of the method. By default, a Rolez method implicitly joins all newly started tasks at the end of the method. Therefore, if there were no child tasks before the method invocation, there are again none afterwards. To override this behavior, a method can be declared as async, which allows newly started tasks to continue to execute after the method has returned. Thus, the program state depends on the presence of the async keyword, which is expressed using the async($m$) function.

The dataflow analysis performs a fixed-point iteration over all statements in a method’s cfg, applying the dataflow equations repeatedly until all in- or out-states are stable. In- and out-states are initialized with the $s_{\text{init}}$ state, except for the in-state of the first statement in a method, which is initialized with $s_{\text{entry}}$ instead. The entry state $s_{\text{entry}}$ depends on the initial assumption of the analysis: for methods, the analysis is performed once with $\top$ and once with $\bot$, while tasks are analyzed once, using $\bot$. When control flow joins, e.g., after an if–else, the combination operator $\sqcup$ combines the out-states of the predecessor statements. Since the analysis is conservative, $\sqcup$ is the logical disjunction, i.e., if there may exist child tasks on at least one incoming path, the analysis must assume that they may also exist after the join.

6 Evaluation

We evaluate the performance impact of the techniques in Sections 4 and 5 by addressing two main questions: (1) How large are the parallel speedups that a vm-independent implementation of a pp language with heavy runtime checking achieves using these techniques? (2) By how much can the
overhead of \( \mathcal{P} \) runtime checks be reduced by the techniques presented in this paper? To answer these questions, we implemented 6 parallel programs using the Rolez implementation and compare their performance to that of equivalent, but manually synchronized Java implementations.

**Experimental Setup** We implemented the following programs: parallel Quicksort and Mergesort; IDEA encryption and Monte Carlo financial simulation, both adapted from the Parallel Java Grande benchmark suite [16]; and two programs based on our implementation of a ray tracer, one that renders static images (simply called Ray Tracer) and one that renders animated scenes (called Animator).

All Java code is compiled using the OpenJdk 8 compiler and executed on the OpenJdk HotSpot JVM, using standard settings (i.e., no flags were specified). The Rolez compiler also performs other optimizations that are specific to the role-based nature of the checks, but these optimizations were disabled for all the experiments performed for this paper.

We measured the performance on a machine with four Intel Xeon E7-4830 processors with a total of 32 cores and 64 GB of main memory, running Ubuntu. To minimize warm-up effects from the jrt compiler in the jvm, we executed every program 5 to 10 times before measuring. Then, we repeated every experiment 30 times inside the same vm, taking the arithmetic mean. We also study different input sizes.

**Parallel Speedup** We compare the performance of all Rolez programs to that of equivalent sequential and manually parallelized Java versions. Figure 10 shows the parallel speedups that the Rolez programs achieve when using both optimization techniques from Sections 4 and 5, and compares them to those achieved by the Java implementations. Both the Rolez and the Java speedups are relative to the (same) single-threaded Java implementations, to show that the performance of Rolez is roughly on par with that of Java. For this experiment, we used the largest input size for all programs.

Three of the programs (IDEA, Monte Carlo, and Ray Tracer) achieve substantial speedups for 32 tasks, from 16x to almost 22x. The Rolez implementations of Animator, Mergesort, and Quicksort achieve slightly less substantial speedups of 2–9x. However, the Java implementations achieve only moderate speedups as well, so these lower speedups are likely caused by the limited scalability that is inherent in these programs.

The speedups for different numbers of tasks achieved by the Rolez programs follow roughly the same patterns as those achieved by the Java versions. For some programs, the lines in Figure 10 are almost indistinguishable, which means that the Rolez programs have very little runtime overhead when compared to Java. For others, the gap between the lines is clearly visible and illustrates that the Rolez runtime checking can sometimes still add a noticeable overhead when compared to manually synchronized implementations. However, with more tasks, substantial parallel speedups compared to single-threaded Java can be achieved nevertheless.

![Figure 10. Comparison of the parallel speedups achieved by the optimized Rolez (Blue) and Java (Gray) implementations, for different numbers of tasks. All speedups are relative to the single-threaded Java implementations.](image)

**Impact of Code-managed Runtime Data and Concurrency Analysis** To understand the impact of the presented techniques, we also compiled and executed the 6 Rolez programs described above with three different levels of optimizations: 1) no optimizations enabled, 2) with task parameters enabled, and 3) with task parameters and concurrency analysis enabled. For the first level, every guard retrieves the current task using a ThreadLocal variable.

Figure 11 shows the execution time overhead of the Rolez programs compared to the equivalent Java implementations, when run with 32 tasks and for three different input sizes. For some programs, like Animator and IDEA, the overhead

3Note that when disabling these optimizations, the compiler still generates task parameters and unguarded methods, but these are not used during execution. As explained earlier, we have found that adding a method parameter has no significant impact on performance.
is moderate (mostly below 20%), for all input sizes and optimization levels. In these programs, there is relatively little access to guarded objects, so few checks are required. For others, like Mergesort and Quicksort, the overhead is more pronounced and can be as high as 300% without optimizations. These programs perform little computation per access to a guarded object (in this case the array to be sorted), so the runtime checks comprise a larger share of execution time.

When comparing the different levels of optimizations, the figure shows that the task parameter technique alone can already reduce the runtime overhead substantially. For example, for Mergesort and Quicksort, the overhead is reduced roughly by 3×. For IDEA, the Rolez overhead can even become negative, which means that the performance difference to Java is so small that other factors, like variance in memory allocation, have a larger impact on execution time.

Adding concurrency analysis, the overhead is again reduced substantially for some programs. For example, for the Ray Tracer program, the overhead is reduced from 30–80% (with task parameters but without concurrency analysis) to about 0–20%. This program is structured like the example program in Figure 6: there is a single level of concurrent tasks that execute almost all the work in parallel. Since none of these tasks have child tasks, concurrency analysis is able to remove almost all guards. IDEA is structured similarly. On the other hand, Mergesort and Quicksort are implemented by forking off one task to sort the first part of the array and sorting the second half in the current task, for every parallel recursion step. Hence, the code that sorts the second part needs guarding, as there is a child task (sorting the first part).

Both techniques combined reduce the overhead of Rolez over Java substantially, for almost all programs. For example, for Quicksort, the overhead is decreased by a factor of about 4–8× and, for Ray Tracer, it is reduced from over 100% to almost 0% for the medium and large input sizes.

7 Related Work

VM-independent PP Runtime Checking Besides Rolez, there are many other languages and systems implemented on top of VMS that rely on a vm-independent approach to do runtime checking for parallelism and concurrency. For example, there are several software transactional memory (STM) implementations that rely on this approach. Hindman and Grossman [7] add an atomic block to Java using source-to-source translation and thus keep the implementation of the atomic block "quite separate from other concerns". Their checks directly use the current Java thread, which is more efficient to retrieve than a ThreadLocal value, so their implementation may not benefit from using dedicated method parameters. Similarly, DEUCE [8] is a Java STM framework, implemented using bytecode instrumentation. It uses the technique described in Section 4 to pass a "transaction context" from method to method. While the reason for using method parameters is not explicitly stated, we assume that this is done for performance reasons. However, their evaluation does not discuss the impact of this optimization; to the best of our knowledge, we are the first to describe this idea as a general technique for efficient PP runtime checking, and to evaluate its performance impact. Another STM-based language implemented using bytecode transformation is presented by Bättig and Gross [1]. Although not mentioned in the paper, their implementation also makes use of a transaction context that is passed using method parameters. In this work, we show that this technique can be used as a powerful optimization for non-transactional PP languages as well.

Concurrency Analysis The analysis presented in Section 5 is an interprocedural dataflow analysis. Such analyses have been researched for a long time, but are traditionally considered whole-program analyses [14]. While there are approaches to make dataflow analysis "modular", e.g., by Roundy et al. [12, 13], these approaches consider much larger "modules", e.g., whole libraries, whereas the presented concurrency analysis is modular on the method level, thanks to the integration with the code generator.

The idea of generating two copies of methods and switching between them has been employed in other contexts before. For example, Pizlo et al. [10] present a compiler optimization that generates multiple versions of the program
code, each optimized for a specific garbage collection (gc) phase. In contrast to the technique presented here, switching between different code versions is not based on a static analysis, but is determined at runtime, based on gc progress. The idea is also used in an stm system by Yoo et al. [17], where two versions of a function are generated: one with memory barriers that is used inside atomic blocks and an optimized version for calls outside atomic blocks. While the technique is similar to ours, the work presented here shows that the idea can be applied more generally to perform optimization based on interprocedural analysis in a modular fashion.

Another technique that is used to perform interprocedural optimization is method inlining [3, 11], and is performed by many jit compilers. Inlining a method can result in more optimization opportunities, as its body can be optimized in the context of the caller code. However, the vm-independent approach for implementing jit runtime checks poses challenges to performing interprocedural optimizations using method inlining, because inlining is done by the jit compiler, whereas the jit checking optimizations have to be applied on the source or bytecode level. A trick that is used by Bättig and Gross [1] is to compile and execute the program once without optimizations, record the inlining decisions made by the jit compiler, and then compile the program again, using this information to inline ahead of time.

8 Conclusion

Virtual machines have proven to be an ideal basis for implementing parallel languages and constructs with a lot of runtime checking. Implementing such checks in a vm-independent way can make them easier to maintain and extend, and simpler to port to other vms. We showed that this approach can nevertheless lead to efficient systems, when designing runtime, code generation, and static analyses together. Both presented techniques are based on such a co-design, the first between code generation and runtime, and the second between code generation and static analysis.

When these techniques are used to optimize Rolez programs, almost all programs achieve substantial parallel speedups and reach performance roughly on par with manually synchronized Java implementations. Both techniques reduce the runtime overhead substantially, sometimes for different programs and sometimes for the same ones. When combined, the techniques reduce the overhead even more; for one program, the the overhead is reduced from over 100% to almost 0%, illustrating the great potential of these techniques.

References