JATO: Native Code Atomicity for Java

Siliang Li[1], Yu David Liu[2], and Gang Tan[1]

[1] Department of Computer Science & Engineering, Lehigh University
[2] Department of Computer Science, SUNY Binghamton

Abstract. Atomicity enforcement in a multi-threaded application can be critical to the application's safety. In this paper, we take the challenge of enforcing atomicity in a multilingual application, which is developed in multiple programming languages. Specifically, we describe the design and implementation of JATO, which enforces the atomicity of a native method when a Java application invokes the native method through the Java Native Interface (JNI). JATO relies on a constraint-based system, which generates constraints from both Java and native code based on how Java objects are accessed by threads. Constraints are then solved to infer a set of Java objects that need to be locked in native methods to enforce the atomicity of the native method invocation. We also propose a number of optimizations that soundly improve the performance. Evaluation through JATO's prototype implementation demonstrates it enforces native-method atomicity with reasonable run-time overhead.

1 Introduction

Atomicity in programming languages is a fundamental concurrency property: a program fragment is *atomic* if its execution sequence—regardless of how the latter interleaves with other concurrent execution sequences at run time—exhibits the same "serial" behavior (*i.e.*, as if no interleaving happened). Atomicity significantly simplifies the reasoning about concurrent programs because invariants held by the atomic region in a serial execution naturally holds for a concurrent execution. Thanks to the proliferation of multi-core and many-core architectures, there is a resurgence of interest in atomicity, with active research including type systems and program analyses for atomicity enforcement and violation identification (*e.g.*, [6, 23, 11, 5]), efficient implementation techniques (*e.g.*, [10]) and alternative programming models (*e.g.*, [1, 21, 3]).

As we adopt these research ideas to serious production settings, one major hurdle to cross is to support atomicity across foreign function interfaces (FFIs). Almost all languages support an FFI for interoperating with modules in low-level languages (*e.g.*, [20, 28, 17]). For instance, numerous classes in `java.lang.*` and `java.io.*` packages in the Java Development Kit (JDK) use the Java Native Interface (JNI), the FFI for Java. Existing atomicity solutions rarely provide direct support for FFIs. More commonly, code accessed through FFIs—called *native code* in JNI—is treated as a "black box."

The "black box" assumption typically yields two implementations, either leading to severe performance penalty or unsoundness. In the first implementation, the behavior of the native code is over-approximated as "anything can happen," *i.e.*, any memory area may be accessed by the native code. In that scenario, a "stop-the-world" strategy

is usually required to guarantee soundness when native code is being executed—all other threads must be blocked. In the second implementation, the run-time behavior of native code is ignored, an unsound under-approximation. Atomicity violations may occur when native code happens to access the same memory area that it interleaves with. Such systems, no matter how sophisticated their support for atomicity for non-native code, technically conform to weak atomicity [22] at best. The lack of atomicity support for native code further complicates the design of new parallel/concurrent programming models. For example, several recent languages [1, 15, 3] are designed to make programs atomic by default, promoting the robustness of multi-core software. Native code poses difficulties for these languages: the lack of atomicity support for it is often cited [1] as a key reason for these languages to design "opt-out" constructs from their otherwise elegant implicit atomicity models.

We present JATO for atomicity enforcement across the JNI. It is standard knowledge that atomicity enforcement requires a precise accounting of the relationship between threads and their accessed memory. JATO is built upon the simple observation that despite rather different syntax and semantics between Java and native code, the memory access of both languages can be statically abstracted in a uniform manner. JATO first performs a static analysis to abstract memory access from both non-native code and native code, and then uses a lock-based implementation to guarantee atomicity, judiciously adding protection locks to selected memory locations. With the ability to treat code on both sides of the JNI as "white boxes" and perform precise analysis over them, our solution is not only sound, but also practical in terms of performance as demonstrated by a prototype implementation. This paper makes the following contributions:

- We propose a novel static analysis to precisely identify the set of Java objects whose protection is necessary for atomicity enforcement. The analysis is constructed as a constraint-based inference, which uniformly extracts memory-access constraints from JNI programs.
- The paper reports a prototype implementation, demonstrating the effectiveness and the performance impact on both micro-benchmarks and real-world applications.
- We propose a number of optimizations to further soundly improve the performance, such as no locks on read-only objects.

2 Background and Assumptions

The JNI allows Java programs to interface with low-level native code written in C, C++, or assembly languages. It allows Java code to invoke and to be invoked by native methods. A native method is declared in a Java class by adding the **native** modifier to a method. For example, the following `Node` class declares a native method named `add`:

```
class Node {int i=10; native void add (Node n);}
```

Once declared, native methods are invoked in Java in the same way as how Java methods are invoked. Note that the Java side may have multiple Java threads running, each of which may invoke some native method.

The implementation of a native method receives a set of Java-object references from the Java side; for instance, the above `add` method receives a reference to `this` object

and a reference to the `n` object. A native-method implementation can interact with Java through a set of JNI interface functions (called JNI functions hereafter) as well as using features provided by the native language. Through JNI functions, native methods can inspect/modify/create Java objects, invoke Java methods, and so on. As an example, it can invoke `MonitorEnter` to lock a Java object and `MonitorExit` to unlock a Java object.

Assumptions. In any language that supports atomicity, it is necessary to define the *atomic region*, a demarcation of the program to indicate where an atomic execution starts and where it ends. One approach is to introduce some special syntax and ask programmers to mark atomic regions – such as atomic blocks. JATO's assumption is that each native method forms an atomic region. This allows us to analyze unannotated JNI code directly. Furthermore, we believe that this assumption matches Java programmers' intuition nicely. Java programmers often view native methods as black boxes, avoiding the reasoning about interleaving between Java code and native code. Finally, the assumption does not affect expressiveness. For instance, an atomic region with two native method invocations can be encoded as creating a third native method whose body contains the two invocations. If there is Java code fragment in between the two invocations, the encoded version can model the Java code by inserting a Java callback between the two invocations. Overall, the core algorithm we propose stays the same regardless of the demarcation strategy of atomic regions.

When enforcing native-method atomicity, JATO focuses on those Java objects that cross the Java-native boundary. It ignores the memory regions owned by native methods. For instance, native code might have a global pointer to a memory buffer in the native heap and lack of protection of the buffer might cause atomicity violations. Enforcing this form of atomicity can be performed on the native-side alone (*e.g.*, [2]). Furthermore, native code cannot pass pointers that point to C buffers across the boundary because Java code does not understand C's type system; native code has to invoke JNI functions to create Java objects and pass references to those Java objects across the boundary. Because of these reasons, JATO focuses on language-interoperation issues and analyzes those cross-boundary Java objects.

3 The Formal Model

In this section, we use an idealized JNI language to describe the core of JATO: a constraint-based lock inference algorithm for ensuring the atomicity of native methods.

3.1 Abstract Syntax

The following BNF presents the abstract syntax of an idealized JNI language where notation $\overline{X}$ represents a sequence of X's. Its Java subset is similar to Featherweight Java (FJ) [13], but with explicit support for field update and let bindings. For simplicity, the language omits features such as type casting, constructors, field initializers, multi-argument methods on the Java side, and heap management on the native side.

$$
\begin{array}{lll}
P & ::= & \overline{\textbf{class}\ \mathsf{c}\ \textbf{extends}\ \mathsf{c}\ \{F\ M\ N\}} & \textit{classes} \\
F & ::= & \overline{\mathsf{c}\ \mathsf{f}} & \textit{fields} \\
M & ::= & \overline{\mathsf{c}\ \mathsf{m}(\mathsf{c}\ x)\{e\}} & \textit{Java methods} \\
N & ::= & \overline{\textbf{native}\ \mathsf{c}\ \mathsf{m}(\mathsf{c}\ x)\{t\}} & \textit{native methods} \\
e & ::= & x \mid \textbf{null} \mid e.\mathsf{f} \mid e.\mathsf{f} := e \mid e.\mathsf{m}(e) \mid \textbf{new}_{\ell}\ \mathsf{c} \mid \textbf{let}\ x = e\ \textbf{in}\ e & \textit{Java terms} \\
t & ::= & x \mid \textbf{null} \mid \texttt{GetField}(t, fd) \mid \texttt{SetField}(t, fd, t) & \textit{native terms} \\
 & & \mid \texttt{NewObject}_{\ell}(\mathsf{c}) \mid \texttt{CallMethod}(t, md, t) \mid \textbf{let}\ x = t\ \textbf{in}\ t & \\
\textit{bd} & ::= & e \mid t & \textit{method body} \\
fd & ::= & \langle \mathsf{c}, \mathsf{f} \rangle & \textit{field ID} \\
md & ::= & \langle \mathsf{c}, \mathsf{m} \rangle & \textit{method ID}
\end{array}
$$

A program is composed of a sequence of classes, each of which in turn is composed of a sequence of fields F, a sequence of Java methods M, and a sequence of native methods N. In this JNI language, both Java and native code are within the definition of classes; real JNI programs have separate files for native code. As a convention, metavariable $\mathsf{c}(\in \mathbb{CN})$ is used for class names, f for field names, m for method names, and x for variable names. The root class is `Object`. We use e for a Java term, and t for a native term. A native method uses a set of JNI functions for accessing Java objects. `GetField` and `SetField` access a field via a field ID, and `CallMethod` invokes a method defined on a Java object, which could either be implemented in Java or in native code. Both the Java-side instantiation expression (**new**) and the native-side counterpart (`NewObject`) are annotated with labels $\ell(\in \mathbb{LAB})$ and we require distinctness of all ℓ's in the code. We use notation $\mathcal{L}_P : \mathbb{LAB} \mapsto \mathbb{CN}$ to represent the mapping function from labels to the names of the instantiated classes as exhibited in program P. We use $mbody(\mathsf{m}, \mathsf{c})$ to compute the method body of m of class c, represented as $x.bd$ where x is the parameter and *bd* is the definition of the method body. The definition of this function is identical to FJ's namesake function when m is a Java method. When m is a native method, the only difference is that the method should be looked up in N instead of M. We omit this lengthy definition in this short presentation.

Throughout this section, we will use a toy example to illustrate ideas, presented in Fig. 1. We liberally use void and primitive types, constructors, and use "$x = e_1; e_2$" for **let** $x = e_1$ **in** e_2. Note that the `Node` class contains a native method for adding integers of two `Node` objects and updating the receiver object. The goal in our context is to insert appropriate locks to ensure the execution of this native method being atomic.

3.2 Constraint Generation: An Overview

Atomicity enforcement relies on a precise accounting of memory access, which in JATO is abstracted as constraints. Constraints are generated through a type inference algorithm, defined in two steps: (1) constraints are generated intraprocedurally, both for Java methods and native methods; (2) all constraints are combined together through a closure process, analogous to interprocedural type propagation. The two-step approach is not surprising for object-oriented type inference, because dynamic dispatch approximation and concrete class analysis are long known to be intertwined in the presence of

```
class Node extends Object {
  int i=10;
  native void add (Node n) {
    x1=GetField(this,<Node,i>);
    x2=GetField(n,<Node,i>);
    SetField(this,<Node,i>,x1+x2);}}
class Thread2 extends Thread {
  Node n1, n2;
  Thread2(Node n1, Node n2) {
    this.n1=n1; this.n2=n2;}
  void run() {n2.add(n1);}}
```

```
class Main extends Object {
  void main() {
    n1=new Node_ℓ1();
    n2=new Node_ℓ2();
    th=new Thread2_ℓth(n1,n2);
    th.start();
    n1.add(n2);
  }
}
```

Fig. 1. A running example

interprocedural analysis [27]: approximating dynamic dispatch – *i.e.*, determine which methods would be used to enable interprocedural analysis – requires the knowledge of the *concrete classes* (*i.e.*, the class of the run-time object) of the receiver, but interprocedural analysis is usually required to compute the concrete classes of the receiver object. JATO performs step (1) to intraprocedurally generate constraints useful for dynamic dispatch approximation and concrete class analysis, and then relies on step (2) to perform the two tasks based on the constraints. The details of the two steps are described in Sec. 3.3 and Sec. 3.4, respectively.

One interesting aspect of JATO is that both Java code and native code will be abstracted into the same forms of constraints after step (1). JATO constraints are:

$$
\begin{array}{rll}
\mathcal{K} ::= & \overline{\kappa} & \textit{constraint set} \\
\kappa ::= & \alpha \xrightarrow{\theta} \alpha' \mid \alpha \leq \alpha' \mid [\alpha.\mathsf{m}]^{\alpha'} & \textit{constraint} \\
\theta ::= & \mathsf{R} \mid \mathsf{W} & \textit{access mode} \\
\alpha ::= & \ell \mid \phi \mid \texttt{thisO} \mid \texttt{thisT} & \textit{abstract object/thread} \\
 & \mid \alpha.\mathsf{f} \mid \alpha.\mathsf{m}^{+} \mid \alpha.\mathsf{m}^{-} &
\end{array}
$$

An *access constraint* $\alpha \xrightarrow{\theta} \alpha'$ says that an (abstract) object α accesses an (abstract) object α', and the access is either a read ($\theta = \mathsf{R}$) or a write ($\theta = \mathsf{W}$). Objects in JATO's static system are represented in several forms. The first form is an instantiation site label ℓ. Recall earlier, we have required all ℓ's associated with the instantiation expressions (**new** or `NewObject`) to be distinct. It is thus natural to represent abstract objects with instantiation site labels. Our formal system's precision is thus middle-of-the-road: we differentiate objects of the same class if they are instantiated from different sites, but reins in the complexity by leaving out more precise features such as nCFA [29] or n-object context-sensitivity [25]. The other forms of α are used by the type inference algorithm: label variables $\phi \in \mathbb{LVAR}$, `thisO` for the object enclosing the code being analyzed, `thisT` for the thread executing the code being analyzed, $\alpha.\mathsf{f}$ for an alias to field f of object α, and $\alpha.\mathsf{m}^{+}$ and $\alpha.\mathsf{m}^{-}$ for aliases to the return value and the formal parameter of a method invocation to method name m of α, respectively.

$$\text{(T-Read)}\ \frac{\Gamma \vdash e : \alpha \backslash \mathcal{K}}{\Gamma \vdash e.\mathsf{f} : \alpha.\mathsf{f} \backslash \mathcal{K} \cup \{\texttt{thisT} \xrightarrow{\mathsf{R}} \alpha\}}$$

$$\text{(T-Write)}\ \frac{\Gamma \vdash e : \alpha \backslash \mathcal{K} \qquad \Gamma \vdash e' : \alpha' \backslash \mathcal{K}'}{\Gamma \vdash e.\mathsf{f} := e' : \alpha' \backslash \mathcal{K} \cup \mathcal{K}' \cup \{\alpha' \leq \alpha.\mathsf{f},\ \texttt{thisT} \xrightarrow{\mathsf{W}} \alpha\}}$$

$$\text{(T-Msg)}\ \frac{\Gamma \vdash e : \alpha \backslash \mathcal{K} \qquad \Gamma \vdash e' : \alpha' \backslash \mathcal{K}'}{\Gamma \vdash e.\mathsf{m}(e') : \alpha.\mathsf{m}^{+} \backslash \mathcal{K} \cup \mathcal{K}' \cup \{\alpha' \leq \alpha.\mathsf{m}^{-},\ [\alpha.\mathsf{m}]^{\texttt{thisT}}\}}$$

$$\text{(T-Thread)}\ \frac{\Gamma \vdash e : \alpha \backslash \mathcal{K} \qquad \mathit{javaT}(\Gamma, e) \text{ is of a thread class}}{\Gamma \vdash e.\texttt{start}() : \alpha \backslash \mathcal{K} \cup \{[\alpha.\texttt{run}]^{\alpha}\}}$$

$$\text{(T-New)}\ \Gamma \vdash \mathbf{new}_{\ell}\ \mathsf{c} : \ell \backslash \emptyset \qquad \text{(T-NewThread)}\ \frac{\mathsf{c} \text{ is of a thread class} \qquad \phi \text{ fresh}}{\Gamma \vdash \mathbf{new}_{\ell}\ \mathsf{c} : \ell \backslash \{\ell \leq \phi,\ \phi \leq \ell\}}$$

$$\text{(T-Var)}\ \Gamma \vdash x : \Gamma(x) \backslash \emptyset \qquad \text{(T-Null)}\ \Gamma \vdash \mathbf{null} : \ell_{\texttt{null}} \backslash \emptyset$$

$$\text{(T-Let)}\ \frac{\Gamma \vdash e : \alpha \backslash \mathcal{K} \qquad \Gamma \rhd [x \mapsto \alpha] \vdash e' : \alpha' \backslash \mathcal{K}'}{\Gamma \vdash \mathbf{let}\ x = e\ \mathbf{in}\ e' : \alpha' \backslash \mathcal{K} \cup \mathcal{K}'}$$

Fig. 2. Java-Side Intraprocedual Constraint Generation

The additional two forms of constraints, $\alpha \leq \alpha'$ and $[\alpha.\mathsf{m}]^{\alpha'}$, are used for concrete class analysis and dynamic dispatch approximation, respectively. Constraint $\alpha \leq \alpha'$ says that α may flow into α'. At a high level, one can view this form of constraint as relating two aliases. (As we shall see, the transitive closure of the binary relation defined by $\leq$ is *de facto* a concrete class analysis.) Constraint $[\alpha.\mathsf{m}]^{\alpha'}$ is a *dynamic dispatch placeholder*, denoting method m of object α is being invoked by thread α'.

3.3 Intraprocedural Constraint Generation

We now describe the constraint-generation rules for Step (1) described in Sec. 3.2. Fig. 2 and Fig. 3 are rules for Java code and native code, respectively. The class-level constraint-generation rules are defined in Fig. 4. *Environment* Γ is a mapping from x's to α's. *Constraint summary* $\mathcal{M}$ is a mapping from method names to constraint sets. Judgement $\Gamma \vdash e : \alpha \backslash \mathcal{K}$ says expression e has type α under environment Γ and constraints $\mathcal{K}$. Since no confusion can exist, we further use $\Gamma \vdash t : \alpha \backslash \mathcal{K}$ to represent the analogous judgement for native term t. Judgement $\vdash_{\texttt{cls}}\ \mathbf{class}\ \mathsf{c} \ldots \backslash \mathcal{M}$ says the constraint summary of class c is $\mathcal{M}$. Operator $\rhd$ is a mapping update: given a mapping U, $U \rhd [u \mapsto v]$ is identical to U except element u maps to v in $U \rhd [u \mapsto v]$.

Observe that types are abstract objects (represented by α's). Java nominal typing (class names as types) is largely orthogonal to our interest here, so our type system

$$\text{(TN-Read)}\ \frac{\Gamma \vdash t : \alpha \backslash \mathcal{K} \qquad fd = \langle \mathsf{c}, \mathsf{f} \rangle}{\Gamma \vdash \texttt{GetField}(t, fd) : \alpha.\mathsf{f} \backslash \mathcal{K} \cup \{\texttt{thisT} \xrightarrow{\mathsf{R}} \alpha\}}$$

$$\text{(TN-Write)}\ \frac{\Gamma \vdash t : \alpha \backslash \mathcal{K} \qquad \Gamma \vdash t' : \alpha' \backslash \mathcal{K}' \qquad fd = \langle \mathsf{c}, \mathsf{f} \rangle}{\Gamma \vdash \texttt{SetField}(t, fd, t') : \alpha' \backslash \mathcal{K} \cup \mathcal{K}' \cup \{\alpha' \leq \alpha.\mathsf{f},\ \texttt{thisT} \xrightarrow{\mathsf{W}} \alpha\}}$$

$$\text{(TN-Msg)}\ \frac{\Gamma \vdash t : \alpha \backslash \mathcal{K} \qquad \Gamma \vdash t' : \alpha' \backslash \mathcal{K}' \qquad md = \langle \mathsf{c}, \mathsf{m} \rangle}{\Gamma \vdash \texttt{CallMethod}(t, md, t') : \alpha.\mathsf{m}^{+} \backslash \mathcal{K} \cup \mathcal{K}' \cup \{\alpha' \leq \alpha.\mathsf{m}^{-},\ [\alpha.\mathsf{m}]^{\texttt{thisT}}\}}$$

$$\text{(TN-Thread)}\ \frac{\Gamma \vdash t : \alpha \backslash \mathcal{K} \qquad md = \langle \mathsf{c}, \texttt{start} \rangle \qquad \mathsf{c} \text{ is a thread class}}{\Gamma \vdash \texttt{CallMethod}(t, md) : \alpha \backslash \mathcal{K} \cup \{[\alpha.\texttt{run}]^{\alpha}\}}$$

$$\text{(TN-New)}\ \Gamma \vdash \mathbf{new}_{\ell}\ \mathsf{c} : \ell \backslash \emptyset$$

$$\text{(TN-NewThread)}\ \frac{\mathsf{c} \text{ is a thread class} \qquad \phi \text{ fresh}}{\Gamma \vdash \texttt{NewObject}_{\ell}(\mathsf{c}) : \ell \backslash \{\ell \leq \phi,\ \phi \leq \ell\}}$$

$$\text{(TN-Var)}\ \Gamma \vdash x : \Gamma(x) \backslash \emptyset \qquad\qquad \text{(TN-Null)}\ \Gamma \vdash \mathbf{null} : \ell_{\texttt{null}} \backslash \emptyset$$

$$\text{(TN-Let)}\ \frac{\Gamma \vdash t : \alpha \backslash \mathcal{K} \qquad \Gamma \rhd [x \mapsto \alpha] \vdash t' : \alpha' \backslash \mathcal{K}'}{\Gamma \vdash \mathbf{let}\ x = t\ \mathbf{in}\ t' : \alpha' \backslash \mathcal{K} \cup \mathcal{K}'}$$

Fig. 3. Native-Side Intraprocedural Constraint Generation

does not include it. Taking an alternative view, one can imagine we only analyze programs already typed through Java-style nominal typing. For that reason, we liberally use function $javaT(\Gamma, e)$ to compute the class names for expression e.

On the Java side, (T-Read) and (T-Write) generate constraints to represent the read-/write access from the current thread (`thisT`) to the object whose field is being read-/written (α in both rules). The constraint $\alpha' \leq \alpha.\mathsf{f}$ in (T-Write) abstracts the fact that e' flows into the field f of e, capturing the data flow. The flow constraint generated by (T-Msg) is for the flow from the argument to the parameter of the method. That rule in addition generates a dynamic dispatch placeholder. (T-Thread) models the somewhat stylistic way Java performs thread creation: when an object of a thread class is sent a `start` message, the `run` method of the same object will be wrapped up in a new thread and executed. (T-New) says that the label used to annotate the instantiation point will be used as the type of the instantiated object. (T-NewThread) creates one additional label variable to represent the thread object. The goal here is to compensate the loss of precision of static analysis, which in turn would have affected soundness: a thread object may very well be part of a recursive context (a loop for example) where one instantiation point may be mapped to multiple run-time instances. The static analysis

$$\text{(T-Cls)}\ \frac{\begin{array}{c}\vdash_{\text{cls}} \textbf{class}\ \mathsf{c}_0 \ldots \backslash \mathcal{M} \\ [\text{this} \mapsto \texttt{thisO},\ x \mapsto \texttt{thisO}.\mathsf{m}^-] \vdash bd : \alpha \backslash \mathcal{K} \text{ for all } mbody(\mathsf{m}, \mathsf{c}) = x.bd \\ \mathcal{K}' = \mathcal{K} \cup \{\alpha \leq \texttt{thisO}.\mathsf{m}^+\}\end{array}}{\vdash_{\text{cls}} \textbf{class}\ \mathsf{c}\ \textbf{extends}\ \mathsf{c}_0\ \{F\ M\ N\} \backslash (\mathcal{M} \rhd \overline{\mathsf{m} \mapsto \mathcal{K}'})}$$

$$\text{(T-ClsTop)}\ \vdash_{\text{cls}} \textbf{class}\ \texttt{Object} \backslash []$$

Fig. 4. Class-Level Constraint Generation

needs to be aware if all such instances access one shared memory location – a soundness issue because exclusive access by one thread or shared access by multiple threads have drastically different implications in reasoning about multi-threaded programs. The solution here is called *doubling* [15, 34], treating every instantiation point for thread objects as two threads. Observe that we do not perform doubling for non-thread objects in (T-New) because there is no soundness concern there. The rest of the three rules should be obvious, where $\ell_{\texttt{null}}$ is a predefined label for **null**. For the running example, the following constraints will be generated for the two classes written in Java:

$$\texttt{Main} : \{\texttt{main} \mapsto \{\ell_{th} \leq \phi_2,\ \phi_2 \leq \ell_{th},\ [\ell_{th}.\texttt{run}]^{\ell_{th}},\ \ell_2 \leq \ell_1.\texttt{add}^-,\ [\ell_1.\texttt{add}]^{\texttt{thisT}}\}\}$$
$$\texttt{Thread2} : \{\texttt{run} \mapsto \{\ell_1 \leq \ell_2.\texttt{add}^-,\ [\ell_2.\texttt{add}]^{\texttt{thisT}}\}\}$$

The native-side inference rules have a one-on-one correspondence with the Java-side rules – as related by names – and every pair of corresponding rules generate the same form of constraints. This is a crucial insight of JATO: by abstracting the two worlds of Java syntax and native code syntax into one unified constraint representation, the artificial boundary between Java and native code disappears. As a result, thorny problems such as callbacks (to Java) inside native code no longer exists – the two worlds, after constraints are generated, are effectively one. The constraints for the `Node` class in the running example are:

$$\texttt{Node} : \{\texttt{add} \mapsto \{\texttt{thisT} \xrightarrow{\mathsf{R}} \texttt{thisO},\ \texttt{thisT} \xrightarrow{\mathsf{W}} \texttt{thisO},\ \texttt{thisT} \xrightarrow{\mathsf{R}} \texttt{thisO}.\texttt{add}^-\}\}$$

3.4 Constraint Closure

Now that the constraint summary has been generated on a per-class per-method basis, we can discuss how to combine them into one global set. This is defined by computing the *constraint closure*, defined as follows:

Definition 1 (Constraint Closure). *The closure of program P with entry method md, denoted as $[\![\, P, md \,]\!]$ is the smallest set that satisfies the following conditions:*

- ***Flows**: $\leq$ is reflexive and transitive in $[\![\, P, md \,]\!]$.*

- ***Concrete Class Approaching**: If* $\{\alpha' \leq \alpha\} \cup \mathcal{K} \subseteq [\![P, md]\!]$*, then* $\mathcal{K}\{\alpha'/\alpha\} \subseteq [\![P, md]\!]$.
- ***Dynamic Dispatch**: If* $[\ell.\mathsf{m}]^{\ell_0} \in [\![P, md]\!]$*, then* $\mathcal{M}(\mathsf{m})\{\ell/\mathtt{thisO}\}\{\ell_0/\mathtt{thisT}\} \subseteq [\![P, md]\!]$ *where* $\mathcal{L}_P(\ell) = \mathsf{c}$ *and* $\vdash_{\mathtt{cls}}$ **class** $\mathsf{c} \ldots \backslash\mathcal{M}$.
- ***Bootstrapping**:* $\{[\ell_{\mathtt{BO}}.\mathsf{m}]^{\ell_{\mathtt{BT}}}, \ell_{\mathtt{BP}} \leq \ell_{\mathtt{BO}}.\mathsf{m}^-\} \subseteq [\![P, md]\!]$ *where* $md = \langle \mathsf{c}, \mathsf{m} \rangle$.

The combination of **Flows** and **Concrete Class Approaching** is *de facto* a concrete class analysis, where the "concrete class" in our case is the object instantiation sites (not Java nominal types): the **Flows** rule interprocedurally builds the data flow, and the **Concrete Class Approaching** rule substitutes a flow element with one "up stream" on the data flow. When the "source" of the data flow – an instantiation point label – is substituted in, concrete class analysis is achieved. Standard notation $\mathcal{K}\{\alpha'/\alpha\}$ substitutes every occurrence of α in $\mathcal{K}$ with α'. **Dynamic Dispatch** says that once the receiver object of an invocation resolves to a concrete class, dynamic dispatch can thus be resolved. The substitutions of `thisO` and `thisT` are not surprising from an interprocedural perspective. The last rule, **Bootstrapping**, bootstraps the closure. $\ell_{\mathtt{BO}}, \ell_{\mathtt{BT}}, \ell_{\mathtt{BP}}$ are pre-defined labels representing the bootstrapping object (the one with method md), the bootstrapping thread, and the parameter used for the bootstrapping invocation.

For instance, if P is the running example, the following constraints are among the ones in the closure from its main method, *i.e.*, $[\![P, \langle \mathsf{c}_{\mathtt{main}}, \mathsf{m}_{\mathtt{main}} \rangle]\!]$:

$$\begin{array}{lll} \ell_{\mathtt{BT}} \xrightarrow{\mathsf{R}} \ell_1 & \ell_{\mathtt{BT}} \xrightarrow{\mathsf{W}} \ell_1 & \ell_{\mathtt{BT}} \xrightarrow{\mathsf{R}} \ell_2 \\ \ell_{th} \xrightarrow{\mathsf{R}} \ell_2 & \ell_{th} \xrightarrow{\mathsf{W}} \ell_2 & \ell_{th} \xrightarrow{\mathsf{R}} \ell_1 \end{array}$$

That is, the bootstrapping thread performs read and write access to object ℓ_1 and read access to object ℓ_2. The child thread performs read access to object ℓ_1 and read and write access to object ℓ_2. This matches our intuition about the program.

3.5 Atomicity Enforcement

Based on the generated constraints, JATO infers a set of Java objects that need to be locked in a native method to ensure its atomicity. JATO also takes several optimizing steps to remove unnecessary locks while still maintaining atomicity.

Lock-all. The simplest way to ensure atomicity is to insert locks for all objects that a native method may read from or write to. Suppose we need to enforce the atomicity of a native method md in a program P, the set of objects that need to be locked are:

$$Acc(P, md) \stackrel{\text{def}}{=} \{\, \alpha \mid (\alpha' \xrightarrow{\theta} \alpha) \in [\![P, md]\!] \wedge (\alpha \in \mathbb{LAB} \vee labs(\alpha) \subseteq \{\, \ell_{\mathtt{BO}}, \ell_{\mathtt{BP}} \,\}) \,\}$$

The first predicate $(\alpha' \xrightarrow{\theta} \alpha) \in [\![P, md]\!]$ says that α is indeed read or written. The α's that satisfy this predicate may be in a form that represents an alias to an object, such as $\ell.\mathsf{f}_1.\mathsf{f}_2.\mathsf{m}^+$, and it is clearly desirable to only inform the lock insertion procedure of the real instantiation point of theobject (the $\alpha \in \mathbb{LAB}$ predicate) – *e.g.*, "please lock the object instantiated at label ℓ_{33}." This, however, is not always possible because the instantiation site for the object enclosing the native method and that for the native method

parameter are abstractly represented as $\ell_{\texttt{BO}}$ and $\ell_{\texttt{BP}}$, respectively. It is thus impossible to concretize any abstract object whose representation is "built around them". For example, $\ell_{\texttt{BO}}.\mathsf{f}_3$ means that the object is stored in field f_3 of the enclosing object $\ell_{\texttt{BO}}$, and access to the stored object requires locking the enclosing object. This is the intuition behind predicate $labs(\alpha) \subseteq \{\, \ell_{\texttt{BO}}, \ell_{\texttt{BP}} \,\}$, where $labs(\alpha)$ enumerates all the labels in α.

For the running example, the set of objects to lock for the native `add` method – $Acc(P, \langle \texttt{Node}, \texttt{add} \rangle)$ – is $\{\, \ell_{\texttt{BO}},\ \ell_{\texttt{BP}} \,\}$, meaning both the enclosing object and the parameter needs to be locked.

Locking all objects in $Acc(P, md)$ is sufficient to guarantee the atomicity of md. This comes as no surprise: every memory access by the native method is guarded by a lock. The baseline approach here is analogous to a purely dynamic approach: instead of statically computing the closure and the set of objects to be locked as we define here, one could indeed achieve the same effect by just locking at run time for every object access.

In the lock-all approach, JATO inserts code that acquires the lock for each object in the set as computed above and releases the lock at the end. The lock is acquired by JNI function `MonitorEnter` and released by `MonitorExit`.

Lock-on-write. In this strategy, we differentiate read and write access, and optimize based on the widely known fact that non-exclusive reads and exclusive writes are adequate to guarantee atomicity. The basic idea is simple: given a constraint set $\mathcal{K}$, only elements in the following set needs to be locked, where $size$ computes the size of a set:

$$lockS(\mathcal{K}) \stackrel{\text{def}}{=} \{\ell \mid size(\{\ell' \mid \ell' \xrightarrow{\mathsf{W}} \ell \in \mathcal{K}\}) \neq 0 \wedge size(\{\ell' \mid \ell' \xrightarrow{\theta} \ell \in \mathcal{K}\}) > 1\}$$

It would be tempting to compute the necessary locks for enforcing the atomicity of native method md of program P as $lockS(\llbracket\, P, md \,\rrbracket)$. This unfortunately would be unsound. Consider the running example. Even though the parameter object `n` is only read accessed in native method `add`, it is not safe to remove the lock due to two facts: (1) in the main thread, `add` receives object ℓ_1 as the argument; (2) in the child thread, object ℓ_1 is mutated. If the lock to the parameter object `n` were removed, atomicity of `add` could not be guaranteed since the integer value in the parameter object may be mutated in the middle of the method. Therefore, it is necessary to perform a global analysis to apply the optimization.

The next attempt would be to lock objects in $lockS(\llbracket\, P, \langle \mathsf{c}_{\texttt{main}}, \mathsf{m}_{\texttt{main}} \rangle \,\rrbracket)$. Clearly, this is sound, but it does not take into the account that native method md only accesses a subset of these objects. To compute the objects that are accessed by md, we define function $AccG(P, md)$ as the smallest set satisfying the following conditions, where $md = \langle \mathsf{c}, \mathsf{m} \rangle$ and $\mathcal{K}_0 = \llbracket\, P, \langle \mathsf{c}_{\texttt{main}}, \mathsf{m}_{\texttt{main}} \rangle \,\rrbracket$:

- If $\{[\ell.\mathsf{m}]^{\ell_0}, \ell_1 \leq \ell.\mathsf{m}^{-}\} \subseteq \mathcal{K}_0$ where $\mathcal{L}_P(\ell) = \mathsf{c}$, then $Acc(P, md)\{\ell/\ell_{\texttt{BO}}\}\{\ell_0/\ell_{\texttt{BT}}\}\{\ell_1/\ell_{\texttt{BP}}\} \subseteq AccG(P, md)$.
- If $\ell \leq \alpha \in \mathcal{K}_0$ and $\alpha \in AccG(P, md)$, then $\ell \in AccG(P, md)$.

In other words, $Acc(P, md)$ almost fits our need, except that it contains placeholder labels such as $\ell_{\texttt{BO}}$, $\ell_{\texttt{BT}}$, and $\ell_{\texttt{BP}}$. $AccG$ concretizes any abstract object whose representation is dependent on them. With this definition, we can now define our strategy: locks are needed for native method md of program P for any object in the following set:

$$AccG(P, md) \cap lockS([\![P, \langle \mathsf{c}_{\texttt{main}}, \mathsf{m}_{\texttt{main}} \rangle]\!]).$$

Lock-at-write-site. Instead of acquiring the lock of an object at the beginning of a native method and releasing the lock at the end, this optimization inserts locking around the code region of the native method that accesses the object. If there are multiple accesses of the object, JATO finds the smallest code region that covers all accesses and acquires/releases the lock only once.

4 Prototype implementation

We implemented a prototype system based on the constraint-based system described in the previous section. Java-side constraint generation in JATO is built upon Cypress [35], a static analysis framework focusing on memory access patterns. Native-side constraint generation is implemented in CIL [26], an infrastructure for analyzing and transforming C code. The rest of JATO is developed in around 5,000 lines of OCaml code.

One issue that we have ignored in the idealized JNI language is the necessity of performing Java-type analysis in native code. In the idealized language, native methods can directly use field IDs in the form of $\langle \mathsf{c}, \mathsf{f} \rangle$ (and similarly for method IDs). But in real JNI programs, native methods have to invoke certain JNI functions to construct those IDs. To read a field of a Java object, native method must take three steps: (1) use `GetObjectClass` to get a reference to the class object of the Java object; (2) use `GetFieldID` to get a field ID for a particular field by providing the field's name and type; (3) use the field ID to retrieve the value of the field in the object.

For instance, the following program first gets `obj`'s field `nd`, which is a reference to another object of class `Node`. It then reads the field `i` of the `Node` object.

```
jclass cls = GetObjectClass(obj);
jfieldID fid = GetFieldID(cls, "nd", "Node");
jobject obj2 = GetField(obj, fid);
jclass cls2 = GetObjectClass(obj2);
jfieldID fid2 = GetFieldID(cls2, "i", "I");
int x1 = GetIntField(obj, fid2);
```

The above steps may not always be performed in consecutive steps; caching field and method IDs for future use is a common optimization. Furthermore, arguments provided to functions such as `GetFieldID` may not always be string constants. For better precision, JATO uses an inter-procedural, context-sensitive static analysis to track constants and infer types of Java references [19]. For the above program, it is able to decide that there is a read access to `obj` and there is a read access to `obj.nd`. To do this, it is necessary to infer what Java class `cls` represents and what field ID `fid` represents.

5 Preliminary evaluation

We performed preliminary evaluation on a set of multithreaded JNI programs. Each program was analyzed to generate a set of constraints, as presented in Sec. 3. Based on

the closure of the generated constraints, a set of objects were identified to ensure atomicity of a native method in these programs. Different locking schemes were evaluated to examine their performance.

All experiments were carried out on an iMac machine running Mac OS X (version 10.7.4) with Intel core i7 CPU of 4 cores clocked at 2.8GHz and with 8GB memory. The version of Java is OpenJDK 7. For each experiment, we took the average among ten runs.

We next summarize the JNI programs we have experimented with. The programs include: (1) a parallel matrix-multiplication (`MM`) program, constructed by ourselves; (2) a Fast-Fourier-Transform (`FFT`) program, adapted from JTransforms [33] by rewriting some Java routines in C; (3) the `compress` program, which is a module that performs multithreaded file compression provided by the MessAdmin [24] project; (4) the `derby` benchmark program is selected from SPECjvm2008 [30] and is a database program. Both `compress` and `derby` are pure Java programs, but they invoke standard Java classes in `java.io` and `java.util.zip`, which contain native methods.

The analysis time and LOC for both Java side and C side on each program are listed below. It is observed that majority of the time is spent on Java side analysis, particularly on Java-side constraint-generation.

Program	LOC (Java)	Time (Java)	LOC (C)	Time (C)
`MM`	275	3.34s	150	10μs
`FFT`	6,654	8.14s	3,169	0.01s
`compress`	3,197	27.8s	5,402	0.05s
`derby`	919,493	81.04s	5,402	0.05s

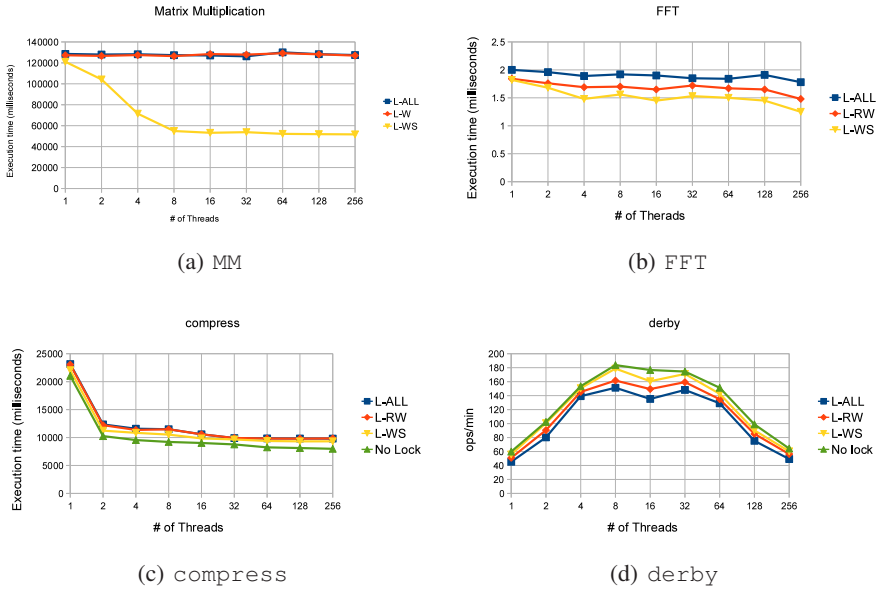
All programs are benchmarked under the three strategies we described in the previous section. L-ALL stands for the lock-all approach. L-W stands for the lock-on-write approach. L-WS stands for the case after applying the lock-on-write and lock-at-write-site optimizations.

Matrix multiplication. The programs takes in two input matrices, calculates the multiplication of the two and writes the result in an output matrix. It launches multiple threads and each thread is responsible for calculating the result of one element of the output matrix. The calculation of one element is through a native method. In this program, three two-dimensional arrays of `double` crosses the boundary from Java to the native code.

For this program, JATO identifies that the native method accesses the three cross-boundary objects. These objects are shared among threads. The input matrices and their arrays are read-accessed whereas the resulting matrix and its array are read- and write-accessed.

Fig. 5(a) presents the execution times of applying different locking schemes. The size of the matrices is 500 by 500 with array elements ranging between 0.0 and 1000.0. L-WS has the best performance overall.

`FFT`. The native method of this program takes in an array of `double` to be transformed and sets the transformed result in an output array. The input array is read-accessed whereas the output array is write-accessed. The arrays are shared among



(a) `MM` (b) `FFT`

(c) `compress` (d) `derby`

Fig. 5. Execution time of the benchmark programs under different locking schemes.

threads. Fig. 5(b) shows the results of `FFT`. Similar to the program of matrix multiplication, L-W improved upon L-ALL, and L-WS performs the best among the three.

`compress`. This program compresses an input file by dividing the file into smaller blocks and assigning one block to one thread for compression. The actual compression is performed in the native side using the `zlib` C library. JATO identifies that a number of objects such as `Deflater` are shared among threads and read/write accessed at the Java side. One exception is `FileInputStream`, where it is only read-accessed in Java but is write-accessed at the native side. In term of the number of locks inserted, there is little difference between lock-all and lock-on-write.

Fig. 5(c) presents the results of `compress`. The file size is about 700MB and the block size is 128K. The performance gain of L-W over L-ALL is negligible. We see there is some minor improvement using L-WS. This is because in the native code, write-access code regions to the locked objects are typically small.

`derby`. It is a multithreaded database. Some `byte` arrays and `FileInputStream` objects are passed into the native code. They are read-accessed between threads from the Java side. On the native side, both kinds of objects are write-accessed.

Fig. 5(d) shows the result of running derby. The experiment was run for 240 seconds with 60 seconds warm-up time. The peak ops/min occurs when the number of threads is between 8 to 32. We can see that in L-WS approach, the performance gains at its peak is about 35% over L-ALL.

For `compress` and `derby`, we also experimented with the no-lock scheme in which no locking is inserted in native methods. Although the uninstrumented programs

run successfully, there is no guarantee of native-method atomicity as provided by JATO. The programs of matrix multiplication and `FFT` would generate wrong results when no locks were inserted for native-method atomicity. For the matrix-multiplication program, even though the native method of each thread calculates and updates only one element of the output matrix, it is necessary to acquire the lock of the output matrix before operating on it: native methods use JNI function `GetArrayElements` to get a pointer to the output matrix and `GetArrayElements` may copy the matrix and return a pointer to the copy [20].

6 Related Work

The benefits of static reasoning of atomicity in programming languages were demonstrated by Flanagan and Qadeer [6] through a type effect system. Since then, many static systems have been designed to automatically insert locks to enforce atomicity: some are type-based [23, 15]; some are based on points-to graphs [11]; some reduce the problem to an ILP optimization problem [5]. Among them, JATO's approach is more related to [15]. Unlike that approach where the focus is on the interaction between language design and static analysis, JATO focuses on static analysis in a mixed language setting.

Atomicity can either be implemented via locks (*e.g.*, the related work above) or by transactional memory (TM) [10]. Related to our work are two concepts articulated in TM research: *weak atomicity* and *strong atomicity* [22]. In a system that supports weak atomicity, the execution of an atomic program fragment exhibits serial behaviors *only* when interleaving with that of other atomic program fragments; there is no guarantee when the former interleaves with *arbitrary* executions. To support the latter, *i.e.*, strong atomicity, has been a design goal of many later systems (*e.g.*, [4]). Most existing strong atomicity algorithms would disallow native methods to be invoked within atomic regions, an unrealistic assumption considering a significant number of Java libraries are written in native code for example. Should they allow for native methods but ignore their impact these approaches would revert back to what they were aimed at solving: weak atomicity.

In a software transactional memory setting where the atomicity region is defined as atomic blocks, *external actions* [9] are proposed as a language abstraction to allow code running within an atomic block to request that a given pre-registered operation (such as native method invocation) be executed outside the block. In the "atomicity-by-default" language AME [1], a `protected` block construct is introduced to allow the code within the block to opt out of the atomicity region. Native methods are cited as a motivation for this construct. Overall, these solutions focus on how to faithfully model the non-atomicity of native methods, not how to support their atomicity.

This work belongs to the general category of improving upon FFIs' safety, reliability, and security. FFI-based software is often error-prone; recent studies found a large number of software bugs in the interface code between modules of different languages based on static analysis [7, 8, 32, 14, 18, 19] and dynamic analysis [31, 16], and new interface languages for writing safer multilingual code (*e.g.*, [12]). JATO performs interlanguage analysis and lock insertion to ensure atomicity of native methods in JNI code. We are not aware of other work that addresses concurrency issues in FFI code.

7 Conclusion and Future Work

JATO is a system that enforces atomicity of native methods in multi-threaded JNI programs. Atomicity enforcement algorithms are generalized to programs developed in multiple languages by using an inter-language, constraint-based system. JATO takes care to enforce a small number of locks for efficiency.

As future work, we will investigate how to ensure locking inserted by JATO does not cause deadlocks (even though we didn't encounter such cases yet during our experiment), probably using the approach of a global lock order as in Autolocker [23]. Moreover, we believe that JATO's approach can be generalized to other FFIs such as the OCaml/C interface [17] and the Python/C interface [28].

Acknowledgements

The authors would like to thank Haitao Steve Zhu for his assistance with running experiments in Cypress. The authors would also like to thank the anonymous reviewers for their thorough and valuable comments. This research is supported by US NSF grants CCF-0915157, CCF-151149211, a research award from Google, and in part by National Natural Science Foundation of China grant 61170051.

References

1. M. Abadi, A. Birrell, T. Harris, and M. Isard. Semantics of transactional memory and automatic mutual exclusion. In *POPL '08*, pages 63–74, 2008.
2. E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: safe multithreaded programming for c/c++. In *OOPSLA '09*, pages 81–96, 2009.
3. R. L. Bocchino, Jr., S. Heumann, N. Honarmand, S. V. Adve, V. S. Adve, A. Welc, and T. Shpeisman. Safe nondeterminism in a deterministic-by-default parallel language. In *POPL '11*, pages 535–548, 2011.
4. B. CarlStrom, A. McDonald, H. Chafi, J. Chung, C. Minh, C. Kozyrakis, and K. Olukotun. The atomos transactional programming language. In *PLDI'06*, June 2006.
5. M. Emmi, J. S. Fischer, R. Jhala, and R. Majumdar. Lock allocation. In *POPL '07*, pages 291–296, 2007.
6. C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *PLDI'03*, pages 338–349, 2003.
7. M. Furr and J. Foster. Checking type safety of foreign function calls. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 62–72, 2005.
8. M. Furr and J. Foster. Polymorphic type inference for the JNI. In *15th European Symposium on Programming (ESOP)*, pages 309–324, 2006.
9. T. Harris. Exceptions and side-effects in atomic blocks. *Sci. Comput. Program.*, 58(3):325–343, Dec. 2005.
10. T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA'03*, pages 388–402, 2003.
11. M. Hicks, J. S. Foster, and P. Prattikakis. Lock inference for atomic sections. In *TRANSACT'06*, June 2006.
12. M. Hirzel and R. Grimm. Jeannie: Granting Java Native Interface developers their wishes. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 19–38, 2007.

13. A. Igarashi, B. Pierce, and P. Wadler. Featherweight java - a minimal core calculus for java and gj. In *ACM Transactions on Programming Languages and Systems*, pages 132–146, 1999.
14. G. Kondoh and T. Onodera. Finding bugs in Java Native Interface programs. In *ISSTA '08: Proceedings of the 2008 International Symposium on Software Testing and Analysis*, pages 109–118, New York, NY, USA, 2008. ACM.
15. A. Kulkarni, Y. D. Liu, and S. F. Smith. Task types for pervasive atomicity. In *OOPSLA '10*, October 2010.
16. B. Lee, M. Hirzel, R. Grimm, B. Wiedermann, and K. S. McKinley. Jinn: Synthesizing a dynamic bug detector for foreign language interfaces. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 36–49, 2010.
17. X. Leroy. *The Objective Caml system*, 2008. `http://caml.inria.fr/pub/docs/manual-ocaml/index.html`.
18. S. Li and G. Tan. Finding bugs in exceptional situations of JNI programs. In *16th ACM Conference on Computer and Communications Security (CCS)*, pages 442–452, 2009.
19. S. Li and G. Tan. JET: Exception checking in the Java Native Interface. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 345–358, 2011.
20. S. Liang. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley Longman Publishing Co., Inc., 1999.
21. Y. D. Liu, X. Lu, and S. F. Smith. Coqa: Concurrent objects with quantized atomicity. In *CC'08: International Conference on Compiler Construction*, March 2008.
22. M. M. K. Martin, C. Blundell, and E. Lewis. Subtleties of transactional memory atomicity semantics. *Computer Architecture Letters*, 5(2), 2006.
23. B. McCloskey, F. Zhou, D. Gay, and E. Brewer. Autolocker: synchronization inference for atomic sections. In *POPL'06*, pages 346–358, 2006.
24. messAdmin. `http://messadmin.sourceforge.net/`.
25. A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41, 2005.
26. G. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *International Conference on Compiler Construction (CC)*, pages 213–228, 2002.
27. J. Palsberg and M. I. Schwartzbach. Object-oriented type inference. In *OOPSLA '91*, pages 146–161, 1991.
28. Python/C API reference manual. `http://docs.python.org/c-api/index.html`, Apr. 2009.
29. O. Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, May 1991. CMU-CS-91-145.
30. SPECjvm2008. `http://www.spec.org/jvm2008/`.
31. G. Tan, A. Appel, S. Chakradhar, A. Raghunathan, S. Ravi, and D. Wang. Safe Java Native Interface. In *Proceedings of IEEE International Symposium on Secure Software Engineering*, pages 97–106, 2006.
32. G. Tan and G. Morrisett. ILEA: Inter-language analysis across Java and C. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 39–56, 2007.
33. P. Wendykier and J. G. Nagy. Parallel colt: A high-performance java library for scientific computing and image processing. *ACM Trans. Math. Softw.*, 37(3):31:1–31:22, Sept. 2010.
34. J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI '04*, pages 131–144, 2004.
35. H. S. Zhu and Y. D. Liu. Scalable object locality analysis with cypress principle. Technical report, SUNY Binghamton, May 2012.