Ribbons: a Partially Shared Memory Programming Model

Kevin Hoffman  Harrison Metzger  Patrick Eugster
Department of Computer Science, Purdue University
{khoffma,hmetzger,peugster}@cs.purdue.edu

Abstract

The need for programs to execute subcomponents in isolation from each other or with lower privileges is prevalent among today’s systems. We introduce ribbons: a shared memory programming model that allows for more implicit sharing of memory than processes but is more restrictive than threads. Ribbons structure the heap into protection domains. Privileges between these protection domains are carefully controlled in order to confine computation. We propose RIBBONJ, a backwards-compatible extension of Java, to easily create or port programs to use the ribbons model. We study the progress and isolation properties of a subset of the language. Building on JikesRVM we implement ribbons by leveraging existing memory protection mechanisms in modern hardware and operating systems, avoiding the overhead of inline security checks and read or write barriers. We evaluate efficiency via microbenchmarks and the DaCapo suite, observing minor overhead. Additionally, we refactor Apache Tomcat to use ribbons for application isolation, discuss the refactoring’s design and complexity, and evaluate performance using the SPECweb2009 benchmark.

1. Introduction

Today’s systems need the ability to execute subcomponents with lower privileges or in (partial) isolation from each other. Web servers, application servers, and web browsers allow for embedded execution of third-party subcomponents such as plugins, which are often targets of exploitation.

Current isolation mechanisms typically apply the following techniques: (1) executing code within a separate process and interacting with it via inter-process communication (IPC), (2) relying on a reference monitor to mediate all accesses from the isolated code to sensitive resources, and (3) static isolation of classes such as through multiple Java class loaders. The first technique tends to increase complexity and introduce significant performance overhead when the interaction between the components is non-trivial. For example, the Multitasking Virtual Machine (MVM) [14] uses the doors IPC technique in Solaris to implement native code isolation and observes a 397× slowdown on Java Native Interface (JNI) calls (an increase from 0.136µs to 54µs). The second technique tends to be application-specific, hampering re-use, and can be expensive when applied to heap objects (e.g., 2× to 8× slowdown [20]). The third technique ties isolation boundaries to lexical scope, precluding dynamic isolation boundaries, and depends on application-specific mechanisms to protect access to data in globally shared classes.

Consider the case of Apache Tomcat, an open-source implementation of the Java Servlet and Java Server Pages (JSP) technologies. Tomcat allows multiple web applications and related components to run in isolation from other each other within the same Java Virtual Machine (JVM). Executing components (servlets, JSPs, tag libraries, etc.) are isolated so that they can not interfere either with the operation of the main Tomcat server nor with other running components. Tomcat enforces isolation through two mechanisms: (a) a separate class loader for each web application, and (b) the Java Security Manager [16], an access monitor enforcing isolation boundaries upon JVM properties, files and directories, networking, and reflection. Using separate class loaders prevents a web application from accessing Tomcat’s internals or other applications’ state, only allowing applications to interact with the host server code through certain classes loaded by the global class loader. Class loaders achieve this isolation by copying loaded classes and giving them names which are logically distinct across loaders. The normal type-safety mechanisms in the JVM thus enforce isolation between uses of a class loaded by different class loaders.

This mechanism, however, is subject to vulnerabilities through bugs in classes loaded by Tomcat’s global class loader (CVE-2009-0783 [1]). The bug allows a web application to escape isolation by replacing Tomcat’s XML parser at runtime, allowing a malicious application to read sensitive files of other applications (e.g., containing passwords). The complexity of Tomcat’s ad-hoc isolation mechanisms makes it hard to verify that any given version is 100% correct and remains so as the code evolves.

Another approach is to run each isolated component in a separate VM or process. This strategy is used by the Google Chrome and Firefox 4 web browsers, which execute tabs
and plugins in distinct processes. These techniques induce runtime overhead due to increased memory usage (e.g., extra copies of objects) and IPC between tabs and the main process (e.g., remote method invocations if using something like Isolates [14][24]). IPC also increases code complexity. Specific solutions such as browser isolation mechanisms [9] cannot easily be applied to other cases such as application servers, or even related plugin architectures such as integrated development environments (e.g., Eclipse).

We propose ribbons to allow the heaps of subcomponents within a process to be fully or partially isolated in well-structured ways, without significant runtime overhead. The heap is divided into an arbitrary and dynamic number of protection domains. Access privileges are tracked pair-wise between protection domains, and threads are grouped into ribbons. Ribbons subsume threads, and protect against inadvertent or malicious access to data and mitigate unbounded heap corruption in unsafe runtime environments. Enforcement during execution relies on standard hardware-level virtual memory and memory protection mechanisms, avoiding per-instruction overheads.

This paper makes the following contributions:

• We define the ribbons programming model.
• We introduce RIBBONJ, a Java language extension, for writing programs using the ribbons model.
• We present RIBBONJ-LITE, a formalism for RIBBONJ’s core, to study progress and isolation properties.
• We evaluate our implementation in JikesRVM, using microbenchmarks and the DaCapo benchmarks [5], with results showing minor runtime overhead (0% to 12%).
• We discuss experiences applying ribbons to small-scale applications, including lusearch and jIRCd.
• We present a case study wherein we refactor Apache Tomcat to isolate web applications using ribbons. This “ribbonization” only required 680 new lines of code. Results indicate a moderate latency increase without throughput decrease.

Some applications may require more than just heap isolation for safeguarding the execution of sub-components (e.g., if sub-components need to execute with different OS-level privileges). For example, web browsers may desire to execute plugins with highly restricted access to the local filesystem. Additionally, using distinct processes for isolation can provide “kill-safety” [21], if designed correctly.

This paper focuses solely on isolation of the heap and does not claim to provide complete isolation of sub-components or kill-safety. Full isolation and kill-safety would require additional support for ribbons within the OS kernel. In this paper, we lay a foundation for such future work by defining the ribbons programming model and language (including a formal model), and by implementing and evaluating the mechanisms necessary for heap isolation. We also demonstrate the immediate value of efficient heap-based isolation mechanisms through the Apache Tomcat case study.

Roadmap. Section 2 details the ribbon model and its concepts in RIBBONJ. We formalize a subset of RIBBONJ in Section 3 to study isolation and progress properties. Section 4 discusses our implementation of RIBBONJ in JikesRVM, and Section 5 evaluates it. Section 6 presents our ribbonization of Tomcat. Section 7 discusses related work. Section 8 presents conclusions.

2. Protection Domains, Ribbons, and RIBBONJ

In this section we first overview the ribbon memory programming model, and then describe RIBBONJ, a backwards-compatible extension of Java that provides the features of the ribbon model.

2.1 Ribbon Memory Programming Model

The ribbons model defines three abstractions: protection domains, domain sharing privileges, and actual ribbons. For illustrative purposes, Figure 1 visualizes how these concepts could be applied to provide heap isolation for components within a web browser. Protection domains are the basic unit from which memory sharing constraints are formed. All memory allocations are associated with some protection domain (implicitly or explicitly), such that each area of memory allocated (each object in object-oriented languages) is placed “within” that domain. In the browser example, the global private and shared states represent two domains.
Every thread in a program is associated with one or more protection domains, which determines its set of privileges as follows: Domain sharing privileges specify the access that threads executing within the context of one protection domain (termed the source domain) have on memory contained within some other protection domain (target domain). A privilege is composed of a source domain identifier SD, a target domain identifier TD, and any number of the following permissions:

**read**: Domain SD can read memory in domain TD.

**write**: Domain SD can write memory in domain TD.

**inject**: Domain SD can allocate memory in domain TD.

As an example, the browser global shared state domain can not write into the global private state. Scripts can only access their tabs.

Finally, a ribbon is a set of protection domains that models the assignment of threads to sets of protection domains. Creating a new ribbon also creates a new thread that is associated with a new set of existing protection domains (programmatically specified). This thread and all threads that it spawns execute within the context of these protection domains. Once created, the set of domain sharing privileges for a ribbon cannot be changed. The privileges acquired by a ribbon upon creation are upper bounded by the privileges of the ribbon of the creating thread. This ensures that once sharing constraints are placed upon a newly created ribbon, all future computation resulting directly or indirectly from that ribbon will never violate the sharing constraints.

In summary, a process contains one or more ribbons, and a ribbon contains one or more threads. Threads spawned within a ribbon share the ribbon's set of protection domains. Upon process creation, a new ribbon is created, and thereby a new thread. Programs that are unaware of ribbons operate as before, as they are contained within a single ribbon, and all threads share the same protection domain.

### 2.2 RIBBONJ

We introduce an extension of Java called RIBBONJ, implementing the ribbon memory programming model. Modeling ribbon concepts at the language level offers improved programmability through type system support and syntactic sugar. Additionally, a carefully designed language supporting ribbons can help the compiler understand information needed for optimization and automatic parallelization (such as aliasing).

The `protectiondomain` keyword is used to define new protection domain types. These types represent entire "classes" of protection domains that may be instantiated at runtime. The type optionally accepts as domain arguments other typed protection domains. The `protectiondomain` definition contains a list of rules that affect privileges when a new domain of that type is created: `grant` adds a new privilege where the source domain is explicitly indicated, and the target domain is the new protection domain; `require` adds a new privilege where the target domain is explicitly indicated, and the source domain is the new protection domain. The list of rules is abstract and is just a policy until an actual protection domain is created at runtime and concrete protection domains (instead of abstract types) are bound to the domain arguments.

Note that `require` is needed in addition to `grant` because privilege specifications are used only when a new protection domain is created, and a new protection domain cannot grant a privilege to a protection domain that has not yet been created. In the web browser example, the browser global shared state domain cannot grant privileges to domains for tabs, because the tab domains do not exist when the global shared state domain is created; rather, the tab domains must require certain privileges from the global shared state domain when a tab domain is created.

Ribbons are modeled in the language by special classes that implicitly inherit from `java.lang.ThreadGroup` and implement the `Ribbon` interface. The `ribbon` keyword declares new ribbon types. The declaration includes a list of named protection domains, termed the `domain arguments`, which represent the protection domains with which threads within the ribbon will be associated. The `primary protection domain` designates the first in this list. The actual domain argument values are not specified until the ribbon is instantiated. The `getdomain` operator is provided to retrieve the protection domain value of a given name for a given ribbon object. `ribbon` classes may optionally extend others, and are instantiated using the normal `new` keyword and constructor call, followed by a `where` clause to specify the domain argument values for the new ribbon. Once instantiated, a ribbon begins execution via the `start` method exposed through the `Ribbon` interface, allowing the application to indicate the `java.lang.Thread` to use as the first thread.

The `new` operator is also extended so that a new object can be allocated within a specific protection domain. If the extended new operator is not used, then the new object is allocated within the primary protection domain of the ribbon associated with the currently executing thread. Finally, the `thisribbon` keyword is added for convenience to get the object representing the ribbon associated with the currently executing thread.

### 2.3 RIBBONJ Example

We give the specifics of the RIBBONJ syntax through another example to broaden the illustration. The example, depicted in Figure 2, represents a master/worker scenario, as it can model application servers. Each master may spawn multiple workers, yet:
Masters may not read/write/inject other masters.
Masters may read data in workers that are spawned by that master but not by other masters.
Workers may not read/write/inject other workers.
Workers may only receive work from their respective masters.

Three “classes” of protection domains are required to model this: MasterDomain (stores information needed to produce items), WorkerDomain (stores information needed to process items), and QueueDomain (stores produced items ready for consumption). Note that multiple masters may be created (e.g., one master for HTTP requests and another for HTTPS). One must distinguish between the protection domains of these masters. This is effectively modeled within RIBBONJ since protection domains are modeled as types that are used as “templates” and not fully instantiated until runtime. Protection domain types structure runtime protection domains, providing rich information for static analyses and compiler optimizations. Below are the protection domain types:

```java
definition ProtectionDomain
    read from (owner)
    write from (q)
    grant {read, write, inject} to {owner};
```

Note how the MasterDomain requires read access to some generic owner domain. Also, note that MasterDomain needs inject access to the QueueDomain to allocate new objects inside that domain, while WorkerDomain only requires write access to remove objects from the worker queue for processing. Also of interest is that a Master needs to be able to read data in Worker domains that it creates. This cannot be specified in the MasterDomain, because the Worker domains do not exist when the Master is created. The grant statement allows a Worker, when it is created, to add to the privileges of the MasterDomain passed to the Worker instantiation. The protection domain types defined above can then be used to instantiate ribbons that operate within them:

```java
ribbon Worker(WorkerDomain wdom)
    static class WorkerThread extends Thread
        public void run()
    
ribbon Master(MasterDomain mdom)
    QueueDomain qdom = mdom.q;
    Queue q = new<qdom> Queue();
    public void createWorker()
        Worker w = new Worker()
        getdomain(thisribbon(), mdom), qdom);
        w.start(new Worker.WorkerThread());

class MasterThread extends java.lang.Thread
    public void produce()
        q.add(new<qdom> WorkItem(...) );
    public void run()
        for (int i=0; i<5; i++)
            createWorker();
```

When a new work item is produced, the Master must allocate it within the QueueDomain so that Worker objects can modify it. This can be done by specifying a concrete protection domain (some object deriving from ProtectionDomain) within angle brackets after the new operator and before the type name.

Also, when a new WorkerDomain is instantiated, first a new WorkerDomain protection domain is created. The constructor for the WorkerDomain requires two domain arguments. Here, they are the protection domain named mdom for the current ribbon, and the qdom protection domain value created earlier. In this way the protection domain “template” is applied to effect the runtime privileges for the new domain. The new WorkerDomain value is used to create a new Worker ribbon. The newly instantiated ribbon is then told to begin execution, using a new WorkerThread thread class as the first thread of execution within the newly executing ribbon. Note that the WorkerThread and MasterThread classes need not have been declared as inner classes of their respective ribbons, but were done so for convenience.
3. RIBBONJ-LITE

To study the isolation properties of RIBBONJ we formalize its core. The resulting syntax and semantics follow in the spirit of other object calculi, e.g., Featherweight Java (FJ) [26], Classic Java [22].

3.1 Syntax and Definitions

The syntax of our core language, RIBBONJ-LITE, is presented below. RIBBONJ-LITE can be viewed as FJ without subclasses (and thus casts) for simplicity, but augmented with: protection domain types \((D)\) and protection domain values \((\pi)\), ribbon types \((R)\) and ribbon values \((r(R))\), location values \(l(\pi,C)\), field assignments \((t,f=t)\), sequences of terms \(\langle t, f\rangle\), threads \(\langle newthread, t\rangle\), and ribbons \(newribbon(R)\).

Protection domain values model first class heaps, and consist of a typed identifier denoting a concrete instance of the protection domain (rather than actually containing the values within the protection domain). Protection domain values are typed, and corresponding declarations instantiated to yield new protection domains (heaps) at runtime. The protection domain types indicate which permissions to grant to or require from existing protection domains when creating a new domain. A global protection domain that always exists is identified as \(\pi_\bot\), which has a protection domain type of \(PD\). Ribbon values model the set of protection domains that should be “active” when executing terms in threads belonging to the ribbon. Ribbon values consist of a sequence storing the values of the named protection domains for that ribbon. Location values \(l(\pi,C)\) uniquely identify how to retrieve an object from the object store. \(C\) denotes the type of the object, while \(\pi\) denotes the protection domain value within which the object is contained. Locations may be written simply \(l\) for brevity when the containing protection domain and type \(C\) are not germane to the context.

In RIBBONJ-LITE we chose not to overload the \texttt{new} operator for readability, but instead introduced different operators to create new ribbons, protection domains, and objects. The \texttt{newribbon} operator creates new protection domains and:

\[
\begin{align*}
\text{fields(thread)} &= \emptyset \\
\text{protectiondomain}(D_{(\bot)}) &= \begin{cases} \\
\end{cases} \\
\text{Z}(A,x) ; \in \mathcal{T} \\
\text{ribbon}(R) &= \begin{cases} \\
\end{cases} \\
\\text{\texttt{grant}(x)} &= \texttt{grant}(\pi) \\
\\texttt{require(x)} &= \texttt{require}(\pi) \\
\end{align*}
\]

Figure 3. Auxiliary definitions.

accepts concrete protection domain values referenced within the relevant protection domain type. \texttt{newribbon} creates new ribbons and accepts the protection domain values within which the new ribbon should operate. The first protection domain value passed to the ribbon constructor is also defined as the default protection domain for the new ribbon. In addition to creating a new memory execution context the new ribbon also implicitly creates a new thread within the new ribbon’s memory context. \texttt{thisribbon} provides access to the ribbon value of the ribbon in which the current thread is executing. \texttt{getdomain} allows retrieval of one of the named protection domain values associated with a given ribbon value (or the default protection domain value). Thread creation is augmented to indicate the ribbon value of the ribbon in which the new thread should execute. Object creation is augmented to accept the protection domain value in which the new object should be created. \texttt{new C(\top)} is syntactic sugar for \texttt{new \langle getdomain(thisribbon, default) \rangle C(\top)}.

3.2 Dynamic Semantics

Figure 4 defines an operational semantics for RIBBONJ-LITE. Global evaluation is of the form \(\langle Q, E, P, \mathcal{R} \rangle \Rightarrow \langle Q', E', P', \mathcal{R}' \rangle\) where \(Q\) is a parallel composition of executing threads \(\langle \tau(-) \rangle\), \(E\) is an object store, \(P\) is the privilege set, and \(\mathcal{R}\) is the ribbon thread context — an ordered set of ribbon values storing the current ribbon value for each active thread. The privilege set tracks permissions for source protection domains \(\pi_{SD}\) on target
domains $\pi_{TD}$ using tuples $(\pi_{SD}, \pi_{TD}, \text{privilege})$. Congruence on global evaluation (CONGR-E) relies on evaluation contexts $E$ defined below:

$$E ::= [] | E.f | E.f.t | v.f.E | E.m(T) | v.m(E) | \pi, E, \tilde{\tau} \mid \text{newribbon } R(E) \mid \text{newribbon } R(v_{1}) (E_{1}) \mid \text{new<}E\text{> }C(E) \mid \text{newthread }<E> (T_{1}) \mid \text{getdomain } (E, x) \mid \text{new<}\pi\text{> }C(E) \mid \text{newdomain } D(E) \mid \text{newthread }<v_{1}> (E_{1})$$

Rules for local evaluation $\langle t, E, P, R \rangle \rightarrow \langle t', E', P', R \rangle$ only include one ribbon value, $R$, corresponding to the ribbon value associated with the thread of the single thread term being reduced. Note that local evaluation never changes $R$.

Rule FORK-E creates a new thread of control operating within the context of the specified ribbon. The ribbon thread context $R=\text{newthread } (R_{1})$ is appended with the given ribbon value, remembering for each thread the ribbon within which the thread is operating. The newthread expression on the creating thread of execution is replaced with an expression to create a new Thread object representing the newly created thread. For brevity, but without loss of validity, we omit a rule for cleaning up completed threads. Rule CONS-E creates a new object within a specified protection domain, verifying that the current ribbon has the inject privilege on the target domain. The access control occurring within the second line of the rule’s antecedent ensures that at least one of the protection domains associated with the current thread’s ribbon has the inject privilege on the protection domain within which the new object is being created. Rules FIELD-ACC-E and FIELD-ASS-E allow field access and assignment and checks for the read or write privilege as appropriate. The rule for method calls, METH-E, verifies that the current ribbon has the read privilege on the target object. Rule SEQUENCE-NEXT-E reduces a sequence to the last value after all terms in the sequence have been fully evaluated.

Rule NEW-DOMAIN-E creates a new protection domain value using the given protection domain type as a template. This template is used to add to the privilege set to both (a) grant other protection domains privileges to the new domain (through grant), and (b) grant the new protection domain privileges to other domains (through require). These effects are modeled through buildnewprivs. Rule NEW-RIBBON-E creates a new ribbon value and forks a thread using this value. The new ribbon value’s named protection domain values are initialized using the domain values passed as arguments. Note that the value to which the newribbon term evaluates is the new ribbon value, not the value that the newthread expression evaluates to. Rule THIS-RIBBON-E exposes the value of the current thread term’s ribbon value. Rule GET-DOMAIN-E retrieves a named protection domain value from a given ribbon value.

**Figure 4.** Global evaluation $\langle Q, E, P, R \rangle \rightarrow \langle Q', E', P', R' \rangle$ and local evaluation $\langle t, E, P, R \rangle \rightarrow \langle t', E', P', R' \rangle$.

### 3.3 Progress

In the absence of casts, the original progress theorem of FIJ is simplified, notwithstanding our additions of locations (since locations explicitly include type information). We define stuck terms to account for the cases where a protection domain privilege check fails.
Theorem 1 (Local Progress). Suppose \( t \) is a closed, well-typed normal form under \( \rightarrow \) and \( t \) is within \( \langle t, \mathcal{E}, \mathcal{P}, \mathcal{R} \rangle \). Then either \( t \) is a value, or \( \exists \mathcal{E} \) such that either of the following holds:

1. \( t = E[\mathsf{new} \leftarrow \pi \cdot C(\pi)] \land \mathbin{\not\in} \left( x_i: \pi_i \right) \in \mathcal{R} \langle \pi_i, \pi, \text{inject} \rangle \in \mathcal{P} \)
2. \( t = E[l(\pi, C) \cdot f_i] \land \mathbin{\not\in} \left( x_i: \pi_i \right) \in \mathcal{R} \langle \pi_i, \pi, \text{read} \rangle \in \mathcal{P} \)
3. \( t = E[l(\pi, C) \cdot f_i \cdot u'] \land \mathbin{\not\in} \left( x_i: \pi_i \right) \in \mathcal{R} \langle \pi_i, \pi, \text{write} \rangle \in \mathcal{P} \)
4. \( t = E[l(\pi, C) \cdot m(\pi)] \land \mathbin{\not\in} \left( x_i: \pi_i \right) \in \mathcal{R} \langle \pi_i, \pi, \text{read} \rangle \in \mathcal{P} \)

Proof. Straightforward by induction on derivation of \( \rightarrow \).

Theorem 2 (Global Progress). Suppose \( Q \) is a closed, well-typed normal form under \( \Rightarrow \). Then either of the following holds:

1. \( Q \) is a parallel composition of non-value terms where each term matches one of the non-value normal forms in Theorem 1.
2. \( Q \) is a parallel composition of zero executing threads.

Proof. Proof by contradiction. There are two cases to consider:

1. Suppose \( Q \) is in normal form and is a parallel composition of thread terms, and at least one of the terms is not a value and is not one of the stuck terms listed in Theorem 1. If this term is a \texttt{newthread} term, then because \( Q \) is well typed and because of type preservation, the rule \texttt{Fork-E} is applicable and progress is made. If this term is not a \texttt{newthread} term then by Theorem 1 this non-value term can be further evaluated under \( \rightarrow \), and thus global progress can be made under \texttt{Congr-E}. Thus, in either case, \( Q \) cannot be in normal form.
2. Suppose \( Q \) is in normal form and is a parallel composition of thread terms, and at least one of the terms is a value. For such a term, the rule \texttt{End-E} is applicable and progress is made. Thus, \( Q \) cannot be in normal form.

3.4 Isolation

With the \texttt{RibbonJ-Lite} semantics above, a new protection domain can grant or require privileges to or from any other domain. While this structures the heap and prevents inadvertent accesses, it is not strict enough to fully isolate or sandbox code executing in ribbons. We provide two extensions for different levels of isolation, ending with an isolation model matching that of \texttt{RibbonJ}.

### 3.4.1 Cooperative Isolation

Under this first model, a new protection domain \( \mathcal{P} \) cannot require a privilege from another protection domain \( \mathcal{Q} \) unless at least one of the protection domains in the creator’s ribbon already has the privilege on domain \( \mathcal{Q} \). Thus, the new protection domain is unable to increase the scope of what the creator’s ribbon is able to access, but the new domain is able to grant privileges to other domains which these did not have on the creator’s ribbon. The creating ribbon thereby controls which protection domains have access to the new domain via the protection domain values it passes to the domain being constructed. While not suitable for sandboxing potentially malicious or insecure components, this isolation level does allow for more flexibility in that a ribbon can create a new protection domain in order to communicate, indirectly, with domains which with it currently does not have privileges to communicate.

The semantics of Figure 4 are retained, but we replace the \texttt{Build-Require-Privs} rule of Figure 4 as follows:

\[
(Z, A, x) \in \text{domprivs}(D) \quad \mathcal{Z} = \text{require} \quad (x: \pi_x) \in \Pi \quad \mathcal{R} = r(R) \quad \exists (x_i: \pi_i) \in r(R) \langle \pi_i, \pi_x, A \rangle \in \mathcal{P} \quad \langle \pi_x, \pi, A \rangle \in \text{buildnewprivs}(\mathcal{P}, \mathcal{R}, \Pi, \pi(D))
\]

This rule has an additional check in the antecedent, ensuring that the creating ribbon already has the privilege that the new protection domain is requiring. Observe that rather than creating a new kind of stuck term, progress is still made if a new protection domain tries to require a privilege it is not allowed to receive. Instead of becoming stuck, such a privilege is not added to the new set. Local evaluation could later get stuck if it tries to access a location in a protection domain that it was not allowed to require a privilege to.

### 3.4.2 Full Isolation

In this level the same restrictions apply as in cooperative isolation. Additionally, a new protection domain \( \mathcal{P} \) cannot grant a privilege to another protection domain \( \mathcal{Q} \) unless domain \( \mathcal{Q} \) already has this privilege on one of the domains of the creator’s ribbon. Under this level of isolation, a new protection domain is fully sandboxed within its creating ribbon. This matches the isolation behavior of \texttt{RibbonJ}. The dynamic semantics of this isolation level are the same as with cooperative isolation, except that \texttt{Build-Grant-Privs} is substituted with:

\[
(Z, A, x) \in \text{domprivs}(D) \quad \mathcal{Z} = \text{grant} \quad (x: \pi_x) \in \Pi \quad \mathcal{R} = r(R) \quad \exists (x_i: \pi_i) \in r(R) \langle \pi_i, \pi_x, A \rangle \in \mathcal{P} \quad \langle \pi_x, \pi, A \rangle \in \text{buildnewprivs}(\mathcal{P}, \mathcal{R}, \Pi, \pi(D))
\]

A privilege check is added to the antecedent of the rule to ensure that a privilege can only be granted when appropriate.
as defined above. The effect on progress is similar to that of cooperative isolation—no new stuck terms are added, but rather local evaluation may become stuck if it tries to use a privilege that was not granted when the protection domain was created.

3.4.3 Adaptable Isolation Levels

The isolation level need not be decided once for the whole program, but can be indicated when a new ribbon is created. Only a same or more strict isolation level than the current ribbon’s level would be allowed, and the rules for constructing the modified privilege set would depend on the isolation level.

We outline the required changes to the dynamic semantics to support adaptable isolation levels. First, ribbon values would be augmented to store the isolation levels associated with them. Ribbon creation would be augmented to allow the new isolation level to be specified. A check would be added to the evaluation rule for \texttt{newribbon} such that the isolation level of a new ribbon \( r \) could not be less restrictive than the isolation level of the ribbon value associated with the thread that is creating \( r \). Finally, the different variants of the \texttt{BUILD-GRANT-PRIVS} and \texttt{BUILD-REQUIRE-PRIVS} rules would become applicable based on the isolation level of the new ribbon.

A narrowing extension \cite{25} provides a yet finer granularity by allowing segments of code to execute with fewer privileges than the rest. This is supported by optionally allowing a ribbon value to be specified as the “context” when calling a method.

4. Implementation of Ribbons

The implementation is 3-fold, comprised of the compiler modifications for \texttt{RIBBONJ} within JastAdd \cite{19}, the modifications to JikesRVM \cite{2}, and the system level support. All code and evaluation data is available for download \cite{25}.

4.1 Language Frontend: JastAdd

The parser and bytecode compiler for \texttt{RIBBONJ} are implemented with JastAdd \cite{19}. JastAdd provides declarative mechanisms for modular and composable compiler extensions. We implemented modules for extending the lexer and parser, \texttt{RIBBONJ}-specific type checking, which were then composed with the unmodified modules for the JastAdd Java 5 compiler. Most of the \texttt{RIBBONJ} features were implemented through straightforward AST rewriting mechanisms to translate to a Java API implemented by the VM. Two exceptions were ribbon initialization after construction (\texttt{where} clause) and the extended \texttt{new} operator, which were implemented by modifying bytecode generation in the compiler backend.

If a specific protection domain is given to the extended \texttt{new} operator, bytecode is generated to call a special static method with the protection domain instance immediately before the new call. This special static method is inlined by the JVM to set a thread-local variable indicating the protection domain from which to attempt to allocate for just the next object allocation on that thread. Ribbon initialization bytecode is generated after the constructor call to duplicate the operand and then call an additional initialization method on the ribbon object with the protection domain arguments.

4.2 VM Implementation: JikesRVM

The JikesRVM modifications are built on top of a lower-level C library and interact with most areas relating to memory management and allocation. Briefly, JikesRVM divides up the heap into “spaces” being managed by \texttt{Space}-derived classes. Spaces acquire and release memory in 4MB chunks in coordination with a map of the virtual address space and in accordance with a \texttt{vmRequest}. We added a new type of \texttt{vmRequest} that indicated memory should be obtained from the lower-level memory domain C library and which memory domain to use. Spaces work in coordination with page resource objects, which are responsible for resource accounting and distributing pages within allocated chunks. All spaces and page resources were modified to let the memory domain C library handle page (un)protection and freeing if the associated virtual address was managed by memory domains.

Each protection domain is modeled as a distinct space object. Previously, JikesRVM would assume a static number of spaces would be created at JVM startup, and the number and type depending on the garbage collector (GC) selected. For example; a large object space, an immortal space, a nursery space, a mature space, etc. Our implementation now allows for an arbitrary number of spaces to be created at runtime. Exactly how many spaces would be created for each memory domain is specific to the GC plan selected. In our implementation we modified the basic mark-sweep GC, which involved modifying the resource accounting, tracing, and sweeping phases to become aware of dynamically created spaces. The GC itself operates within a ribbon that has privileges to all memory domains within the JVM. Free pages identified by the GC are freed from their corresponding memory domain as appropriate. Modifying a copying or generational collector is planned, but introduces complexities that are beyond the scope of this paper.

Thread-local data was added to remember if the next object allocation was for a specific protection domain. This thread-local field is reset to null after each allocation. Additionally, each thread remembers the “default” protection domain to use for allocation. Allocation in JikesRVM is modeled using classes derived from \texttt{Allocator} which are permanently attached to a particular \texttt{Space} instance upon al-
A generalization of the clone first thread) by a system call to Linux’s actions (read, write, etc.). JikesRVM interacts with the sys-mapping between a memory domain and ribbon to a set of memory, along with the mmap a user-space prototype possible. We employ POSIX shared nel, the rich functionality of POSIX and Linux APIs make While the system level support ultimately belongs in the ker-

4.3 Operating System (OS) Support: Linux

While the system level support ultimately belongs in the kernel, the rich functionality of POSIX and Linux APIs make a user-space prototype possible. We employ POSIX shared memory, along with the mmap, mprotect, and clone system calls. Conceptually, the implementation consists of three concepts. A memory domain, which corresponds to an instantiated protection domain, is a set of virtual memory pages. A ribbon is a container of threads. A privilege is a mapping between a memory domain and ribbon to a set of actions (read, write, etc.). JikesRVM interacts with the system via a strict API, such that future work could swap this implementation for one in the kernel without modification.

A ribbon in our implementation represents a separate and distinct virtual memory space. It is created (along with its first thread) by a system call to Linux’s clone. The calling thread is cloned; however, a special flag is passed to indicate that the new clone should not share the same virtual address space mappings but will share all other properties (e.g., shared file table) of a new thread. In this way, a rib-

As a memory domain grows (or shrinks) it allocates (or releases) pages from the reserved memory area claimed at startup. Accesses to this area are serialized such that no two memory domains could allocate the same page, as this would become problematic should a single ribbon need to access both memory domains simultaneously mapped into it. During execution, a memory domain could grow (by claiming pages from the reserved area) because a ribbon attempted to allocate an object inside it. These additional pages will subsequently be mapped into the ribbon such that the actual hardware permits access to them. However, another ribbon could also have access privileges to this memory do-

Implementing ribbons within JikesRVM required the merging of JikesRVM’s custom SIGSEGV handler with the memory domain SIGSEGV handler. Additionally, thread startup and bootstrap was modified to understand the ribbon concept, with a new “ribbon initialization” mode for thread creation that uses the ribbon C library instead of pthreads. A support class for the language frontend is also provided that tracks the privileges between protection domains instantiated at runtime and then upon ribbon creation applies the proper isolation checks (as described in Section 3.4) while setting up the memory domain privileges for the new ribbon.

The size of this area is configurable, but in our implementation it far ex-
ceeds the system’s RAM. However, because of Linux’s on-demand paging and memory over-commit, pages in this area from the OS’s perspective remain unused until a write attempt has been made to it.

2 A generalization of the fork system call used to spawn new threads.
domains as this is better addressed in the kernel. Possible solutions such as sending signals to other ribbons could be used. We simply avoid the issue by not shrinking any memory domains. Throughout our evaluation this never caused a problem.

A limitation of the implementation is that isolation only strongly applies to the “JIT-ed” Java bytecode, which we term as untrusted code. We assume that all native code is inherently trusted. Thus, any untrusted code must not be allowed to invoke native (JNI) method calls that are not known to be trusted (e.g., part of the class library implementation). This can be enforced by either the JIT compiler or by the class loader. Further care is taken to protect the metadata for ribbons, memory domains, and privileges. The pages where this metadata exist are unmapped from all ribbons during normal execution, and only mapped in during certain system level code. This provides some protection against untrusted native code.

With the metadata in user-space, isolation is only ensured for pure Java code compiled by the JIT – natively compiled code can potentially modify this metadata, thus manipulating protection boundaries. In order for isolation to be guaranteed for native code, there must be support in the kernel. This paper focuses on defining the ribbon concept, the RIBONJ semantics, and the JVM implementation issues. The details of kernel support is beyond its scope.

5. Evaluation

In this section we evaluate the performance of our implementation by comparing the performance of the version of JikesRVM that we started (from SVN version 15779) to our custom version. We report on first experiences with “ribbonizing” small-scale programs.

5.1 Methods and Benchmarks

All benchmarks were run on a dedicated system with 6-core AMD Phenom II 1090T CPU with 8 GB DDR3 RAM, running Gentoo Linux 2.6.36 SMP. JikesRVM was configured with the mark-sweep GC with the optimizing compiler. It was also configured to run with a fixed heap size and to use 6 threads for parallel GC (one per core). To avoid variations introduced by the adaptive optimization system (AOS), AOS was disabled and loaded classes were immediately compiled by the fully optimizing compiler on first access.

Three JVMs were evaluated: (1) unmodified JikesRVM (2) “ribbonized” JikesRVM with lazy mapping (RibbonRVM-lazy), (3) ribbonized JikesRVM where all pages are immediately mapped in upon allocation (RibbonRVM-nolazy). Ten benchmark iterations were performed. Each iteration executed all benchmarks, running all 3 JVMs for the same benchmark back to back. Alternating JVMs and benchmarks between iterations minimizes bias due to systematic disturbance. In each benchmark run, two warmup runs of the benchmark were taken (without JVM restarting) to allow all methods to be fully compiled and for the system to reach a stable state before the timed run began. All data points represent the mean over 10 iterations and are given with 99% t-test confidence intervals.

Overhead was evaluated using the DaCapo benchmark suite [5] (version 2006-10-MR[4]) as well as with an implementation of the Master/Worker example from Section[1]. We also ribbonized lusearch to use 32 ribbons instead of 32 threads (lusearch-rib). The Master/Worker benchmark (mw) consists of master threads producing work that is consumed by worker threads. In the benchmark, a synchronized queue is filled with homogenous items (objects). Each item contains an array of long integers. The queue is prepopulated with enough items to fill 32 MB of memory. All master threads shared the same ribbon, but each worker had its own ribbon and protection domain, just as in the example. The masters produce an item by copying the array data into the appropriate worker item in the queue. The workers consume an item by copying the array data out of the queue item and into a work buffer and then summing all of the longs in that item. The benchmark was designed to be memory-bound as opposed to being CPU-bound. On our benchmark system we found maximum performance was achieved with 2 master threads (1 ribbon) and 2 worker threads (2 ribbons), so this configuration was used for all runs. We varied the work item size from 512 bytes to 1 MB to vary the effects of memory bandwidth and synchronization.

5.2 Benchmark Results

Figure[5] presents the wall-clock running times for the benchmarks. No significant overhead is observed for antlr, hsqldb, and xalan. The overhead of other benchmarks is between 2% and 6%. Interestingly, the degree of parallelism does not seem to correlate with observed overhead. The overhead for the parallel benchmarks (mw, hsqldb, lusearch, xalan) varies widely. Additionally, the outlier is jython (single threaded), with an overhead of 10%.

The overhead on the DaCapo benchmarks (except lusearch-rib) cannot be due to additional memory mapping, page faults, or context switching caused by ribbonization, because these benchmarks are still running within the context of a single ribbon. For DaCapo performance overheads are caused more by the additional complexities introduced into the fast allocation path. Currently the RIBONJ compiler uses a thread-local field (in the MutatorContext instances) to store which protection domain a new object should be allocated within (note that this field is only set

---

if a non-default domain is used for allocation). The fast allocation path must retrieve this additional field and also reset it after the allocation. There is also an additional conditional to test if a non-default protection domain needs to be used for allocation. To further substantiate the above theory, we removed the thread-local field accesses from the fast allocation path and the overhead on jython lowered to 6%. However, when we instead removed the extra conditional, performance on jython did not improve.

While DaCapo primarily measures overhead in the context of a single ribbon, mw is a better test for overhead caused by having multiple ribbons. The mw benchmark avoids object allocation almost entirely during the timed run by allocating all objects during the warmup phase. The same items are then re-used (being put back into the queue where they can be re-populated by a Master thread). Overheads are thus instead caused by increased memory mapping operations (each ribbon has to map in the appropriate memory) and corresponding page faults, and changes in scheduling and context switching (the kernel currently views ribbons as separate processes). We found runtimes varied more widely than for DaCapo, so we increased the iteration count to 15 for these benchmarks.

In the smallest case, a slowdown of 3% was observed. For the medium item size, results were not statistically significant. Notably, for the 1MB item size case performance actually consistently improves. We re-ran this case (another 15 iterations), and the results were consistent (5% improvement). Our theory to explain this performance increase is that because ribbons are treated as processes instead of as threads the scheduling (by the kernel) is more favorable in this case. Another possibility is that ribbons slightly changes the virtual address memory layout of the JVM, which could affect caching behaviors (as cachegrind and other tools do not support our special use of the clone system call, it is difficult to better substantiate this theory). We ran the benchmark on a machine with a different configuration (Dual 2.8 GHz Quad Core Intel Xeon) but observed an overhead of 2% to 3% instead.

The lusearch-rib benchmark runs with 32 ribbons and measures overhead from all factors, since it performs significant object allocation and is highly concurrent. lusearch had the most overhead of any parallel benchmark in the threaded version, and it is also the most concurrent benchmark. Thus, overhead of ribbonization is likely to be the highest of any of the parallel benchmarks. Overhead for lusearch was just over 12%, highlighting potential for future performance optimizations. Kernel-level support for ribbons is anticipated to significantly reduce overhead for highly concurrent, fast allocating programs.

### 5.3 Security

To verify that privileges were properly enforced we modified the Master/Worker example to reference memory that it should not have privileges for. For example, we had a Worker try to write to the queue data domain, which the Worker only had read access to. This produced the following RuntimeException:

```java
org.mmtk.vm.Memdom$ViolationException
    at MasterWorker$QueueDomain.take(...)  
    at MasterWorker$WorkerRibbonThread.run(...) 
```

### 5.4 Small-scale Refactoring Experiences

We refactored several applications to gain experience using RIBBONJ in practice, and briefly summarize our experiences thus far. First, we implemented the master/worker example in 291 lines of Java. Ribboning this program to isolate each worker required changing 23 lines of code (13 lines for protection domains, 2 lines for ribbons, 3 lines for changed allocation sites, and 5 for changes to threads). We also ribbonized the lusearch DaCapo benchmark, in this case not to gain any isolation but just to introduce ribbons for benchmarking. Only 3 lines of code had to be changed (1 line for the ribbon declaration, 2 to instantiate the ribbon).

jIRCd is an open source Java IRC daemon, consisting of 9,556 lines of code and 121 classes. We applied a two-tier isolation model where each connection operates within its own protection domain, and there is a global protection domain for shared state. Ribbonization required 19 lines of code. Most of the client-specific state was allocated in just a few places, so using the appropriate default protection

---

**Figure 5.** Wall-clock running times for the benchmarks. Times are normalized to the running time of the original version of JikesRVM.
domain for each ribbon allowed us to keep modification of allocation sites to a minimum.

We note that despite the significant difference in size between the master/worker example (291 lines) and jIRCd (9,556) the code changes required to ribbonize the application is about the same (23 lines vs 19 lines). For these relatively smaller applications, it is clear that the complexity of ribbonization is not determined by program size but rather determined by how “cleanly” the heap can be segmented into “classes” of objects (in this case, isolation classes). To understand whether this principle extends onto more industrial-strength applications we ribbonized the Tomcat web server, as detailed in Section 6.

6. Case Study: Ribbonization of Tomcat

This section presents a case study in ribbonizing the Apache Tomcat web server. We discuss the minor program extensions necessary and compare the performance of our ribbonized version to the original one.

6.1 Apache Tomcat Overview

As discussed in the introduction, Tomcat is a web server implementing the Java Servlet and JSP standards, and is used in many enterprise applications. Tomcat was first released in 1999 and has grown to become a mature open-source project, comprising over 410,000 lines of Java code.

To use Tomcat, developers package servlets and JSPs as applications into WAR files, which can then be dynamically deployed or undeployed to or from a running Tomcat server. Applications deployed as WAR files typically run in complete isolation from each other. This isolation is enforced in Tomcat through private class loaders (one per application context) and the Java Security Manager.

Using a private class loader for each context provides strong isolation, but also results in memory overhead from redundant class metadata for identical classes shared by distinct web applications. Additionally, private class loaders cause increased startup costs, due to having to decode, verify, and compile several copies of the same method instead of just one copy of each unique method.

While memory savings per instance may be relatively minor (e.g., in the JVM system [13] they found a savings of 5MB-7MB per additional deployed application for the libraries they studied), these savings can add up significantly considering that there may be hundreds or even thousands of deployed applications on a single Tomcat server. As individual servers grow increasingly powerful (and can therefore accomodate more users and deployed apps), the memory and CPU overhead from duplicate metadata and class compilation becomes increasingly relevant.

To reduce these inefficiencies we refactored Tomcat version 5.5.31 with RibbonJ to use protection domains to isolate the heaps of deployed web applications and to use ribbons to enforce this isolation while an application processes a request. Our prototype, TomcatRJ, uses our modified version of JikesRVM for its runtime, and supports the same features and deployment configurations as the original version of Tomcat that we started with.

6.2 Architecture and Coding Overview

The left side of Figure 6(a) shows the class loader hierarchy for the unmodified version of Tomcat. While there are several class loaders in the hierarchy, most are used only to isolate sensitive parts of the Tomcat runtime from any shared state and private application state, and once their initial set of classes are loaded they do not load many additional classes. The exceptions are the application private class loaders, of which there can be an arbitrary number, and so we focus our efforts on eliminating these class loaders through ribbonization.

The right side of Figure 6(a) shows the straightforward division of the Tomcat heap into protection domains, wherein each application-private class loader is replaced by an application-private protection domain. Note how only one global protection domain is used for all of the “fixed” class loaders. Within this global protection domain the class loaders provide the isolation, and so we do not need to isolate further using protection domains. Additionally, in the new architecture classes loaded by applications are loaded by the Shared class loader, ensuring there is only one copy of relevant class metadata.

Figure 6(b) shows how we map ribbons onto the protection domains and also the domain sharing privileges. All of the threads in the original design of Tomcat continue to run in the main Tomcat ribbon, termed the daemon ribbon. During the processing of a request, we need to ensure that the request is processed within an isolated context. To do this, we maintain an additional processing thread pool for each deployed application, hand the request off through a shared queue for processing on the application-private ribbon, and wait for the result. Hand-off and waiting is accomplished through normal Java synchronization primitives.

Tomcat follows the chain-of-responsibility software design pattern for request processing. A Pipeline models a sequence of actions required to process a request. Actions are modeled by Valve objects. This design lends itself quite nicely to ribbonization. We created a new type of Valve that offloaded requests to a private processing ribbon and waited for the response. Valve objects are aware of lifecycle, and so during startup and shutdown the appropriate protection domain, ribbon, and thread pool could be constructed or cleaned up, as appropriate. Then if the application is configured to use ribbonization for isolation instead of a private
class loader, a ribbonizing Valve would be inserted (at runtime) into the processing Pipeline for that application.

Note that in our design we did not modify the architecture of Tomcat’s original request processing queue—Tomcat will accept a new connection and perform application dispatching from within the daemon ribbon, and only once it has been dispatched to the proper application will it then be redirected to the appropriate application-specific ribbon. The main challenge in ribbonizing the original thread pool itself is that we do not know which application needs to process a request until the request has been partially decoded. Because a thread cannot “switch” to a different ribbon, the request must be passed off to a different thread in the desired ribbon; we thus require two extra context switches (one to pass off the request, and one to receive the response once processed). The overhead of this technique is measured in detail below.

Finally, certain class metadata must not be shared, such as the values of static fields and the locks of objects, in order to provide the same level of isolation as private class loaders. In our implementation, we rewrite the bytecode to mediate accesses to static fields and redirect them to ribbon-local values of these fields instead.

Our prototype does not address the redirection of object locking; however, this could likely be addressed at the VM level without too much difficulty by redirecting to a ribbon-local lock for the object when an object’s lock is promoted to a heavyweight lock. Additionally, the benchmark we used to evaluate the performance of our prototype (SPECweb2009) does not make use of locks on isolated objects, so our performance numbers would not change significantly if locks were fully isolated. The benchmark was designed to be representative of the designs of modern web applications. In Tomcat web applications most shared state is contained in either the session object (which is not managed by the application-private class loader), or in a database (which does not rely on Java object synchronization for concurrency control). Thus, we did not focus our efforts on isolating locks for this refactoring of Tomcat.

6.3 Assessment of Refactoring Efforts

The refactoring required relatively little coding and was exhibited purely as new lines of code (no existing lines of code needed to be changed). In total there were 680 new lines of code, most of which were for the implementation of the ribbonizing Valve. AspectJ was used to write a short yet powerful aspect to redirect static field accesses to a ribbon-local value cache. Only two of the original Tomcat Java files had to be changed—21 lines of code were inserted to check if an application should be ribbonized, and if so, to insert the ribbonizing Valve into the appropriate Pipeline.

This confirms the results found in Section 5.4 that the complexity required to ribbonize a program does not depend on program size, but rather how “cleanly” the heap can be coarsely partitioned at the source level. Tomcat’s heap already had very clear boundaries of division, which combined with its use of design patterns allowed for a highly modular implementation of ribbonization.

6.4 Performance Evaluation

We used the SPECweb2009\(^\text{\textregistered}\) benchmark to evaluate the performance of TomcatRJ under realistic workloads. In this paper we focus on the Banking workload of SPECweb2009, which was modeled after the actual workload and design of a major banking system in Texas. The benchmark is divided into three tiers: backend, application logic, and static content. The backend tier is simulated by an Apache FastCGI module written in C. The application logic is written in JSP and Java, which we host on Tomcat. Finally, static content such as bank check images are pre-generated and stored on the filesystem. We also use Tomcat to serve static content.

The benchmark is driven by one or more multithreaded clients, which make requests to the static web and application server, simulating the behavior of actual users. We configured the benchmark to eliminate any simulated “user
think time,” such that once a response to a request was returned the client would immediately make another request; this puts a higher than normal load on the web and application server as all clients are making requests as fast as possible. We used one client running on its own system with a 2.8 GHz Quad-Core Intel Xeon processor and 8GB of RAM, running OSX 10.5. The client was configured to spawn 40 concurrent request generating threads. During the benchmark the CPU on the client system remained under 50%, so the client was not a bottleneck. The client system was connected to the server system through a single Gigabit switch.

The server system had a 2.8 GHz Quad-Core Intel Xeon processor and 4GB of RAM, running x86 Linux kernel 2.6.32-22-generic #36-Ubuntu SMP. Both Tomcat and the backend simulator were run on the same system. Throughout the benchmark we observed the backend simulator using less than 5% of the CPU, so it was not a bottleneck.

We ran three iterations of the banking workload, interleaving between two configurations: first, the unmodified version of JikesRVM (fully optimized build) running the original Tomcat, and second, the ribbonized version of JikesRVM (fully optimized build) running TomcatRJ. The SPECweb2009 benchmark strives to maintain a constant level of throughput and then measures request latency to determine if requests were processed within a “good” amount of time, “tolerable” amount of time, or “unacceptable” amount of time. In all runs all requests were processed in a “good” amount of time. The benchmark slowly ramps up throughput to the target throughput, then warms up the server by maintaining the target throughput, and finally performs the actual benchmark run. Every 10 seconds data is recorded on the aggregate throughput and average request latency. Additionally, we collected statistics on the individual processing times of requests within Tomcat. In TomcatRJ we recorded the time required to actually process the request itself as well as the processing time including the time to offload the request to another ribbon and wait for the response to be returned to the calling ribbon. In this way we could accurately measure the additional latency caused by ribbonization.

<table>
<thead>
<tr>
<th>Lines of Code</th>
<th>Tomcat</th>
<th>TomcatRJ</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>New+Modified LOC</td>
<td>N/A</td>
<td>680</td>
<td>0.122% of total</td>
</tr>
<tr>
<td>Total # of Requests</td>
<td>5,537</td>
<td>5,562</td>
<td>+25</td>
</tr>
<tr>
<td>Median Latency</td>
<td>2.97ms</td>
<td>3.60ms</td>
<td>+0.63ms</td>
</tr>
<tr>
<td>Mean Latency</td>
<td>4.42ms</td>
<td>6.11ms</td>
<td>+1.69ms</td>
</tr>
<tr>
<td>Latency 99th Percentile</td>
<td>23.5ms</td>
<td>31.5ms</td>
<td>+8ms</td>
</tr>
<tr>
<td>Latency 99.9th Percentile</td>
<td>51.4ms</td>
<td>259.2ms</td>
<td>+207.8ms</td>
</tr>
</tbody>
</table>

**Table 1. Summary of data from the evaluation of Tomcat**

Figure 7 offers insight into the differences in request processing latency. From the boxplot, we see that TomcatRJ median latency is less than 1ms more than for Tomcat (0.63ms, see Table 1). In both cases, the mean is above the upper quartile (75th percentile), indicating there are very large outliers on the upper end. The upper quartile stretches farther for TomcatRJ, indicating the upper half of the distribution is more evenly distributed, and thus interruptions in processing time are more likely. Undoubtedly, this extra source of variation is caused by the extra context switches. In most cases, the extra context switches happen relatively quickly, but in the TomcatRJ case small delays happen more frequently, and the duration of rare delays is larger.

From the CDF we can see that for approximately 80% to 85% of all requests the difference in latency is less than 1ms. However, past this point, the outliers become more dramatic in TomcatRJ. The 99th and 99.9th percentile for latency in Table 1 provide insights into the differences in these extreme outliers. At the 99th percentile, the difference is still not so dramatic (8ms longer for TomcatRJ), but at the 99.9th percentile, the outlier for TomcatRJ is more than 4
times the outlier for Tomcat. When the system is under heavy load (having 40 clients make requests as fast as possible to the server application), it appears that in very rare cases one or both of the threads processing a request is starved while waiting to be scheduled to receive the processed result.

Table 2. Summary of additional latency in request processing due to ribbonization in TomcatRJ

Table 2 and Figure 8 present the observed additional latency caused by ribbonization. The additional latency is calculated by taking the request processing time as observed in the main daemon pool for a thread and subtracting the time it took to actually process the request on the isolated thread. This can be obtained by having the final Valve in the Pipeline (which executes on the isolated ribbon) record the time it takes to process the request just on that thread. The difference thus includes both the extra time required to pass off a request and the extra time waiting to be scheduled to receive the processed result.

Figure 8. Visualizes the distribution for the additional time required to process requests in Tomcat due to ribbonization during SPECweb2009 Banking. The vertical line at 1ms on the CDF marks the 90th percentile.

Figure 9. Throughput for each 10-second interval during SPECweb2009 Banking, averaged over 3 runs. Excludes warmup and rampdown.

Measured throughput over the life of the workload is shown in Figure 9. It is important to note that despite the increase in latency in TomcatRJ, there is no statistically significant difference in total throughput. This indicates that the additional context switch is not causing significant overhead in terms of CPU cycles. This is not surprising considering that a modern CPU on a modern OS can perform millions of context switches per second. This result also indicates that the additional logic in the allocation fast path in the ribbonized version of JikesRVM is not causing significant overhead for this workload.

Surprisingly, even the same general shape of the throughput graph is similar over time. We postulate that each benchmark client thread is seeded deterministically, and that because the throughput of the benchmark is carefully regulated, similar patterns of throughput over time can be observed even when averaging data points across several distinct runs.

In conclusion, we have observed that for almost all requests, the cost of ribbonization was less than 1ms and that total throughput is maintained at previous levels. However, the additional context switch introduces variation into the total processing time of a request, such that 1 out of every 1000 requests is expected to experience an additional delay between 200ms and 550ms and 1 out of every 100 requests is expected to experience an additional delay of 8ms.

7. Related Work

Many proposals for fine- and course-grained memory protection based on hardware modifications have appeared, including Mondrian [39], Legba [38], InfoShield [32], and others (e.g., [40]). These systems only provide protection mechanisms at the lowest level, and do not address programmability within such models. Traditional capability-based access control mechanisms (e.g., [13, 28]) are not suitable for pro-
tecting memory regions as they can be shared in an unrestricted fashion across threads.

Both hardware-based [18, 27, 31, 36] and software-based [10, 29] techniques have been proposed for guarding against heap corruption and memory bugs. These techniques target specific types of flaws and do not provide general-purpose protection. Also, they either do not distinguish between threads in a process [18, 27, 29, 31] or are based solely on temporal constraints [10, 56]. All hardware-based techniques except [27] require hardware modifications.

Region-based memory management has been investigated in various programming paradigms, e.g., functional [35], imperative [23], as well as in real-time systems [8, 7]. The main driving force is to avoid costly heap allocation and garbage collection overheads, which is achieved by introducing structure into the memory such as to deallocate entire regions in single steps. Alas, region-based memory management approaches force rigid hierarchical organizational patterns on the heap whose lifetime is tied to code scope, and focus on performance but provide no protection features. Memory structuring is also the motivation for isolates introduced in the Multitasking Virtual Machine (MVM) [14, 24], which reduces the memory footprint by allowing code sharing without state. This leads to security benefits, but at a very coarse grain, and still requires IPC mechanisms for communication between isolates.

Ownership types [12, 6] and derivatives attempt to control the effects of aliasing in object settings, by limiting accesses to objects to their owners. The latter entities — objects themselves — are given certain access rights on the former entities, while the same rights are denied to others. Ideally, ownership relationships lead to a hierarchical structure of the heap, or can even allow for stack allocation instead of heap allocation. Ownership types target at static enforcement of access rules which is quite restrictive because communication patterns are fixed statically by type — different threads acting in different roles may not access the same type differently. Most importantly, however, in the present context, ownership types as such cannot fully guard against malicious components and cannot protect low-level system components.

The implementation for lightweight protection domains in Nooks [35] and others (e.g., [11, 44]) also utilizes page table manipulation and/or hardware segmentation. The protection domains and corresponding privileges were fixed in these systems, and so are not easily extended to support ribbons. Opal [8] is an OS with features for processes with partially-shared memory address spaces. Opal requires a single system-wide shared virtual address space, and does not model allocation, which is important for security and allocation optimization. Its dynamic grant/deny segmentation model is potentially less secure (accidental capability leakage via bugs, manipulation) and prevents analysis of programs for automatic parallelization. Others (e.g., [17, 15, 30]) have similar limitations.

Pure software techniques based on code rewriting [37], inline monitors [20], and static enforcement through typing [1, 4], termed software-isolated processes (SIPs), have been proposed and show promise for efficient enforcement of fine-grained protection mechanisms. Code rewriting and inline monitors rely on either a trusted rewriter or a trusted static verifier to ensure correctness. Additionally, these techniques either have fixed protection domains [8] or require slow paths with excessive overheads (2 × to 8 × slowdown [20]), and may be highly architecture specific, as was the case with [37]. SIPs in [1] require completely disjoint protection domains, and do not allow for arbitrarily flexible communication patterns that can be safely adjusted at runtime. None of these prior techniques can be easily adapted to efficiently provide different protection domains for different threads within the same process.

8. Conclusions

Herein we presented Ribbons, a new model for programming components with full or partial heap isolation. We defined RIBBONJ, a realization of this model in Java, and explored its formal properties in RIBBONJ-LITE. We know of no other language that supports (1) typed protection domains instantiated at runtime and associated with a group of threads, (2) the inject privilege, and (3) adaptable isolation levels.

We discussed the design and implementation of the RibbonJ compiler, supporting JVM, and Linux user-space runtime. The full RibbonJ compiler and ribbonized JikesRVM are freely available for download and experimentation [25]. Our initial system demonstrates reasonable overheads for many workloads. In our case study of Apache Tomcat we discussed the design, implementation, and evaluation of isolating web applications using ribbons. We found that isolation could be implemented using ribbons without complex or intrusive code changes. We also found that typical overhead is less than 1ms per request, although ribbonization did increase latency significantly for outliers.

Future work will focus on kernel-level support for ribbons. We anticipate kernel-level support providing (a) significant performance improvements through tighter integration with virtual memory protection mechanisms and the OS scheduler, and (b) the potential to provide full or partial isolation of ribbons beyond the heap. Many open questions surround how to best model the isolation of non-heap resources using ribbons, including how to do so in such a way as to allow for “kill-safety” of individual ribbons running within a process. We envision the ribbons programming model providing a comprehensive framework for the efficient isolation of components at a language, language runtime, and systems level.
References


