Capsule-oriented Programming

Hridesh Rajan, Steven M. Kautz, Eric Lin, Sarah Kabala, Ganesha Upadhyaya, Yuheng Long, Rex Fernando, and Loránd Szakács

TR #13-01
Initial Submission: February 28, 2013.

Keywords: concurrency, modularity, synergy
CR Categories:
D.2.10 [*Software Engineering*] Design
D.1.5 [*Programming Techniques*] Object-Oriented Programming
D.2.2 [*Design Tools and Techniques*] Modules and interfaces,Object-oriented design methods
D.2.3 [*Coding Tools and Techniques*] Object-Oriented Programming
D.3.3 [*Programming Languages*] Language Constructs and Features - Control structures

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1041, USA

Capsule-oriented Programming

Hridesh Rajan, Steven M. Kautz, Eric Lin, Sarah Kabala, Ganesha Upadhyaya,
Yuheng Long, Rex Fernando, and Loránd Szakács
Iowa State University, Ames, IA, 50011, USA
{hridesh,smkautz,eylin,skabala,ganeshau,csgzlong,fernanre,lorand}@iastate.edu

ABSTRACT

Many programmers find writing and reasoning about concurrent programs difficult and can benefit from better abstractions for concurrency. A promising class of such concurrency abstractions combines state and control within a single linguistic mechanism and uses asynchronous messages for communications, e.g. active objects or actors. One hurdle is the need to adapt to an asychronous style of programming. We believe that most benefits of actor-like abstractions can be brought to sequentially-trained programmers via a more familiar synchronous model. We call this model *capsule-oriented programming*, where programmers describe a system in terms of its modular structure and write sequential code to implement the operations of those modules using a new abstraction that we call *capsule*. Capsule-oriented programs look like familiar sequential programs but they are implicitly concurrent. We present Panini, a capsule-oriented programming language, and its compiler, which help programmers avoid two classes of concurrency errors: sequential inconsistency and data races due to sharing. We have refactored the Java Grande and NPB benchmarks (>134,000 LOC) using Panini, leading to simpler and shorter programs that perform as well as the parallel versions provided with the benchmarks.

1. INTRODUCTION

Modern software systems tend to be distributed, event-driven, and asynchronous, often requiring components to maintain multiple threads for message and event handling. In addition, there is increasing pressure on developers to introduce concurrency into applications in order to take advantage of multicore processors to improve performance. Yet concurrent programming stubbornly remains difficult and error-prone. First, a programmer must partition the overall system workload into tasks. Second, tasks must be associated with threads of execution in a manner that improves utilization while minimizing overhead; note that this set of decisions is highly dependent on characteristics of the platform, such as the number of available cores. Finally, the programmer must manage the dependence, interaction, and potential interleaving between tasks to maintain the intended semantics and avoid concurrency hazards, often by using low-level primitives for synchronization. For the programmer to address these issues, better abstractions are needed that can hide the details of concurrency and allow them to focus on the program logic.

The significance of better abstractions for concurrency is not lost on the research community. Some key ideas from the last two decades include: *guardians* [25], *active objects* [23, 32], and *actors* [3], all of which combine state and control within a single linguistic mechanism and use asynchronous messages for communications. We believe that a major gap remains. There is an impedance mismatch between sequential and implicitly concurrent code written using actor-like abstractions that is hard for a sequentially trained programmer to overcome. These programmers typically rely upon the sequence of operations to reason about their programs.

Contributions. We present *capsule-oriented programming* to address the challenges of concurrent programming. A central goal of this programming style is to provide tools to enable programmers to simply do what they do best, that is, to describe a system in terms of its modular structure [34] and write sequential code to implement the operations of those modules. To achieve this, we introduce a new abstraction that we call *capsule*. A capsule is similar to a process in CSP [21]; it defines a set of public operations, and also serves as a memory region for some set of ordinary objects.

One goal in capsule-oriented programming is that the programmer should get the benefits of asynchronous execution without being forced to adapt to an asynchronous, message-passing style of programming. To the programmer, inter-capsule calls look like ordinary method calls. Capsule-oriented programs are implicitly concurrent. There are no explicit threads or synchronization locks; if necessary or beneficial, concurrency is introduced by the compiler. Capsule-oriented programming eliminates two classes of concurrency errors: sequential inconsistency and race conditions due to shared data.

We have realized the basic ideas behind capsule-oriented programming in an extension of Java that we call Panini (§3) and have implemented a compiler for Panini by extending the industry standard `javac` compiler (§4). Our experience with over 134,000 lines of capsule-oriented programs shows that they are simpler, shorter, and have performance comparable to explicitly concurrent versions written by expert benchmark programmers (§5 and §6).

2. MOTIVATION

To illustrate the challenges of concurrent program design, consider a simplified navigation system. The system consists of four components: a route calculator, a maneuver generator, an interface to a GPS unit, and a UI. The UI requests a new route by invoking a `calculate` operation on the route calculator, assumed to be computationally intensive. When finished, the route is passed to the maneuver generator via method `setNewRoute`. The GPS inter-

Java program with explicit concurrency

```
1  class ManeuverGen {
2    private Route currentRoute;
3    private Position currentPosition;
4    private UI ui;
5    public synchronized void setNewRoute(Route r) {
6      currentRoute = r;
7    }
8    public synchronized void updatePosition(Position p) {
9      currentPosition = p;
10     final Position temp = p;
11     Runnable r = new Runnable() {
12         public void run() {ui.updatePosition(temp);}
13     };
14     SwingUtilities.invokeLater(r);
15     final Instruction inst = nextManeuver();
16     if (inst != null) {
17         r = new Runnable() {
18             public void run() { ui.announceNextTurn(inst); }
19         };
20         SwingUtilities.invokeLater(r);
21     }
22   }
23   public synchronized Position getCurrentPosition() {
24     return currentPosition;
25   }
26   private Instruction nextManeuver() {/* ... */}
27 }
28 interface Calculator {void calculate(Position dst);}
29 class Shortest implements Calculator {
30   private ManeuverGen mg;
31   public Shortest(ManeuverGen mg) {this.mg = mg;}
32   public void calculate(final Position dst){
33     Thread t = new Thread(new Runnable() {
34       public void run() {
35         Route r = helper(mg.getCurrentPosition(), dst);
36         mg.setNewRoute(r);
37       }
38     });
39     t.start();
40   }
41   private Route helper(Position src, Position dst) {/* */}
42 }
43 class GPS {
44   private ManeuverGen mg;
45   public GPS(ManeuverGen mg) {this.mg = mg;}
46   public void runLoop() {
47     while (true) mg.updatePosition(readData());
48   }
49   private native Position readData();
50 }
```

Java program, con't

```
51 class UI { /* provides updatePosition, announceNextTurn */}
52 class Navigation {
53   public static void main(String[] args) {
54     ManeuverGen mg = new ManeuverGen();
55     Calculator rc = new Shortest(mg);
56     final GPS gps = new GPS(mg);
57     Thread t = new Thread(new Runnable() {
58       public void run() { gps.runLoop(); }
59     });
60     t.start();
61       //
62       // Also create and start UI, details not shown
63       //
64   }
65 }
```

Implicitly concurrent Panini program

```
66 capsule ManeuverGen (UI ui) {     // Requires an instance of capsule UI
67   Route currentRoute = null;      // A capsule state
68   Position position = null;
69   void setNewRoute(Route r) { currentRoute = r; }    // A capsule procedure
70   void updatePosition(Position p) {
71     position = p;
72     ui.updatePosition(p);         // Inter-capsule procedure call
73     Instruction inst = nextManeuver();
74     if (inst != null) ui.announceNextTurn(inst);
75   }
76   Position getCurrentPosition() { return position; }
77   private Instruction nextManeuver() {/* ... */}     // A helper procedure
78 }
79 signature Calculator { void calculate(Position dst); }
80 capsule Shortest (ManeuverGen m) implements Calculator {
81   void calculate(Position dst) {
82     Route r = helper(m.getCurrentPosition(), dst);
83     m.setNewRoute(r);
84   }
85   private Route helper(Position src, Position dst) {/* ... */}
86 }
87 capsule GPS (ManeuverGen mg) {
88   void run() {
89     while (true) mg.updatePosition( readData() );
90   }
91   private native Position readData();
92 }
93 capsule UI { /* provides updatePosition, announceNextTurn */}
94 system Navigation {
95   UI ui; ManeuverGen m; Shortest r; GPS g; // Capsule instances
96   m(ui); r(m); g(m);                       // Wiring capsule instances
97 }
```

Figure 1: Programs for a simplified navigation system. Classes `Position`, `Route`, and `Instruction` are elided.

face continually parses the data stream from the hardware and updates the maneuver generator with the current position via method `updatePosition`. The maneuver generator checks the position against the current route and generates a new turn instruction for the UI if needed (not computationally intensive).

The modular structure of the system is clear from the description above, and it is not difficult to define four Java classes with appropriate methods corresponding to this design. However, the system will not yet work. The programmer is faced with a number of non-trivial decisions: Which of these components needs to be associated with an execution thread of its own? Which operations must be executed asynchronously? Where is synchronization going to be needed? A human expert might reach the following conclusions, shown in the listing in Figure 1.

- A thread is needed to read the GPS data (lines 57-60)
- The UI, as usual, has its own event-handling thread. The calls on the UI need to pass their data to the event handling thread via the UI event queue (lines 10–14 and 17–20)
- The route calculation needs to run in a separate thread; otherwise, calls to calculateRoute will "steal" the UI event thread and cause the UI to become unresponsive (lines 33–39)
- The `ManeuverGen` class does not need a dedicated thread, however, its methods need to be synchronized, since its data is accessed by both the GPS thread and the thread doing route calculation (lines 5, 8, and 23)

None of the conclusions above, in itself, is difficult to implement in Java. Rather, in practice it is the process of visualizing the interactions between the components, in order to reach those conclusions, that is extremely challenging for programmers [28, 2].

3. THE PANINI LANGUAGE

A central goal of capsule-oriented programming and the Panini language is to help sequentially trained programmers deal with the challenges of concurrent program design.

Overview. The Panini programmer specifies a system as a collection of *capsules* and ordinary object-oriented classes. A *capsule*

is an abstraction for decomposing a software into its parts and a *system* is a mechanism for composing these parts together.[1][2] A capsule is like Parnas's information-hiding module [34] in that it defines a set of public operations, hides the implementation details, and could serve as a work assignment for a developer or a team of developers. Beyond these standard responsibilities, a capsule also serves as a memory region, or ownership domain [11, 10], for some set of standard object instances and behaves as an independent logical process [21]. Inter-capsule calls look like ordinary method calls to the programmer. The object-oriented features are standard, but there are no explicit threads or synchronization locks in Panini.

3.1 Declarations in Panini

A program in Panini can have zero or more signature declarations, zero or more class declarations, zero or more capsule declarations, and a system declaration.

Signature. A signature declaration in Panini contains one or more abstract procedure signatures. An example signature declaration `Calculator` appears on line 79 in Figure 1. Each procedure signature has a return type, a name, and zero or more formal parameters. This syntax is similar to interfaces in Java, except that for simplicity we do not allow signature inheritance.

Capsule. A capsule declaration consists of the keyword 'capsule', the name of the capsule, zero or more formal parameters representing dependencies on other capsules, and zero or more signatures representing services that the capsule provides, followed by the capsule's implementation. This is similar to both the Mesa [31] and nesC [16] languages. Each procedure declaration in every signature implemented by the capsule must match in entirety with exactly one capsule procedure. Panini does not have capsule inheritance but does have class inheritance. The primary mechanism for reuse of capsules is composition.

The example in Figure 1 contains four capsule declarations. The capsule `ManeuverGen` requires a `UI` capsule instance on line 66, (`UI ui`). Similarly, the capsule `Shortest` declares that it requires an instance of `ManeuverGen` capsule and it provides the `Calculator` signature on line 80. After a capsule instance is correctly initialized, expressions inside a capsule instance may access these imported capsule instances using their names, e.g., `ui`. Like ML modules [29], Panini capsule instances are not first-class values so capsule instances may not be passed as arguments to methods or stored in capsule states or object fields.

A capsule implementation consists of zero or more *state declarations*, zero or more *capsule procedures*, and zero or more internal class declarations. A state declaration has a class type, a name, and optionally an initialization expression, so in that sense it is similar to a field in traditional class declarations, except that a capsule instance controls all accesses to the object graph reachable from its states, i.e., a capsule instance acts as a dominator for the graph [11]. All state declarations are private to a capsule, therefore, no visibility modifiers are necessary. Two examples appear on lines 67-68 in Figure 1.

A capsule procedure has a return type, a name, zero or more arguments, and a body. Helper procedures can be declared by qualifying them with a modifier `private`. All capsule procedures, except helper procedures, constitute the interface of the capsule. There is one designated optional capsule procedure `run` representing that the capsule can start computation without external stimuli.

[1]The intent of capsule, a modeling notation in UML-RT and ROOM [39] is similar in that there is an activity, a state machine and an interface, but Panini capsules are a language feature.

[2]Unrelated to CAPSULE [27] a stream processing framework.

A capsule can also contain standard object-oriented class declarations. These class declarations are considered internal to the capsule and are not visible outside the capsule. A class declaration that is used across two or more capsules should be declared outside a capsule, as is usual in object-oriented languages.

System Declaration. A system declaration is a declarative static specification of the topology of capsule instances that would be present in the program. The system declaration for the navigation system appears on lines 94-97 in Figure 1; line 95 specifies the capsule instances that will be participating in this system, e.g., an instance of `UI`; and line 96 specifies how these instances are connected, e.g., the `GPS` instance `g` is connected to the `ManeuverGen` instance `m`. The topology of capsules is fixed for a program and does not change dynamically, which facilitates more precise static analysis of capsule interactions (§4). Arrays of capsule instances of fixed length can also be declared.

Design Rationale for Capsules. At first glance a capsule declaration may look similar to a class declaration, thus naturally raising the question as to why a new syntactic category is essential, and why class declarations may not be enhanced with the additional capabilities that capsules provide, namely, confinement (as in Erlang [4]) and an activity thread (as in previous work on concurrent OO languages [8, 44, 40]).There are three main reasons for this design decision in Panini. First, we believe based on previous experiences that objects may be too fine-grained to think of each one as a potentially independent activity [33]. Second, we wanted to specify a system as a set of related capsules with a fixed topology, in order to make it feasible to perform static analysis of the system graphs described in §4; this implies that capsules should not be first-class values. Third, there is a large body of OO code that is written without any regard to confinement. Changing the semantics of classes would have made reusing this vast set of libraries difficult, if not impossible. In the current design of Panini, since syntactic categories are different, sequential OO code can be reused within the boundary of a capsule without needing any modification.

3.2 Informal Semantics

Any capsule with a `run` procedure begins executing independently as soon as the initialization and interconnection of all capsules is complete and may generate calls to the procedures of other capsules. For example, referring to Figure 1, capsule `GPS` will run code on lines 88-90. Capsules without a `run` procedure, such as `Shortest` and `ManeuverGen`, perform computation only when their procedures are invoked.

As discussed in more detail in §4, even a capsule without a `run` procedure may be associated with an independent execution thread. Thus, in any nontrivial Panini program there will be issues of thread-safety to address for all capsules. Thread-safety is primarily ensured by confining the object graphs rooted at each capsule's states, i.e., only a single execution thread has access to the states of a capsule. In inter-capsular calls, built-in and immutable types are passed and returned by value, whereas reference types are passed by linear transfer of ownership for the object graph rooted at the parameter object or result object.

Although capsule procedures may execute asynchronously, the programmer does not have to program in a message-passing style. A call to a capsule procedure behaves like an ordinary method call. Calling a capsule procedure may have side-effects on the state of the capsule instance, e.g., reading or writing state, and may also lead to external calls to other capsule procedures. For two consecutive procedure calls on the same capsule instance, the side-effects of the first procedure call are always visible to the second procedure call to provide sequential consistency. However, it is also pos-

sible that two calls on *different* capsules ultimately lead to effects within a single capsule, and we also ensure that effects of consecutive calls, as observable within a given capsule, are always seen in the order intended by the programmer.

Navigation system revisited. Compared to the explicitly concurrent Java program on lines 1-65 in Figure 1, the Panini program on lines 66-97 is an implicitly concurrent program. Owing to the declarative nature of some Panini features, this program is much shorter compared to the Java program. Most importantly, this example illustrates some of the key advantages of the Panini approach for programmers. These are:

- They don't need to create explicit threads or specify whether a given capsule needs its own thread of execution.
- They don't need to recognize or reason about potential data races.
- They work within a familiar method-call style interface with a reasonable expectation of sequential consistency.
- All synchronization-related details are abstracted away and are fully transparent to them.

4. THE PANINI COMPILER

To show the feasibility of realizing capsule-oriented programming in an industrial-strength compiler, we extended the OpenJDK Java compiler (javac) to add support for capsule-oriented programming to create the Panini compiler. We considered library and annotation-based approaches instead of extending syntax for Java, but the notation burden of these alternatives was significant enough to hamper usability, so we went with syntax changes.

There were three major challenges: how to detect confinement violations, how to detect the possibility of sequential inconsistency, and how to realize capsules while minimizing overhead.

Confinement. Within a capsule, thread-safety is primarily ensured by confinement: only a single capsule instance has access to the objects belonging to the instance. Confinement in itself is generally insufficient, however, since mutable objects can still be passed as arguments to inter-capsule procedure calls and returned as values. Unlike Erlang, which enforces that all data is immutable [4], or Scala actors with capabilities [19], which use an ownership type system [11, 10], we have adapted a static analysis for confinement [18] that tracks variables used as parameters in inter-capsule procedure calls and in return statements of public procedures.

Sequential consistency. In Panini, all instances of capsules in a system and all interconnections between them are declared statically; there is no inheritance for capsules and references to them cannot be passed as procedure arguments. The compiler exploits these properties to efficiently construct a *system graph* showing dependencies between capsule instances. In addition, when compiling a system declaration the compiler produces a more detailed *interprocedural system graph* in which the nodes are capsule procedures and edges are interprocedural calls. The compiler detects cases in which two calls lead to observable effects within the same capsule instance and provides a warning to the programmer. In some cases, that warning may be benign, and we provide an annotation `Commutes` for the programmer to express that. Our current prototype does not check whether this annotation is correctly placed, however, Diniz and Rinard's commutativity analysis could be applied [13]. If two calls are made to the same capsule instance, the FIFO ordering of messages (see below) ensures that effects within that capsule instance will occur in order.

Execution model. A capsule instance is a potential *execution domain* as well as an ownership domain for its state. An important feature of Panini is that the specification of capsules by the programmer is decoupled from the decisions about how to map each capsule's activities to OS threads. Logically, each capsule is an independent process-like entity, and the invocation of a capsule procedure triggers creation of a request object that is placed on a FIFO queue (the message ring buffer shown in Figure 2). In the default implementation, requests are executed in FIFO order by a single thread associated with the capsule instance.

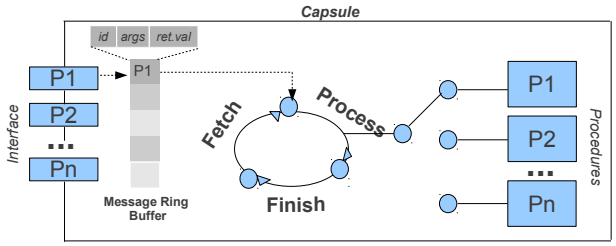

Figure 2: A capsule. Procedure invocations are enqueued as requests on the Message Ring Buffer. Requests are processed by an execution thread in FIFO order.

From the caller's point of view, invoking a capsule procedure looks like an ordinary method call. The call returns immediately and the caller receives a *future* as a proxy for the actual return value (void return values are allowed). The future is completely transparent to the programmer, that is, it is an automatically generated type that is consistent with the expected return value [35]. If the value is not used immediately, the caller can continue execution. An attempt by the caller to invoke a method on the returned future will cause the caller to block until the callee has finished executing the procedure, automatically providing a synchronization point.

5. FURTHER APPLICATIONS

In coarse-grained concurrent applications, such as the simplified navigation system illustrated in Figure 1, the main motivation is not necessarily to achieve parallel execution but rather to correctly and safely model components that are "logically autonomous" [24]. These kinds of asynchronous, event-driven systems are the obvious candidates for Panini. However, the capsule abstraction also adapts easily to other styles of parallel programming, while retaining Panini's advantages of abstracting away the concurrency control mechanisms and ensuring data confinement.

Master-worker example. As an example, Figure 3 is a simple parallel program in which a number of "worker" tasks execute a Monte Carlo approximation of π in parallel; a "master" task combines the results as the workers finish. Each call to `compute` on line 17 executes asynchronously in an instance of a `Worker` capsule; the returned `Number` object is transparently replaced by a future for the eventual result. The futures provide an implicit barrier; that is, in the call to `value` on line 19, the execution of the `run` procedure in master capsule blocks until the corresponding *Worker* has finished computing its result.

The explicitly concurrent Java program has the applications's concerns tangled with the concurrency concerns, whereas the Panini program abstracts away the details of concurrency. As Figure 3 shows, the performance of the Panini program is comparable to that of the thread-based program. A more significant potential benefit is that the Panini compiler can be employed to guard against race conditions when parallelism is introduced into an application.

Thread pool example. Figure 4 shows the Java and Panini code for a simplified server using a thread pool implemented using a

Java program with threads and synchronization

```
1  class Worker implements Runnable {
2    long num;
3    private final CountDownLatch doneSignal;
4    Worker(long num, CountDownLatch doneSignal) {
5      this.num = num;
6      this.doneSignal = doneSignal;
7    }
8    Random prng = new Random ();
9    Number _circleCount = null; //Emulates return value of worker
10   Number getCircleCount() { return _circleCount; }
11   public void run() {
12     _circleCount = new Number(0);
13     for (long j = 0; j < num; j++) {
14       double x = prng.nextDouble();
15       double y = prng.nextDouble();
16       if ((x * x + y * y) < 1) _circleCount.incr () ;
17     }
18     doneSignal.countDown();
19   }
20 }
21 class Master {
22   void assign(long totalCount, int numWorkers) {
23     CountDownLatch l = new CountDownLatch(numWorkers);
24     Worker[] workers = new Worker[numWorkers];
25     for (int i = 0; i < numWorkers; ++i) {
26       workers[i] = new Worker(totalCount/numWorkers, l);
27       new Thread(workers[i]).start () ;
28     }
29     try {
30       l.await() ;
31     } catch (InterruptedException e) { /* Error recovery */ }
32     Number[] results = new Number[numWorkers];
33     for (int i=0; i< numWorkers; i++)
34       results[i] = workers[i].getCircleCount();
35     long total = 0;
36     for (Number result: results) total += result.value() ;
37     double pi = 4.0 * total / totalCount;
38   }
39 }
40 public class Pi {
41   public static void main(String[] args) {
42     Master master = new Master();
43     master.assign(50000000,10);
44   }
45 }
```

Panini program

```
1  capsule Worker (int num) {
2    Random prng = new Random ();
3    Number compute() {
4      Number _circleCount = new Number(0);
5      for (int j = 0; j < num; j++) {
6        double x = prng.nextDouble();
7        double y = prng.nextDouble();
8        if ((x * x + y * y) < 1) _circleCount.incr () ;
9      }
10     return _circleCount;
11   }
12 }
13 capsule Master (int totalCount, Worker[] workers) {
14   void run(){
15     Number[] results = new Number[workers.length];
16     for (int i = 0; i < workers.length; i++)
17       results[i] = workers[i].compute();
18     long total = 0;
19     for (Number result: results) total += result.value() ;
20     double pi = 4.0 * total / totalCount;
21   }
22 }
23 system Pi {
24   Master master; Worker workers[10];  // Statically declared capsule array
25   master(50000000, workers);          // Wiring capsules together
26   for (Worker w : workers) w(5000000); //Giving initial value to num
27 }
```

Performance results

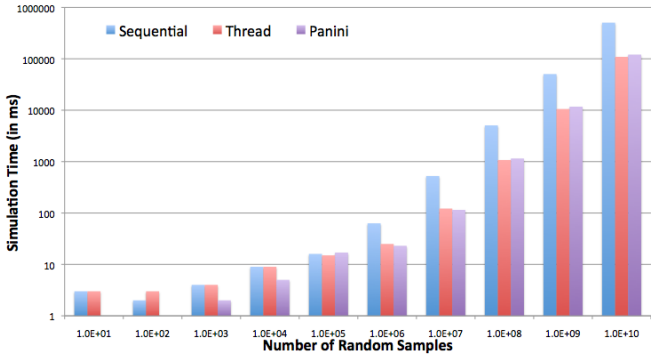

Figure 3: Parallel programs for Monte Carlo calculation of π. The user-defined `Number` class is the same in both Java and Panini.

leader-followers pattern [38]. The Java code is based on the implementation of the Tomcat 6 web server [1]. The thread pool `Pool` consists of a queue of idle threads. When the pool recieves a request from the `Host` via the `runIt` method to handle a connection, a `Worker` is removed from the queue and awakened with the `doRunIt` method. When finished handling the connection, the worker calls `available` to put itself back on the idle queue and then waits for the next task. If there are no idle threads in the queue when the host calls `runIt`, the main thread will block until a worker is available to handle the request, automatically providing a form of throttling to ensure that existing connections are handled before any new connections are accepted. (Note that this example, while faithful to the Tomcat implementation, does not precisely match the leader-followers pattern as described in [38] in which the worker threads also take turns listening for connections.)

The Panini version is functionally the same, but is dramatically simpler since all of the queueing, waiting, and notification is implicit. When a `Worker` calls the `getConnection` procedure, the call returns immediately with a future representing the Socket object, and a task corresponding to the procedure body is queued for execution in the `Host`. When a worker attempts to use the socket in its `handleConnection` helper, it blocks until the `Host` provides the actual socket.

Pipeline example. Panini's features can also be useful for implementing applications that can benefit from pipeline parallelism. To illustrate, consider the problem of maintaining the running average and maximum of a stream of numbers, e.g. average and max price of a stock in a day. Figure 5 presents a model with four capsules: `Source`, `Average`, `Max`, and `Sink`. Each of these capsules implement the signature `Stage` that provides only one procedure `consume`. The capsule `Source` generates a stream of random numbers and sends it down the pipeline, where `Average` updates its running average, `Max` updates the maximum, and `Sink` counts. Finally, on line 33, instances of these capsules are connected as a pipeline `src -> avg -> sum -> max -> snk`.

This code is very similar to how one would write a sequential program to model the same scenario, so the structure of this Panini program would be familiar to a sequential programmer. This code is also free of any concurrency-related concerns, such as setup and teardown threads for running each stage in the pipeline concurrently and synchronization between adjacent stages to hand off the input to the next stage, which is typical of a pipeline pattern. This code would, however, have all of the benefits of the explicitly concurrent implementation. Therefore, we believe that a sequential programmer would have a greater chance of success.

Java

```
1  public class Server {
2    public static void main(String[] args) {
3      new Host().runServer();
4    }
5  }
6  class Pool {
7    Queue<Worker> idle = new LinkedList<Worker>();
8    public void addWorker(Worker w) {
9      idle.add(w);
10   }
11   public synchronized void runIt(Socket s) {
12     while (idle.isEmpty()) wait();
13     Worker w = idle.remove();
14     w.doRunIt(s);
15   }
16   public synchronized void available(Worker w){
17     idle.add(w);
18     notify();
19   }
20 }
21 class Worker extends Thread {
22   private Pool p;
23   private Socket s;
24   public Worker(Pool p){this.p = p;}
25   public void run() {
26     while (true) {
27       synchronized(this) {
28         while (s == null) wait();
29         handleConnection(s);
30         s = null;
31       }
32       p.available(this);
33     }
34   }
35   public synchronized void doRunIt(Socket s){
36     this.s = s;
37     notify();
38   }
39   private void handleConnection(Socket s){ /*...*/}
40 }
```

Java program, con't

```
41 class Host {
42   private Pool p;
43   public Host(){
44     p = new Pool();
45     for (int i = 0; i < 10; ++i) {
46       p.addWorker(new Worker(p));
47     }
48   }
49   public void runServer() {
50     ServerSocket ss = new ServerSocket(8080);
51     while (true) {
52       Socket s = ss.accept();
53       p.runIt(s);
54     }
55   }
56 }
```

Panini

```
57 capsule Host() {
58   ServerSocket ss = new ServerSocket(8080);
59   Socket getConnection() {
60     Socket s = ss.accept();
61     return s;
62   }
63 }
64 capsule Worker(Host h) {
65   void run() {
66     while (true) {
67       Socket s = h.getConnection();
68       handleConnection(s);
69     }
70   }
71   private void handleConnection(Socket s) { /*...*/ }
72 }
73 system Server {
74   Host h; Worker workers[10]; // Statically declared capsule array
75   for (Worker w: workers)
76     w(h);                     // Wiring capsules together
77 }
```

Figure 4: Server with a leader-followers style thread pool (based on Tomcat 6). Exception handling code is elided.

6. EVALUATION: PERFORMANCE

We now investigate three critical features of Panini to determine its viability as a language and an abstraction for non-trivial, implicitly concurrent programming. First, in §6.2 we translate 15 popular benchmarks shown in Figure 6 into Panini and determine the lines of code changed by minimal rewriting. Second, in §6.3 we test the speedup of these Panini benchmarks against the original serial and parallel versions. Third, in §6.4 we examine the memory overhead for Panini benchmarks compared with the originals.

6.1 Benchmarks

Our selected benchmark suites, JavaGrande (JG) [41] and NPB [15], offer explicitly-threaded and explicitly-serial reference programs. Our rewriting translated only concurrency-related code and structure from the original programs and did not alter or optimize code not related to concurrency. We translated every Java benchmark in NPB and every benchmark in JG that had both a serial and a threaded version. For the rest of our evaluation, we refer to these explicitly-sequential and explicitly-parallel programs as SERIAL and PARALLEL, respectively. Figure 6 summarizes several characteristics of these programs gathered with SOOT [42] and lists what problem sizes our experiments used.

6.2 Code Size

Explicitly concurrent code can be some of the most intricate code in a program and challenging to write. Reducing this volume of code eases the programmer's burden and limits development bugs.

Research Question 1: *Do Panini programs require less code to implicitly achieve parallel and serial execution?*

Methodology. The translation of NPB and JG benchmarks from Java to Panini allowed many lines of explicitly-concurrent code to be replaced by implicitly-concurrent lines of code. We used the following non-intrusive guidelines for rewriting:

- Change thread objects to Panini capsules
- Change synchronized methods and blocks to Panini capsule methods and calls
- Create Panini capsule fields for top-level class instances
- Remove explicitly-concurrent and serial-only code

We used standard techniques for counting lines of code, which ignore blank lines and comments and count multi-statement lines with a weight equal to the number of actual statements.[3]

Results. The lines of code added and deleted by our rewriting are shown in Figure 7.

Analysis. The translated Panini benchmarks show a significant drop in lines of concurrency-related code because synchronization patterns are more concise in Panini. For example, execution of parallel workers is synchronized regularly in NPB with 11 lines in a master object and 13 lines in a worker object. Panini realizes this pattern implicitly with only 2 and 4 capsule lines, respectively.

[3] `http://reasoning.com/downloads/java_line_count_estimator.html`

```
signature Stage { void consume(long n); }
capsule Source (Stage next, long num) {
  Random prng = new Random ();
  void run() {
    for (int j = 0; j < num; j++) {
      long n = prng.nextInt(1024);
      next.consume(n);
    }
  }
}
capsule Average (Stage next) implements Stage {
  long average = 0;
  long count = 0;
  void consume(long n) {
    next.consume(n);
    average = ((count * average) + n) / (count + 1);
    count++;
  }
}
capsule Max (Stage next) implements Stage {
  long max = Long.MIN_VALUE;
  void consume(long n) {
    next.consume(n);
    if ( n > max) max = n;
  }
}
capsule Sink(long num) implements Stage {
  long count = 0;
  void consume(long n) { count++; }
}
system Pipeline {
  Source src; Average avg; Max max; Sink snk;
  src(avg,500); avg(sum); max(snk); snk(500);
}
```

Figure 5: An Example of Pipeline Parallelism in Panini.

	Benchmark	Abbr.	Size	# Methods	# Statements
JavaGrande	Crypt	cr	C	30	1,567
	LUFact	lu	C	33	1,737
	MolDyn	md	B	35	2,417
	MonteCarlo	mc	B	110	2,252
	RayTracer	rt	B	68	2,303
	Series	ser	B	28	873
	SOR	sor	C	26	771
	SparseMatmult	sm	C	27	818
NPB	BT	–	A	44	34,804
	CG	–	A	31	3,434
	FT	–	A	37	4,831
	IS	–	A	27	2,358
	LU	–	A	44	36,736
	MG	–	A	41	7,818
	SP	–	A	44	28,098
Total				625	130,817

Figure 6: Static Characteristics of Evaluation Benchmarks

By reducing the volume and complexity of concurrent code, Panini promises to boost programmer productivity and success during concurrent program design. When rewriting these benchmarks in Panini, our student programmers gave frequent feedback that the Panini code they created was easier for them to write and understand than the explicitly-concurrent code of the original benchmarks.

6.3 Speedup

To ensure that Panini program performance is comparable to that of explicitly-parallel programs, we examined the speedup that Panini benchmarks achieve with reference to the original serial versions and original parallel versions. We define Panini speedup as:

$$Speedup = Reference\ Time \ / \ \text{Panini}\ Time$$

Because speedup is a ratio, its average is correctly computed as a geometric mean rather than as an arithmetic mean.

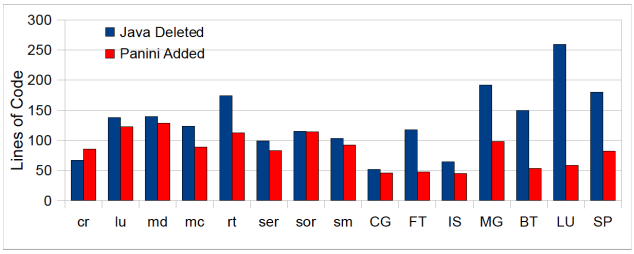

Figure 7: Change in Benchmark Code Size

General Methodology. Following the methodology of Georges *et al.*, we tested the speedup of Panini benchmarks against the originals using two different metrics [17]. *Start-up performance* is measured for "one VM invocation and a single benchmark iteration." *Steady-state performance* is measured using "one VM invocation and multiple benchmark iterations," where each benchmark is repeated within the VM until its performance reaches steady-state. Iteration time measurements are taken after steady-state is reached.

For our experiments, steady-state performance is reached when the coefficient of variation of the most recent three iteration times of a benchmark fall below 0.02. Some changes to the original benchmarks were necessary to reset static result variables, restart timers, join created threads, and cut object reference loops in support of repeated iterations in a single VM.

We repeated our comparison on multiple evaluation platforms with a number of cores ranging from 2 to 12 including an Intel Core 2 Duo with two cores, an Intel Core i5 with four cores, an AMD Opteron 2431 with six cores, and an AMD Opteron 2431 with twelve cores. We also considered several different values for the number of threads used by each benchmark in order to compare performance across a range of thread-to-core ratios.

6.3.1 *Start-up Time*

When a Java program is run only once, a previous adaptive compilation of that program is not available to the VM. If the program is fairly short, the execution time will include JIT-ing but will not be long enough to benefit much from it. In these cases, start-up time is a critical performance measure.

Research Question 2: *Do Panini programs have better start-up performance than sequential programs?*

Methodology. We took 30 start-up time samples for each benchmark on each evaluation platform and at each number of threads of interest. The number of threads ranged from "0," which implies serial execution, up to 24, which is twice the number of cores on our largest evaluation machine. Time was measured as the OS-reported CPU time used to execute the program once in a fresh VM. Because the NPB benchmarks BT, LU, and SP take orders of magnitude longer to run than all other benchmarks, we sampled them only 3 times and not on our 2-core platform. We computed 95% confidence intervals (CI's) using the Student's t-inference to verify the reliability of our observed summary statistics.

Results. At a glance, Figure 8 shows the speedup for all test cases and all Panini benchmarks with reference to SERIAL. As white changes to blue, speedup increases from 1.0 to 12.0. As white changes to yellow, speedup decreases from 1.0 to 0.0.

Analysis. Figure 8 is dominated by white and blue, which shows that Panini speedup is near 1.0 or above 1.0, in most cases. Only a few test cases show a slowdown with Panini in light yellow.

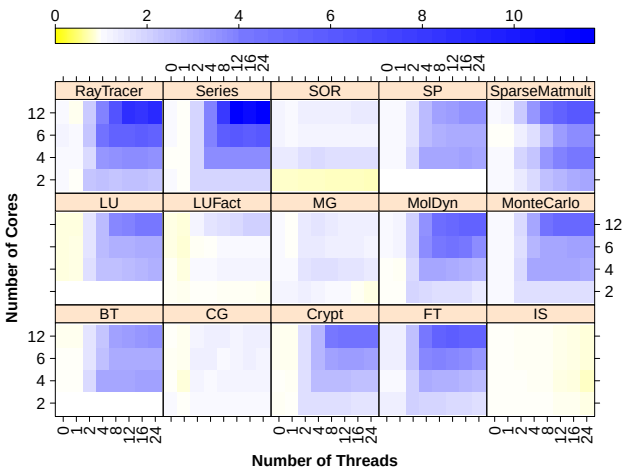

Figure 8: Start-up Time Speedup For Panini vs. SERIAL. (Stronger blues show that Panini performance is better.)

This overview gives quick visual evidence that a Panini program is as good as or better than SERIAL. Not all programs exhibit start-up time speedup; so, we examine head-to-head speedup at each experiment setting to confirm that Panini's implictly-concurrent performance is on par with PARALLEL.

Research Question 3: *Is the start-up time speedup of Panini programs comparable to that achieved by human experts?*

Methodology. To answer this research question, we used the test data that was gathered for the previous question.

Results. Figure 9 shows the head-to-head speedup of each Panini benchmark against its explicit PARALLEL or SERIAL version. The confidence intervals for start-up time speedup were very tight; so, they are not shown.

Analysis. The majority of Panini benchmarks show close start-up time performance to their explicitly-parallel counterparts. SOR from JavaGrande exhibits unpredictable behavior while the others follow consistent speedup patterns. As the number of machine cores increases, speedup tends to vary more. The 2-core machine shows speedup constricted around 1.0 while the 12-core machine shows speedup ranging more loosely between about 0.8 and 1.2.

The Panini speedup for SOR is unique and somewhat erratic. We inspected the PARALLEL code for SOR and discovered that a busy wait on shared data values is used for thread synchronization. This has rising costs as the number of threads increases; so, Panini's automatic synchronization mechanism performs much better at non-ideal thread-to-core ratios. It may be that this behavior is intended for benchmarking purposes. Other JG benchmarks make use of a TournamentBarrier, and show better behavior as thread-count increases.

Assuming a log-normal distribution for speedup, statistical t-inference yields a 99% confidence interval of (0.997,1.021) for speedup across all start-up time experiments. Repeating the t-test for an alternative hypothesis that true speedup is less than 1.0 yields p-value against the alternative of 0.9699. So, even when restricted to the same compilation decision as human experts, there is strong evidence that a Panini program with compiler-generated synchronization will execute as fast as or faster than a vetted Java program with manually-written synchronization.

6.3.2 Steady-State Time

When a Java program runs for a long time, adaptive compilation and class-loading performed during start-up have much less impact on overall performance. This can be approximated for shorter programs with repeated executions in a single VM. For long-running concurrent applications such as server programs, steady-state performance is of greater interest than start-up time.

Research Question 4: *Do Panini programs have better steady-state performance than sequential programs?*

Methodology. We took 10 steady-state time samples for each benchmark on the same combination of platforms and program thread counts as the preceding start-up time experiment. Time was measured as the average clock time of three iterations of the program inside an existing VM after steady-state performance was reached. Because the NPB benchmarks BT, LU, and SP take orders of magnitude longer to run than all other benchmarks, we sampled them 2 times and not on our 2-core evaluation platform. We again computed 95% confidence intervals.

Results. Figure 10 shows an overview of steady-state time speedup of each Panini benchmark with reference to SERIAL.

Analysis. Almost entirely white and blue, Figure 10 establishes that Panini programs are as fast or faster than SERIAL. Not all Panini benchmarks show speedup against SERIAL; so, we again more closely examine steady-state time in head-to-head comparisons of Panini benchmarks with the originals.

Research Question 5: *Is the steady-state time speedup of Panini programs comparable to that achieved by human experts?*

Methodology. To answer this research question, we used the test data that was gathered for the previous question.

Results. Head-to-head steady-state speedup for Panini with reference to PARALLEL and SERIAL is shown in Figure 12.

Analysis. As before, most Panini benchmarks perform virtually the same as their counterparts. The variation in speedup increases with the number of cores on the platform, exhibiting little difference from 1.0 on a 2-core machine and a range of about 1.3 to 0.7 on a 12-core machine. Though not shown, the 95% CI's remain between 0.6 and 1.6 for all benchmarks other than SOR, whose busy-sync behavior is again a problem for PARALLEL.

The expected steady-state speedup overall is 1.003, with a 99% confidence interval of (0.988,1.018). Though not as clearly equivalent as it was for start-up time, there remains evidence that a Panini program with automatic, implicit concurrency will execute as fast as an expert's explicitly-parallel Java program in steady-state.

6.4 Memory Overhead

Efficient use of both time and memory is essential for applications to reduce energy costs and allow other programs to be productive on a shared platform. Having shown that the size of concurrency code is decreased and that the execution time remains generally unchanged for Panini programs, we consider finally what memory overhead, if any, is introduced by Panini.

Research Question 6: *How much memory overhead is introduced by Panini and what are the primary sources of this overhead?*

Methodology. Using our 4-core evaluation platform, we logged garbage collector activity for 10 runs of each benchmark at each number of threads used in the speedup experiments of §6.3. Because the NPB benchmarks BT, LU, and SP take orders of magnitude longer to run, we sampled them only 3 times.

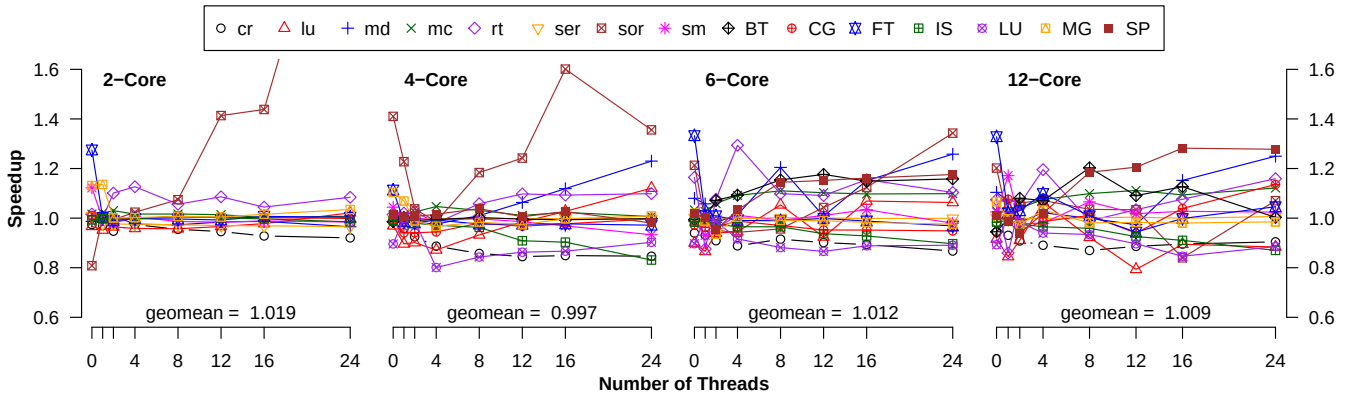

Figure 9: Start-up Time Speedup For Panini vs. Originals — Equivalent, overall, with no direct concurrent programming.

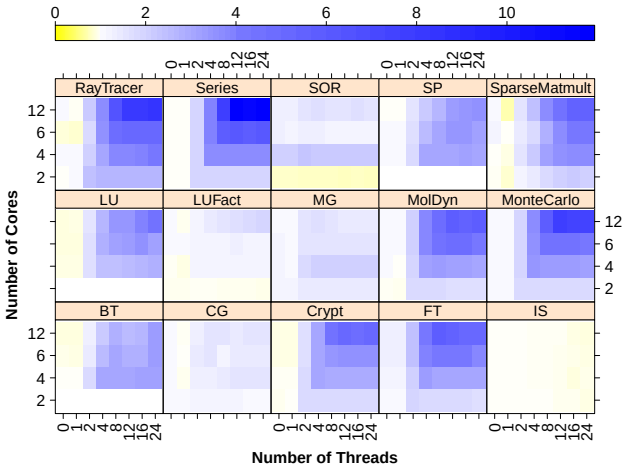

Figure 10: Steady-state Time Speedup For Panini vs. SERIAL (Stronger blues show that Panini performance is better.)

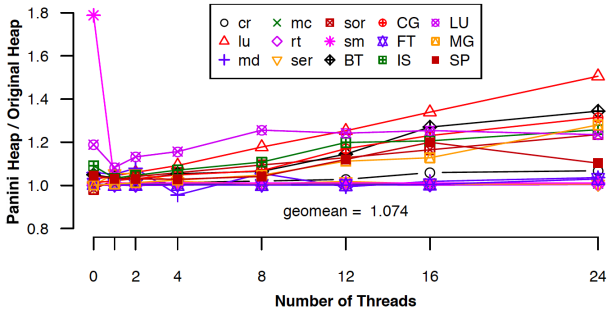

Figure 11: Change in Java Heap Used For Panini vs. Originals

Results. Figure 11 shows the head-to-head ratio of the heap memory used by Panini over PARALLEL and SERIAL on our 4-core evaluation platform.

Analysis. Panini's automatically returned future objects are small; however, when the memory used by a program is also small per synchronization, this overhead becomes more apparent. In particular, LUFact uses shared objects throughout its execution yet has several barriers per iteration.

Our Panini version of SparseMatmult was based on the PARALLEL version, which creates a copy of 3 large matrices in each worker object. The SERIAL version uses a single master copy of its matrices. So, SERIAL consumes only about half as much Java heap as the Panini version, which has 1 serial worker capsule.

6.5 Summary

Panini programs automatically achieve start-up and steady-state time performance that is likely indistinguishable from hand-crafted, explicitly-parallel programs. All in fewer lines of code and with *no* explicitly concurrent code. Compiler-generated synchronization introduces minimal memory overhead, which becomes trivial for sufficient computation per capsule method. Thus, Panini is a very attractive choice for a simple, implicitly-concurrent development language that yields no performance loss.

7. RELATED IDEAS

The idea of exposing concurrency at the boundary of components has appeared in one form or another in much previous work, e.g., guardians [25], active objects [32], and actors [3] as in Erlang [4] in Scala [20] and in AmbientTalk [12]. Many of the early proposals were in the context of distributed systems; nevertheless, they are relevant to current and ongoing efforts in the design, semantics, and implementation of concurrent language features [4, 20], including capsules. Bal, Steiner, and Tanenbaum present an excellent overview of this research [5]. The observation that improving parallelism requires better abstractions has also appeared in literature, e.g., X10 [9], Habanero [7], and Galois [22]. A primary goal of Panini is to decrease the impedance mismatch between sequential and implicitly concurrent code to help sequentially trained programmers who may struggle with concurrency. In contrast to prior work that provides an asynchronous programming model, calls to capsules are treated as logically synchronous in Panini. Many decisions in the design of Panini and its implementation are driven by this fundamental difference in philosophy.

Active Objects. Lavendar and Schmidt [23], Nierstrasz [32] among others, investigated the notion of active objects. Most recently Schäfer and Poetzsch-Heffter [37] as well as Clarke *et al.* [10] have further investigated this design. The notion of a "domain" in Hybrid [32] is closely related to a capsule. A domain encapsulates a set of objects, whereas a capsule acts as a dominator for the object graph reachable via the capsule [11]. Like a capsule, a domain can only have a single active thread of control called an "activity," which is like a token that is moved from one domain to another; in Panini a capsule logically retains an active thread of control throughout its lifetime. In Hybrid all calls (except for "activity start") were remote procedure calls with blocking send, whereas in Panini intra-capsular calls are synchronous and inter-capsular calls

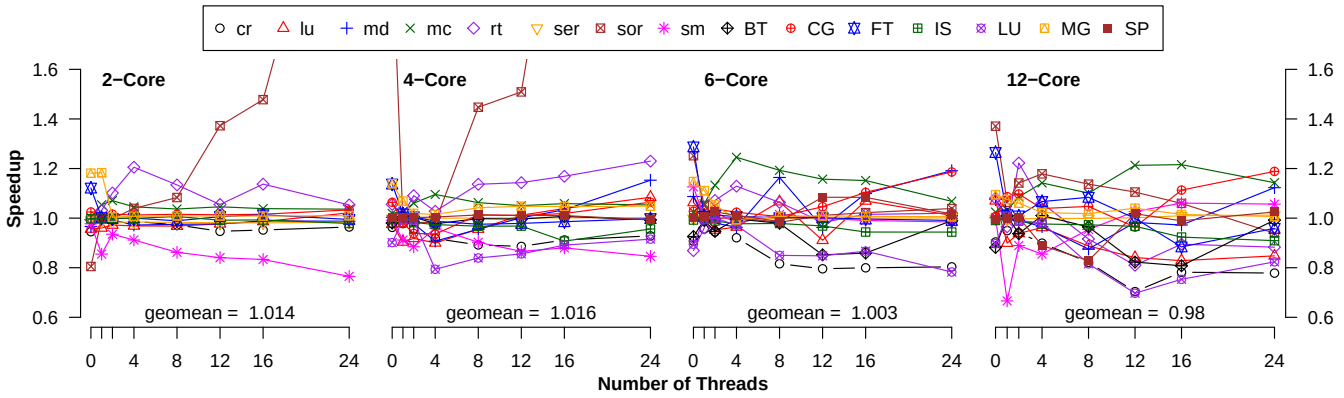

Figure 12: Steady-state Time Speedup For Panini vs. Originals — Equivalent, overall, with no direct concurrent programming.

are logically synchronous. MOAO [10] distinguishes between "active" and "passive" objects and uses futures to introduce concurrency for calls on active objects. Transparent futures using proxy objects similar to those generated by the Panini compiler are described in [35].

Argus. The notion of capsules in Panini is also related to the notion of *guardians* in Argus [25]. Like guardians, capsules encapsulate and control access to resources. However, unlike guardians, sending a request does not create a concurrent process in Panini, avoiding concurrency-related issues within a capsule's boundary. Unlike the dynamic creation of guardians in Argus, capsules are statically created and configured in Panini, which helps provide sequential consistency via a compile-time analysis.

More declarative concurrency. Unlike Jade [36] and similar approaches like Cilk [14] that focus on fine-grained parallelism, Panini combines both concurrent tasks and data into the capsule abstraction to provide coarse-grained implicit concurrency. Capsule are also more general compared to async event types that are specific to implicit-invocation design style [26].

Concurrent object-oriented languages. There is also a rich body of work on such concurrent object-oriented languages such as COOL [8], Seuss [30], Concurrent Smalltalk [44] and BETA [40]. See Papathomas's survey for an overview [33]. Unlike the works above, in Panini objects do not execute in the context of a local process, which avoids creating too many processes [33]. Panini's design also avoids the inheritance anomaly [33].

StreamIt. Capsules in Panini can also be used like filters and streams in the StreamIt language as we show in §5; however, since focus of StreamIt is streaming applications it provides some specialized (and highly optimized) features for this domain, e.g. splitters, joiners. Panini is intended to be a general purpose language in which these features can be defined by the programmer, e.g. consider a capsule `Splitter` in Figure 5 that connects to two other capsules and calls the procedure `consume` on both.

Explicitly concurrent languages. Compared to explicitly concurrent features like threads in Java and C#, and approaches like unified parallel C (UPC) [6], Titanium [43], Panini provides implicit concurrency at the capsule boundary. The advantage of Panini's approach compared to these ideas is the ease of use for a sequentially trained programmer, whereas a disadvantage is the lack of fine-grained control over concurrent structures.

Modules in sequential languages. One of the earliest languages to have the notion of modules with exported and imported interfaces was Mesa [31], a language designed and developed at Xerox Palo Alto Research Center in the 70's and early 80's. The notions of modules and signatures in ML are similar [29]. Panini's capsules are based on Mesa, but also provide implicit concurrency.

8. CONCLUSION AND FUTURE WORK

Programmers come in two shapes — those who have mastered reasoning about the interleaving of concurrent tasks and those that find it difficult. There are many in the second camp [28, 2]. There is a significant body of research on helping programmers in the first camp, typically HPC programmers [9, 7, 22]. Programmers who find concurrency hard to master can benefit from better abstractions that provide implicit concurrency [3, 25, 32]. While the jury is still out on whether the asynchronous message passing model for concurrency is the best path going forward, it is clear that the existing software development workforce was not educated to adopt that model [2].

We have argued for research on programming language mechanisms that adhere to assumptions that a sequentially trained programmer would rely upon, such as the sequential semantics of procedure calls. However, we do not promise concurrency for free; rather, we rely upon the observation that capable, sequential programmers have received some formal training in modularization-related concepts in both basic and in advanced courses. To harness this knowledge towards exposing modularization-guided implicit concurrency in program design, we have shown how to fuse the notion of a module [31, 16, 29] with the notion of an activity [32] and an ownership domain [11] such that separating program logic into capsules naturally contributes toward implicit concurrency. Capsules provide simultaneous benefits in terms of both software evolution and program performance.

In Panini, the specification of capsules by the programmer is decoupled from decisions regarding how to associate them with threads of execution. The default implementation associates a dedicated OS thread with each capsule. However, in some cases a purely thread-based execution model may introduce unnecessary overhead. Our immediate future work focuses on a compilation strategy that, based on a set of heuristics and program metrics we have identified, selects for each capsule either a thread-based implementation, a purely sequential implementation, or a task-based implementation in which one execution thread is shared by a group of similar capsules.

A key challenge in this work was to provide a safe and efficient implementation while retaining familiar semantics. We tried out the Panini language on several benchmarks, where it showed speedup comparable to manually parallelized versions, while providing the additional benefit of simpler, more modular code. Since concurrency is implicit, Panini hides these concerns from programmers, allowing them to focus on the task at hand. By bringing most benefits of actor-like abstractions to sequentially-trained programmers via a familiar synchronous model, Panini could help these programmers overcome the tough hurdle of concurrent program design.

Acknowledgements

This work was supported in part by the NSF under grants CCF-08-46059, and CCF-11-17937. We thank Mehdi Bagherzadeh, Ferosh Jacob, Gary T. Leavens, Robyn Lutz, Sean Mooney, Henrique Rebêlo, Laurel Tweed, and David Weiss for comments.

9. REFERENCES

[1] Apache Tomcat. `http://tomcat.apache.org/`.

[2] ACM/IEEE-CS Joint Task Force. Computer science curricula 2013 (CS2013). Technical report, ACM/IEEE, 2012.

[3] G. Agha. Actors: a model of concurrent computation in distributed systems. Technical Report AITR-844, 1985.

[4] J. Armstrong, R. Williams, M. Virding, and C. Wikstroem. *Concurrent Programming in ERLANG*. Prentice-Hal, 1996.

[5] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum. Programming languages for distributed computing systems. *ACM Comput. Surv.*, 21(3):261–322, Sept. 1989.

[6] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren. *Introduction to UPC and language specification*. Center for Computing Sciences, Institute for Defense Analyses, 1999.

[7] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. Habanero-Java: the new adventures of old X10. In *PPPJ*, pages 51–61, 2011.

[8] R. Chandra, A. Gupta, and J. L. Hennessy. COOL: An object-based language for parallel programming. *Computer '94*, 27.

[9] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05*.

[10] D. Clarke, T. Wrigstad, J. Östlund, and E. B. Johnsen. Minimal ownership for Active Objects. In *APLAS '08*.

[11] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA '98*.

[12] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. DâĂŹHondt, and W. De Meuter. Ambient-oriented programming in AmbientTalk. *ECOOP*, pages 230–254, 2006.

[13] P. C. Diniz and M. C. Rinard. Commutativity analysis: a new analysis technique for parallelizing compilers. *ACM Trans. Program. Lang. Syst.*, 19(6):942–991, 1997.

[14] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI*, pages 212–223, 1998.

[15] M. Frumkin, M. Schultz, H. Jin, and J. Yan. Implementation of the NAS Parallel Benchmarks in Java. 2002.

[16] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *PLDI*, pages 1–11, 2003.

[17] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous Java performance evaluation. In *OOPSLA '07*, pages 57–76.

[18] C. Grothoff, J. Palsberg, and J. Vitek. Encapsulating objects with confined types. *ACM Trans. Program. Lang. Syst.*, 29(6), Oct. 2007.

[19] P. Haller and M. Odersky. Capabilities for uniqueness and borrowing. In *ECOOP'10*.

[20] P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci. '09*, 410.

[21] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, Aug. 1978.

[22] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *PLDI '07*.

[23] R. Lavender and D. Schmidt. Active object – an object behavioral pattern for concurrent programming. In *Pattern languages of program design 2*, 1996.

[24] D. Lea. *Concurrent Programming in Java*. Addison-Wesley, Boston, MA, USA, 1999.

[25] B. Liskov and R. Scheifler. Guardians and Actions: Linguistic support for robust, distributed programs. *TOPLAS '83*, 5.

[26] Y. Long, S. L. Mooney, T. Sondag, and H. Rajan. Implicit invocation meets safe, implicit concurrency. In *GPCE*, pages 63–72. ACM, 2010.

[27] G. Losa, V. Kumar, H. Andrade, B. Gedik, M. Hirzel, R. Soulé, and K.-L. Wu. CAPSULE: language and system support for efficient state sharing in distributed stream processing systems. In *DEBS '12*.

[28] D. Meder, V. Pankratius, and W. F. Tichy. Parallelism in curricula an international survey. Technical report, University of Karlsruhe, 2008.

[29] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1990.

[30] J. Misra. A simple, object-based view of multiprogramming. *Form. Methods Syst. Des.*, 20(1):23–45, 2002.

[31] J. G. Mitchell, W. Maybury, and R. Sweet. Mesa language manual. Technical Report CSL-79-3, Xerox Palo Alto Research Center, 1979.

[32] O. M. Nierstrasz. Active objects in Hybrid. In *OOPSLA '87*.

[33] M. Papathomas. Concurrency in object-oriented programming languages. In *Object-oriented software composition*. Prentice Hall, 1995.

[34] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, Dec. 1972.

[35] P. Pratikakis, J. Spacco, and M. Hicks. Transparent proxies for Java futures. In *OOPSLA '04*, pages 206–223.

[36] M. C. Rinard and M. S. Lam. The design, implementation, and evaluation of Jade. *TOPLAS '98*, 20.

[37] J. Schäfer and A. Poetzsch-Heffter. JCoBox: generalizing active objects to concurrent components. In *ECOOP'10*.

[38] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture, Volume 2*. Wiley, New York, NY, USA, 2000.

[39] B. Selic, G. Gullekson, and P. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, 1994.

[40] B. Shriver and P. Wegner. Research directions in object-oriented programming, 1987.

[41] L. Smith, J. Bull, and J. Obdrizalek. A parallel Java Grande benchmark suite. In *ACM/IEEE Conf. on Supercomputing*, pages 6–6, 2001.

[42] R. Vallee-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot-a Java bytecode optimization framework. In *the conference of the Centre for Advanced Studies on Collaborative research*, page 13, 1999.

[43] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, et al. Titanium: A high-performance Java dialect. *Concurrency Practice and Experience*, 10(11-13):825–836.

[44] A. Yonezawa and M. Tokoro. Object-oriented concurrent programming, 1990.