Tasks: Language Support for Event-driven Programming

Jeffrey Fischer Rupak Majumdar Todd Millstein
Department of Computer Science, University of California, Los Angeles
{fischer,rupak,todd}@cs.ucla.edu

ABSTRACT

The *event-driven* programming style is pervasive as an efficient method for interacting with the environment. Unfortunately, the event-driven style severely complicates program maintenance and understanding, as it requires each logical flow of control to be fragmented across multiple independent *callbacks*.

We propose *tasks* as a new programming model for organizing event-driven programs. Tasks are a variant of cooperative multi-threading and allow each logical control flow to be modularized in the traditional manner, including usage of standard control mechanisms like procedures and exceptions. At the same time, by using method annotations, task-based programs can be automatically and modularly translated into efficient event-based code, using a form of continuation passing style (CPS) translation. A linkable scheduler architecture permits tasks to be used in many different contexts.

We have instantiated our model as a backward-compatible extension to Java, called TaskJava. We illustrate the benefits of our language through a formalization in an extension to Featherweight Java, and through a case study based on an open-source web server.

1. INTRODUCTION

A wide variety of applications, from high-performance servers to enterprise applications to GUIs to embedded systems, rely on an *event-based* programming style. Event-driven programming implements a stylized programming idiom where programs use non-blocking I/O operations, and the programmer breaks the computation into fine-grained *callbacks* (or *event handlers*) that are each associated with the completion of an I/O call (or *event*). This approach permits the interleaving of many simultaneous logical tasks with minimal overhead, under the control of an application-level cooperative scheduler. Each callback executes some useful work and then either schedules further callbacks, contingent upon later events, or invokes a *continuation*, which resumes the control flow of its logical caller. The event-driven style has been demonstrated to achieve high throughput in server applications [16, 22], resource-constrained embedded devices [8], and business applications [3].

Unfortunately, programming with events comes at a cost: event-driven programs are extremely difficult to understand and maintain. Each logical unit of work must be manually broken into multiple callbacks scattered throughout the program text. This manual code decomposition is in conflict with higher-level program structuring. For example, calls do not return directly to their callers, so it is difficult to make use of procedural abstraction as well as a structured exception mechanism.

Threads represent an alternative programming model commonly used to interleave multiple flows of control. Since each thread maintains its own call stack, standard program structuring may be naturally used, unlike in the event-driven style. However, threads have disadvantages as well, including the potential for race conditions and deadlocks, as well as high memory consumption [21]. Within the systems research community, there is currently no agreement that one approach is better than the other [16, 20, 21, 1]. In addition, in some contexts, threads either cannot be used at all (such as within some operating system kernels) or can only be used in conjunction with events (such as thread-pooled servers for Java Servlets [3]). Thus, we believe that events are here to stay and are an important target for programming language support.

In this paper, we introduce *tasks* as a new programming language construct for event-driven applications. A task, like a thread, encapsulates an independent unit of work. The logical control flow of each unit of work is preserved, and standard program structures like procedures and exceptions may be naturally used. However, unlike threads, tasks can be automatically implemented by the compiler in an event-driven style, thereby obtaining the low-overhead and high-throughput advantages of events. Our compilation strategy is a restricted form of *continuation-passing style* (CPS), a well-studied compiler transformation that is popular for functional programming languages [2]. We have instantiated our concept of tasks as a backward-compatible extension to Java called TaskJava and have implemented the TaskJava compiler in the Polyglot compiler framework [15].

Tasks are a variant of *cooperative multitasking*, a form of interleaved execution where context switches only occur upon explicit yields. TaskJava provides several technical contributions over existing cooperative multitasking systems.

- First, TaskJava's modular static type system tracks the set of methods whose execution might yield, requiring each to have a new `async` modifier. Aside from serving as useful documentation for clients, these annotations tell the compiler exactly where CPS translation is required (and where it is not). In contrast, existing systems must allow for yields anywhere, which requires either low-level stack manipulation (which is not possible in virtual machine-based languages), maintaining the stack on the heap, or copying the stack onto the heap as necessary.
- Second, TaskJava is scheduler-independent: TaskJava programs can be "linked" against any scheduler that provides the semantics of a new `wait` primitive, which yields control to the scheduler. This design permits the benefits of tasks to be accrued across multiple event domains (GUI events, web server events, etc.). Prior approaches are tied to a specific scheduler and notion of events.
- Finally, TaskJava properly handles the interactions of `wait` with Java language features including checked exceptions and method overriding, and TaskJava's implementation adheres to the constraints imposed by the Java virtual machine.

We evaluate TaskJava in two ways. First, we have formalized the language and its compilation strategy via CoreTaskJava (CTJ), a core language that extends Featherweight Java [11]. We provide a direct operational semantics for CTJ, whereby `wait` calls block until an appropriate event is signaled, as well as a translation relation from CTJ to Featherweight Java, which formalizes the continuation-passing transformation performed by the TaskJava compiler. We have proven CTJ's type system sound, and as corollaries of this property, we show that a well-typed CTJ program is guaranteed to avoid two significant classes of errors that may occur in event-driven programs, which we dub the *lost continuation* and *lost exception* problems.

The lost continuation problem occurs when a callback has an execution path in which the callback's continuation is neither invoked nor passed along to the next callback in the event chain. A lost continuation causes the intended sequential behavior of the program to be broken, often producing errors that are difficult to trace to their source. The lost exception problem occurs when an exceptional condition produced by a callback is not properly handled by the subsequent continuation, potentially causing the program to crash or continue executing in undefined ways.

Second, to evaluate TaskJava's benefits in practice, we extended Fizmez [4], an open source web server, to use interleaved computation. We implemented two versions: one using a manual event-driven style and the other using TaskJava. The TaskJava version maintains the same structure as the original web server, while the event-driven version requires its logic to be fragmented across many callback classes, obscuring the control flow. At the same time, the TaskJava version pays only a modest performance penalty versus the hand-coded one.

While many of the ingredients of our approach have been present in the programming languages literature, we believe that our design and implementation of TaskJava shows how these techniques can be combined effectively to provide demonstrable benefits in an important class of systems applications.

The rest of the paper is organized as follows. In section 2, we informally present tasks and TaskJava by example and contrast with event-driven programs. In section 3, we describe the CoreTaskJava formalisms. In section 4 we overview the implementation of the TaskJava compiler, and in section 5 we discuss our web server case study. We then survey related work in section 6 and conclude in section 7.

2. PROGRAMMING WITH TASKS

2.1 Event-driven Programming

The event-driven programming style is frequently used in server programming in conjunction with *non-blocking I/O*. Non-blocking I/O libraries (such as Java's NIO package) permit input/output operations to be scheduled so that they do not block inside the operating system. Thus, independent requests can be executed in an overlapping fashion without preemptive multi-threading.

Non-blocking I/O libraries generally provide two types of calls. First, a *selection* call permits waiting for one or more channels/sockets to be ready for a new request. Examples include the Unix `select` call and the `Selector.select` method in Java's NIO package. Second, calls are provided to initiate the actual I/O operations (e.g., read and write) once the associated channel has become ready. Unlike a standard blocking read or write request, non-blocking read and write calls generally complete only the portion of a request that can be accomplished without blocking.

Selection calls are usually incorporated into a user-defined scheduler framework. Rather than calling the selection API directly, clients of the scheduler *register* to receive notification when the state of a given channel/socket changes. The scheduler then calls the selection API on behalf of all clients, notifying clients of events via callbacks. The control logic of each client is broken across a series of these callbacks and thus is interleaved by the scheduler with the callbacks of the other clients. This approach permits independent activities to cooperatively share the process's CPU and I/O resources.

2.2 Event-driven Writer

Figure 1 shows a simple program fragment, written in an event-driven style, which sends a buffer of data on a non-blocking channel. `Writer`'s `run` method first obtains the data to be written (not shown), which is stored in a buffer `buf`. The method then calls `Scheduler.register`, which registers a callback to be invoked upon a write-ready or error event on the channel `ch`. The `run` method returns immediately after the `register` call — execution of this logical control flow must be resumed by the scheduler.

When an event occurs on channel `ch`, the scheduler invokes the `run` method of the callback it was given (an instance of `WriteReadyCB`). This method performs a write on the channel and then checks to see if more data needs to be written. If so, the callback re-registers itself with the scheduler. Otherwise, it calls the *continuation* method `restOfRun` on the original `Writer` object, which resumes the logical control flow. If an error event is returned by the scheduler, the callback prints an error message. Since no callback is registered or continuation method invoked, the logical control flow is effectively terminated in that case.

Even this simple example illustrates the violence that the event-driven style does to a program's natural flow of control. The code in `restOfRun` logically follows the buffer write, but they must be unnaturally separated because of

```
01 public class Writer {
02   ByteChannel ch;
03   ...
04   /* The main body of our task */
05   public void run() {
06     // get the data to write
07     ByteBuffer buf = ...;
08     /* wait for channel to be ready */
09     Scheduler.register(ch, Event.WRITE_RDY_EVT,
10                        Event.ERR_EVT,
11                        new WriteReadyCB(ch, buf, this));
12   }
13   /* After the write has completed, we continue with what
14      we were doing.  The event-driven style forces this
15      in a separate method.  */
16   public void restOfRun() { ...  }
17   /* Callback which does the write and then registers
18      itself if there still is data left */
19   class WriteReadyCB implements Callback {
20     ...
21     public WriteReadyCB(ByteChannel ch, ByteBuffer buf,
22                         WriteTask caller) {...}
23     public void run(Event e) {
24       switch (e.type()) {
25        case Event.WRITE_RDY_EVT:
26         ch.write(buf);
27         if (buf.hasRemaining())
28           Scheduler.register(ch, Event.WRITE_RDY_EVT,
29                              Event.ERR_EVT, this);
30         else caller.restOfRun();
31         break;
32        default:
33         System.out.println(e.toString());
34       } } } }
```

Figure 1: Implementation of an event-driven writer

```
public class WriterTask implements Task {
  ByteChannel ch; ...
  /* The main body of our task */
  public void run() {
    // get the data to write
    ByteBuffer buf = ...;
    // write the buffer
    do {
      Event e =
        Scheduler.wait(ch, Event.WRITE_RDY_EVT,
                       Event.ERR_EVT);
      switch (e.type()) {
       case Event.WRITE_RDY_EVT:
        ch.write(buf);
        break;
       default:
        System.out.println(e.toString());
        return;
      }
    } while (buf.hasRemaining())
    /* the write is completed, so continue
       with the rest of the method */
    ...
  } }
```

Figure 2: Implementation of the writer in TaskJava

the intervening event registration. Similarly, performing the buffer write conceptually involves a loop that writes to the channel until the entire buffer has been written. In `WriteReadyCB.run`, this loop must be unnaturally simulated by having the callback re-register itself repeatedly.

Without care, it is easy for a programmer to introduce errors that go undetected. For example, if the call to `restOfRun` is accidentally omitted on line 30, then `Writer`'s control flow will never be resumed after the write. If the re-registration on line 28 is omitted, the write will not even be completed. These are examples of *lost continuation* problems.

2.3 Task-based Writer

Figure 2 shows a TaskJava implementation of the same program fragment. The class `WriterTask` is declared as a *task* by implementing the `Task` interface. Tasks are the unit of concurrency in TaskJava, serving a role similar to that of a thread in multi-threaded systems. Instances of a task may be created by using the `spawn` keyword, which is followed by a call to one of the task's constructors (e.g., `spawn WriterTask()`). A `spawn` causes a new instance of the task to be created and schedules the instance's `run` method for execution.

The logical control flow of our writer is now entirely encapsulated in `WriterTask`'s `run` method. The `register` call from `Writer` is replaced with a `wait` call, which conceptually blocks until one of the requested events has occurred, returning that event. In this way, explicit callback functions are not needed, so the code need not be unnaturally fragmented across multiple methods (e.g., `restOfRun`). Similarly, the logic of the buffer write can be implemented using an ordinary `do-while` loop.

The ability to use traditional program structures to express the control flow of a task avoids the lost continuation problem. The programmer need not manually ensure that the appropriate callback is registered or continuation method is invoked on each path. This work is done by the TaskJava compiler, which translates the `WriterTask` into a continuation-passing style that is very similar to the `Writer` code in figure 1. In particular, `wait` calls are translated to `register` calls, and the portion of the `run` method after the `wait` call is placed in a separate continuation method.

TaskJava allows programmers to define their own scheduler class, their own event type and implementations, and their own type of event "tags" (e.g., `WRITE_RDY_EVT`). As long as the scheduler defines a `register` method for event registrations, TaskJava allows the scheduler to be treated as if it has a corresponding `wait` method. This approach allows existing scheduler frameworks to obtain the benefits of TaskJava without any modification. For example, the scheduler used in the manual version in figure 1 may be reused in figure 2. This approach also allows multiple scheduler frameworks to be used in the same program.

2.4 Asynchronous Methods

The TaskJava implementation of our writer also naturally supports procedural abstraction. For example, figure 3 shows a refactoring of our task whereby the code to write the buffer is encapsulated in its own method, allowing that code to be easily used by multiple clients. Implementing this `write` method in the manual event-driven version of the code would be much more unwieldy, because event-driven programming breaks the standard call-return discipline. To return control back to caller, therefore, such a `write` method would have to take an explicit continuation argument to be

```
public class WriterTask implements Task{
  ByteChannel ch; ...
  /* The main body of our task */
  public void run() {
    // get the data to write
    ByteBuffer buf = ...;
    try {
      write(ch, buf);
    } catch (IOException e) {
       System.out.println(e.getMessage());
    }
  }
  public async void write(ByteChannel ch, ByteBuffer b)
    throws IOException {
    do {
      Event e = Scheduler.wait(ch, Event.WRITE_RDY_EVT,
                                 Event.ERR_EVT);
      switch (e.type()) {
       case Event.WRITE_RDY_EVT:
        ch.write(buf);
        break;
       default:
        throw new IOException(e.toString());
    } } while (buf.hasRemaining())
  } }
```

Figure 3: Use of asynchronous methods in TaskJava

called upon completion of the write.

Figure 3 also shows that tasks are compatible with regular Java exception handling. The `write` method throws an `IOException` when an error event is signaled, allowing its caller to handle the error as appropriate. As in Java, the TaskJava compiler ensures that all (checked) exceptions are caught. In contrast, a manual event-driven version of the `write` method would have to signal the error to its caller in an *ad hoc* manner, for example by setting a flag or invoking a special `error` continuation method. This approach is tedious and loses the static assurance that all exceptions are caught, resulting in the potential for *lost exception* problems at run time.

Methods that directly or transitively invoke `wait`, like our `write` method, are called *asynchronous* methods. Such methods (other than a task's distinguished `run` method) must have the `async` modifier. To programmers, this modifier indicates that the method has the potential to block. To the TaskJava compiler, this modifier indicates that the method must be translated into continuation-passing style.

Asynchronous methods, like regular Java methods, interact naturally with inheritance. For example, a subclass of `WriterTask` can override the `write` method to support a different or enhanced algorithm for writing a buffer. Making the same change to the `Writer` class in figure 1 is less natural due to the fragmentation inherent in the event-driven style. For example, modifications to the logic for writing the buffer would require a new subclass of the `WriteReadyCB` callback class, and this modification then requires a new subclass of `Writer` whose `run` method creates the new kind of callback.

3. FORMALIZING TASKS

We formalize TaskJava and prove our theorems in a core calculus extending Featherweight Java (FJ) [11]. We do this in two steps: first, we define FJ^+, an extension to FJ with exceptions and a built-in set datatype; second, we define CoreTaskJava (CTJ), which extends FJ^+ with support for tasks and events. This section overviews our formalism and associated metatheory; full details are available in a companion technical report [6].

3.1 FJ^+

The syntax of FJ^+ is described in Figure 4. An FJ^+ program consists of a *class table* mapping class names to classes, and an initial expression. As in FJ, the notation $\bar{\texttt{D}}$ denotes a sequence of elements from domain `D`. A class consists of a list of *fields*, a *constructor*, and a list of *methods*. We assume there exist built-in classes `Object` and `Throwable`. The class `Throwable` is a subclass of `Object` and both have no fields and no methods. The metavariable `C` ranges over class names, `f` over field names, `m` over method names, and `x` over formal parameter names. An expression is either a formal, a field access, a method call, an object allocation, a type cast, a set, the `throw` of an exception, or a `try` expression. The only values are objects (instances of classes) and set values.

Exceptions have a significant impact on the semantics of TaskJava and our translation strategy. Thus, including them enables us to capture more of the issues that must be addressed by a full implementation. Sets are used extensively in our modeling of the event scheduler. The addition of first-class set literals simplifies our notation without changing the semantics of our formalization.

Figure 5 shows the computation rules from the small-step operational semantics of FJ^+. The first three rules are taken directly from Featherweight Java and define the evaluation of fields, method calls, and constructors. Rule E-Set$_1$ defines the computation of casts for set literals. Given a cast to `Set<T>`, where T is an element type, if the type of each element is a subtype of T, then the cast can be erased. The next seven rules define the semantics of nested `throw` expressions — the entire expression is replaced by the enclosed `throw`. This propagates exceptions up the dynamic evaluation context. Finally, rules E-Try$_1$, E-Try$_2$, and E-Try$_3$ define the semantics of `try..catch` expressions. If the body of a `try` block evaluates to a non-exception value, the `catch` block is dropped. If the body of the `try` block evaluates to a caught exception, the `catch` block is evaluated. If the `try` block evaluates to an uncaught exception, the entire `try..catch` expression is replaced with a `throw` of the exception. The congruence rules for FJ^+ are straightforward and may be found in the technical report.

For the static semantics, the typing judgment for expressions now has the form $\Gamma \vdash e : T|\bar{\tau}$, where Γ is a type environment as usual and $\bar{\tau}$ is set of exception classes that may escape the expression. This set of exceptions is used by the rules in order to ensure that every method has an appropriate `throws` clause, as in Java. Also, the set of exception classes resulting from typechecking a program's initial expression must be empty. These checks ensure that all run-time exceptions are caught.

3.2 CoreTaskJava

CoreTaskJava (CTJ) is a core calculus that extends FJ^+ with support for event-driven programming. The syntax extensions for CoreTaskJava (CTJ) are shown in Figure 4: a `spawn` expression for creating tasks; a `wait` primitive, which accepts a set of events and blocks until one of them occurs; and asynchronous methods via the `async` modifier. A task in CTJ subclasses from a built-in class `Task`:

FJ$^+$	Program	P	::=	$\overline{CL}$ `return e;`
	Class List	CL	::=	`class C extends C {`$\bar{\texttt{T}}\ \bar{\texttt{f}}$`; K `$\bar{\texttt{M}}$`}`
	Constructor	K	::=	`C(`$\bar{\texttt{T}}\ \bar{\texttt{f}}$`) { super(`$\bar{\texttt{f}}$`); this.`$\bar{\texttt{f}}$`= `$\bar{\texttt{f}}$`;}`
	Method	M	::=	`T m(`$\bar{\texttt{T}}\ \bar{\texttt{x}}$`) throws `$\bar{\texttt{C}}$` {return e;}`
	Type	T	::=	`C | Set<T>`
	Expressions	$\texttt{e}_{base}$	::=	`x | e.f | e.m(`$\bar{\texttt{e}}$`) | new C(`$\bar{\texttt{e}}$`) | (C)e`
				`| {`$\bar{\texttt{e}}$`} | throw e | try {e} catch (C x) {e}`
	Values	v	::=	`new C(`$\bar{\texttt{v}}$`) | {`$\bar{\texttt{v}}$`}`
CTJ	Async methods	M	::=	`... | async T m(`$\bar{\texttt{T}}\ \bar{\texttt{x}}$`) throws `$\bar{\texttt{C}}$` {return e;}`
		e	::=	`... | spawn C(e) | wait(e)`

Figure 4: Syntax of FJ$^+$ and CTJ.

$\boxed{e \longrightarrow e'}$

$$\frac{fields(C) = \bar{\texttt{T}}\ \bar{\texttt{f}}}{(\texttt{new } C(\bar{\texttt{v}})).f_i \longrightarrow v_i}\ (\text{E-Fd}_1) \qquad \frac{mbody(m, C) = (\bar{\texttt{x}}, e_0)}{(\texttt{new } C(\bar{\texttt{v}})).m(\bar{\texttt{v}}_e) \longrightarrow [\bar{\texttt{v}}_e/\bar{\texttt{x}}, \texttt{new } C(\bar{\texttt{v}})/\texttt{this}]e_0}\ (\text{E-App}_1) \qquad \frac{C <: C'}{(C')(\texttt{new } C(\bar{\texttt{v}})) \longrightarrow \texttt{new } C(\bar{\texttt{v}})}\ (\text{E-Cst}_1)$$

$$\frac{\emptyset \vdash \bar{\texttt{v}} : \bar{\texttt{T}}|\tau \qquad \forall \texttt{T}_i \in \bar{\texttt{T}}.\texttt{T}_i <: \texttt{T}}{(\texttt{Set} < \texttt{T} >)\{\bar{\texttt{v}}\} \longrightarrow \{\bar{\texttt{v}}\}}\ (\text{E-Set}_1) \qquad \texttt{new } C(\bar{\texttt{v}}, \texttt{throw } v_e, \bar{\texttt{e}}) \longrightarrow \texttt{throw } v_e\ (\text{E-Th}_1) \qquad (\texttt{throw } v_e).m(\bar{\texttt{e}}) \longrightarrow \texttt{throw } v_e\ (\text{E-Th}_2)$$

$$v.m(\bar{\texttt{v}}, \texttt{throw } v_e, \bar{\texttt{e}}) \longrightarrow \texttt{throw } v_e\ (\text{E-Th}_3) \qquad \{\bar{\texttt{v}}, \texttt{throw } v_e, \bar{\texttt{e}}\} \longrightarrow \texttt{throw } v_e\ (\text{E-Th}_4) \qquad (\texttt{throw } v_e).f \longrightarrow \texttt{throw } v_e\ (\text{E-Th}_5)$$

$$(C)(\texttt{throw } v_e) \longrightarrow \texttt{throw } v_e\ (\text{E-Th}_6) \qquad \texttt{throw throw } v_e \longrightarrow \texttt{throw } v_e\ (\text{E-Th}_7) \qquad \texttt{try } \{v;\} \texttt{ catch } (C_e\ x)\ \{e;\} \longrightarrow v\ (\text{E-Try}_1)$$

$$\frac{v = \texttt{new } C(\bar{\texttt{v}}) \qquad C <: C_e}{\texttt{try } \{\texttt{throw } v;\} \texttt{ catch } (C_e\ x)\ \{e;\} \longrightarrow [v/x]e}\ (\text{E-Try}_2) \qquad \frac{v = \texttt{new } C(\bar{\texttt{v}}) \qquad C \not<: C_e}{\texttt{try } \{\texttt{throw } v;\} \texttt{ catch } (C_e\ x)\ \{e;\} \longrightarrow \texttt{throw } v}\ (\text{E-Try}_3)$$

Figure 5: Computation rules for FJ$^+$.

```
class Task extends Object {
  Object run(Object retVal) { return new Object(); } }
```

A task's `run` method contains the body of the task and is invoked after the task is spawned.

Figure 6 lists the operational rules for CoreTaskJava. The metavariable E represents an *evaluation context* (i.e., an expression containing a *hole*), which formalizes the next subexpression to be evaluated. We write $E[e]$ to represent an expression E with a subexpression e in the next evaluation position.

The operational semantics of CTJ is given with respect to a *program state*, which consists of (1) an expression representing the in-progress evaluation of the currently executing task, and (2) a set $\mathcal{B}$ of (`Set<Event>`, $E[]$) pairs representing the events that each task is blocked on and the task's current continuation (a CTJ evaluation context). Such a program state is denoted $e|\mathcal{B}$.

We define a two-level operational semantics for CTJ programs. First, the $\longrightarrow_c$ relation on expressions extends the FJ$^+$ $\longrightarrow$ relation with congruence rules to evaluate the arguments of `wait` and `spawn` calls, as shown at the top of the figure. Second, we define a new relation $\Longrightarrow_c$ on program states, which makes use of the $\longrightarrow_c$ relation. We use this two-level approach to distinguish the evaluation steps that have no interaction with the currently blocked tasks (the $\longrightarrow_c$ relation) from those that do (the $\Longrightarrow_c$ relation).

Rule E_c-Con is used to evaluate the current task as much as possible via $\longrightarrow_c$. When this rule is no longer applicable, there are three cases. If the current task contains a `wait` call to be evaluated next, rule E_c-Wait adds the blocked task to the program state and replaces the current expression with a new dummy value. Rule E_c-Spn models a `spawn` call similarly, treating the new task's execution (via `run`) as blocked on an empty event set. Finally, if the current task has been evaluated to a value, a blocked task is nondeterministically selected from the program state and its continuation is resumed, nondeterministically passing one of the waited-for events (rule E_c-Run). A `NullEvent` is passed if the task waits for an empty set of events (rule E_c-$\eta_\emptyset$Run).

The key enhancement for static typechecking is the handling of asynchronous methods. In particular, the static type system ensures that any method that potentially invokes `wait` or a method declared `async` is either a task's `run` method or is itself declared `async`. Further, if a method m_1 overrides another method m_2, then either both or neither must be declared `async`.

3.3 Type Soundness for CoreTaskJava

We have proven type soundness for CTJ. We state the main theorems here; full proofs are provided in [6].

THEOREM 1. [$\Longrightarrow_c$ *Subject Reduction] If* $\vdash e : \texttt{T}_e|\bar{\tau}_e$ *and* $\mathcal{B}$ *OK and* $e|\mathcal{B} \Longrightarrow_c e'|\mathcal{B}'$*, then* $\vdash e' : \texttt{T}_{e'}|\bar{\tau}_{e'}$ *and* $\mathcal{B}'$ *OK.*

The above theorem is a variant of the standard "subject reduction" theorem, which says that evaluation preserves typing. However, in our case the type of the expression resulting from a step of $\Longrightarrow_c$ may be totally unrelated to the type of the original expression. This can occur because the two expressions may derive from two completely independent tasks. For example, if the step uses rule E_c-Run, then a task that has finished evaluating is replaced by the current continuation for one of the blocked tasks in the program state.

The progress theorem states that well-typed CTJ programs cannot get stuck, except when a bad cast occurs.

THEOREM 2. [$\Longrightarrow_c$ *Progress] Suppose* $\vdash e : T|\bar{\tau}$ *and* $\mathcal{B}$ *OK. Then one of the following must be true:*

$$\boxed{e \longrightarrow_c e'}$$

$$\frac{e_0 \longrightarrow_c e_0'}{\texttt{wait}\ e_0 \longrightarrow_c \texttt{wait}\ e_0'}\ (E_c\text{-}\textsc{Wt}) \qquad \texttt{wait throw}\ v \longrightarrow_c \texttt{throw}\ v\ (E_c\text{-}\textsc{WtT}) \qquad \frac{e_0 \longrightarrow_c e_0'}{\texttt{spawn}\ e_0 \longrightarrow_c \texttt{spawn}\ e_0'}\ (E_c\text{-}\textsc{Sp}) \qquad \texttt{spawn throw}\ v \longrightarrow_c \texttt{throw}\ v\ (E_c\text{-}\textsc{SpT})$$

$$\boxed{e|\mathcal{B} \Longrightarrow_c e'|\mathcal{B}'}$$

$$\frac{e \longrightarrow_c e'}{e|\mathcal{B} \Longrightarrow_c e'|\mathcal{B}}\ (E_c\text{-}\textsc{Con}) \qquad \frac{w = \texttt{wait}\ \{\overline{\mathtt{v}}\}}{E[w]|\mathcal{B} \Longrightarrow_c \texttt{new Object()}|\mathcal{B} \cup (\{\overline{\mathtt{v}}\}, E[])}\ (E_c\text{-}\textsc{Wait}) \qquad \begin{array}{c} E[\texttt{spawn}\ C(\overline{\mathtt{v}})]|\mathcal{B} \Longrightarrow_c \\ E[\texttt{new}\ C(\overline{\mathtt{v}})]|\mathcal{B} \cup (\emptyset, (\texttt{new}\ C(\overline{\mathtt{v}}))\texttt{.run([])}) \end{array}\ (E_c\text{-}\textsc{Spn})$$

$$\frac{(\{\overline{\mathtt{v}}\}, E[]) \in \mathcal{B} \qquad \eta \in \{\overline{\mathtt{v}}\}}{v|\mathcal{B} \Longrightarrow_c E[\eta]|\mathcal{B} \setminus (\{\overline{\mathtt{v}}\}, E[])}\ (E_c\text{-}\textsc{Run}) \qquad \frac{(\emptyset, E[]) \in \mathcal{B} \qquad \eta_0 = \texttt{new NullEvent()}}{v|\mathcal{B} \Longrightarrow_c E[\eta_0]|\mathcal{B} \setminus (\emptyset, E[])}\ (E_c\text{-}\eta_\emptyset\textsc{Run})$$

Figure 6: Operational Semantics of CTJ.

- *e is a value and $\mathcal{B} = \emptyset$.*
- *e is of the form $E[(T)v]$, where $E[]$ is an evaluation context and v is a value whose type is not a subtype of T.*
- *There exists an expression e' and set of blocked tasks $\mathcal{B}'$ such that $e|\mathcal{B} \Longrightarrow_c e'|\mathcal{B}'$.*

Now we can state the main result for this section, namely type soundness for CTJ. Let $\Longrightarrow_c^*$ denote the reflexive, transitive closure of the $\Longrightarrow$ relation.

THEOREM 3. [*Type Soundness*] *If $P_c = \overline{CL}$ `return` e is a CTJ program and $\vdash^c P_c$ OK, then one of the following must be true:*

- *$e|\emptyset \Longrightarrow_c^* v|\emptyset$*
- *$e|\emptyset$ diverges*
- *$e|\emptyset \Longrightarrow_c^* E[(T)v]|\mathcal{B}$, where the type of v is not a subtype of T.*

As corollaries of this theorem, it is easy to show that CTJ avoids the lost exception and lost continuation problems. First, the only way a well-typed program terminates is via either reduction to a value or via a bad cast. Therefore, a well-typed program cannot terminate with an uncaught exception. Second, a well-typed program continues execution unless it terminates or encounters a cast error (it does not get "stuck"). From this, it can be shown that, if program execution reaches an asynchronous method call or wait call, either evaluation of the calling expression is eventually resumed (with the results of the call), execution stops due to a runtime cast error, or the program diverges. In other words, asynchronous calls always complete, unless the entire program fails to complete.

3.4 Translating CoreTaskJava

A CTJ program is translated to FJ$^+$ by rewriting tasks and asynchronous methods to use a continuation-passing style. We describe this translation formally using a set of syntax transformation rules. Our technical report has the full details on the formal translation and its properties. The next section informally describes the translation performed by our implementation.

4. IMPLEMENTATION

4.1 Compiling TaskJava Programs to Java

The TaskJava compiler implements a source-to-source translation of TaskJava programs to (event-driven) Java programs. We refer to invocations of `wait` and `async` methods collectively as *asynchronous calls*. Note that this translation is only needed for methods containing asynchronous calls — all other methods are left unchanged.

In this section, we provide a high level overview of our compiler's implementation, using a series of small examples. A more complete example of the translation output may be found in the companion technical report [6].

CPS transformation of Tasks. The compiler uses continuation-passing style to break up the `run` methods of tasks into a part that is executed up to an asynchronous call and a continuation. Rather than implement the continuation as a separate class, we keep the continuation within the original method. The body of a task's `run` is now enclosed within a `switch` statement, with a case for the initial code leading up to the first asynchronous call and a case for each continuation. Thus, the `switch` statement acts as a kind of structured `goto`. We use this approach instead of separate methods to avoid building up the call stack when a loop's body contains an asynchronous call, since Java does not optimize tail-recursive calls.

Task state. Any state which must be kept across asynchronous calls (e.g., the next step of the switch statement) is stored in a new `_state` field of the task. An inner class is defined to include these new fields.

If local variables are declared in a block that becomes broken across continuations, they must be declared in a scope accessible to both the original code and the continuation. Currently, we solve this problem by changing all local variables to be fields of the `_state` object.

Calls to `spawn`. The spawning of a new task is implemented by creating a new task object and then registering this object with the scheduler, which will then call the task's `run` method.

Calls to asynchronous methods. When an asynchronous method is called, a callback object is created and passed to the callee. The `run` method of this callback should be invoked upon completion of the callee method. The caller returns immediately upon return from the callee, to be resumed later by the callback. For example, consider the following asynchronous call which returns a concatenated string:

```
...
x = concat(``abc'', ``xyz'');
```

This would be translated to:

```
case 1:
  ...
```

```
    concat(``abc'', ``xyz'', new run_callback(this, 2));
    return;
  case 2:
    x = (String)this._state._retVal;
```

Here, `concat` is passed a third parameter, a new callback object. The callback is initialized with a reference to the calling task (`this`) and the `switch` step to resume upon completion of the call. The actual assignment of `x` now occurs in the following `switch` step.

Callback classes are created by the compiler. To resume a task, the callback simply assigns to two compiler-generated fields in the task and re-invokes the task's `run` method. The first compiler-generated field, `_state._step`, indicates the `switch` case to resume (2 in our example). The second field, `_state._retVal`, contains the result of the asynchronous call (the concatenated string, in our example).

We introduce temporary variables in situations where breaking up an expression at asynchronous calls becomes difficult. For example, a nested asynchronous call, such as in `concat(concat(x, y), z)`, is first assigned to a temporary variable, which is passed to the outer call. Temporaries are also used when an asynchronous call occurs inside an `if` or loop condition.

Calls to `wait`. Calls to `wait` may be translated in a similar manner to asynchronous method calls, replacing the `wait` call itself with a scheduler event registration. In our implementation, we take a slightly different approach, described in section 4.2.

Asynchronous methods. The signature of an asynchronous method is changed to include an additional callback parameter. This callback is called upon completion of the method. Any return value is passed to the callback, instead of being returned to the asynchronous method's caller.

The bodies of `async` methods are translated in a similar manner to tasks. However, since simultaneous calls of a given method are possible, the `_state` object is passed as a parameter to the method, rather than added as a field to the containing class. To achieve this, the main body of the method is moved to a separate (private) continuation method. The original (externally callable) method just constructs a state object, stores the method arguments in this state, and then calls the continuation method. As with tasks, asynchronous methods return immediately after calling an asynchronous method or `register`, and are resumed through a callback.

Loops. If an asynchronous call occurs within a loop, the explicit loop statement is removed and replaced with a "branch and goto" style of control flow, simulated using steps of the `switch` statement. The entire `switch` statement is then placed within a `while` loop.

For example, consider the following call to `concat`:

```
  String s = ``''; int i = 0;
  while (i<5) {
    s = concat(s, 'a');  i = i + 1;
  }
  ...
```

This would be translated as follows:

```
  while (true) {
    switch (_state._step) {
    case 1:
      _state.s = ``''; _state.i = 0;
    case 2:
      if (!(_state.i < 5)) { _state._step = 4; break; }
      concat(_state.s, 'a', new run_callback(this, 3));
      return;
    case 3:
      _state.s = (String)_state._retVal;
      _state.i = _state.i + 1;
      _state._step = 2; break;
    case 4:
      ...
```

In the first case, we see the translated initialization assignments. The local variables have been made into fields of the the task's `_state` member. We fall through to the second case, which implements the "top" of the original `while` loop. If the original loop condition is false, we simulate a `goto` to step 4 by setting the step variable to 4 and breaking out to the enclosing `while` loop. Otherwise, we call `concat`, passing a new callback object, and then return. Upon completion of `concat`, the callback will set the step to 3 and invoke the task's `run` method. This gets us back to case 3 of our `switch` statement. At the end of this case, we simulate a `goto` back to the top of the loop by setting the step variable to 2 and breaking out of the enclosing `switch`.

Exceptions. Due to the CPS translation, asynchronous methods cannot simply throw exceptions to their callers. Instead, exceptions are passed from callee to caller via a separate `error` method on the callback. The body of an asynchronous method which may throw exceptions is enclosed in a `try..catch` block. If an exception is thrown, the `error` method of the callback is called (instead of the normal control flow's `run` method), with the exception passed as a parameter.

The callback's `error` method assigns its exception to a compiler-generated `_error` field of the `_state` object and then resumes the associated caller. When an asynchronous call may have thrown an exception, the continuation code of the task or asynchronous method then checks whether the `_error` field has been set. If so, it re-throws the exception.

Consider the following example:

```
  try {
    x = concat(``abc'', ``xyz'');
  } catch (IOException e) {
    System.out.println(``error!'');
  }
  ...
```

This would be translated as:

```
  case 1:
    concat(``abc'', ``xyz'', new run_callback(this, 2));
    return;
  case 2:
    try {
      if (_state._error!=null) throw _state._error;
      x = (String)_state._retVal;
    } catch (IOException e) {
      System.out.println(``error!'');
    }
    ...
```

We initiate the `concat` asynchronous call as before. However, upon resumption of the caller, we check the `_error` field to see if an exception occurred. If so, we re-throw the exception. The continuation block is enclosed in a `try` statement. Thus, if the callee throws an `IOException`, the appropriate `catch` block is invoked.

4.2 The scheduler

An important design goal for TaskJava is to avoid making the language dependent on a specific scheduler implementation and its definition of events. One approach (used in the examples of section 2) is to specify one or more schedulers to the compiler, perhaps as a command-line option. The compiler then replaces `wait` calls with event registrations for this scheduler.

We chose a more flexible approach in our implementation. We do not include a `wait` call at all, but instead provide a second type of asynchronous method — `asyncdirect`. From the caller's perspective, an `asyncdirect` method looks like an asynchronous method with an implicit (rather than explicit) callback. However, the declaration of an `asyncdirect` method must contain an explicit callback. No translation of the code in the method's body is performed — it is the method's responsibility to call the callback upon completion. Typically, an `asyncdirect` method registers an event, stores a mapping between the event and the callback, and then returns. Upon completion of the event, the mapping is retrieved and the callback invoked.

This approach easily permits more than one scheduler to be used within the same program. Also, existing scheduler implementations can be easily wrapped with `asyncdirect` methods and used by TaskJava.

For our experiments, we implemented a single-threaded scheduler on top of Java's nonblocking I/O package (`java.nio`). Clients may register a callback to be associated with events on a given channel (socket). The scheduler then registers interest in the requested events with the Java `nio` layer and stores an association between the events and callbacks in a map. The scheduler's main loop blocks in the `nio` layer, waiting for an event to occur. Upon waking up, the scheduler iterates through the returned events and calls each associated client callback.

We have also implemented a thread-pooled scheduler which can concurrently process events. Event registrations are transparently mapped to threads by hashing on the associated channel. This scheduler provides the same API as our single-threaded scheduler, permitting applications which do not share data across tasks to take advantage of thread-pooling without any code changes.

5. CASE STUDY

Fizmez. As a proof-of-concept for TaskJava, we modified an existing program to use interleaved computation. We chose *Fizmez* [4], a simple, open source web server, which originally processed one client request at a time. We first extended the server to interleave request processing by spawning a new task for each accepted client connection. To provide a basis for comparison, we also implemented an event-driven version of Fizmez.

Task version. In this version, each iteration of the server's main loop accepts a socket and spawns a new `WsRequest` task. This task reads HTTP requests from the new socket, retrieves the requested file and writes the contents of the file to the socket.

The original Fizmez server used standard blocking sockets provided by the `java.io` package. To port Fizmez to TaskJava, we needed to convert the server to use our event scheduler. We used TaskJava's asynchronous methods to build an abstraction on top of our scheduler with an API that mirrors that of the `java.io` package. This approach allowed us to convert I/O calls to TaskJava simply by changing class names in field and method argument declarations.

Client threads	Latency(ms) Event	Latency(ms) Task	Throughput(req/sec) Event	Throughput(req/sec) Task
1	33.0	31.1	30.2	32.1
25	76.8	79.2	322.1	306.3
50	112.4	120.0	443.4	413.5
100	187.6	197.0	351.0	262.2
200	317.3	345.8	403.5	225.8
300	455.8	462.4	324.2	328.6
400	601.4	695.9	216.0	212.0

Table 1: Web server performance test results

Overall, we were able to maintain the same organization of the web server's code as was used in the original implementation. The main change we made was to refactor the request-processing code out of the main web server class and into a new class. This change was necessary since requests are now processed concurrently, so each request must maintain its own state.

Explicit event version. The event-driven implementation required major changes to the original Fizmez code. The web server no longer has an explicit main loop. Instead, a callback re-registers itself with the scheduler to process the next connection request. More seriously, the processing of each client request, which is implemented in a single method in the original and TaskJava implementations, is split across six callback classes and a shared state class in the explicit event implementation.

5.1 Performance Experiments.

We compared the performance of the TaskJava and explicit event-driven web server implementations using a multi-threaded driver program that submits 25 requests per thread for a 100 kilobyte file (stored in the web server's cache). Latency is measured as the average time per request and throughput as the total number of requests divided by the total test time (not including client thread initialization).

The performance tests were run on a Dell PowerEdge 1800 with two 3.6Ghz Xeon processors and 5 GB of memory. Table 1 shows the experimental results. The columns labeled "Event" and "Task" represent results for the event-driven server and the TaskJava server, respectively.

The overhead that TaskJava contributes to latency is within 10%, except at 400 client threads, where it reaches 16%. The throughput penalty for TaskJava is low up through 50 threads, but then becomes more significant, reaching 44% at 200 threads. Above 200 threads, the total throughput of both implementations drops, and the overhead becomes insignificant.

These results are not surprising, as we have not yet made any efforts to optimize the continuation-passing code generated by our compiler. There are two main differences between the TaskJava compiler-generated code and the hand-optimized event code. First, compared to the event version, each TaskJava asynchronous call involves one extra method call, extra assignments (for the `_step`, `_retVal`, and `_error` fields), and an extra `switch` statement. Second, the event-driven server pre-allocates and reuses callbacks. For example, in the event-driven implementation, we associate reused callbacks with each connection, as we know that, by design, there will be only one read or write request pending on a given connection at a time. In contrast, the TaskJava compiler currently allocates a new callback for each asynchronous call.

We are investigating approaches to reduce this overhead for a future version of the TaskJava compiler. To reduce the cost of the `switch` statement and extra assignments, we can embed the continuation code directly in a callback, except when it occurs within a loop. Alternatively, we may achieve more flexibility in structuring control flow by compiling directly to JVM bytecode. Without an interprocedural analysis, we cannot remove all extra callback allocations. However, we can allocate a single callback per enclosing method rather than per called method. This will eliminate the majority of runtime allocations that occur in our web server.

6. RELATED WORK

Event-driven programming is pervasive in many applications, including servers [16, 22], GUIs, and sensor networks applications [8, 10]. In [1], event-based and thread-based styles are broken into two distinct differences: manual vs. automatic stack management and manual vs. automatic task management. Threads provide automatic stack and task management, while events provide manual stack and task management. By this classification, TaskJava provides manual task management and automatic stack management. A hybrid cooperative/preemptive approach to task management is also possible in TaskJava by using a thread-pooled scheduler. Asynchronous methods in TaskJava make explicit when a method may yield control, addressing the key disadvantage of automatic stack management cited by [1].

Cooperative multitasking. The introduction compared TaskJava's approach with the concept of cooperative multitasking. Many implementations exist for cooperative multitasking in C and C++. In fact, [5] lists twenty such implementations. Aside from the differences discussed in the introduction, context switching in these systems is typically implemented through C or assembly-level stack manipulation. Stack manipulation is not possible for virtual machine-based languages, like Java, so TaskJava uses the CPS approach instead. While this approach is more complicated, it can be advantageous. In particular, the stack-based approach requires a contiguous stack space to be allocated per thread, which may result in a significant overhead when many tasks are created.

The C library and source-to-source compiler Capriccio [21] provides cooperative threading, implemented using stack manipulation. It avoids the memory consumption problems common to most cooperative and operating system thread implementations by using a whole-program analysis and dynamic checks to reduce the stack memory consumed by each thread. This downside of this approach is the loss of modular compilation. Capriccio also suffers from the other weaknesses of cooperative threading — difficulty implementing on top of a VM architecture and lack of scheduler flexibility.

Functional Programming Languages. The functional programming community has explored the use of continuations to preserve control flow in the context of concurrent programming. [9] and [18] describe the use of Scheme's first-class continuations to avoid the inversion of control in web programs. Concurrent ML [19] builds pre-emptive threads on top of continuations. Concurrent ML also adds first-class events to the SML language, including a `choose` primitive, which can be used to build constructs equivalent to TaskJava's `wait`.

TaskJava's asynchronous methods can be viewed as a limited form of continuation. Although asynchronous methods do not support some programming styles possible with continuations, providing a more limited construct enables the TaskJava compiler to statically and modularly determine which calls may be saved and later resumed. This limits the performance penalty for supporting continuations (such as storing call state on the heap) to those calls which actually use this construct.

The functional programming community has also worked to extend the Continuation Passing Style transformation to better serve the needs of concurrent programs. [7] describes *trampolined style*, a programming style and transformation which permits the interleaving of tail recursive functions. [14] describes a sequence of code transformations which avoid inversion of control issues in web programs, without requiring language support for continuations. Neither approach provides a limited, modular translation which can coexist with existing codebases. In addition, both papers describe translations in the context of late-bound, functional languages, as opposed to a statically-typed, object-oriented, non-tail-recursive language like Java.

In [17], a transformation for Scheme programs is described, which permits the implementation of first class continuations on top of a non-cooperating virtual machine. The transformation installs an exception handler around the body of each function. When the current continuation is to be captured, a special exception is thrown. The handler for each function saves the current function's state to a continuation. This approach avoids changing function signatures, permitting some interoperability between translated and non-translated code. However, if a non-translated function appears on the stack when a continuation is captured, a runtime error is thrown. By using method annotations to direct the translation, TaskJava avoids this issue while still permitting interoperability between translated and non-translated code.

Languages and Tools for Embedded Systems. nesC [8] is a language for embedded systems with direct language support for writing in a continuation passing style. As such, it suffers from the lost continuation problem — there is no guarantee that a completion event will actually be called. This approach was chosen by the designers of nesC because it can be implemented with a fixed-size stack and without any dynamic memory allocation.

[13] describes a source-to-source translator for Java Card applications. It uses a whole-program translation to convert code interacting with a host computer to a single large state machine. Like TaskJava, it must break methods up at blocking calls (limited to the Java Card communication API) and must handle classes and exceptions. However, the translator does not support concurrent tasks or recursive method calls. In addition, rather than use method annotations, method bodies are split at each method call, regardless of whether they contain blocking calls. These limitations, appropriate to an embedded environment, significantly simplify the translation algorithm. Tasks in TaskJava are more general and thus useful in a wider range of applications.

Simplifying event systems through meta-

programming. The Tame framework [12] implements a limited form of CPS transformation through C++ templates and macros. The goal of Tame, like TaskJava, is to reuse existing event infrastructure without obscuring the program's control flow. Continuations are passed explicitly between functions. However, the code for these continuations is generated automatically and the calling function rewritten into case blocks of a switch statement, similar to the transformation performed by the TaskJava compiler. Thus, Tame programs can have the benefits of events without the software engineering challenges of an explicit continuation passing style.

By using templates and macros, Tame can be delivered as a library, rather than requiring a new compiler front-end. However, this approach does have disadvantages: the syntax of asynchronous calls is more limited, exceptions are not supported, template error messages can be cryptic, and the implementation only works against a specific event scheduler. Tame favors flexibility and explicit continuation management over safety. As such, it does not prevent either the lost continuation or the lost exception problems.

7. CONCLUSION

We have described the task programming model and its instantiation in the TaskJava extension to Java. Our approach provides three advances over prior work: 1) a modular translation enabled by method annotations, 2) the idea of a linkable scheduler, which has been formalized through a two-level operational semantics and implemented in our TaskJava compiler, and 3) the design of a CPS translation that works with Java language features including exceptions and subclassing.

TaskJava is the first step toward our goal of writing robust and reliable programs for large-scale asynchronous systems. We plan to improve our compiler implementation and extend the TaskJava language. For example, we would like to add Tame's fork and join constructs to TaskJava, in a manner that is compatible with exceptions, and supports the static guarantees currently provided by our language.

We also plan to investigate how the explicit control flow of TaskJava programs can improve static analysis tools. We expect analyses for TaskJava programs to be more precise when compared to analyses of general event-driven programs, which must reason about event flow through function pointers and objects. In addition, our translation approach may also yield insights about how event-programming frameworks may better support analysis tools. For example, a TaskJava program interacts with the scheduler in two ways: through `spawn` and through `wait`. These interactions are both translated into event registrations. In a program written directly using callbacks, making the distinction between these cases explicit yields more information about the programmer's intent, which may help static analyses.

8. REFERENCES

[1] A. Adya, J. Howell, M. Theimer, W.J. Bolosky, and J.R. Douceur. Cooperative task management without manual stack management. In *Proc. Usenix Tech. Conf.*, 2002.

[2] A. Appel. *Compiling with continuations.* Cambridge University Press, 1991.

[3] Enterprise Java Beans. http://java.sun.com/products/ejb/.

[4] David Bond. Fizmez web server. http://sourceforge.net/projects/fizmezwebserver.

[5] R. Engelschall. Portable multithreading - the signal stack trick for user-space thread creation. In *Proc. USENIX Tech. Conf.*, June 2000.

[6] J. Fischer, R. Majumdar, and T. Millstein. Preventing lost messages in event-driven programming, January 2006. http://www.cs.ucla.edu/tech-report/2006-reports/060001.pdf.

[7] S. Ganz, D. Friedman, and M. Wand. Trampolined style. In *ICFP '99*, pages 18–27, 1999.

[8] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC Language: A Holistic Approach to Network Embedded Systems. In *PLDI '03*, pages 1–11, June 2003.

[9] P. Graunke, S. Krishnamurthi, S. Van Der Hoeven, and M. Felleisen. Programming the web with high-level programming languages. *LNCS*, 2028:122–137, 2001.

[10] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *ASPLOS '00*, pages 93–104. ACM, 2000.

[11] A. Igarashi, B. Pierce, and P. Wadler. Featherweight java: a minimal core calculus for java and gj. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.

[12] M. Krohn, E. Kohler, F. Kaashoek, and D. Mazieres. The Tame event-driven framework. http://www.okws.org/doku.php?id=okws:tame.

[13] P. Li and S. Zdancewic. Advanced control flow in java card programming. In *LCTES '04*, pages 165–174. ACM Press, 2004.

[14] J. Matthews, R. Findler, P. Graunke, S. Krishnamurthi, and M. Felleisen. Automatically restructuring programs for the web. *Automated Software Eng.*, 11(4):337–364, October 2004.

[15] N. Nystrom, M.R. Clarkson, and A.C. Myers. Polyglot: An extensible compiler framework for java. In *CC '03*, LNCS 2622, pages 138–152. Springer, 2003.

[16] V.S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable web server. In *Proc. USENIX Tech. Conf.*, pages 199–212. Usenix, 1999.

[17] G. Pettyjohn, J. Clements, J. Marshall, S. Krishnamurthi, and M. Felleisen. Continuations from generalized stack inspection. In *ICFP '05*, pages 216–227, 2005.

[18] C. Queinnec. Inverting back the inversion of control or, continuations versus page-centric programming. *SIGPLAN Not.*, 38(2):57–64, 2003.

[19] J. Reppy. Cml: A higher concurrent language. In *PLDI '91*, pages 293–305, New York, NY, USA, 1991. ACM Press.

[20] R. von Behren, J. Condit, and E. Brewer. Why events are a bad idea (for high-concurrency servers). In *HotOS IX*, 2003.

[21] R. von Behren, J. Condit, F. Zhou, G. Necula, and E. Brewer. Capriccio: scalable threads for internet services. In *SOSP '03*, pages 268–281. ACM, 2003.

[22] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. In *SOSP '01*. ACM, 2001.