Tasks: Language Support for Event-Driven Programming

Jeffrey Fischer, Rupak Majumdar, and Todd Millstein
UCLA Computer Science Dept.

Outline

- The problem
- Example: Threads vs. Events
- The TaskJava language
- Related work
- Applications

The Problem

- Current solutions for interleaved computation suffer a number of drawbacks
  - Multi-threaded servers
    - Introduce concurrency
    - Performance issues when using large number of threads
    - Not suitable for some contexts (embedded systems, some OS kernels, business processes)

The Problem

- Current solutions for interleaved computation suffer a number of drawbacks
  - Event-driven servers
    - Must manually translate to *continuation-passing style*
    - Difficult to follow control flow
      - May lead to bugs!
    - Cannot easily take advantage of language features such as inheritance and exceptions

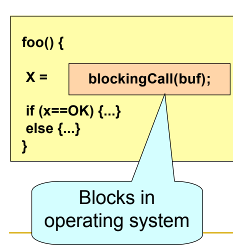
Our solution: TaskJava

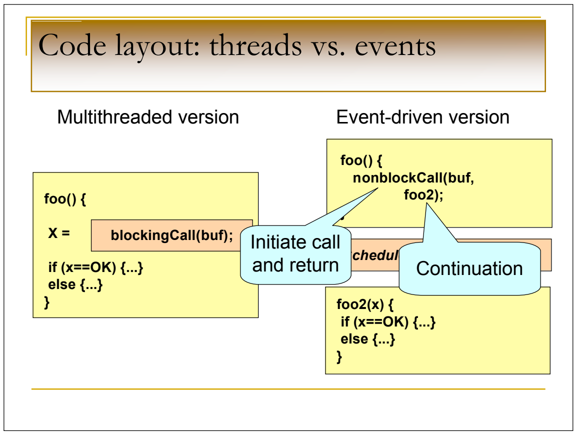
High performance concurrency while preserving program structure.

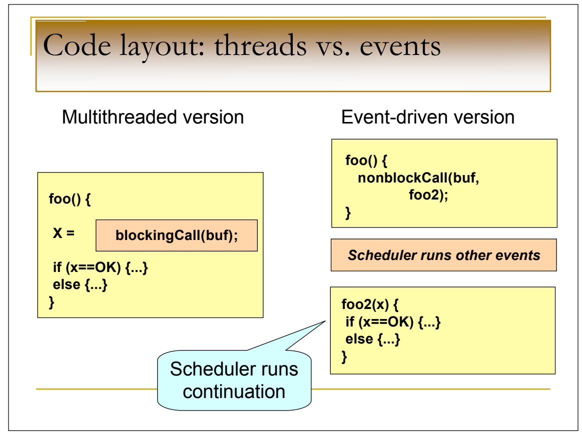
- Extension to Java
- Programming model: Tasks "look like" threads, but run like events
- Compiler performs modular CPS translation

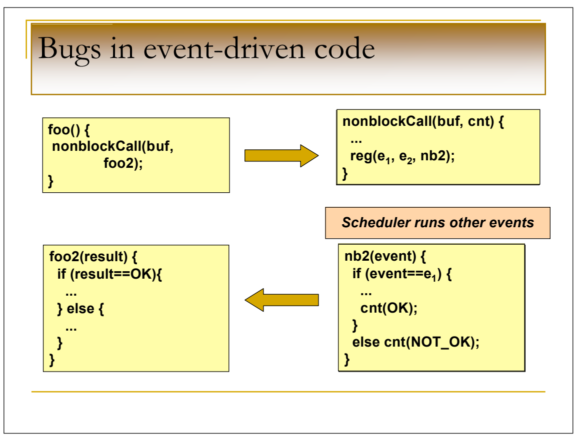
Code layout: threads vs. events

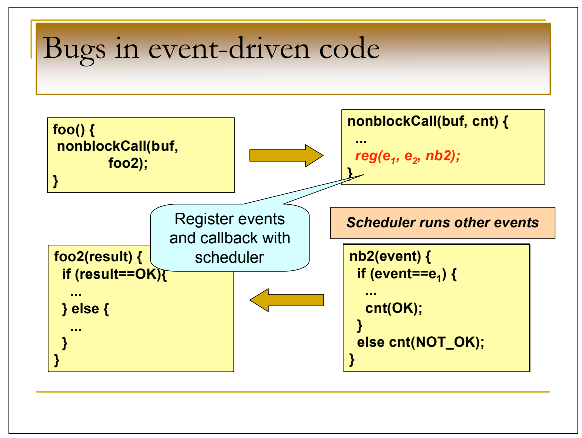
Multithreaded version

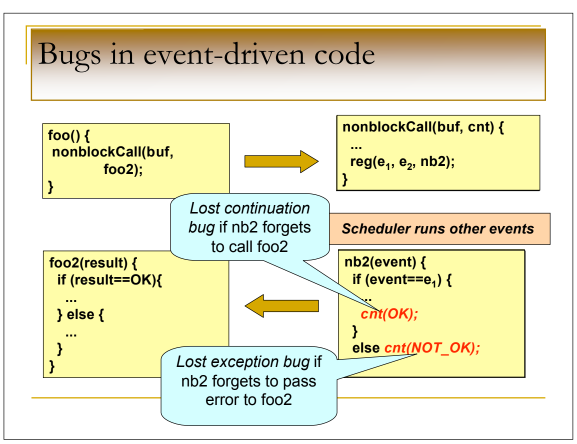


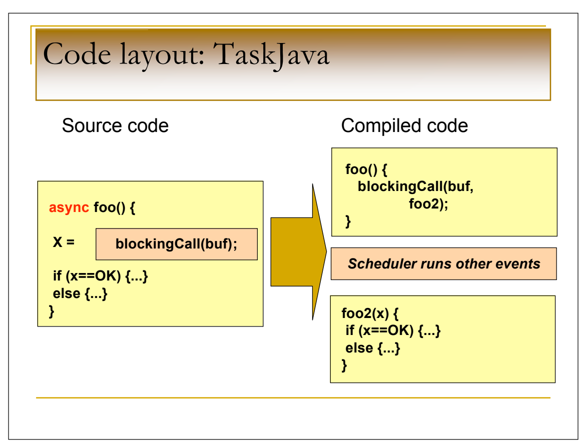

Code layout: threads vs. events
Multithreaded version
Event-driven version
foo() {
X =
blockingCall(buf);
if (x==OK) {...}
else {...}
}
foo() {
nonblockCall(buf,
foo2);
Initiate call
and return
Continuation
foo2(x) {
if (x==OK) {...}
else {...}
}


Code layout: threads vs. events
Multithreaded version
Event-driven version
foo() {
X =
blockingCall(buf);
if (x==OK) {...}
else {...}
}
foo() {
nonblockCall(buf,
foo2);
}
Scheduler runs other events
foo2(x) {
if (x==OK) {...}
else {...}
}
Scheduler runs
continuation


Bugs in event-driven code
foo() {
nonblockCall(buf,
foo2);
}
nonblockCall(buf, cnt) {
...
reg(e1, e2, nb2);
}
Scheduler runs other events
foo2(result) {
if (result==OK){
...
} else {
...
}
}
nb2(event) {
if (event==e1) {
...
cnt(OK);
}
else cnt(NOT_OK);
}


Bugs in event-driven code
foo() {
nonblockCall(buf,
foo2);
}
nonblockCall(buf, cnt) {
...
reg(e1, e2, nb2);
}
Register events
and callback with
scheduler
Scheduler runs other events
foo2(result) {
if (result==OK){
...
} else {
...
}
}
nb2(event) {
if (event==e1) {
...
cnt(OK);
}
else cnt(NOT_OK);
}


Bugs in event-driven code
foo() {
nonblockCall(buf,
foo2);
}
nonblockCall(buf, cnt) {
...
reg(e1, e2, nb2);
}
Lost continuation
bug if nb2 forgets
to call foo2
Scheduler runs other events
foo2(result) {
if (result==OK){
...
} else {
...
}
}
nb2(event) {
if (event==e1) {
cnt(OK);
}
else cnt(NOT_OK);
}
Lost exception bug if
nb2 forgets to pass
error to foo2


Code layout: TaskJava
Source code
Compiled code
async foo() {
X =
blockingCall(buf);
if (x==OK) {...}
else {...}
}
foo() {
blockingCall(buf,
foo2);
}
Scheduler runs other events
foo2(x) {
if (x==OK) {...}
else {...}
}

TaskJava features

```
Class WriteTask implements Task {
  void run() { ...
    do { write(buffer);
    } while (buffer.hasRemaining()); ...
  }
  async void write(CharBuffer buffer)
    throws IOException {
      Event e = wait(channel, Event.WRITE,
                Event.ERROR);
      switch (e.type()) {
        case Event.WRITE: ch.write(buffer); break;
        case Event.ERROR: throw new IOException();
      }
} }
```

The **Task** interface looks like Java's **Thread** interface

TaskJava features

```
Class WriteTask implements Task {
  void run() { ...
    do { write(buffer);
    } while (buffer.hasRemaining()); ...
  }
  async void write(CharBuffer buffer)
    throws IOException {
      Event e = wait(channel, Event.WRITE,
                Event.ERROR);
      switch (e.type()) {
        case Event.WRITE: ch.write(buffer); break;
        case Event.ERROR: throw new IOException();
      }
} }
```

Asychronous method call

Method annotation

TaskJava features

```
Class WriteTask implements Task {
  void run() { ...
    do { write(buffer);
    } while (buffer.hasRemaining()); ...
  }
  async void write(CharBuffer buffer)
    throws IOException {
      Event e = wait(channel, Event.WRITE,
                Event.ERROR);
      switch (e.type()) {
        case Event.WRITE: ch.write(buffer); break;
        case Event.ERROR: throw new IOException();
      }
} }
```

Yield until one of the events occurs

Errors signaled by throwing exceptions

Translating TaskJava to Java

- Bodies of Task `run` and `async` methods split into continuation passing style
  - Explicit callbacks introduced by compiler
- wait (Set) → <Scheduler>.register(Set, new cb(...))
- Scheduler class provided as option to compiler

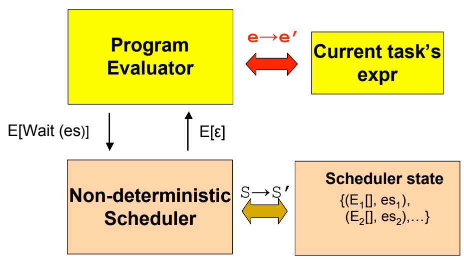
Translating TaskJava: the tricky bits

- Local variables
  - Move to heap if used across async calls
- Nested asynchronous calls
  - Introduce temporary variables
- Loops
  - Flatten and use switch statement to simulate goto's
- Exceptions
  - Separate callbacks for error control flow

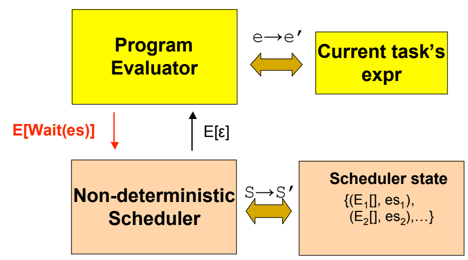
Formalizing TaskJava

- Defined semantic rules and type system
- Prove key properties:
  - Type soundness
  - No lost continuations or lost exceptions
- Translation to Java

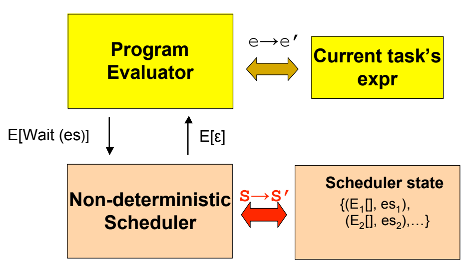
Formalizing TaskJava



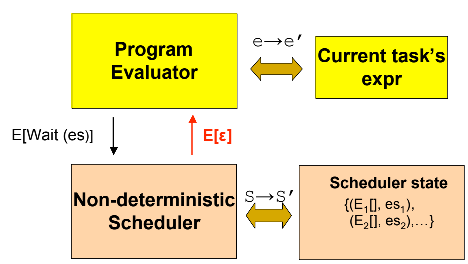
Formalizing TaskJava



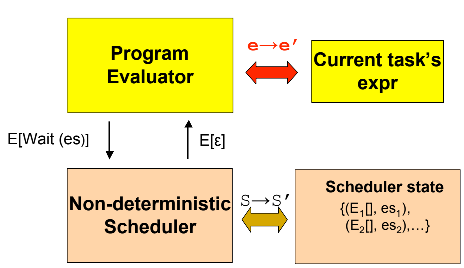
Formalizing TaskJava



Formalizing TaskJava



Formalizing TaskJava



So, what's new?

- Type system tracks blocking methods
- Compiler translates only blocking methods
  - Safe
  - Coexists with existing libraries
- Compiler generates scheduler-independent code
  - Case study: web server with pure-event and thread-pooled schedulers

Related work

- Cooperative multitasking
- Functional programming languages
- Other language approaches

Related work: Cooperative Multitasking

- User-level threads through stack manipulation
- Many implementations
  - E.g. [Engelschall 00], [von Behren, et. al. 03]
- Does not work for most VM-based languages
- Scheduler is fixed

Related work: Functional programming languages

- Scheme: avoid inversion of control issues in web programming
  - First-class continuations [Graunke 01], [Queinnec 03]
  - Whole program CPS transformations [Matthews, et. al. 04]
- Concurrent ML [Reppy 91]
  - User-level pre-emptive threads and first class events
  - Built on top of continuations

Related work: Functional programming languages

- "Continuations from generalized stack inspection" [Pettyjohn, et al. 05]
  - Implements Scheme continuations on .NET VM
  - Uses exception handlers and stack copying

Related work: other language approaches

- TAME: C++ Library for event-driven programming [Krohn and Kohler 06]
  - Implements a localized CPS-transform via templates
  - Emphasizes flexibility over safety
- MAWL [Ball and Aktins 99]
  - DSL for Web applications
- Scala actor library
  - Programming provides continuation as a closure
  - Type system ensures that "async" call does not return values

Applications of TaskJava

- TaskJava in embedded environments
- Web applications

TaskJava for embedded applications

- Embedded systems generally cannot use threads
  - Larger and non-deterministic memory usage
  - Non-deterministic scheduling
- TaskJava could address these issues
- Possible targets:
  - Virgil, a Java-like language with static allocation
  - Sqawk Java VM

Embedded TaskJava: static memory allocation

- Annotations help in analyzing stack usage
- Restrictions
  - No recursion
  - No variables used across asynchronous calls
  - Static number of tasks (how?)
- Is TaskJava still useful when compiling directly to hardware instruction set?

TaskJava for web applications

- Challenges in server code for web applications:
  - Servers make blocking calls to clients
    - Java Servlet model is event-driven to avoid tying up threads
  - Client makes control flow decisions
  - Browser uses non-standard control flow model
    - Backwards control flow via “back button”
    - Forking control flow via “new window” and “new tab”

How TaskJava can help

- From the scheduler’s perspective, callbacks are first class continuations
  - Callback may be saved, copied, called at any time
  - Can build on ideas from web frameworks in languages like Scheme and Smalltalk

Session management

- Keep callbacks in a map, indexed by session id
- Options for continuation management:
  - One continuation per session
  - One continuation per web page
- Alternatively, encrypt callback and send all state to client

For more information

- “Tasks: Language Support for Event-driven Programming”, PEPM 2007
- http://cs.ucla.edu/~fischer