A Thread of One's Own

Sriram Srinivasan
University of Cambridge Computer Laboratory, 15 JJ Thomson Ave,
Cambridge, CB3 0FD, UK
Sriram.Srinivasan@cl.cam.ac.uk

Abstract. This paper demonstrates an architecture for suspending and resuming methods in Java using a restricted form of continuation passing style (CPS) transformation. It describes Kilim[1], a toolkit to portably weave threads of control called Fibers, through Java code. The chief contributions of this paper are the set of design choices made for both space and time efficiency in getting one-shot continuations to work on the JVM (in some cases, 60x faster than competing approaches) and to address some tough issues traditionally passed over by others, such as handling of local subroutines and constructors. We are able to support hundreds of thousands of threads of control with switching times of the order of 3 to 4 µs on a low-powered laptop with Sun's JVM.

Keywords: suspend, resume, lightweight threads, actors, concurrency, java

1 Background

It is almost a truism nowadays that "concurrent programming is hard", that it is error-prone and not scalable. Many in the research community have repeatedly pointed out that it doesn't *have* to be this way [7], that both problems are due to shared-state concurrency and not due to multiple threads of control. The other generally accepted notion is that threads are heavyweight. Again, it doesn't *have* to be this way, as many real industrial implementations such as Erlang [18] and Windows Fibers [12] and the Singularity project at Microsoft [21] have demonstrated.

Actors (also *active objects*) are a different (and far saner) way to deal with concurrency. They encapsulate data, code *and* a thread of control of their own and communicate by sending messages, like mini processes hooked together using pipes. The Kilim project at the University of Cambridge seeks to introduce Java programmers to the joys of active objects and message passing concurrency.

This paper concerns itself with the issue of making possible hundreds of thousands of lightweight threads of control, one per actor; we consider a thread *lightweight* if a programmer can start one without much ado as with Erlang processes or Ada tasks. Threads in Java, .NET and pthreads package all fail this test, limiting us to a few hundred threads per operating system process.

We want each active object to be able to make blocking calls such as `sleep` and `receive` without consuming a heavyweight Java thread. The traditional approach of structuring the code in a callback-oriented style results in the programmer having to

[1] A Kilim is a Turkish carpet where the fibers are woven tightly to create a pileless (flat) rug.

manually save relevant data into a global or a callback data structure before unrolling the stack. Not only is it tedious to program, it obfuscates flow of control.

This paper describes in detail a way of portably transforming straightforward sequential code containing blocking primitives into one that allows it to voluntarily suspend itself and to cede control to another runnable actor.

2 Kilim

The Kilim package supplies a *Weave* tool that recognizes invocations of blocking methods and transforms the code in the caller. It is a simple batch process that works directly on java bytecode:

```
java kilim.Weave –d destdir classnames ...
```

How does it know which methods are *pausable* (can potentially block) and which can't? A method is pausable if it invokes the static method `pause()` of a class called `kilim.Fiber`, or calls another pausable method, or overrides a pausable method; the notion of being pausable is thus an interface contract, similar in spirit to a checked exception. In order to avoid having to analyze the program as a whole or to maintain a database of pausable methods, we require the programmer to explicitly mark a method with a `@Pausable` annotation. This is an artificial restriction that can be eased within environments such as Eclipse.

2. 1 Transforming the Code

The transformation is conceptually simple and is shown in the next listing[2], with the original on the left and the transformed code on the right. In this example, `a()` calls a pausable method `b()`. Code injected in the three areas marked as *prelude*, *pre-call* and *post-call* helps the method pack up its local operand stack, its registers and the location of the program counter and return a status to its caller, which in turn does the same thing. Each method gets an extra argument of type Fiber; it is this that collects information about each activation frame as the stack is unwound and also represents an out-of-band channel of information that tells a caller whether its callee has signaled a pause or whether it has returned normally. The fiber is thus a continuation object that has all the information required to restore the call hierarchy and the data in the stack and registers, to allow the actor to resume from where it left off.

The injected code in the prelude consults the fiber and starts either at the original starting point START in the normal case or jumps straight to the pre-call stage of `b()` when being resumed. All pausable method invocations are sandwiched between calls to `Fiber.down` and `Fiber.up`; these help the fiber keep track of the current position in the call hierarchy and ensure that a copy of the current activation frame's program counter (pc) and status are readily available as member variables of the Fiber

[2] Note that we show only the transformation of `a()`. The one for `b()` would be very similar. The transformation is on bytecode, hence the use of gotos.

object. The post-call section examines the fiber's status after `b`'s return. The status conveys two orthogonal pieces of information: (i) whether or not `b()` wants to yield mid-flow and (ii) whether we already captured the current activation frame's state in an *earlier* suspend/resume cycle.

```
// original
@Pausable
void a() {
  x = ...
  b(); // b is pausable
  print (x);
}
```

```
// transformed code
void a(Fiber f) {
  switch (f.pc) {  // prelude
   case 0: goto START;
   case 1: goto CALL_B}
 START:
   x = ...
 CALL_B:           // pre_call
   f.down()
   b(f);
   f.up()          // post-call
   switch (f.status) {
   case NOT_PAUSING_NO_STATE:
      goto RESUME
   case NOT_PAUSING_HAS_STATE:
     restore state
     goto RESUME
   case PAUSING_NO_STATE :
     capture state, return
   case PAUSING_HAS_STATE:
     return
   }

   RESUME:
   print (x);
}
```

There are several optimizations in this process. First only those methods that contain invocations to pausable methods are modified (unlike typical CPS transformations). Second, live-variable analysis is performed to ensure that only those variables used downstream after a resumption point are captured. Third, the flow of constant values and of duplicates through the registers and stack is tracked. Clearly, these don't need to be saved; at resumption time, these are restored swiftly using JVM instructions that can push constants. Fourth, `return` is used to unwind the stack as opposed to using exceptions as a *longjmp* mechanism because exceptions are expensive by a couple of orders of magnitude. Not only do exceptions have to be caught and rethrown at each level of the stack chain, they clear the operand stack as well. This unnecessarily forces one to take a snapshot of the operand stack before making a call assuming optimistically that the callee is going to block.

2.2 Custom State objects

Each call site may have its own set of data items (some primitive types, some references) to be saved and restored. How do you store say, one object, two integers and one double in one place, and a single integer at another call site? It is expensive to box all primitive types and store them into an array of Objects.

We decided instead to take the approach of creating a custom State class to store exactly what is needed, one field per data item. Two such classes are shown next. The fiber that's passed down a call hierarchy accumulates a linked list of these state objects as the stack is unwound.

```
class S_I extends State {
   int f0;
}
```

```
Class S_OI2D extends State
{
  Object f0;
  int    f1;
  int    f2;
  double f3;
}
```

The novel part of this scheme is that the class names and the data layout are canonical; any call site that requires a single integer to be saved (as with `a()` in the previous example) will save it in an instance of class `S_I`. At first, it seems like this scheme would generate hundreds of such custom classes, one per pausable method invocation site, but an analysis of the JDK and several popular application servers show surprisingly low variation. The performance of this scheme is on par with other approaches that use arrays for each primitive type and one for reference types, but its chief advantage is that the code that saves and restores state doesn't need to worry about array overflow or the fact that arrays once grown needlessly occupy space.

3 Living with the JVM Verifier

The JVM verifier performs type and liveness analysis to ensure that the operand stack has the same number and type of data items independent of the path taken to get to a particular point and that a register or a stack element has been properly initialized with the correct type before it can be accessed. This means that our simple prelude shown earlier cannot arbitrarily jump into the middle of code. We solve this by having the prelude insert dummy constants of the appropriate type to fix the stack before doing the jump. We prefer not to restore the real state unless absolutely necessary (of course, in the rare but worst case, we have to pop off the dummy constants and insert the real state before resuming).

The post-call stage similarly ensures the stack has exactly one item of the appropriate return type before executing `return`. In the case of a pausing `return`, the returned value is a dummy constant of the method's return type.

3.1 Java Subroutines and *jsr*

Most JVM instructions are very simple to handle from the point of view of type and data flow analysis. The `jsr` instruction ("jump to subroutine") is a notable exception that causes headaches well in excess of its usage.

The Java Virtual Machine specification has an ill-defined notion of a local subroutine (different from a method). A subroutine is intended to support the try/catch/finally construct where the block in the *finally* section must be executed regardless of whether the *try* block completed normally or threw an exception that may or may not have been caught. In all three cases, the JVM specification suggests that the compiler emit the finally block as a *subroutine*, to which control can be transferred using a special jump instruction called `jsr`. A corresponding `ret` instruction is used to return to the instruction following the `jsr`.

The problems with `jsr` are numerous and well documented [11], so we will be brief. The instruction pushes a JVM internal address in the stack that is subsequently used by `ret`. This internal address is treated differently from other data types; it can't be stored in an instance variable, for example. This means that if a pausable method is called within a subroutine, we won't be able to jump directly to it in the prelude because we'll soon run into a `ret` instruction that expects an address to return to.

There is another problem: traditional liveness analysis (that figures out which values are still used downstream) is only concerned with intra-procedural data flow, not inter-procedural; that is, it can deal with hard-coded jump labels, but not a `ret` instruction that says "go back to where you came from".

Our solution was to inline (and thus duplicate) subroutines calls. As it turns out, it isn't as bad as it sounds; most virtual machines do the same thing internally and perhaps more importantly, most modern java compilers don't emit the jsr instruction any more.

3.2 *new* Challenges

The `new` expression in Java is split in two parts at the bytecode level; the first part allocates an empty object, the second part calls the constructor.

```
; new Foo(10, bar()) is translated to
new Foo         ; allocate object
ldc 10          ; load 10
invokestatic bar  ; call bar(), which returns int
invokespecial Foo.<init>(II) ; call constructor
```

The JVM tracks the fact that the empty object is uninitialized prior to the constructor's invocation. If `bar()` is a pausable method, we cannot have the prelude jump directly to it because the JVM prevents any jumps to a target between a `new` and the corresponding constructor. Our solution is equivalent to an ANF transformation, where

```
new Foo(10, bar())
```

is translated to:

```
tmp = bar()
new Foo(10, tmp)
```

The call to `bar` can now be paused.

4 Performance

We compared Kilim's Fibers to the JavaFlow project (discussed later) currently in development under the aegis of the Apache Jakarta project. These tests were performed on a 1.33 GHz Apple PowerBook with 1G RAM, running Mac OS X (10.4).

The following test shows a simple micro-benchmark that has `a()` calling `b()` in a loop 100000 times, with the times shown in milliseconds. This test measures the overhead of winding and unwinding the stack. `b()` manipulates three data items before and after the call to `pause()` one *long* type and two strings (one of which is a duplicate of the other). The first column shows the timings of a method that is not pausable but does the same work as `b()`, the second and third show the timings of `b()` when it pauses on every iteration and when it doesn't pause.

Times in μs. Lower is better.	Not pausable	Pausable	
		Not pausing	Pausing
Kilim	21	29	156
JavaFlow	37	29	7829

JavaFlow (discussed in the next section) is quite a bit slower when pausing: 78 μs vs. Kilim's 1.5 μs per iteration. The reasons for this are discussed later.

The next measurement shows how the two stack up (pun intended) with `b()` recursing to a stack depth of 5 and 10, the numbers showing elapsed time in milliseconds measured over 10000 iterations.

Times in μs. Lower is better	Stack depth	Not pausable	Pausable	
			No pause	Pause
Kilim	5	8	72	866
JavaFlow	5	57	53	15773
Kilim	10	14	142	1041
JavaFlow	10	109	113	27438

This shows that Kilim is roughly 20x faster than JavaFlow on this benchmark.

The comparison to non-pausable methods is not strictly a fair comparison in the tests above because we measured it without running it through the weaver, merely to study the overhead of adding the extra code. For a true comparison, one must compare it to a hand-coded state machine that actually implements the same functionality. In practice, the overhead of pausing is dwarfed by the other things a system needs to do and is round-off error when used in the context of network or disk–intensive split phase operations

5 Discussion

There are two popular ways of building suspend/resume systems; one is continuations and the other is coroutines.

There are several reasons why CPS (continuation passing style) transformation runs into rough weather in the context of the Java VM. First, CPS transformation modifies each procedure such that it never returns; it jumps ahead to the next procedure. This process doesn't need a stack and indeed, is quite at odds with an environment that forces a stack on it, as is the case with a JVM. The JVM architecture is quite stack-centric; security on the JVM relies on stack inspection and exception handlers are installed for exceptions thrown from within a stack hierarchy. CPS transformation also relies on a uniform transformation of all code, which would render reflection impossible.

As for coroutines. Ana de Moura et al's paper [13] discusses a taxonomy of coroutines and presents the argument that a certain class of them ("stackful") is equivalent to one-shot continuations. In their taxonomy, the continuations presented in this paper are stackful (because we suspend and resume a nested call hierarchy) and asymmetric (the Fiber API implies a caller/callee relationship). The Kilim framework also supplies an active object framework built atop the Fiber framework that offers fully symmetric support.

Coroutines and continuations can be implemented very efficiently in a language such as C that has direct access to memory, but even the C programming model forces a stack view of things. This is the reason for Haskell's GHC compiler to supply a Perl script called "the evil mangler" to hack GCC's output to make it support tail-calls.

We have considerably less leeway with the JVM that does not allow any form of stack manipulation (such as swizzling stack pointers), which necessarily forces us into the considerably more inefficient route of rewinding and restoring the stack. Within these constraints, we have focused our attention on maximizing run-time performance.

That said, the portability of our approach allows this framework to be readily used in a wide-variety of applications that have built-in latencies due to human interaction and/or networks; examples are web-servlet frameworks, workflow engines and user interfaces. The approach can even be used in such a heavily constrained as the Java applet sandbox provided by a web browser.

We have deliberately chosen not to expose fibers as a first-class feature although that functionality is there for the taking. Our chief preoccupation is to get many threads of control that we intend to schedule automatically and allow them to be individually cancelable or upgradeable from a management console. These properties are difficult to achieve if the code is in direct control of yielding. Besides, it has been our experience that programmers find it difficult to comprehend and debug the generality of first class continuations, a feeling supported by the number of tutorials on the subject.

The ability of continuations to translate a call chain into a list of State objects on the heap has tempted some to use them for serializing threads. We are skeptical of this particular application for a number of reasons, mainly because there often are non-serializable objects such as database connections and sockets that can't be packaged away.

6 Related Work

6.1 Lightweight Threads

There have been a number of projects that create fast, lightweight threads at a lower level. Capriccio 12 is a notable example. They modified the portable *pthreads* library to avoid massive pre-allocation of heap/stack space, relying instead on a static analysis of code to figure out the appropriate size and the appropriate locations in the code to allocate more heap space. It would be nice to have this facility available in all JVMs along with tail–call optimization.

6.2 First-Class Continuations

Continuations are powerful constructs that can be used as primitives to create all forms of branching, including portable user-level threads, exception handling, backtracking and others. Much literature exists on the properties of continuations, their taxonomy and on compiling with continuations [3])

Scheme and Standard ML are among many functional languages to support *first-class* continuations in the form of *call/cc* (call with current continuation). Implementations of these languages in Java have run into the constraints imposed by the JVM -- JScheme, JRuby, Kawa and Scala all punt on this feature to various degrees. Wadler et al also reflect on this aspect [6].

Pettyjohn et al [1][2] prove and demonstrate a technique for achieving first-class support for continuations in environments that don't support stack inspections and manipulation, such as Java and .NET. They also generalize all previous approaches of achieving continuations on the JVM. Their basic idea is to break up the code into fragments (as top level methods) where the last instruction of any fragment is a call to the next fragment in the chain. Correspondingly, they have specialized continuation objects that maintain the state needed for each fragment and an overridden `Invoke` method to invoke the corresponding fragment. It is structurally similar to our solution, with a generated State class per fragment that knows exactly which fragment to invoke.

Their approach while theoretically sound is inefficient and incomplete, as indicated by the prototype [2]. It is inefficient because it uses exceptions to capture state although the authors note that one could use distinguished return values or some out-of-band signal instead to indicate pausing. Other inefficiencies include creating a custom object per call invocation site, splitting the code into top-level procedures which results in loops being split into virtual function calls and the stack restoration involves a recursive invocation of Continuation frames, which means there are two virtual function calls for every one in a scheme that already splits up into many fragments. The transformation also incurs a function call's overhead on every "return".

Their suggestion of distinguished return values pose the problem that you need to introduce a return type for methods that were originally `void` and you also have the problem of separating an ordinary value returned by the application from a distinguished value.

The paper also does not tackle code transformation in the presence of exception handlers and java subroutines, the traditional stumbling blocks, but there is no reason why our techniques can't be employed.

Their solution isn't efficient for the JVM because they rely on the JVM to provide tail-call optimization (the last statement in each fragment is a method call to the next fragment).

These problems are identifiable in earlier approaches also.

There are three frameworks that transform Java bytecode internally into a style similar to ours, but don't surface it as a first class primitive. They are RIFE [14], PicoThreads [24] and JavaFlow [23], the last being the most promising and under active development.

The JavaFlow project uses thread-local variables to pass the continuation instead of modifying method signatures, as we do. While clean, this approach is error-prone because in the absence of any information from the interface (or some global knowledge of all the classes in the program) it is impossible to know whether the called interface or virtual method is pausable or not. A non-transformed piece of code would not know to look for yielding returns. In our case, the verifier would refuse to load the class because of the signature mismatch.

JavaFlow correctly transforms a method if it can reach a `suspend()` invocation. But it unnecessarily transforms all non-pausable methods reachable from there as well, leading to substantial code bloat.

None of these projects do liveness analysis as of this writing. This means they must store all local variables regardless of whether they will be used after resumption. Analysis of some popular Java projects and the JDK indicate that only about 30-40% of the information on average is used across method invocations. Finally, none of these projects handle subroutines and constructor calls with pausable methods. It is worth noting here that we explicitly disallow pausable methods within constructors, because an object should not be receiving messages until it is fully initialized, but we do allow a pausable method invocation in an expression that supplies a parameter to `new`.

6.3 Bytecode Analysis and Insertion

Xavier Leroy's paper [11] neatly sums up all the challenges of bytecode verification and provides formalizations and algorithms for doing type analysis. The Kilim weaver does value analysis in addition to types, to track duplicate values and constants. We settled on the ASM toolkit [17] (in preference to SOOT and BCEL) for its speed and compactness, but used our own verification and analysis engine.

7 Future

In order to make an active object framework a compelling alternative to the traditional locking and shared-memory mindset, we need to have fast and lightweight threads. Speed and portability are usually at odds with each other. Our approach works readily on cell phones and big iron. On the speed end of the spectrum, there is Occam 19, with process creation times in the 20 ns range on an 800 Mhz Pentium III, but with far less portability. The search for a happy medium will be multi-pronged; we are investigating modifications to the JVM and to the GCJ framework (GNU compiler for Java 20), whilst exposing a consistent interface to the programmer. As the American baseball player Yogi Berra once said, "When you come to a fork in the road, take it".

Acknowledgments
I owe a considerable debt to my advisor Dr. Jean Bacon and to Reto Kramer, Dr. Samuel Kounev and Lauri Pesonen for their support and feedback. I wish to thank Dr. Alan Mycroft in particular for his insights and enthusiasm. This work is supported by EPSRC grant GR/T28164.

References

1. Greg Pettyjohn, John Clements, Joe Marshall, Shriram Krishnamurthi, Matthias Felleisen. Continuations from generalized stack inspection. ICFP 2005: 216-227
2. Greg Pettyjohn. A Technique for implementing First-Class Continuations. At http://www.ccs.neu.edu/scheme/pubs/stackhack4.html
3. Robert Hieb, R. Kent Dybvig, Carl Bruggeman. Representing Control in the Presence of First-Class Continuations. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation.*
4. Tim Lindholm and Frank Yellin. The Java Virtual Machine Specification. Addison Wesley. 1997.
5. Sekiguchi, T., T. Sakamoto and A. Yonezawa. Portable implementation of continuation operators in imperative languages by exception handling, volume Advances in Exception Handling Techniques, pages 217–233. Springer-Verlag, 2001
6. M. Odesky and P. Wadler. *Pizza into Java*: Translating theory into practice. POPL. 1997 MPI
7. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative task management without manual stack management. In Proceedings of the 2002 Usenix ATC, June 2002.
8. Tao, W. A portable mechanism for thread persistence and migration. PhD thesis, University of Utah, 2001.

9. Edoardo Biagioni, Ken Cline, Peter Lee, Chris Okasaki, Chris Stone. Safe-for-Space Threads in Standard ML. In Higher-Order and Symbolic Computation, pages 209-225, Vol. 11, No. 2, 1998.
10. Alfred Aho, Ravi Sethi, and Jeffrey Ullman. Compilers Principles, Techniques, and Tools. Addison Wesley, 1986.
11. Xavier Leroy. Java bytecode verification: algorithms and formalizations. Journal of Automated Reasoning, 2003
12. R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: Scalable threads for internet services, in 19th ACM Symposium on Operating Systems Principles, 2003
13. Ana L´ucia de Moura, Roberto Ierusalimschy. Revisiting Coroutines. Technical report MCC15/04. Computer Science Department, PUC-Rio, June 2004
14. RIFE. http://rifers.org/
15. JavaFlow. http://jakarta.apache.org/commons/sandbox/javaflow/
16. Andrew Begel, Josh MacDonald, Michael Shilman. PicoThreads: Lightweight Threads in Java. At http://citeseer.ist.psu.edu/385966.html
17. ASM bytecode toolkit
http://asm.objectweb.org/index.html
18. Erlang. http://www.erlang.org
19. occam-pi project at the University of Kent. http://www.cs.kent.ac.uk/projects/ofa/kroc/
20. GNU compiler for Java (GCJ) http://gcc.gnu.org/java/
21. Microsoft Research Singularity Project at http://research.microsoft.com/os/singularity
22. RIFE. http://rifers.org/
23. JavaFlow. *http://jakarta.apache.org/commons/sandbox/javaflow/*
24. Andrew Begel, Josh MacDonald, Michael Shilman. PicoThreads: Lightweight Threads in Java. At *http://citeseer.ist.psu.edu/385966.html*