Technische Universität Berlin

Forschungsberichte der Fakultät IV – Elektrotechnik und Informatik

ICOOOLPS'2007

Proceedings of the Second ECOOP Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems

July 30, 2007, TU Berlin, Germany

Olivier Zendra, Eric Jul, Michael Cebulla (Eds.)

Bericht-Nr. 2007 – 5

ISSN 1436-9915

ICOOOLPS'2007

Proceedings of the Second ECOOP Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems

July 30, 2007
Berlin (Germany)
`http://icooolps.loria.fr`

Contents

Continuations in the Java Virtual Machine

Iulian Dragoş[1], Antonio Cunei[2], and Jan Vitek[2]

[1] École Polytechnique Fédérale de Lausanne, Switzerland
[2] Purdue University, USA

Abstract. Continuations have received considerable attention lately as a possible solution to web application development. Other uses proposed in the past, such as cooperative threading, coroutines or writing iterators, made them an attractive feature of dynamically typed languages. We present issues involved in adding continuations to a statically typed, object-oriented language like Java, and its implementation in the Java Virtual Machine. We propose three different flavors of continuations, and study their interactions with the base language, focusing on Java's concurrency model. We describe our implementation in Ovm, a realtime Java Virtual Machine, and discuss open issues.

1 Overview

Continuations are a way to represent the "rest of the computation" at a given point in the program [1]. Most languages that have first-class continuations represent them as functions that, when called, cause the immediate transfer of control from the caller to the point where it was *captured*. The computation will resume in the same state as when it was captured. It has to be noted that, unlike *setjmp/longjmp* in C, a continuation can be called at any point during program execution, sometimes long after the activation frame where it was captured has been left.

Continuations are used to encode coroutines, cooperative threading, to write iterators (C# iterators or Python generators) and have been proposed as a natural abstraction for interactive web applications [2, 3]. For instance, a tree traversal iterator, which might be a non-trivial task to code in plain Java, can be as simple as shown in Fig. 1. `Yield` suspends the current method and returns a value to the caller. Subsequent calls to `next` resume the traversal immediately after the previous call to `yield`. This method could not be coded in Java without continuations. For brevity, we don't include the definition of this method.

Our goal when designing continuations for Java was to confine changes to the virtual machine. Previous work has shown that continuations can be added in languages that target uncooperative virtual machines like the JVM or the .NET [4, 5]. However, implementing continuations in the virtual machine is likely to be more efficient and make them available to all languages targeting that platform. Furthermore, we chose to expose them through library functions rather than new opcodes, so that existing compilers need not be modified to take advantage of this new feature.

```
public class System {
  public static Object callcc(
    Runnable1 r);
  public static Object callccBounded(
    ContinuationBound cb, Runnable1 r);
  public static Object callccOneShot(
    Runnable1 r);
  //..
}

public interface Runnable1 {
  public void run(Continuation k);
}

public final class Continuation {
  public void call(Object v);
}
```

(a) Additions to the java.lang package.

```
public Iterator getIterator() {
  return new Generator() {
    protected void generate() {
      preorder(Tree.this);
    }

    private void preorder(Tree t) {
      if (t.isLeaf())
        yield(t.value);
      else {
        preorder(t.left);
        preorder(t.right);
      }
    }};
}
```

(b) A tree iterator.

Fig. 1.

Continuations are exposed through the library function `callcc`[3]. This follows common practice in languages that have first-class continuations (Scheme, Smalltalk, Ruby, ML, etc.). Fig. 2 shows an example of using `callcc` in our system. The list of additions to the `java.lang` package is shown in Fig. 1 (a different class than `System` could have been used for adding `callcc` but we chose to make it clear it is a VM service).

```
System.callcc(new Runnable1() {
  public void run(Continuation k) {
    k.call(new Integer(42));
  }
});
```

Fig. 2. Using callcc

1.1 Call/cc and Call

We follow standard semantics for the call/cc mechanism. In the following text the term "execution context" will refer to the call stack, local variables and instruction pointer during the execution of a method. Global state and objects on the heap are not saved/restored by continuations.

[3] Call with current continuation

System.callcc This method captures a continuation `k` which represents the current execution context, with the instruction pointer pointing immediately after this call. It then proceeds to call method `Runnable1.run` on its parameter.
Return value: If the call to `Runnable1.run` finishes without a call to `k.call` being made, `System.callcc` returns `null`. Otherwise, it returns the value passed as argument to `k.call` (see below).

Continuation.call A call to this method restores the execution context captured by the given continuation. The current execution context is dropped and execution continues at the point where `System.callcc` was called, returning the value passed as argument to `call`. This method never returns.

2 Continuations

There are several flavours of continuations presented in literature [6–8]. We have chosen to implement three kinds of continuations: full continuations for their expressivity, one–shot continuations because they are fast and serve a common use–case, and delimited continuations as a good compromise between efficiency and expressivity.

First–class continuations This is the most general kind of continuations. There are no limitations with regard to the number of times a continuation can be called, or its lifetime. Since they need to copy the full execution stack, they are more costly than the other kinds of continuations.

Delimited continuations The user first obtains a *continuation bound* which is then used to obtain a delimited continuation. Such a continuation is valid for as long as the execution stack does not exit its bound. An invalid continuation throws an exception when called. Valid delimited continuations can still be called any number of times. A delimited continuation costs less than a full continuation, since they only need to copy a subsection of the execution stack, given by the continuation bound and the point where `callcc` is called.

One–shot continuations A one shot continuation can be called a single time. They are valid for as long as the execution does not leave the `callcc` frame[4]. They are the "cheapest" continuations available, since the system only has to save the instruction pointer.

The user has the choice of which kind of continuations to use by calling the corresponding `callcc` method. Full continuations can be used when the whole execution context needs to be saved and restored, for instance to code threads. One–shot continuations can be used fast for non–local returns while delimited continuations are a good choice when only a limited part of the execution context should be saved/restored (for instance, to write coroutines).

[4] This can be regarded as an implicit application of that continuation by callcc itself, after the runnable that was passed as argument returns.

3 Interactions with the Base Language

Existing Java language features turn out to interact in unexpected ways with continuations. Exceptions, language support for synchronization through monitors, threads and the Java security model all present possible issues when continuations are added to the language. We will consider each one in turn and present the problems we identified and our solutions.

3.1 Exceptions

The try-catch-finally statement raises issues in the presence of continuations: *finally* handlers are guaranteed to run, be it through normal or abrupt (exception thrown) completion of its protected block. Consider the code shown in Fig. 3 (a).

```
Handle h = getNativeHandle();
try {
 // use h in various ways
 System.callcc(
   new Runnable1() {
     public void run(Continuation k) {
       this.dangerousK = k;
     }});
 // use h again
} finally {
  releaseNativeHandle(h);
}
```

(a)

```
System.callcc(new Runnable1() {
  public void run(Continuation k) {
    this.k1 = k;
    try {
      System.callcc(new Runnable1() {
        public void run(Continuation k) {
          this.k2 = k;
          //..
          k1.call();
        }});
      // k2 points here
    } finally {
      //..
    }
    }});
// k1 points here
```

(b)

Fig. 3. Continuations and "finally"

In usual Java, this code ensures the native handler is released under all circumstances. It also saves a continuation which can later be called, and make use of the handle *after* it has been released. Furthermore, it will reach the end of the try–finally block and run the handler once more — releasing the native handler a second time.

The problem comes from the fact that `callcc` breaks the assumption that a *finally* handler is run just once. The converse, calling a continuation while *finally* handlers are active has a similar problem: should they be run? The answer is not easy, since it depends on the continuation: in Fig. 3 (b) calling `k1` leaves the *finally* handler, while calling `k2` does not.

Additionally, a VM-only solution cannot be adopted, since the Java VM [9] has no notion of *finally* handlers. It is the compiler's job to generate proper code

on all control flow paths to invoke the finally code. Thus, Java with continuations has to either forbid such cases, or relax the guarantees of `finally`. To forbid them, a simple extension to the type checker could ensure that all methods that call `callcc` or `Continuation.call` are annotated with a special annotation, and that no such methods are called from within blocks protected by a `finally` handler. Relaxing the guarantees for `finally` is the road taken by Ruby and Smalltalk, who will not honor their equivalent of `finally` when continuations are involved.

An interesting alternative to try-finally is Scheme's `dynamic-wind` [10], which works like a try-finally with a prelude: whenever control enters the block (either normally or through a continuation), the *prelude* is run; the same goes for leaving the block and the *postlude*. Such a method can be easily written when having `callcc` [11].

The best way to deal with continuations in the presence of Java's try-finally is an interesting research question, and has to be dealt with by any implementation that adds continuation to the language. However, our focus in this paper is the VM side of things, so we will not explore further.

3.2 Synchronization

Capturing a continuation inside a `synchronized` block means that such a block can be re-entered any number of times through that continuation. Code inside that block assumes that a number of locks have been acquired and therefore, whenever it executes, they have to actually be held.

Our solution is that `Continuation.call` fails (with an `IllegalMonitorState exception`) if the current thread does not hold exactly the same monitors as the one which captured the continuation. To see why, we need to notice that finalizers and monitors are closely related. A `synchronized` block implicitly defines a finally handler which will release that monitor. It follows that we have the following restrictions on callers of a continuation captured inside a synchronized block:

Destination compatibility The caller must be able to release at least the same monitors as the ones active at the point of capture. This is because it will run the special finally handlers.

Source compatibility The caller must not hold monitors other than the ones active at the point of capture. This is because it will drop the current execution context, and its own finally handlers will not be executed leading to unreleased monitors.

It follows that the two sets of monitors have to be equal. Our implementation keeps track of the monitors acquired by a thread and makes the necessary checks when a continuation si captured or applied. User code can test whether a continuation is valid in the current context by calling `Continuation.isValid`. This problem seems to have gone unnoticed by the creators of RiFE [5].

3.3 Threads

Continuations refer to the thread that created them. What happens if a thread calls a continuation captured by another thread? Delimited and one–shot continuations fail at runtime. Since they carry only a part of the execution context, they can't recreate it on the target thread. Full continuations could, in principle, be called in a different thread than the one who captured them (our current implementation does not allow this).

3.4 Java Security Model

The Java security model [12] uses a stack–walking algorithm for deciding whether the access to some resource should be permitted or not. Access is granted if *all* the code on the stack has the required permission (we ignore doPrivileged for simplicity, as it does not affect this issue). Since continuations capture (portions of) the stack, they carry around the security context of the code who captured them, and makes it more difficult to reason about security. Indeed, it's not only control flow (who calls whom), but also data flow (what continuations can reach a given `call`) that has to be taken into account. Imagine a continuation that points to some security sensitive operation (like formatting the hard drive) that reaches untrusted code (by careless programming). Untrusted code can call this continuation and no one could stop disaster from happening. This problem is similar to that of thread creation: a new thread has an empty stack, therefore it could perform some sensitive actions which, later, could leak to the code that created the thread. That code might not have had the necessary permissions to carry on those operations itself. The security model handles this by making new threads inherit the security context of their parent.

A similar solution for continuation would be to record for each thread the security contexts at all points where a continuation was called, and modify the algorithm to take them into account. However, this is not a satisfactory solution since it implies an ever increasing chain of security contexts. Our implementation does not address this issue, which should however be kept into consideration when dealing with security-sensitive applications.

4 Implementation

We implemented continuations in Ovm [13], a framework for building virtual machines, coming with several implementation of VM services (garbage collection, monitors, or execution engine) which can be combined to build a working JVM. We used the *j2c* execution engine, which is an ahead-of-time compiler that uses C++ as target.

4.1 Garbage Collection

We used a conservative, mostly copying garbage collector. Using C++ as a target has the disadvantage that the garbage collector has to be able to deal with lack

of precision. While pointers from objects can be precisely identified, pointers from the stack have to be handled conservatively, and the pointed objects have to be marked as not movable (pinned). Since continuations are not ordinary objects, but carry stacks with them, they need to be visited *before* traversing the reachability graph, and their "neighbours" pinned. This last requirement comes from the fact that an object can be reached before its continuation is visited, and therefore moved before it had the chance to be pinned. Note that this issue does not appear in VMs that use heap–allocated activation frames, such as [14], as they can be treated as ordinary objects.

We modified the mostly copying GC to handle continuations correctly. We keep around a list of *live* continuations which is updated each time a continuation is captured (continuation capture, as well as continuation calls are atomic operations). When the GC starts, it visits all live continuations and treats their stacks and registers as conservative roots. A subtle problem arises: since Ovm provides no weak references, and continuations are referenced from the GC itself, they will never be collected. Our solution is to "manually" garbage collect the live continuation list. We use a simple mark and sweep algorithm which marks continuations in the live list when they are visited during normal GC walk. At the end of the GC the list is swept. This way we guarantee that dead continuations will be collected in the second GC run after they are dead.

4.2 Monitors

In order to satisfy the monitor-affinity property of continuations, we need to keep track of the acquired locks. Each thread maintains a list of entered monitors, which is updated on monitor enter and exit. When a continuation is captured, the list of monitors is saved. When a continuation is called, we check that the monitors entered by the current thread are the same as the ones saved in the continuation. If this is not true, an `IllegalMonitorState` exception is thrown. Continuations are invalidated eagerly, as soon as the most recently acquired monitor has been released. This allows the programmer to test a continuation before attempting to call it.

A current limitation is that thin locks [15] are not supported. Thin locks use a partial word in objects to perform fast locking when there is no contention.

4.3 Performance

We present some preliminary performance results of our prototype. Figure 4 shows the cost of different operations involving continuations. Our testing configuration is real time Ovm with the j2c backend, and a mostly–copying garbage collector. Each data point in the figure is an average over ten measurements.

Continuation capture is the most expensive operation. Its cost increases lineary with the stack depth, and up to depths of 50 activation frames it is lower than the cost of creating a thread in our system. Its cost is roughly 8 times the cost of a normal call. One–shot continuations show a constant cost relative to stack depth, about the same as a normal method call. It is interesting to note

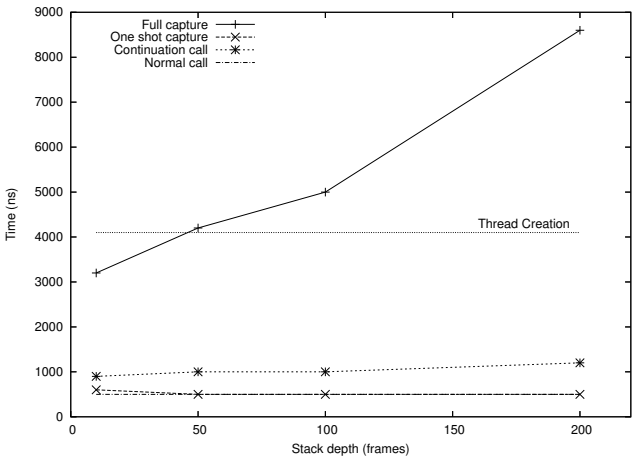

Fig. 4. Cost of continuation operations in different settings. We used a dual–core, Intel Xeon 3GHz machine with 2GB RAM, running Linux 2.6.15.

that the cost of a continuation call is almost constant relative to stack depth, roughly 2 times more expensive than method calls. We believe the difference between capture and call is due to memory allocation that takes place during capture. The cost of continuation capture is an area where we expect our future implementation to show significant improvement. We didn't include measurements for bounded continuations as their behavior relative to stack depth is similar to that of full continuations.

5 Conclusions

We have presented a way to integrate first–class continuations in the Java language. We studied the interactions between existing language features and continuations and suggested possible approaches to reconcile the existing semantics of Java regarding exceptions, threads, monitors and the security model. We have implemented continuations in a Java VM, we have a system that handles monitors correctly, and we conducted preliminary performance measurements. We showed how a conservative copying GC can be extended to deal with continuations. As far as we know, this is the first implementation of continuations in a Java VM.

6 Future Work

We are planning to conduct further work on the interaction between continuations and the "finally" exception mechanism, in order to obtain an efficient implementation that does not sacrifice the expected exception semantics.

Continuation capture is eagarly copying the whole execution stack. We plan to implement a more efficient scheme using lazy copying [16].

References

1. John C. Reynolds. The discoveries of continuations. *Lisp and Symbolic Computation*, 6, 1993.
2. Paul Graunke, Shriram Krishnamurthi, Steve Van Der Hoeven, and Matthias Felleisen. Programming the Web with high-level programming languages. *Lecture Notes in Computer Science*, 2028:122, 2001.
3. Stéphane Ducasse, Adrian Lienhard, and Lukas Renggli. Seaside – a multiple control flow web application framework. *ESUG International Smalltalk Conference*, September 2004.
4. Greg Pettyjohn, John Clements, Joe Marshall, Shriram Krishnamurthi, and Matthias Felleisen. Continuations from generalized stack inspection. In *International Conference on Functional Programming*, 2005.
5. Rife continuations. http://rifers.org/wiki/display/RIFECNT/Home.
6. Matthias Felleisen, Daniel P. Friedman, Bruce Duba, and John Merrill. Beyond continuations. Technical Report 216, Indiana University, 1987.
7. Carl Bruggeman, Oscal Waddell, and R. Kent. Dybvig. Representing control in the presence of one-shot continuations. In *Conference on Programming Language Design and Implementation*. ACM SIGPLAN, 1996.
8. Christian Queinnec. A library of high-level control operators. *Lisp Pointers, ACM SIGPLAN Special Interest Publ. on Lisp*, 6(4):11–26, 1993.
9. Tim Lindholm and Frank Yellin. *The Java(TM) Virtual Machine Specification (2nd Edition)*. Prentice Hall PTR, April 1999.
10. R. Kelsey, W. Clinger, and J. Rees. *Revised^5 Report on the Algorithmic Language Scheme*, chapter 6.4. 1998.
11. Dorai Sitaram. Unwind-protect in portable scheme. In *Scheme Workshop 2003*, November 2003.
12. Li Gong. Java 2 security architecture. 1998.
13. The ovm project. http://ovmj.org.
14. John H. Reppy. A high-performance garbage collector for Standard ML. Technical report, Murray Hill, NJ, 1993.
15. David F. Bacon, Ravi B. Konuru, Chet Murthy, and Mauricio J. Serrano. Thin locks: Featherweight synchronization for java. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 258–268, 1998.
16. William D. Clinger. Implementation strategies for first–class continuations. *Higher–Order and Symbolic Computation*, 12:7–45, 1999.

One Method At A Time Is Quite A Waste Of Time

Andreas Gal, Michael Bebenita, and Michael Franz

Computer Science Department
University of California, Irvine
Irvine, CA, 92697, USA
{gal,bebenita,franz}@uci.edu

Abstract. Most just-in-time compilers for object-oriented languages operate at the granularity of methods. Unfortunately, even "hot" methods often contain "cold" code paths. As a consequence, just-in-time compilers waste time compiling code that will be executed only rarely, if at all. We discuss an alternative approach in which only truly "hot" code is ever compiled.

1 Introduction

Many modern object-oriented languages such as Smalltalk [7], Java [10, 1] and C# [8] have a virtual machine-based execution model. Just-in-time compilation is often used to translate the virtual machine bytecode into native machine code for faster execution.

Just-in-time compilers used in virtual machines are often quite similar in structure to their static counterparts. In case of static compilation, the compiler processes the program code method by method, constructing a control-flow graph (CFG) for each method, and performing a series of optimization steps based on this graph. In the final step the compiler traverses the CFG and emits native code.

Most dynamic compilers behave essentially identically: pick a method, construct its CFG, and generate native code for it. In order to strike a balance between startup latency and long term efficiency, JIT compilers often operate in a mixed mode environment instead of compiling the entire program. In the case of Java, bytecode is first executed through an interpreter. Methods that are invoked often are identified as "hott" and are dynamically compiled into native code.

In a static compiler, using methods as compilation units is a natural choice. In static compilation there is usually no profiling information available that could reveal whether any particular part of a method is "hotter" and thus more "compilation worthy" than others, it actually makes perfect sense to always compile entire methods and all possible paths through them. After all, for a static compiler, all these different paths look equally likely to be taken at runtime.

This is dramatically different in case of a dynamic compiler. In contrast to its static counterpart, a dynamic compiler has access to runtime profile information that the virtual machine can collect easily while it interprets code. With this profile information, the dynamic compiler can decide which parts of a method actually contribute to the overall runtime, and which parts are rarely taken and are in fact irrelevant from a global perspective as far as optimization potential is concerned.

Region-based compilation was proposed to address this issue. Suganuma et al. [13], proposed a region-based just-in-time compiler for Java. It uses runtime profiling to select code regions for compilation and uses partial method inlining to inline profitable parts of method bodies only. The authors observed not only a reduction in compilation time, but also achieved better code quality due to rarely executed code being excluded from analysis and optimization.

However, region-based compilation really only addresses the symptoms by trying to exclude unprofitable code areas from compiled methods, instead of addressing the actual problem, which is the choice of an unsuitable compilation unit in the first place.

The unsuitability of methods as compilation units becomes even more apparent when trying to deal with long running hot code regions inside a method that is currently being interpreted. Once such a method is discovered to be hot, it is subsequently compiled and optimized to directly executable native code. The runtime system then has to perform an expensive and complex process called *on-stack replacement* [9] to substitute the newly compiled code for the interpreted version. For a virtual machine, being able to deal with on-stack replacement means having to support side-exits from running interpreted methods, and side re-entries into compiled method bodies. Both of these processes are so complex that very few open-source or research virtual machines actually support on-stack replacement. Therefore it is a feature found mostly only in commercial VMs and large-scale research efforts such as Jikes RVM [2, 4].

We have been exploring a different approach to building dynamic compilers in which no CFG is ever constructed, and no source code level compilation units such as methods are used. Instead, we use runtime profiling to detect frequently executed cyclic code paths in the program. Our compiler then records and generates code from dynamically recorded *code traces* along these paths. It assembles these traces dynamically into a tree-like data-structure that covers frequently executed (and thus compilation worthy) code paths through hot code regions. [5, 6]

Our tree-based code representation has a number of advantages. On the one hand, it subsumes and greatly simplifies on-stack replacement. In our scheme, compiled code is always entered independently of method boundaries. Thus replacing interpreted code with the compiled equivalent becomes trivial. When a trace has been recorded, the interpreter stops at the loop header (which is the entry point of the associated trace tree) and the entire tree is recompiled. The trace tree can immediately be executed, replacing the previously compiled copy. Since recording is only performed after the native code has side exited into the interpreter, its not necessary to actually implement an on-stack replacement mechanism that translates from one compiled state into another.

The other major benefit of our approach is that our trace tree data structure only contains actually relevant code areas. Edges that are not executed at runtime (but appear in the static CFG) are not considered in our representation, and are delegated to the interpreter in the rare cases they are taken. Unlike compilers that use basic-block based CFG analysis where advanced optimizations are expensive, our tree-based compiler can perform advanced optimizations quickly. The lack of control flow merge points in our tree-based representation simplifies optimization algorithms.

2 Trace Compilation

Our trace-based JIT compiler targets the JVM bytecode language and starts the execution of class files through an interpreter, much like traditional JIT compilers do. To detect "hot" code areas that are suitable candidates for dynamic compilation, we use a simple heuristic first introduced by Bala et al. [3]. The interpreter is augmented to keep track of frequently executed backwards branches. The targets of such branches are often the loop headers of hot code areas. Once such a loop header is discovered, we attempt to record a trace through the loop region associated with the header. Starting at the loop header, the interpreter records subsequent instructions until the original loop header is reached again.

During the recording phase, branch instructions are recorded as *guard* instructions and indicate possible loop exits that must be safeguarded against during native code execution. Our JIT compiler emits appropriate code to check that a previously observed result of a guard condition still applies during future executions of the compiled trace. If the condition fails, a side exit is performed and the compiled code hands control back to the interpreter and resumes it in a state consistent with the side exit detected during the native code execution.

During trace recording, method invocations are inlined into the trace. In case of a static method invocation, the target method is fixed and no additional runtime checks are required. For dynamically dispatched methods, the trace recorder inserts a guard instruction to ensure that the same actual method implementation is reached as was found during trace recording. If the guard fails, a regular dynamic dispatch is repeated in the interpreter. If it succeeds, we have effectively performed method specialization on a predicted receiver type. Since our compiler can handle multiple alternative paths through a trace, eventually our compiler specializes method invocations for all commonly occurring receivers.

Guard instructions that appear in inlined methods must carry enough information to be able to reconstruct the interpreter's state in case a side exit occurs. In case of a side exit in the same method scope that the trace was entered, all that has to be done is to write back any values held in machine registers into the appropriate stack and local variable locations. In case of a "deep" side exit inside a method invocation that is being inlined, guard instructions must store enough information to allow the virtual machine to fully construct the interpreter state, which includes constructing method frames on the virtual machine stack.

During native code execution, if a guard instruction fails and a side exit occurs, our JIT compiler must resume execution through the interpreter. Since this switch can be expensive, our JIT compiler attempts to include traces that splinter off at side exits into the original trace. For this, at every side exit we resume interpretation, but at the same time also re-start the trace recorder to record instructions starting at the side exit point. These secondary traces are completed when the interpreter revisits the original loop header. This results in a tree of traces, spanning all observed paths through the original program's loop.

Figure 1 shows the control flow graph of a nested loop, and the corresponding trace tree that is recorded once instruction 2 is detected as a loop header. From left to right, straight line tree paths represent the traces that were recorded. The first trace to be

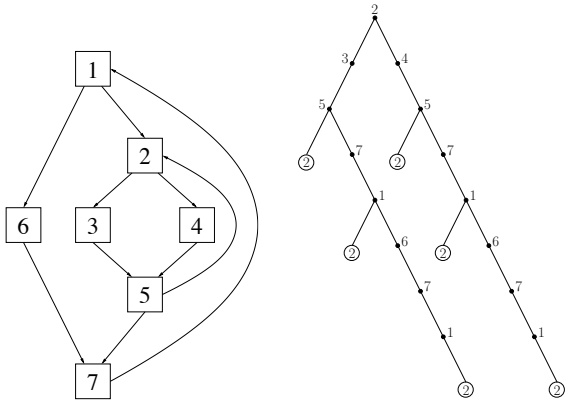

Fig. 1. Example of a nested loop and a set of suitable traces through the nested loop. Each basic block in the control-flow graph of the loop (left) is represented as a numbered node. (1) is the loop header of the outer loop, (2) the loop header of the inner loop. Since (2) is executed more frequently than (1), it will become the anchor node of the trace tree (right) that is recorded for this loop. In the trace tree every trace starts at the anchor node and also terminates at it. Since (2) is the anchor node of the trace tree, every trace ends with node (2). The loop header of the outer loop is inlined into the trace tree as just another trace originating and terminating at node (2).

recorded is $\{2, 3, 5, 2\}$ followed by $\{2, 3, 5, 7, 1, 6, 7, 1, 2\}$ and then $\{2, 3, 5, 7, 1, 2\}$, etc. After every secondary trace that is recorded and added to the trace tree, our JIT compiler recompiles the entire trace tree and emits new native code for it. At the next entry of the compiled code through the loop header, this new version is executed until another side exit is encountered, in which case the tree can be extended further.

The first recorded trace is usually the hottest path through the loop. Secondary traces are less likely to occur during execution and thus benefit less from compilation. Our compiler limits the growth of trace trees to an experimentally chosen number of traces.

3 Evaluation

We have built a dynamic compiler for Java based on the described trace tree compilation method. The compiler was implemented as part of the JamVM [11] Java Virtual Machine, which is a fast Java interpreter but previously lacked a just-in-time compiler.

To detect trace tree entry points we modified JamVM's interpreter to keep track of the execution frequency of JVML branch instructions. Once a certain threshold is surpassed the trace recorder will attempt to record a primary trace starting at the destination of such frequently executed branch instructions.

After compiling the primary trace, the trace is executed in compiled form. The trace recorder is started whenever a side exit occurs, attempting to grow the associated trace tree.

Our current compiler prototype only performs a limited set of optimizations including constant propagation, copy propagation, arithmetic optimizations, loop invariant

code motion and array bounds check elimination. The optimized intermediate representation of the trace tree is then converted to PowerPC machine code.

With the current set of optimizations and our rather simplistic PowerPC code generator that does not perform instruction scheduling, we are able to speed up the execution of section 2 of the Java Grande benchmark [12] set by factor 5 to 12.

On average, our system generates approximately 1500 bytes of machine code for each benchmark program in section 2, which is significantly smaller than the code emitted by method-based just-in-time compilers such as Sun's Hotspot JVM [14]. The latter generates approximately 30kB of code for each Section 2 benchmark program, but also achieves an additional speedup of 1.5 to 2 over the performance of our prototype compiler.

For most benchmarks in section 2 the recorded trees achieve near-ideal coverage of the performance critical part of the loop with 2-4 traces. This means that adding additional traces will no longer produce any substantial speedup since all frequently executed paths are covered. The largest trace tree is produced for *HeapSort*. The second and third trace added to the tree produce a speedup of 2.5 over the initial trace by reducing side exit frequency. The fourth trace produces another speedup of 2. Adding additional traces beyond this point will not produce any visible speedup.

Also noteworthy about our prototype compiler is the compilation speed. Our system spends 100 to 750 times less time in the just-in-time compiler than Sun's Hotspot JVM. While in part this can be explained with the fact that we compile less code in general, this would only explain a compilation time speedup of approximately factor 20. Thus, the overall compilation performance highlights that our trace-based compilation approach is not only more selective when deciding what code to compile, but is also less time consuming per instruction compiled.

4 Discussion

Traditional dynamic compilers are often built based on the same principles and ideas that were invented for their static counterparts. We introduced a novel intermediate representation which we call trace trees that eliminates the inherent weaknesses of using methods as compilation units in a dynamic compiler. The trace tree data structure introduced in this paper enables our just-in-time compiler to incrementally discover alternative paths through a loop and then optimize the loop as a whole, regardless of a possible partial overlap between some of the paths.

When programs execute, the dynamic view of basic blocks and control flow edges that one encounters can be quite different from the static control flow graph. Our trace-tree representation captures this difference and provides a representation that solely addresses "hot" code areas and "hot" edges between them. All other basic blocks and instructions never become part of our compiler's intermediate representation and therefore do not create a cost for the compiler.

Using trace trees as compilation units instead of source code constructs such as methods also provides a intuitive boundary between "cold" code that is interpreted and "hot" code that needs to be compiled, and defines a simple and efficient method to transition between such areas (trace entry and trace side exits).

5 Acknowledgement

This research effort is partially funded by the National Science Foundation (NSF) under grant CNS-0615443. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the National Science Foundation (NSF), or any other agency of the U.S. Government.

References

1. K. Arnold and J. Gosling. *The Java Programming Language (2nd ed.)*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1998.
2. M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive optimization in the Jalapeño JVM. *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 47–65, 2000.
3. V. Bala, E. Duesterwald, and S. Banerjia. Transparent dynamic optimization: The design and implementation of Dynamo. *Hewlett Packard Laboratories Technical Report HPL-1999-78. June 1999*, 1999.
4. M. Burke, J. Whaley, J. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. Serrano, V. Sreedhar, and H. Srinivasan. The Jalapeño dynamic optimizing compiler for Java. *Proceedings of the ACM 1999 conference on Java Grande*, pages 129–141, 1999.
5. A. Gal. *Efficient Bytecode Verification and Compilation in a Virtual Machine*. PhD thesis, University of California, September 2006.
6. A. Gal, C. W. Probst, and M. Franz. HotpathVM: An effective JIT compiler for resource-constrained devices. In *VEE '06: Proceedings of the 2nd International Conference on Virtual Execution Environments*, pages 144–153, New York, NY, USA, 2006. ACM Press.
7. A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1983.
8. A. Hejlsberg, P. Golde, and S. Wiltamuth. *The C# Programming Language*. Addison-Wesley Professional, 2003.
9. U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming Language Design and Implementation (PLDI)*, pages 32–43, New York, NY, USA, 1992. ACM Press.
10. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.
11. R. Lougher. JamVM virtual machine 1.4.3. `http://jamvm.sf.net/`, May 2006.
12. J. A. Mathew, P. D. Coddington, and K. A. Hawick. Analysis and development of Java Grande benchmarks. In *Proceedings of the ACM 1999 Java Grande Conference, San Francisco*, 1999.
13. T. Suganuma, T. Yasue, and T. Nakatani. A region-based compilation technique for dynamic compilers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(1):134–174, 2006.
14. Sun Microsystems. The Java Hotspot Virtual Machine v1.4.1, Sept. 2002.

Type Feedback for Bytecode Interpreters

Position Paper
ICOOOLPS 2007

Michael Haupt[1], Robert Hirschfeld[1], and Marcus Denker[2]

[1] Software Architecture Group
Hasso-Plattner-Institut
University of Potsdam, Germany
[2] Software Composition Group
Institute of Computer Science and Applied Mathematics
University of Berne, Switzerland
{michael.haupt,hirschfeld}@hpi.uni-potsdam.de, denker@iam.unibe.ch

Abstract. This position paper proposes the exploitation of type feedback mechanisms, or more precisely, polymorphic inline caches, for purely interpreting implementations of object-oriented programming languages. Using Squeak's virtual machine as an example, polymorphic inline caches are discussed as an alternative to global caching. An implementation proposal for polymorphic inline caches in the Squeak virtual machine is presented, and possible future applications for online optimization are outlined.

1 Introduction

Bytecode interpreters are small in size and comparatively easy to implement, but generally execute programs much less efficiently than just-in-time (JIT) compilers. Techniques like threaded interpretation [9, 11, 2] focus on speeding up bytecode interpretation itself, and caching [4, 5, 1] improves the performance of message sends—the most common operation in object-oriented software [7].

It is interesting to observe that, while threading mechanisms are used naturally to a varying degree in bytecode interpreter implementations, such systems usually employ only *global caching* to speed up dynamic method dispatch. A global cache is clearly beneficial with respect to overall performance. Still, it does not provide optimal support for polymorphic message send sites, and it does not allow for exploiting type information (we provide details on these issues in the following section). In our opinion, the employment of *polymorphic inline caches* (PICs) [5] instead can provide means for achieving significant speedups in bytecode interpreters while exhibiting only a moderate increase in memory footprint and implementation complexity.

In the next section, we briefly discuss global caches. The bytecode interpreter we use as a case study throughout this paper is that of the Squeak [8, 12] virtual machine (VM) [13]. Section 3 proposes an approach to the implementation of

PICs in the Squeak bytecode interpreter. Finally, section 4 gives a summary of the paper and an outlook on possible future optimizations in bytecode interpreters that are encouraged by the introduction of type feedback mechanisms.

2 Global Caching: Discussion

The Squeak VM bytecode interpreter uses a global cache for improving method lookup performance. This cache has a fixed size and maps <*target class, selector*> pairs to concrete method implementations (compiled methods). It significantly contributes to the overall performance of the Squeak interpreter.

However, such a cache has several shortcomings. Since it is global, collisions are relatively frequent and lead to longer method lookup times. The cache has to be flushed as soon as changes in the class hierarchy or in method implementations occur. For changes in method implementations, the cache is not entirely flushed, but only for the entries that refer to implementations of the *selector* in question.

Moreover, flushing is required whenever the garbage collector performs heap compaction, as hashing is done based on target class and selector object *addresses*. After a flush, the cache needs to be repopulated, during which and overall performance is lower.

The global cache suffers from being global. It cannot react to local changes in an adequate way—i. e., by an update operation that is quasi-local in its effect on cache contents. A local change, such as a class overwriting an inherited method, actually affects only a small part of the entire class hierarchy. Nevertheless, method lookup data for large parts of the class hierarchy needs to be restored.

Also, the global cache is, due to its mapping scheme, generally not able to provide local information, that is information *per send site*. Such information typically comprises of the concrete receiver types (classes) of a message at a given polymorphic send site. It is called *type feedback information* [6] and is very interesting with regard to optimizations (cf. Sec. 4).

The performance of the Squeak bytecode interpreter is good. Still, we believe that it can benefit from a caching mechanism that supports local type feedback. These so-called *inline caches* [4] are usually used in environments that employ JIT compilation and bring great benefit in terms of dynamic message dispatch performance.

Inline caches store the most recently looked-up method address at each given send site. The address is cached at the send site, replacing the call instruction to the lookup method with a direct jump to the code of the method. Since there is a jump to the cached method, no lookup needs to be done at all. There is only a slight overhead resulting from a check at the beginning of the method verifying that the class of the receiver is the correct one. In case the receiver class has changed, the standard lookup is used instead. Obviously, simple inline caches are not an ideal solution for supporting polymorphic send sites since they already fail if the same message is sent to an alternating list of objects.

PICs [5] store, for a given send site, the method addresses of N past message sends, where N is the cache size. A PIC stores <*receiver class, method address*>

key-value pairs. The Strongtalk [14] VM is a prominent example of such an environment. It is comprised of a bytecode interpreter and a JIT compiler; the interpreter already stores type feedback information in PICs to speed up message sending. Information stored in these PICs is later exploited by JIT-compiled native code.

3 Polymorphic Inline Caching in Bytecode Interpreters

We believe that PICs can be beneficial also in solely interpreting VM implementations, such as the Squeak VM [13]. In this section, we outline an implementation proposal for PICs in that environment.

An interpreter does not generate binary code for methods, thus PICs cannot store memory addresses of code. In Squeak, the bytecode is stored in compiled method objects. Here, PICs can store a reference to the compiled method object instead.

The implementation affects both the Squeak VM and reflectively the Smalltalk *image*. At the image level, each Smalltalk method is represented as an instance of the `CompiledMethod` class. The format of `CompiledMethod` instances needs to be modified to store send site type feedback information. In the VM, the interpreter logic must be augmented to support storing and updating of said information.

Squeak `CompiledMethod`s have the layout shown in Fig. 1. The standard *object header* provides information about the object itself, its class, hash value, etc. The subsequent *method header* contains information on the method in question, such as the number of arguments, local variables and literals. After that, there are several pointers to the method's *literals*, each referencing a given constant occurring in the method. For example, all send bytecodes reference their selector (the name of the method to call) as an offset in the literal frame. The *bytecode* array represents the method code, and the *trailer* carries additional information about the method's source code location.

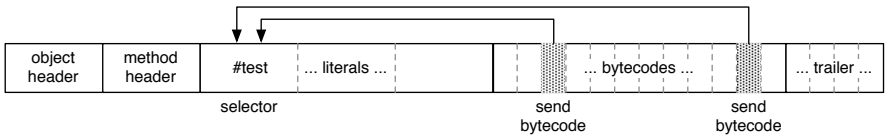

Fig. 1. Object layout of a Squeak `CompiledMethod` instance.

For the PIC implementation in Squeak, the *literals* region in `CompiledMethod` instances is of special interest. In the current Squeak system, the compiler generates one slot in the literal frame for any unique selector. This means that send bytecodes sending the same selector reference the same slot in the the literal array (cf. Fig. 1). The Smalltalk compiler needs to be modified to generate one

entry in the literal array for each send, without the sharing property mentioned above.

Thus we have as many literal slots storing selectors as there are send bytecodes in the `CompiledMethod` (cf. Fig. 2). Each of these slots can hold a reference to an object carrying type feedback information. To this end, several dedicated classes (described below) are to be introduced into the image.

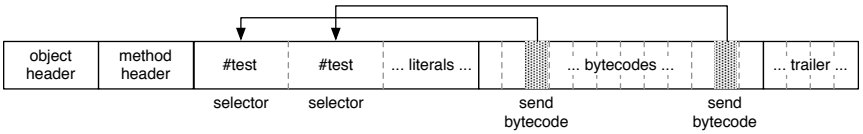

Fig. 2. A `CompiledMethod` instance without selector sharing.

Initially, all selector slots contain selectors. Once a send site is visited by the bytecode interpreter and the corresponding message is sent, the selector is replaced by a reference to an inline cache (IC) object (cf. Fig. 3), an instance of the `InlineCache` class. An IC object contains five values: the selector, the most recent target class of the send, the address of the most recently looked-up method for the selector, a hotness counter, and a trip counter.

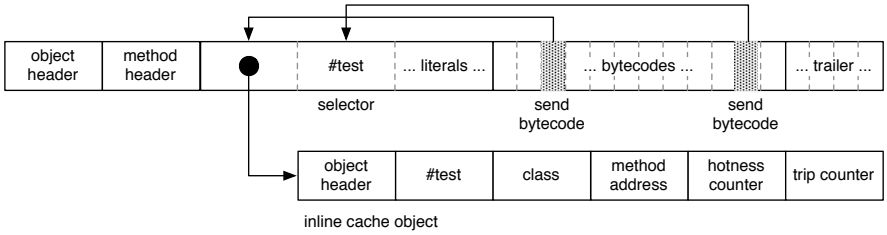

Fig. 3. A former literal slot referencing an IC object.

The bytecode interpreter increases the hotness counter each time it executes the corresponding message send. It then also checks whether the target class is the same as it was when the send was executed the last time. If so, the stored method address is used to retrieve the method implementation to be executed. If the receiver class check fails, the trip counter is increased, and the correct class and implementation are looked up and stored.

If such a send site causes actual lookups too often, its IC object reference can be replaced with a PIC object reference (cf. Fig. 4). A PIC object is organized much like an IC object: it also carries the selector in question and type feedback

information. The notable difference is that a PIC object carries up to 8 triples of receiver class, method address, and hotness counter.

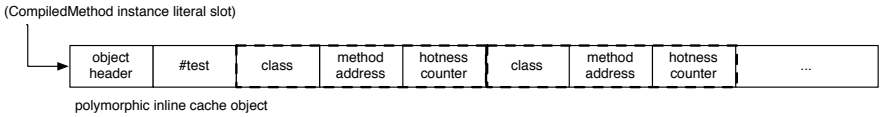

Fig. 4. A PIC object referenced from a `CompiledMethod` instance.

The bytecode interpreter, when executing the corresponding send, iterates over the PIC and checks for the correct receiver class. Once it finds one already stored, the respective stored method is executed and the pertaining hotness counter is increased. If no matching receiver class is found, lookup proceeds as usual and the result is stored in a free slot of the PIC object if available. It is not necessary to store a hotness counter alongside with each PIC object entry, but future optimizations can benefit from this information (see below).

4 Summary and Future Optimizations

In the previous sections, we have discussed caching optimization mechanisms for bytecode interpreters. In our opinion, global caches are, although helpful with regard to performance, not fully supportive of dynamic optimizations possible in interpreters. For that we propose the introduction of PICs based on local type feedback.

The *locality* of type feedback information is a feature of PICs that can be exploited beyond performance improvements in the VM. Type feedback information made available at the *image level* facilitates optimizations above the abstraction barrier imposed by the Squeak VM.

The AOStA (Adaptively Optimizing Smalltalk Architecture) project [10] relies on type feedback from the VM to dynamically and adaptively optimize Smalltalk bytecodes in the image using bytecode manipulation. An example of such an optimization is method inlining: based on type feedback information and hotness counter data, methods frequently invoked from a given send site can be inlined directly at the image level, without the need to create stack frames and method context objects.

Originally, AOStA has been conceived for the VisualWorks VM [3]. The VisualWorks JIT compiler generates PICs at send sites. A system like AOStA can be beneficial also to purely interpreted systems like Squeak Smalltalk if the underlying interpreter supports type feedback using PICs, as proposed in this paper.

References

1. M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney. A Survey of Adaptive Optimization in Virtual Machines. *Proceedings of the IEEE*, 93(2), 2005.
2. M. Berndl, B. Vitale, M. Zaleski, and A. D. Brown. Context Threading: A Flexible and Efficient Dispatch Technique for Virtual Machine Interpreters. In *CGO '05: Proceedings of the international symposium on Code generation and optimization*, pages 15–26. IEEE Computer Society, 2005.
3. Cincom Home Page. `http://www.cincomsmalltalk.com/`.
4. L. P. Deutsch and A. M. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *POPL '84: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 297–302. ACM Press, 1984.
5. U. Hölzle, C. Chambers, and D. Ungar. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *ECOOP '91: Proceedings of the European Conference on Object-Oriented Programming*, pages 21–38. Springer-Verlag, 1991.
6. U. Hölzle and D. Ungar. Optimizing Dynamically-Dispatched Calls With Run-Time Type Feedback. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 326–336. ACM Press, 1994.
7. U. Hölzle and D. Ungar. Do Object-Oriented Languages Need Special Hardware Support? In *ECOOP '95: Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 283–302. Springer-Verlag, 1995.
8. D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the Future: the Story of Squeak, a Practical Smalltalk Written in Itself. In *Proc. OOPSLA 1997*, pages 318–326. ACM Press, 1997.
9. P. Klint. Interpretation Techniques. *Software—Practice and Experience*, 11(9):963–973, 1981.
10. E. Miranda. A Sketch for an Adaptive Optimizer for Smalltalk written in Smalltalk. unpublished, 2002.
11. I. Piumarta and F. Riccardi. Optimizing Direct Threaded Code by Selective Inlining. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 291–300. ACM Press, 1998.
12. Squeak Home Page. `http://www.squeak.org/`.
13. Squeak Virtual Machine Home Page. `http://www.squeakvm.org/`.
14. Strongtalk Home Page. `http://www.strongtalk.org/`.

Language and Runtime Implementation of Sessions for Java

Raymond Hu[1], Nobuko Yoshida[1], and Kohei Honda[2]

[1] Imperial College London
[2] Queen Mary, University of London

1 Introduction

The purpose of this work is to incorporate the principles of *session types* into a concrete object-oriented language, specifically an extension of Java, as a basis for communications-based programming for distributed environments. Building on preceding theoretical studies of this topic, we present the first practical implementation of such a language, including a treatment of asynchronous communication, higher-order sessions (delegation) and session subtyping. This paper summarises the key design ideas of this work including the runtime architecture. Benchmark results for our current implementation demonstrate that our design introduces minimal overheads over the underlying transport, and is competitive with, in many cases outperforming, RMI, the standard API for typed interprocess communication in Java. A detailed version of this paper [12] is available online [11].

1.1 Background

Sockets Communication is becoming one of the central elements of application development in object-oriented programming languages. A frequent programming pattern in communications-based applications arises when processes interact via some *structured* sequence of communications, which as a whole forms the natural unit of a *conversation*. The *socket* (in particular, the TCP socket) is one of the most widely used programming abstractions for expressing such conversation structures. Available through APIs in many object-oriented languages such as Java and C♯, sockets represent the communication *endpoints* of a bidirectional byte stream abstraction, typically thought of as a *connection*. Although well suited to this purpose, socket-based programming suffers from several disadvantages.

- The byte stream abstraction is too low-level: no direct language level abstraction is provided for what each chunk of raw data means, let alone the structure of a conversation as a whole.
- Control flows in a pair of communicating processes should together realise a consistent conversation structure: the lack of an abstraction for the conversation as a whole means a programmer can easily fail to, for example, handle a specific incoming message or send a message when expected, with no way to detect such mistakes before runtime.

- The socket abstraction is directly coupled to a specific transport mechanism. Thus streams are tied to a physical connection, which complicates, for example, the delegation of an ongoing conversation.

These observations motivate the search for an abstraction mechanism for object-oriented programming that can naturally represent diverse conversation structures and be efficiently mapped to representative transport mechanisms, whilst preserving the familiar programming style of socket. Note Java RMI supports type-safe communication, but the rigid shape of method call makes it difficult to express general communication patterns using RPC.

Session Types A *session* is essentially a unit of sequential conversation, and the associated *session type* is an abstraction of the conversation structure and the messages exchanged, against which the communication behaviour of a program can be validated. Session types have been studied in many contexts in the last decade, including π-calculus-based formalisms [9]; multi-threaded functional languages [16]; CORBA [15]; operating systems [7]; and Web services [2].

With respect to the present problem, recent studies [6, 5, 4, 3] have demonstrated a clean integration of session type theory with object-oriented languages, through formalisms distilling selected object-oriented concepts for accurate analysis. Our work furthers these studies by contributing the design and implementation of a concrete, distributed language with session communication primitives and type system, including the key components of the runtime architecture such as the protocols that guarantee type-safe session delegation.

1.2 Problem Outline

The task at hand can be divided into three main problems.

Session programming abstractions. The abstractions should be expressive, enabling the representation of diverse conversation structures, and moreover *usable*, which stipulates a combination of clear syntax and intuitive (unsurprising) semantics, generating programming patterns natural to OOP. Naive implementation of the simplified theory has limitations other than usability, for instance the object calculi mentioned above [6, 5, 4, 3] do not permit operations on different sessions to be interleaved, precluding many real-world communication patterns. Exception handling for session operations is another such issue not addressed in the preceding theoretical work but certainly required for practical use.

Integration of session types. Session type theory has often focused on type inference, whereas Java has explicit type declaration: an implementation of session types closer to the latter would probably be easier for Java programmers to understand. Hence, syntax for session type declaration and an algorithm for static type checking (including new features such as interleaving and exceptions) are required. In addition, we consider how the standard imperative constructs in Java should combine with the chosen session programming abstractions, as reflected by our extended type system.

Runtime architecture. One of the main technical contributions of the present work is the design of the runtime support for asynchronous session communication, with the use of session type information as a fundamental element. Firstly, a safe execution of communication cannot be verified for distributed systems at compiletime, as a communicating party cannot statically assess the behaviour of peers discovered only at runtime. We solve this problem using a validation mechanism at session initiation, in which the two parties exchange their session types to determine whether or not their interactive behaviours are compatible. This session type information is used throughout the established session by both parties; for example, session types play a crucial part in coordinating the protocols for higher-order communication, *session delegation*. To validate the feasibility of our approach, we measure the performance of our primitives using several benchmark programs, comparing their cost over the untyped transport.

2 Approach

We outline the approach of our current design-implementation framework for the proposed language [11]. A core design feature is the use of session types in decoupling user description of session operations (abstraction concerns) and their execution mechanism (implementation concerns), analogous to high-level control flows and the underlying machine instructions in structured sequential programming. The key elements of our approach, addressing the issues described in the previous section, are as follows.

1. A **type syntax** for sessions based on [5, 6, 4, 3], but with enhanced readability and conformance to Java syntax.
2. Object-based **session programming primitives** that present an API-style interface. The fundamental abstraction is the *session-typed socket*, which represents a session endpoint.
3. A **new programming discipline/style**, derived from the first two points, for communications-based programs with guaranteed type and communication safety, which begins with a specification of intended communication structures using session types.
4. **Static session type checking**, implemented using the Polyglot compiler framework [14], coupled with **dynamic compatibility validation** at session initiation through a handshake protocol. Both components utilise session subtyping [8, 2].
5. The **runtime support** for the session abstraction, which encapsulates a variety of communication mechanisms with minimal overhead whilst abstracting from physical connections. The runtime incorporates the protocol for session initiation as well as delegation and close, and makes extensive use of session type information from point 3.

Session type declaration. The session type syntax abstracts the basic (object) send and receive actions, conditional behaviour through branching and selection

(rather than binary if-statements, as in [5, 6, 4, 3]), and unbounded behaviour through while-loop iteration. We illustrate using a small example.

```
protocol p {
  begin.          // Session initiation
  ?[              // Iteration
    ?{              // Branch, two possible subconversations (labelled)
      GET: !(T),      // Send (object) type T
      PUT: ?(T)       // Receive (object) type T
    }
  ]*.
  end
}
```

`T` may itself be a session type, representing session delegation.

Session-typed sockets. We augment the standard socket to support the session communication primitives and session type checking. A session-typed socket, hereafter referred to as simply *session socket*, represents a session endpoint, over which session operations are performed like method calls to the socket object. We continue the above example.

```
STSocket s = STSocketImpl.create(p); // 'p' as declared above
s.request(host, port);                // begin.
T t = ...;
s.inwhile() {                        // ?[
  s.branch() {                         // ?{
    case GET: { s.send(t); }             // !(T),
    case PUT: { t = s.receive(); }       // ?(T)
  }                                    // }
}                                    // ]*.
s.close();                           //end
```

Whilst session programming is similar to standard API-based socket programming, the `branch` and `inwhile` operations (also, the corresponding `select` and `outwhile` operations), are new language constructs with intuitive semantics. The branch waits for the opposing session party to select a label, and inwhile iterates according to a control message implicitly communicated between the two session parties; these operations can be thought of as distributed versions of the standard switch and while statements that maintain synchronisation of control flow across both parties.

Session type checking. The type checker tracks the implementation of a session against the specified protocol, observing the correspondence between session operations and their types as demonstrated in the above example. For receive operations, both type inference, from the declared type, and checking, through explicit casting of the received object, are supported. The implementation may *subtype* the specification [8, 2]: a branch can offer more, and a select can use

less, labels than specified. Type checking delegation operations is uniform with normal message types, but cannot occur within an iterative context.

The above issues relate to checking structural correspondence; we must also preserve session *linearity*. For example, aliasing of session sockets is forbidden, and session operations are not permitted within iterative constructs other than in/outwhile. Session implementation may diverge over conditional constructs provided there is a common supertype across all branches: the statement is typed as the lowest such type if it exists, for instance

```
// Session type: !{GET:?(T), PUT:!(T)}
if(...) {
  s.select(GET) { T t = s.receive(); } // !{GET:?(T)}
}
else {
  s.select(PUT) { s.send(new T()); }  // !{PUT:!(T)}
}
```

The type system allows session sockets to be passed as method arguments, which can be thought of as a "local" delegation: methods that accept session sockets specify the expected session type of the socket in place of `STSocket` in their declaration. We also have a treatment of exception handling for sessions.

```
try {
  s.request(host, port);
  ... // Body of session implementation
}
catch(SessionIncompatibleException sie) { ... }
catch(IOException ioe) { ... }
finally {
  try { s.close(); } catch (IOException x) {}
}
```

Sessions should be implemented using the try-catch construct to handle the listed exceptions (or within a method that throws these exceptions). The first exception signals that session initiation has failed because the opposing party has an incompatible behaviour for interaction; this is determined by a *duality* relation on session types (`!(T)` *duals* `?(T)`, etc.) that also permits subtyping e.g. a client that requires just one service may enter a session with a server offering several services. `IOException` is inherited from standard Java socket programming to signal communication failure during a session. For linearity, a session may not span multiple try-catch blocks unless delegated or passed as a method argument; thus, the occurrence of an exception necessarily terminates the session at both session parties. However, session interleaving is freely supported within a single try-statement, which expresses some semantic dependency between such sessions: an exception on any of the sessions is implicitly signalled to the others. Nested session exceptions can be thrown to an outer level, and the type checker permits only the close operation to be performed within the finally-block of a try-statement. The design of the session exception mechanism is ongoing work.

Runtime layer. The runtime layer encapsulates the underlying communication mechanisms; the interface to the runtime layer is the device by which session abstraction is decoupled from actual implementation. This enables exploitation of the transport, using session type information, for efficient communication with minimal overhead. The runtime is currently implemented in Java as the STSocket API: the compiler translates user code to the target API as a source-to-source translation. We describe some of the key components of the runtime.

- **Initiation handshake.** Session initiation involves a duality check between the session types of the two parties; if incompatible, the specified exception is raised at both parties. The current implementation uses literal class naming, which is sufficient for many examples. An earlier implementation [10] has support for class downloading; we plan to further investigate extensions that combine runtime verification of class compatibility and class downloading. Optimisations such as piggy-backing user messages on the handshake for short sessions are possible.
- **Delegation protocol.** The runtime incorporates an implementation of the delegation protocol [12], which governs the interaction between parties in bringing about session delegation in a transparent manner. Session types play a crucial role in treating asynchrony, one of the main design factors of the delegation protocol which includes the case for simultaneous (double) delegation of a session by both parties. Our protocol allows the delegating party to immediately close the delegated session, precluding (indefinite) message forwarding by a proxy agent (an alternative design).
- **Closing protocol.** An additional handshake at session closure is required to handle certain delegation cases, specifically when the passive party of a delegated session performs only output operations, namely send, select and outwhile. This is because this party may asynchronously complete his/her session contract before the delegating party is actually able to perform the delegation operation: the session must be kept "alive" until both parties agree that it has ended. The collaboration between the delegation and closing protocols is non-trivial, and to maintain asynchrony, the latter is performed in a separately spawned thread.

The current implementation of our work includes session sockets based on the Java Socket and NIO libraries. Performance results of several benchmark programs [12, 11] demonstrate that the session-based programming principles proposed in this work can be realised with low overhead. Indeed, our implementation in many cases exhibits better performance than RMI, the standard method for typed inter-process communication in Java. One factor is that the RMI supports additional features such as class downloading; however, the benchmark programs do not use this feature, and so the overheads incurred by RMI are minimal for this point. Moreover, the presence of session types and the information they convey, such as communication direction and (bounded) message size, suggest the potential for further optimisation at both the user and transport level.

3 Conclusion and Related Work

The present work clarifies, through a concrete implementation, the significant impact that the introduction of sessions and session types into object-programming languages can have, on both programming discipline and runtime architecture. Below we summarise the key technical contributions of the present work over the preceding (mainly theoretical) work on session types.

- **Practical programming methodology for session types.** Starting from protocol declaration, we ensure type safety through combined type checking and inference, extending typability with session subtyping.
- **Integration of session types into OOP.** This is realised by the design of session sockets, extending the type system to prevent aliasing of the socket objects, and through a natural and consistent integration with standard imperative constructs and exception handling, allowing session interleaving.
- **Runtime architecture.** The design of the runtime mechanisms is key to the practical use of session types, which in turn are a fundamental element of this design. The runtime encapsulates the underlying message transport and session communication protocols, including the session initiation handshake, where session types are exchanged and validated; and the delegation protocol, which separates the session abstraction from physical connections.

In the following we discuss some of the related works; a more complete discussion such as a comparison with several typed languages based on process calculi (Pict, Polyphonic C♯, Cω, the Concurrency and Coordination Runtime (CCR), JavaSeal, Occam-pi and X10) can be found in [12].

One of the first applications of session types in practice is found in Web Services. The Web Service Description Language (WS-CDL) [17], developed by a W3C standardisation working group, employs a variant of session types to address the need for static validation of business protocols. A WS-CDL description is realised as the interactions of distributed endpoints written in languages such as Java or WS4-BPEL [1], or that proposed by the present work. Recent work [2] has studied the principles behind deriving sound and complete endpoint implementations from a CDL description.

A variant of session types has been combined with a derivative of C♯ for systems programming, playing a crucial role in the development of Singularity OS for shared memory uni/multiprocessor environments [7]. Session types, referred to as contracts, are used to specify the interaction between OS modules as message-passing conversations. Reflecting the hardware and software assumptions of this work (i.e. shared memory and homogeneity of OS modules), features for distribution, such as the session initiation handshake, are not significant. Further, their work does not support subtyping, another requirement of practical distributed applications, and promotes a programming methodology more tightly coupled to the underlying execution mechanism than our approach.

An implementation of a session type system in Haskell is studied in [13] through an encoding of a simple session calculus. Again, this work is targeted

at a concurrent, but not distributed, environment. It may be difficult to realise the session compatibility check or type-safe delegation since the encoding does not directly type I/O channels.

References

1. WS-BPEL OASIS Web Services Business Process Execution Language. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-specification-draft.html.
2. Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured Communication-Centred Programming for Web Services. In *ESOP '07*, LNCS. Springer-Verlag, 2007.
3. Mario Coppo, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Asynchronous Session Types and Progress for Object-Oriented Languages. To appear in FMOOSE'07. http://www.di.unito.it/ dezani/papers/cdy.pdf, 2007.
4. Mariangiola Dezani-Ciancaglini, Sophia Drossopoulou, Elena Giachino, and Nobuko Yoshida. Bounded Session Types for Object-Oriented Languages. Submitted for publication. http://www.di.unito.it/ dezani/papers/ddgy.pdf, 2007.
5. Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopoulou. Session Types for Object-Oriented Languages. In *ECOOP '06*, volume 4067 of *LNCS*, pages 328–352. Springer-Verlag, 2006.
6. Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, Alex Ahern, and Sophia Drossopoulou. A Distributed Object Oriented Language with Session Types. In *TGC '05*, volume 3705 of *LNCS*, pages 299–318. Springer-Verlag, 2005.
7. Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen C. Hunt, James R. Larus, , and Steven Levi. Language Support for Fast and Reliable Message-based Communication in Singularity OS. In *EuroSys 2006*, ACM SIGOPS, pages 177–190. ACM Press, 2006.
8. Simon Gay and Malcolm Hole. Subtyping for Session Types in the Pi-Calculus. *Acta Informatica*, 42(2/3):191–225, 2005.
9. Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language Primitives and Type Discipline for Structured Communication-Based Programming. *LNCS*, 1381:122–??, 1998.
10. Raymond Hu. Implementation of a Distributed Mobile Java. Master's thesis, Imperial College London, 2006.
11. Raymond Hu, Nobuko Yoshida, and Kohei Honda. Online appendix of this paper. http://www.doc.ic.ac.uk/ rh105/sessiondj.html.
12. Raymond Hu, Nobuko Yoshida, and Kohei Honda. Type-safe Communication in Java with Session Types (18pp. draft). March 2007.
13. Matthias Neubauer and Peter Thiemann. An Implementation of Session Types. In *PADL*, volume 3057 of *LNCS*, pages 56–70. Springer-Verlag, 2004.
14. Polyglot home page. http://www.cs.cornell.edu/Projects/polyglot/.
15. Antonio Vallecillo, Vasco T. Vasconcelos, and António Ravara. Typing the Behavior of Objects and Components using Session Types. In *FOCLASA '02*, volume 68(3) of *ENTCS*, pages 439–456. Elsevier, 2002.
16. Vasco T. Vasconcelos, António Ravara, and Simon Gay. Session Types for Functional Multithreading. In *CONCUR '04*, volume 3170 of *LNCS*, pages 497–511. Springer-Verlag, 2004.
17. Web Services Choreography Working Group. Web Services Choreography Description Language. http://www.w3.org/2002/ws/chor/.

A Survey of Scratch-Pad Memory Management Techniques for low-power and -energy

Maha Idrissi Aouad and Olivier Zendra

INRIA-Lorraine / LORIA, Building C,
615 Rue du Jardin Botanique, BP 101,
54602 Villers-Lès-Nancy Cedex,
FRANCE
Maha.IdrissiAouad@loria.fr, Olivier.Zendra@inria.fr

Abstract. Scratch-Pad Memories (SPMs) are considered to be effective in helping reduce memory energy consumption. However, the variety of SPM management techniques complicates the choice of the right one to implement. In this paper, we first give a synthesis on existing SPM management techniques for low-power and -energy outlining their comparative advantages, drawbacks and trade-offs. Then, we propose a new general classification which encompasses most existing research works. This classification has the advantage of clearly exhibiting lesser explored techniques, hence providing hints for future research.

1 Introduction

Reducing energy consumption of embedded systems is a topical and very crucial subject. Many systems are energy-constrained and, despite batteries progress, these systems still have a limited autonomy. This mainly concerns numerous daily life objects such as cell phones, laptops, PDAs, MP3 players, etc.

Different options to save energy, hence increase autonomy, exist but we can't detail them here due to the lack of space. The interested reader can refer to [Graybill and Melhem, 2002; Zendra, 2006] for a more comprehensive view. These various approaches can be classified in two main categories: hardware optimizations and software optimizations. Hardware techniques fall beyond the scope of this paper, but a large amount of literature about them is available (see first parts of [Graybill and Melhem, 2002]). Some works interestingly couple hardware techniques with software ones, such as [Poletti *et al.*, 2004] (that relies on Direct Memory Access, or DMA, to reduce the copy cost between SPM and DRAM), or [Benini *et al.*, 2000] (using Application-Specific Memory, ASM).

In this paper, however, we will focus on software, compiler-assisted techniques. Cache memories, although they help a lot with program speed, do not always fit in embedded systems: they increase the system size and its energy cost (cache area plus managing logic). In contrast, SPMs have interesting features. Like caches, they consist of small, fast SRAM, but the main difference is that SPMs are directly and explicitly managed at the software level, either by the developer or by the compiler, whereas caches require extra dedicated

circuits. Compared to cache, SPM thus has several advantages [Zendra, 2006]. SPM requires up to 40% less energy and 34% less area than cache [Banakar *et al.*, 2002]. Additionally, SPM cost is lower and its software management makes it more predictable, which is an important feature for real-time systems.
According to [Adiletta *et al.*, 2002; Brash, 2002], a large variety of chips with SPM is available today in the market. Moreover, trends [LCTES, 2003] indicate that the dominance of SPMs is likely to continue in the future. Consequently, many authors have tried to profit from the advantages of SPMs and various related research directions have been investigated. These techniques and algorithms, synthesized in [Benini and Micheli, 1999], try to optimally allocate application code and/or data to SPM in order to reduce the energy consumption of embedded systems. The interested reader can look at [Benini and Micheli, 2000] for a comprehensive list of references. Although some of the research works we present in this paper have not been targeted specifically to Object-Oriented Languages (OOL), we think their underlying principles still apply to OOL.

The rest of the paper is organized as follows. Section 2 describes some software optimization techniques for SPM. Section 3 presents a discussion with a new classification. Finally, section 4 concludes.

2 Software Optimization Techniques for SPM

Numerous research works focus on SPM optimized management techniques. In this section, we present a survey and a classification of these techniques. In order to manage the SPM space, some approaches try to answer the question of which data to allocate to which memory type. Others are based on optimizing data locality. All these methods rely on profiling information to place the most frequently used or most cache conflicting data in fast memory, and other data in slower memory. The main trade-offs of these approaches revolve around the objects considered (arrays, loops, global, heap or stack variables...).

2.1 Techniques Focusing on Data Placement in Memory According to Memory Type

The first category of SPM management techniques comprises those that can be characterized as focusing on data placement in memory according to memory type. These approaches try to answer the question of which program variables should be allocated to which memory or memory bank. In these techniques, because of the reduced size of SRAM, the lesser-used variables are first allocated to slower memory banks (DRAM), while the most frequently used variables are kept in fast memory (SRAM) as much as possible. These methods use profile data to gather access frequency information in order to place frequently used data in fast memory, and other data in slower memory. To do so, most authors model the problem as a 0/1 integer linear programming (ILP) problem and then use an available IP solver to solve it.

[Avissar *et al.*, 2002] considers global and stack variables and chooses between SPM and cache, while [Steinke *et al.*, 2002b; Wehmeyer *et al.*, 2004] consider global variables, functions and basic blocks and choose between SPM banks. Instead of using one single large SPM, the simulated results obtained by [Wehmeyer *et al.*, 2004] have shown that by using a partitioned SPM improvements of up to 22% in the energy consumption of the memory subsystem can be obtained.

These techniques are all based on the frequency of data accesses. In contrast, [Panda *et al.*, 1997] considers arrays and scalar variables and focuses on data that is the most cache-conflict prone. The authors rely on profile data to place the most conflicting data in SRAM. All of these approaches require knowledge of the SPM size at compile time but [Nguyen *et al.*, 2005] presents a compiler method whose resulting executable is portable across SPMs of any size. It consists on discovering SPM size first, either by making an OS or low-level system call if available, or by probing addresses in memory using a binary search pattern and observing the latency to find the range of addresses belonging to SPM. Then, the memory allocation algorithm is the same as in [Avissar *et al.*, 2002].

2.2 Techniques Focusing on the Locality of Memory Access

The techniques presented in this section can be considered as a refinement of those of section 2.1. Indeed, in addition to finding the best data placement with respect to memory types, it is interesting to investigate the locality of memory accesses in order to further optimize energy usage.

Spatial Locality Some methods are based on optimizing spatial locality, that is on ensuring that successive SPM accesses use the same SPM bank as much as possible. Indeed, increasing the spatial locality for a set of SPM banks clearly increases the duration of idleness for other SPM banks, which in turn helps to amortize the cost of placing a bank into low-power mode and then later transitioning it back to normal operation mode.

[Athavale *et al.*, 2001] explores the energy consumption of array allocation mechanisms in Java. Using a set of array-dominated benchmarks and a partitioned memory architecture with multiple low-power operating modes, the authors study two data optimization techniques: memory layout modification and array interleaving. This memory layout modification consists in changing the storage order of data inside an array in order to improve its spatial locality. In addition, array interleaving groups together in the same memory module elements that belong to different multi-dimensional arrays, thus increasing the inter-access interval (time between two references to the same module) for the unused modules. This provides an opportunity to operate the unused memory modules in a lower power mode for a longer time. Their experimental results show that using layout transformation and array interleaving optimization provides an average of 9.68% and 14.96% energy savings, respectively.

The compiler-based strategy proposed in [Kandemir *et al.*, 2005] is also effective in reducing leakage energy of on-chip SPMs and has the advantage of

dealing with arrays and loops in general without a restriction to a specific language. In addition to having some similarities with [Athavale *et al.*, 2001], the technique presented in [Kandemir *et al.*, 2005] also seems general enough to be applicable to any object-oriented language. The idea in [Kandemir *et al.*, 2005] is to divide SPM into banks and use compiler-guided data layout optimization and data migration to maximize SPM bank idleness, thereby increasing the chances of placing banks into low-power state. This work focuses on reducing leakage consumption of on-chip SPMs without hurting performance.

Temporal Locality Other methods are based on temporal locality, that is the fact that recently accessed SPM banks are likely to be accessed again in a near future. [Verma *et al.*, 2004] presents a profile based approach which, on the basis of live ranges of both variables and code segments, replenishes the content of the SPM. These elements are optimally chosen in order to minimize the energy overheads due to spilling memory object to and from the SPM. This technique also computes addresses within the SPM address range where variables and code segments have to be copied. These addresses are computed such that a large number of variables and code segments fit in the same SPM space.

2.3 Comprehensive Techniques Dealing with all Memory Objects

The techniques we mentioned in the previous sections have the drawback of not taking into account all kinds of objects. In the current section, we will focus on Udayakumaran and Barua's works, which conversely deal with *all* memory objects: arrays, loops, global, heap and stack variables. Udayakumaran and Barua's approach is based on works by Kandemir *et al.* and tries to improve them.

Both methods move data back and forth between DRAM and SPM under compiler control, but two improvements are brought by [Udayakumaran and Barua, 2003] over [Kandemir *et al.*, 2001].

First of all, [Kandemir *et al.*, 2001] considers global and stack array variables only and has the three additional following restrictions. One, the programs should primarily access arrays of the innermost loops. Two, the loops must be well-structured and must not have any other control flow such as *if-else*, *break* and *continue* statements. Three, the codes containing these constructs must be well written, that is to say without any of the hand-made optimizations often found in many such codes, because these optimizations consider not only the loop nest in question, but also a much larger context. Combining these three restrictions, Kandemir *et al.*'s method applies to well-structured scientific and multimedia codes. However, as underlined by Udayakumaran and Barua, most programs in embedded systems do not fit within these strict restrictions. [Udayakumaran and Barua, 2003] has improved the generality of the method and applies it to all global and stack variables, and all access patterns to those variables. The method thus becomes more general and is able to exploit locality for all codes, including those with irregular accesses patterns, variables other than arrays, code with pointers and irregular control flow.

The second improvement brought by [Udayakumaran and Barua, 2003] is that [Kandemir *et al.*, 2001] considers each loop nest independently, whereas Udayakumaran and Barua's method is a whole-program analysis across all control structures. This has several consequences. One is that the method presented by Kandemir *et al.* is locally optimized for each loop, while the method of Udayakumaran and Barua is globally optimized for the entire program. Another consequence is that with the method of Kandemir *et al.* the entire SPM is available for each loop nest. In contrast, the approach of Udayakumaran and Barua might choose to do this, but is not constrained to do so. It may choose to use part of the SPM for data that is shared between successive control constructs thus saving on transfer time and energy to DRAM.

Note however that for arrays, it is possible to bring in parts of an array instead of considering the whole array with [Kandemir *et al.*, 2001], whereas this is impossible with [Udayakumaran and Barua, 2003].

Udayakumaran and Barua have extended their work in other papers. For example, the approach in [Udayakumaran *et al.*, 2006] also handles code objects and provides some measurements for energy consumption. Their results from simulation show that their scheme reduces runtime by up to 39.8% and energy by up to 31.3% on average for their benchmarks, depending on the SRAM size used, when compared to [Avissar *et al.*, 2002].

The much cited [Dominguez *et al.*, 2005] is also a very interesting piece of work because it is, to our best knowledge, the only work that considers heap data and has an SPM management policy at runtime that allows fixed moves (as shown in Table 1). Their simulation results show that this method reduces the average runtime by 34.6% and the average power consumption by 39.9% for the same size of SPM fixed at 5% of total data size, when compared to placing all heap variables in DRAM and only global and stack data in SPM.

Finally, [Udayakumaran and Barua, 2006] extends the work done by investigating SPM allocation for arrays. This last paper modifies the algorithm already proposed by adding a code to identify partial variables such as a row, a column or even a collection of elements belonging to an array variable that is accessed by a loop nest, using an affine analysis pass. The aim of this pass is to enable allocation of parts of an array, for instance when the whole array does not fit into the SPM. However, this paper presents results according to runtime only, energy consumption is not considered. We think this should be addressed.

3 Discussion

In this paragraph, we try to bring a fresh look at the SPM management techniques. Thus, we have done a general study that allows us to propose a new classification presented in Table 1 which considers two criteria.

The first criterion deals with the way information is collected and the second criterion refers to the SPM management policy at runtime. For the *Information Collection* criterion, *Compilation* means that the code is analyzed at compile

Table 1. A Classification of SPM Management Phases

References	**Information Collection**	**SPM Management Policy at Runtime**
[Kandemir *et al.*, 2001]	Compilation + Profiles	Static
[Udayakumaran and Barua, 2003]	Compilation	Free Moves
[Udayakumaran *et al.*, 2006]	Compilation	Free Moves
[Udayakumaran and Barua, 2006]	Compilation	Free Moves
[Nguyen *et al.*, 2005]	Compilation + Profiles	Static
[Egger *et al.*, 2006]	Compilation + Profiles	Static
[Absar and Catthoor, 2005]	Compilation	Static
[Poletti *et al.*, 2004]	Compilation	Static
[Steinke *et al.*, 2002a]	Compilation	Static
[Verma *et al.*, 2003]	Compilation	Static
[Verma *et al.*, 2004]	Compilation	Free Moves
[Dominguez *et al.*, 2005]	Compilation	Fixed Moves
[Avissar *et al.*, 2002]	Compilation + Profiles	Static
[Steinke *et al.*, 2002b]	Compilation	Static
[Wehmeyer *et al.*, 2004]	Compilation + Profiles	Static
[Panda *et al.*, 1997]	Compilation + Profiles	Static
[Athavale *et al.*, 2001]	Compilation	Static
[Kandemir *et al.*, 2005]	Compilation + Profiles	Static
[Hiser and Davidson, 2004]	Compilation + Profiles	Static

time, whereas *Profiles* indicates the use of runtime profiling. The *SPM Management Policy at Runtime* can be *Static* which means that data could be overwritten but not moved to another place. With *Moves* some existing variables in SPM could be evicted to make space for incoming ones. In this way, data is never lost. On the one hand, *Fixed Moves* always place data at the same offset in the SPM or DRAM. On the other hand, with *Free Moves* the SPM allocation can be dynamically adapted at runtime by placing most frequently used data at any free location in the SPM or DRAM.

Several methods are *Static* and are based on *Compilation* information only [Absar and Catthoor, 2005; Steinke *et al.*, 2002b; Athavale *et al.*, 2001]. However, with execution *Profiles* an accurate view of the data access patterns can be obtained, since the profiles contain information about which variables are accessed during which program phase. In this context, [Egger *et al.*, 2006; Avissar *et al.*, 2002; Kandemir *et al.*, 2001] provided experimental results which generally improve the ones they had obtained with the *Static* approach. Furthermore, SPM management policies based on *Moves* [Dominguez *et al.*, 2005; Verma *et al.*, 2004; Udayakumaran and Barua, 2003] are more effective than *Static* ones, because they optimize the use of the SPM space and allow to change the content of the SPM at runtime. In other words, just like in a cache, data is moved back and forth between DRAM and SPM, but under compiler control. The energy overhead of performing a move is compensated by a better placement.

As we can see from Table 1, there are different combinations of *Information Collection* and *SPM Management Policy at Runtime*. However, our synthesis shows that there is no method merging compilation and profiles information with an SPM management policy based on moves. This thus seems a potentially interesting area for new research. Furthermore, most of the presented works are not targeted to OOL but apply to them nonetheless. We consider it would be interesting to also study SPM management techniques by taking into account some more specific features of OOL such as object memory layout.

4 Conclusion and perspectives

In this paper, we have given in section 2 a global structure of the use of optimized SPM management techniques. The classification we have proposed enables to make a synthesis of most SPM management techniques, which makes it possible to have a more global and precise view of these techniques aiming at reducing energy and/or power consumption in embedded systems. Indeed, our classification in section 3 exhibits very clearly the fact that combining *Compilation*, *Profiles* information and an SPM management policy based on *Moves* at runtime has not been explored yet. In our point of view, this could be more effective in reducing energy consumption as the results obtained separately are interesting. We plan to explore this in our future works, as well as to study SPM with respect to some specific features of OOL.

References

[Absar and Catthoor, 2005] M. J. Absar and F. Catthoor. Compiler-based approach for exploiting scratch-pad in presence of irregular array access. In *DATE*, 2005.

[Adiletta *et al.*, 2002] M. Adiletta, M. Rosenbluth, D. Bernstein, G. Wolrich, and H. Wilkinson. The Next Generation of Intel IXP Network Processors. *Intel Technology Journal*, 6(3), Aug. 2002.

[Athavale *et al.*, 2001] R. Athavale, N. Vijaykrishnan, M. T. Kandemir, and M. J. Irwin. Influence of array allocation mechanisms on memory system energy. In *IPDPS*, page 3, 2001.

[Avissar *et al.*, 2002] O. Avissar, R. Barua, and D. Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. *Transaction. on Embedded Computing Systems.*, 1(1):6–26, 2002.

[Banakar *et al.*, 2002] R. Banakar, S. Steinke, B. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *CODES*, pages 73–78, New York, NY, USA, 2002. ACM Press.

[Benini and Micheli, 1999] L. Benini and G. De Micheli. System-level power optimization: Techniques and tools. In *ISLPED-99:ACM/IEEE*, pages 288–293, 1999.

[Benini and Micheli, 2000] L. Benini and G. De Micheli. System-level power optimization: techniques and tools. *IEEE Design and Test*, 17(2):74–85, 2000.

[Benini *et al.*, 2000] L. Benini, A. Macii, E. Macii, and M. Poncino. Increasing Energy Efficiency of Embedded Systems by Application Specific Memory Hierarchy Generation. *IEEE Design and Test*, 17(2):74–85, 2000.

[Brash, 2002] D. Brash. The ARM architecture Version 6 (ARMv6). In *ARM Ltd.*, January 2002. White Paper.

[Dominguez *et al.*, 2005] A. Dominguez, S. Udayakumaran, and R. Barua. Heap data allocation to scratch-pad memory in embedded systems. *ACM Transactions on Design Automation of Electronic Systems*, 5(2):115–192, 2005.

[Egger *et al.*, 2006] B. Egger, J. Lee, and H. Shin. Scratchpad Memory Management for Portable Systems with a Memory Management Unit. In *EMSOFT*, 2006.

[Graybill and Melhem, 2002] R. Graybill and R. Melhem. *Power aware computing*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[Hiser and Davidson, 2004] J. D. Hiser and J. W. Davidson. EMBARC: An Efficient Memory Bank Assignment Algorithm for Retargetable Compilers. In *ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 182–191. ACM Press, 2004.

[Kandemir *et al.*, 2001] M. T. Kandemir, I. Ramanujam, M. J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. Dynamic Management of Scratch-pad Memory Space. In *Pmc. DAC*, 2001.

[Kandemir *et al.*, 2005] M. T. Kandemir, M. J. Irwin, G. Chen, and I. Kolcu. Compiler-guided leakage optimization for banked scratch-pad memories. *IEEE Trans. VLSI Syst.*, 13(10):1136–1146, 2005.

[LCTES, 2003] LCTES. Compilation Challenges for Network Processors. In *Compilers and Tools for Embedded Systems*. Industrial Panel, ACM Conference on Languages, June 2003.

[Nguyen *et al.*, 2005] N. Nguyen, A. Dominguez, and R. Barua. Memory allocation for embedded systems with a compile-time-unknown scratch-pad size. In *CASES*, 2005.

[Panda *et al.*, 1997] P. R. Panda, N. Dutt, and A. Nicolau. Efficient utilization of scratch-pad memory in embedded processor applications. In *DATE*, 1997.

[Poletti *et al.*, 2004] F. Poletti, P. Marchal, D. Atienza, L. Benini, F. Catthoor, and J. M. Mendias. An integrated hardware/software approach for run-time scratchpad management. In *DAC*, pages 238–243, 2004.

[Steinke *et al.*, 2002a] S. Steinke, N. Grunwald, L. Wehmeyer, R. Banakar, M. Balakrishnan, and P. Marwedel. Reducing Energy Consumption by Dynamic Copying of Instructions onto Onchip Memory. In *ISSS*, 2002.

[Steinke *et al.*, 2002b] S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *DATE*, page 409. IEEE Computer Society, 2002.

[Udayakumaran and Barua, 2003] S. Udayakumaran and R. Barua. Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. In *CASES*, pages 276–286. ACM Press, 2003.

[Udayakumaran and Barua, 2006] S. Udayakumaran and R. Barua. An integrated scratch-pad allocator for affine and non-affine code. In *DATE*, pages 925–930, 2006.

[Udayakumaran *et al.*, 2006] S. Udayakumaran, A. Dominguez, and R. Barua. Dynamic allocation for scratch-pad memory using compile-time decisions. *Embedded Comput. Syst*, 5(2):472–511, 2006.

[Verma *et al.*, 2003] M. Verma, S. Steinke, and P. Marwedel. Data Partitioning for Maximal Scratchpad Usage. In *ASPDAC*, 2003.

[Verma *et al.*, 2004] M. Verma, L. Wehmeyer, and P. Marwedel. Dynamic Overlay of Scratchpad Memory for Energy Minimization. In *CODES+ISSS*, 2004.

[Wehmeyer *et al.*, 2004] L. Wehmeyer, U. Helmig, and P. Marwedel. Compiler-optimized usage of partitioned memories. In *WMPI*, 2004.

[Zendra, 2006] O. Zendra. Memory and compiler optimizations for low-power and -energy. In *ICOOOLPS*, 2006.

Ensuring that User Defined Code does not See Uninitialized Fields

Anders Bach Nielsen

Dept. of Computer Science, University of Aarhus, Denmark

Abstract. Initialization of objects is commonly handled by user code, often in special routines known as constructors. This applies even in a virtual machine with multiple concurrent execution engines that all share the same heap. But for a language where run-time values play a role in the type system, no user defined code can be allowed to use a field before it is initialized. This paper presents an approach which ensures that user code will not see uninitialized fields. It uses a dual-mode execution model to maintain a reasonable level of performance.

1 Introduction

There are many approaches to object initialization in modern languages. Some do it like Java [9], where default values are assigned to fields in the newly created object and then run a special method, a constructor, to set the fields to user defined values. Not that many do initialization of objects like gbeta [5] does. In gbeta there is no constructor and each field in an object has a special block of instructions to initialize that particular field. The reason why the Java way does not suffice in gbeta is because fields that are significant for typing, must be initialized before user code, using those types, is allowed to run. The fact that gbeta types depend on run-time values is at the core of the type system feature known as family polymorphism [6,11,12,8].

An object in gbeta is a list of part-objects and each part-object can hold a number of fields. These fields can either be mutable references to other objects or immutable references to objects or patterns. When creating a new object, we do not want anyone to know it exists before all fields are completely initialized. If this was running in a virtual machine with several concurrent execution engines, e.g. interpreters, it would break the type system if an uninitialized object somehow leaked to the heap and some other running execution engine used one of the uninitialized fields, in this leaked object.

The issue is now to create a virtual machine, which has multiple concurrent execution engines and a shared heap, and will make sure no references to a new object is leaked to the heap before it is completely initialized. In a *normal heap* no field refers to a partially initialized object, but in an *initializing heap* there may be some fields referring to a partially initialized object.

The approache presented in this paper will use a dual-mode execution model. We are not the first to present at multi mode execution model [2] and it is not to be mistaken for Just-in-Time compilation done in the Java HotSpot [3,10] virtual machine. In Java HotSpot they have two execution modes 1) interpreted and 2) running native code. There is no semantic difference between these two modes. The execution engine

running native code just runs faster. We want to have two execution modes depending on the state of the heap.

2 Class and Object Layout

The class and object layout described in this section is a simplified version of the gbeta pattern and object layout. We have also removed the notion of methods, because they do not contribute anything.

The layout of a class, as seen in Fig. 1, is a list of field initialization code blocks (*icode blocks*). Each icode block is a list of instructions needed to initialize the corresponding fields in the new object. The object has a pointer to its class and a list of fields. Each field can be either a mutable reference or a immutable object or class.

In Fig. 1 the icode blocks of a class are denoted *ICB* one to five. The objects class pointer is marked as *CLP* and the fields are denoted as *SL* one to five.

The instructions in an icode block can be as simple as writing a zero or null value to a field. It can also be as complex as creating new object and calling methods on this newly created object. This is why field initialization is important.

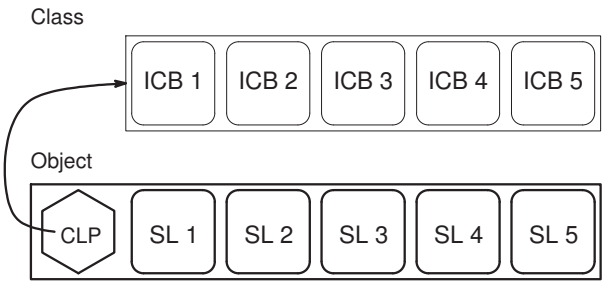

Fig. 1. Simplified Class and Object Layout

3 Normal versus Initializing

When creating a new object from a class, we evaluate the instruction in each icode block. There is no special instruction block that acts as a constructor, the evaluation of all icode blocks gives the initialized object. The instructions in a given icode block are targeted at a specific field in the new object. This gives us the ability to rearrange or concatenate icode blocks to larger instruction blocks.

Initialization of a field can involve running any number of instructions, as mentioned in section 2. The instructions can be divided into two groups. *Non-Leaking* instructions are those that read values from other fields or those that write only to their targeted field. It is especially those that do not leak a (direct or indirect) pointer to the object being initialized. *General* instructions are those which can write any value anywhere. This way they can leak a pointer to the heap of the object being initialized.

There are basically two kinds of icode blocks. The icode blocks which only contain Non-Leaking instructions are classified as *normal* and the icode blocks which contain one or more General instructions would require more careful initialization, so those are classified as *initializing*.

The structure of a class is a list of icode blocks, as seen in Fig. 2 a). An icode block can use values of other fields, initialized by other icode blocks in the same object. This gives dependencies among icode blocks and some fields need to be initialized before others. The initializing icode blocks may depend on fields initialized by either normal or initializing icode blocks, but normal icode blocks may only depend on fields initialized by other normal icode blocks. If an normal icode block *(A)* depends on a field initialized by an initializing icode block *(B)*, then icode block *(A)* should be classified as an initializing icode block as well. If we have a situation where two icode blocks depend on each other, a cyclic dependency, we will just abort with a proper error. More sophisticated approaches can be used to handle many kinds of cyclic dependencies, but in this context we focus on handling the statically non-cyclic cases efficiently, and the more complex cases are postponed as future work. If we have no cyclic dependencies, we can rearrange the icode blocks so all normal icode blocks are grouped first and the initializing icode blocks last. This rearrangement will take into account the icode block dependencies. After such a rearrangement a class could look like Fig. 2 b). To improve initialization performance we can concatenate the normal icode blocks to one large instruction block and do the same for the initialization icode blocks. We now have two large instruction blocks, as seen in Fig. 2 c). To be able to find the initialization block, we tag the block with a start marker instruction and an end marker instruction, seen as the black blocks in Fig. 2 c).

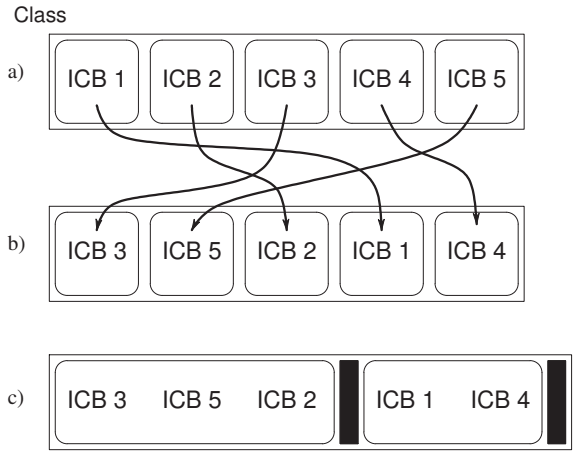

Fig. 2. Rearranging and concatenating icode blocks in a class

4 The `ExecutionBase`

The virtual machine consists of one global space and one heap that is shared among several running execution engines. Each execution engine has its own activation record stack, where the current activation record has an evaluation stack and a temporary array. Each of these execution engines is running in its own thread. The heart of each execution engine (`ExecutionBase`) is sketched in Fig. 3 and a more detailed description of what should happen in this loop will be explained in section 5.

The code presented in Fig. 3 does not look like an interpreter and that is because it is not. The actual interpretation is hidden in the method call `run()` on line 104.

This action of switching execution engine, in this case interpreter, could look like an expensive action and it probably is. But if we look at how often we have icode blocks that need the execution engine to run in initializing mode, then this is a rare situation. Most of the time the execution engine will run in the initial normal mode.

5 Running in Two Different Modes

In this section we will give a more detailed description of a scenario where the virtual machine is running multiple execution engines in normal mode and one of the execution engines encounters a initializing block of instructions.

1. Let us assume that the virtual machine has started, no initializing blocks have been encountered and we have started one or more execution engine threads. In this case all execution engines are running in normal mode and all reachable objects in the heap are fully initialized. One of the execution engines is now instructed to create a new object from a class on the evaluation stack.
2. The execution engine starts creating the object by evaluating the field initialisation instructions in a sequential order.
3. The execution engine reads an instruction telling it that the next block is to be run in initializing mode and it should thus switch to run a initializing execution engine. But not only does this execution thread have to run in initializing mode, all other threads have to run in initializing mode as well. This is because all threads may potentially try to evaluate an uninitialized field once a partially initialized object has been leaked. This triggers a series of events:
   (a) The execution engine will now create a restore point and return to the `ExecutionBase`, see Fig. 3 line 104, with an argument telling it to switch to a execution engine in initializing mode to resume execution.
   (b) The `ExecutionBase` gets a message that it should start an initializing execution engine. It then reads a global counter that tells how many initializing blocks are executing at the moment.
      i. If this global counter is zero, then the other execution engines have to switch to run in initializing mode as well. The `ExecutionBase` then sends a message to the other running execution engines, telling them to switch. It then increments the global counter by one and waits for the other interpreters to signal back that they have switched to initializing mode.

```
1   // this is a shared resource and has to be guarded by some
2   // concurrency control. This is left out to ease the readability
3   // of this example.
4   static int noOfInitializing = 0;
5
6   enum TerminationState {terminated,
7                          willgoinitializing,
8                          willgonormal};

100 Interpreter* interp = new NormalInterpreter();
101
102 int ret;
103 while (true) {
104    ret = interp->run();
105    if (ret == terminated) {
106       // interpreter has reached normal end of execution
107       break;
108    } else if (ret == willgoinitializing &&
109             noOfInitializing == 0) {
110       // send message to other threads to run
111       // in initializing mode
112       noOfInitializing++;
113       // wait for other thread to signal they have switched
114       interp = new InitializingInterpreter();
115    } else if (ret == willgoinitializing &&
116             noOfInitializing > 0) {
117       noOfInitializing++;
118    } else if (ret == willgonormal) {
119       noOfInitializing--;
120       if (noOfInitializing == 0) {
121          // send message to other threads to run
122          // in normal mode
123          interp = new NormalInterpreter();
124       }
125    }
126 }
```

Fig. 3. The `ExecutionBase`

ii. If the global counter is larger than zero, then all execution engines are running in initializing mode and it will just increment the global counter by one and continue.

(c) If it had to switch, it will now create a new initializing execution engine and continue running the thread where the other execution engine left off. The system is now only running initializing execution engines and every time we want to look up an immutable field we always need to check if its not null. All though some immutable fields can be initialized to null, but that would cause a run time error, if its not handled by the programmer.

4. The execution engine reaches the end of the initializing block and read an instruction telling it that this initializing block has ended. Again this will trigger a series of events.
   (a) The execution engine will create a restore point and return to the `ExecutionBase` with an argument telling it to create a execution engine running in normal mode to resume execution.
   (b) The `ExecutionBase` gets a message that it should switch to run as a normal execution engine. It then reads the global counter to see if there are any other threads running in a initializing block.
      i. First of all it will decrement the global counter by one.
      ii. If the global counter is zero, then this was the last thread in an initializing block. It will now send a notify message to the other threads telling them that they can return to normal mode. This thread will switch to a normal execution engine immediately and continue execution.
      iii. If the global counter is larger than zero, then this means that another execution engine has entered an initializing block and this thread cannot switch to normal mode just yet. It will continue execution in initializing mode until signalled.
   (c) If it had to make a switch, it will create a new normal interpreter and continue execution from where the other left off. So either all threads are running normal execution engines again or one had entered a initializing block, so they are still in initializing mode.

6 Implementation Status

We are currently implementing a new virtual machine for gbeta, in its current state of development. The language has evolvedin several ways [4,7] since it was described in the PhD thesis [5] by Erik Ernst.

The method described in this paper is ongoing work and some parts of it are already implemented in the currently running version of our virtual machine.

7 Related Work

An immutable field in Java is declared `final` and given its value in the constructor. This is not to be confused with a field declared `final` and given its value in the declaration. Such a field is compiled into constants and special rules apply [9]. In the Java

Language Specification [9] in section 17.5 the authors state: *That an object is considered to be completely initialized when its constructor finishes.* They also state: that the usage model for final fields is simple. *Set the final fields for an object in that object's constructor. Do not write a reference to the object being constructed in a place where another thread can see it before the object's constructor is finished.*

A great deal of work has been put into strengthening the Java Memory Model [1] and how synchronisation of threads should be done. But the problem of comparing Java immutable fields to gbeta fields is that in Java the system would still run. In Java you would observe odd behavior when reading `final` files that have not been initialized. In gbeta you use run time values of fields as part of the type system. In this way a given class resides within a object stored in a field. If this field was uninitialized the system would fail to find the class and the system would break down.

This is why we have to make sure that user defined code does not see these uninitialized fields of objects.

8 Conclusion

In this paper we have proposed a way of handling initialization of objects, where we respect the type system, in a way that no object is visible to the system until it is completely initialized. This way user defined code will only see completely initialized objects and thereby all promises made by the type system about fields will hold. The proposed method, of having a dual-mode execution engine, depends on an analysis where we can identify regions in the field initializing code that could potentially leak a pointer to the newly created object to the heap. This method is being tried out in practise in the ongoing implementation effort.

Acknowledgements

I would like to thank Erik Ernst for pointing out some important details in my work and the anonymous reviewers for valuable feedback and suggestions.

References

1. Sarita Adve, Jeremy Manson, and Bill Pugh. Jsr-133: Java memory model and thread specification. Technical report, Sun Microsystems, 2004. http://jcp.org/en/jsr/detail?id=133.
2. C. Chambers, D. Ungar, and E. Lee. An efficient implementation of SELF a dynamically-typed object-oriented language based on prototypes. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 24, pages 49–70, New York, NY, 1989. ACM Press.
3. T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson, and M. Wolczko. Compiling java just in time. *Micro, IEEE*, 17(3):36–43, May/Jun 1997. ISSN: 0272-1732.
4. E. Ernst. Higher-order hierarchies. In *Proceedings European Conference on Object-Oriented Programming (ECOOP 2003)*, LNCS, pages 303–329, Heidelberg, July 2003. Springer Verlag.

5. Erik Ernst. *gbeta – a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, Department of Computer Science, University of Aarhus, Århus, Denmark, 1999.
6. Erik Ernst. Family polymorphism. In J. L. Knudsen, editor, *ECOOP 2001*, number 2072 in LNCS, pages 303–326. Springer Verlag, 2001.
7. Erik Ernst. Reconciling virtual classes with genericity. In David E. Lightfoot and Clemens A. Szyperski, editors, *JMLC*, volume 4228 of *Lecture Notes in Computer Science*, pages 57–72. Springer, 2006.
8. Erik Ernst, Klaus Ostermann, and William R. Cook. A virtual class calculus. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 270–282, New York, NY, USA, 2006. ACM Press.
9. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley, May 2005.
10. Andreas Krall. Efficient JavaVM just-in-time compilation. In Jean-Luc Gaudiot, editor, *International Conference on Parallel Architectures and Compilation Techniques*, pages 205–212, Paris, 1998. North-Holland.
11. Mira Mezini and Klaus Ostermann. Conquering aspects with caesar. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 90–99, New York, NY, USA, 2003. ACM Press.
12. Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In Luca Cardelli, editor, *ECOOP*, volume 2743 of *Lecture Notes in Computer Science*, pages 201–224. Springer, 2003.