A Simplified Java Bytecode Compilation System for Resource-Constrained Embedded Processors*

Carmen Badea, Alexandru Nicolau, Alexander V. Veidenbaum
Center for Embedded Computer Systems
University of California, Irvine
{carmen, nicolau, alexv}@ics.uci.edu

Abstract

Embedded platforms are resource-constrained systems in which performance and memory requirements of executed code are of critical importance. However, standard techniques such as full just-in-time(JIT) compilation and/or adaptive optimization (AO) may not be appropriate for this type of systems due to memory and compilation overheads.

The research presented in this paper proposes a technique that combines some of the main benefits of JIT compilation, superoperators(SOs) and profile-guided optimization, in order to deliver a lightweight Java bytecode compilation system, targeted for resource-constrained environments, that achieves runtime performance similar to that of state-of-the-art JIT/AO systems, while having a minimal impact on runtime memory consumption. The key ideas are to use profiler-selected, extended bytecode basic blocks as superoperators (new bytecode instructions) and to perform few, but very targeted, JIT/AO-like optimizations at compile time ***only*** *on the superoperators' bytecode, as directed by compilation "hints" encoded as annotations. As such, our system achieves competitive performance to a JIT/AO system, but with a much lower impact on runtime memory consumption. Moreover, it is shown that our proposed system can further improve program performance by* ***selectively*** *inlining method calls embedded in the chosen superoperators, as directed by runtime profiling data and with minimal impact on classfile size.*

For experimental evaluation, we developed three Virtual Machines(VMs) that employ the ideas presented above. The customized VMs are first compared (w.r.t. runtime performance) to a simple, fast-to-develop VM (baseline) and then to a VM that employs JIT/AO. Our best-performing system attains speedups ranging from a factor of 1.52 to a factor of 3.07, w.r.t. to the baseline VM. When compared to a state-of-the-art JIT/AO VM, our proposed system performs better for three of the benchmarks and worse by less than a factor of 2 for three others. But our SO-extended VM outperforms the JIT/AO system by a factor of 16, on average, w.r.t. runtime memory consumption.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*compilers, optimization*

*This work was supported in part by the National Science Foundation under grant NSF CCF-0311738

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CASES'07, September 30–October 3, 2007, Salzburg, Austria.
Copyright 2007 ACM 978-1-59593-826-8/07/0009 ...$5.00.

General Terms

Performance, Algorithms

Keywords

Superoperators, Java Virtual Machine, Profile-Guided Optimization, Embedded Systems, Adaptive Optimization

1. INTRODUCTION

Embedded systems and related devices have become ubiquitous in today's world and most of them run Java-based applications. Java Virtual Machines (JVMs) provide an easy way to distribute applications in an architecture independent format: the bytecode format ([1]). However, this comes at the price of accepting the underlying virtual stack-based architecture of the JVM which, as opposed to register-based architectures commonly found in real processors, adds inherent overhead during the execution of applications ([2]). At the same time, the performance of the executed code is of critical importance and is often the limiting factor in both the capabilities of the system and user perception.

In order to alleviate the inefficiencies of the stack-based architecture, many research projects have focused on developing high-end optimizations, such as advanced garbage collection schemes or adaptive optimizations that require constant monitoring. However, this type of optimizations may not be suitable for interpreters or compilers running on embedded platforms since these are resource-constrained environments and the use of high-end optimizations would considerably increase resource requirements.

An alternative solution to these optimizations is the use of interpreters with *superoperators* ([3], [4], [5]), where a commonly executed sequence of bytecode instructions is converted into a new bytecode instruction called **superoperator**. One benefit of this technique is the reduction in the number of bytecode fetches and decodes. Another advantage is that by considering a group of bytecode instructions as a new single bytecode instruction, more optimization possibilities arise during the assembly code generation for the new bytecode instruction.

However, in order for the superoperators to improve the execution of an application, they have to be frequently executed pieces of code. Also, a balance between the generation and use of superoperators against possible space, time or power constraints needs to be achieved. For instance, too many superoperators could increase the size of the JVM beyond the current memory constraints. Superoperators that are not carefully chosen (w.r.t. execution frequency) may in

turn have no effect or even decrease the performance of an application.

This paper proposes a *lightweight, profile-guided bytecode compilation system* that employs superoperators and targets resource-constrained environments. In order to make our system a viable alternative to JIT compilers and adaptive optimization systems for resource-constrained environments, w.r.t. runtime performance and memory consumption, we use profile-selected, extended basic blocks (see section 3.1) as superoperators that are optimized by applying basic block-level optimizations similar to standard JIT optimizations (register allocation, common subexpression elimination and copy propagation).

In order to select a most profitable set of superoperators, we developed a fast profiling method based on edge counter information that indicates which extended basic blocks are important in each application (see section 3.3). The information necessary to apply the profile-guided optimizations described above would be stored as annotations and processed by our system at compile time. These compilation "hints" would enable the lightweight, JIT-like, runtime compiler in our system to use a most profitable set of superoperators as new bytecode instructions and to generate optimized assembly code corresponding to these superoperators.

We want our SO-extended compilation system to deliver competitive performance w.r.t. state-of-the-art VMs. As such, we decided to perform *selective, profile-guided inlining* of method calls in order to further improve the speedups achieved by the SO-extended compilation system, while having a minimal impact on memory consumption.

In order to evaluate the benefits of superoperators, we have developed three customized VMs that incrementally employ optimized superoperators:

- PJ-extended VM, which dynamically detects (in software) and optimally executes **all** PicoJava folding patterns present in a Java application. (see section 4.2 for more details).
- SO-extended VM, which employs optimized, profile-selected, extended basic blocks as superoperators.
- SOPJ-extended VM, which combines the benefits of employing optimized, application-customized superoperators and executing all "general-purpose" PicoJava folding patterns.

First, we evaluated the speedups attained by our three customized VMs w.r.t. an interpreter-like, fast-to-develop VM configuration (denoted as baseline VM), on a set of six "embedded-type" Java applications. However, we want our SO-extended compilation system to deliver competitive performance w.r.t. state-of-the-art VMs. As such, we then compared the performance yielded by our best performing compilation system (the SOPJ-extended VM) to that of a VM that employs an optimizing compiler, paired with an adaptive optimization system (denoted as AOS VM in the rest of the paper).

Among the optimizations performed in the AOS configuration is adaptive inlining, which we believe to contribute considerably to the speedups achieved by the AOS VM. Thus, in order to improve the selected SOs and consequently boost the performance attained by our SOPJ-extended VM, we applied selective, profile-guided inlining of method calls embedded in the chosen superoperators for each application in the benchmark suite and evaluated the performance results (section 5.3).

Since the target platforms of our proposed compilation system are resource-constrained environments, we also evaluated the runtime memory consumption of our SO-extended compilation system vs. the amount of runtime memory needed by the AOS VM.

The rest of the paper is organized as follows: in Section 2 we discuss related work on superoperators and improving execution performance of Java applications. Section 3 presents our profile-guided, SO-extended approach and describes the matters involved in choosing a most profitable set of superoperators for an application. Section 4 presents in more detail the different VM configurations that we have developed (listed above). In Section 5 we discuss the experimental setup for our tests and the performance results we have obtained. Section 6 concludes and presents directions for future work.

2. RELATED WORK

The Java language implements the write-once-run-anywhere methodology. Once an application is developed, it can be distributed to remote sites where a special runtime environment takes care of tasks like code translation or optimization that are necessary for the proper execution of the application on the target platform. Under these circumstances code translation is performed either by an interpreter or by a JIT compiler.

Both approaches have advantages and shortcomings: an interpreter does a simple instruction-by-instruction program translation, it is relatively easy to implement and it does not require a lot of memory, while a JIT compiler is a much more complex translator that compiles a whole method on its first encounter prior to execution. Consequently interpreters have low execution performance when compared to JIT compilers and can run a Java program 10 to 20 times slower. Still, interpreters have a few considerable advantages over JIT compilers in the context of embedded systems. First, they require less memory space when compared to a JIT compiler, which makes them an attractive choice due to present memory constraints. Second, they are easier to port to new architectures as opposed to JIT compilers, feature that contributes positively to the reduction of the time-to-market metric that is so important in the development of embedded devices.
Still, interpreters run code much slower than JIT compilers, giving rise to the need of adding optimizations to Java interpreters. One possibility is to use superoperators to speed up the execution of applications.

This paper presents a profile-guided, SO-based, simplified bytecode compilation system. It is built upon an unoptimized VM with a fast-to-develop compiler that simply generates unoptimized native code, thus resembling an interpreter.

Superoperators (SOs) were first introduced by Proebsting in [3] as an optimization strategy for a C interpreter. These specialized instructions were chosen from commonly occurring patterns in the tree-like intermediate representation generated by the lcc compiler. In our work, the superoperators are optimized bytecode basic blocks, prioritized based on execution frequency. His C interpreter, customized with SOs, performed 2 to 3 times faster than the initial switch-based interpreter on his benchmarks. He also conjectured

that up to 20 SOs should be enough to ensure the maximum performance improvement from this technique, which has also been experimentally supported in [4]. We tried to keep the number of the superoperators implemented low (ranging between 1 and 12 for different benchmarks) to preserve the simplicity and ease of maintainability while maximizing the executed code coverage.

Stephenson & Holst in [6], [7] selected and implemented multicodes based on execution traces collected during execution of programs. The patterns to be implemented are chosen based on a similar criterion to ours (frequency of occurrence multiplied by pattern length). A maximum of 48 multicodes are implemented for the Java Kaffe Interpreter with multicodes of length up to 20 tailored for each application, resulting in an execution time improvement of up to 24%. The maximum length for our superoperators is 550 (Blowfish benchmark), thus allowing considerable opportunities for optimizations.

In [8] Power & O'Donoghue evaluate the benefits of selecting and implementing a generic set of superinstructions on the CaffeineMark benchmark. They search for valuable patterns contained in basic blocks across all the applications in the benchmark as opposed to our superoperators (that are basic blocks) specific to each application. Also the code generated for each selected pattern is an unoptimized concatenation of the code generated separately for each instruction in the superinstruction. The implementation scheme of the superinstructions is similar to ours.

In [9] the authors present the design and implementation of a system of superinstructions for the Java interpreter in Sun's CVM. In selecting the valuable patterns, they compare both static and dynamic profiling approaches. They also optimize the stack code and the stack pointer updates as we do with our superoperators. They do not prioritize patterns and add up to 1024 superinstructions to the interpreter.

In [4] the author investigates different approaches to dynamically compiling or interpreting Java bytecode. The commonalities between [4] and our approach are: the use of superoperators; we generate similar optimizing information (register allocation and superoperators' boundaries); we also investigate the benefits of have the VM execute PicoJava patterns as superoperators. The differences are: we use extended bytecode basic blocks for superoperators instead of classic bytecode basic blocks; we execute all PicoJava folding patterns present in an application instead of selecting a few valuable ones as superoperators; we use an edge counter-based profiling method to select the most profitable set of superoperators ; we want our superoperator-based compilation system to be developed only on one VM infrastructure, unlike in [4] where a Java-to-bytecode compiler is used to generate optimization information and a different bytecode-to-native code interpreter is used to take advantage of the superoperators.

One issue that is discussed in [10] and [11] is the quickening optimization in the context of superinstructions, which refers to the process of rewriting slow instructions (i.e. that refer to the constant pool) into equivalent quick instructions upon their first execution. In our proposed compilation system, since the profiler we used provides information ahead of time about the slow instructions that are included in the selected superoperators, we are able to use the equivalent quick instructions when generating the (optimized) assembly code corresponding to the superoperators.

In [12] the authors present a continuous path and edge profiling scheme, which is considerably more complex than our profiling method since it uses hybrid instrumentation and sampling approach. Moreover, it is implemented in the optimizing compiler of the VM we used, while our profiling method is integrated in VM's baseline compiler that our system is built upon.

The compilation system proposed in this paper employs classfile annotations to encode the necessary superoperator information. Annotations have been previously used to convey optimization information such as null pointer check elimination ([13]) or superoperator boundaries ([4]).

An alternative to virtual execution of Java applications performed by interpreters or JIT compilers is a Java processor that implements the JVM in silicon and performs direct execution of Java bytecode, thus delivering better performance of up to 20 times faster than a general-purpose processor employing a JIT compiler or an interpreter. Examples of such processors are the PicoJava I and II processors developed by Sun ([14], [15]).

3. PROFILE-GUIDED COMPILATION

The compilation system presented in this paper proposes the following approach:

- To employ superoperators as new bytecode instructions, as well as selective, profile-guided method inlining, in order to achieve competitive level of performance w.r.t. a full JIT/AO compilation system.
- To select superoperators, based on runtime profiling information, from extended bytecode basic blocks (see sec. 3.1) of the current application.
- To apply few, but very targeted, superoperator-level optimizations in the process of generating assembly code corresponding to the selected superoperators only.

We believe that the **profile-guided**, SO-extended compilation system described in this paper is a better alternative to a full JIT compilation and/or adaptive optimization approach for a resource-constrained system. This claim is supported by the results obtained, with and without profile-guided optimization(PGO), posted on the SPEC website ([16])[1].

In our approach, the following components are needed:

- profiling method to detect a most profitable set of superoperators. We developed a new profiling method, based on dynamic edge-counter information. It is simpler and has lower overhead that other existing profiling methods ([12]).
- annotating module to process the profiling information obtained from the previous step and to add compilation "hints" (in the form of superoperator boundaries annotations and register allocation annotations) to the application's classfile.
- modifications to the VM to enable it to detect the annotations accompanying the bytecode of the application at classfile loading time, process them and generate optimized assembly code for the superoperators accordingly.

[1]e.g. for the Sun Fire X4200 system and for the SPEC CPU2006 benchmarks, the speedup ratio without PGO is 12.2, compared to a speedup ratio of 14.7 with PGO (the ratios are computed as a geometric mean across the CPU2006 suite)

Thus, our approach works as follows: first, the application is run to have the edge-counter profiler generate the profiling information. Next, the annotating module processes the profiling information and adds compilation "hints" to the classfile. On the next run (on the resource-constrained target platform), the annotations added to the classfile in the previous step are read and processed and the application runs with its specific superoperators enabled.

The advantages of our scheme are: reusability of profiling information, since it is stored in the form of annotations in the classfile, and improved runtime performance, with minimal impact on runtime memory consumption. The disadvantage is that, at least one time, the application needs to be run twice, in order to generate the profile information annotations. We believe that this approach is well suited for embedded systems, due to their resource-constrained characteristics.

3.1 Selecting superoperators

Our approach proposes to choose the superoperators from (what we called) *extended bytecode basic blocks* of the application to be executed, as directed by the profiling method described in section 3.3. A standard bytecode basic block is defined as a sequence of consecutive bytecode instructions that is terminated by a conditional branch, unconditional branch, method call or a branch target. For extended bytecode basic blocks, method calls do **not** represent a basic block boundary. The reasons why we decided to use extended bytecode basic blocks instead of classic basic blocks are:

[1] Basic blocks in classical form have proven to be promising as superoperators ([4], [5]); however, the sizes of classic basic blocks are quite small for most embedded applications. By using extended basic blocks as superoperators, the number of bytecode fetches and decodes decreases much more than for classic basic blocks, as does the dispatch overhead.

[2] Since extended basic blocks are considerably larger than classic basic blocks, code optimization opportunities increase considerably during the generation of assembly code corresponding to the new bytecode instruction (e.g., values that were kept in certain registers before a method call in an extended basic block can be reused after the call without having to load them again).

[3] Extended bytecode basic blocks are amenable to inlining method calls located within superoperators denoted as very profitable by our edge counter-based profiling method. In section 5.3 we discuss method inlining in more detail and show the improvements in execution performance due to inlining method calls present in the selected superoperators.

In the rest of the paper we will refer to extended bytecode basic blocks simply as basic blocks.

3.2 Annotations' processing

One of the components needed by our system is the annotating module that processes profiling information provided by the edge-counter profiler and adds compilation "hints" (in the form of superoperator boundaries annotations and register allocation annotations) to the application's classfile. This module can be incorporated in our JIT-like compiler or placed in the VM. The annotations generated by it and added to the application's classfile (see example in Fig. 1) will direct the performing of the optimizations presented in sections 3.2.1 and 3.2.2, as well as indicate the boundaries of superoperators. The impact of annotations on application's size is minimal: for the benchmark suite presented in section 5.2, we determined an upper bound on application's size increase of 2%. Note that the aforementioned annotations will be ignored by any annotation-unaware JVM.

In order to implement the superoperators selected for a particular application, we need new routines that generate optimized assembly code (that is more register-oriented rather than stack-oriented) for each superoperator. The assembly code-emitting methods (that we have developed) have been added to the platform-dependent compiler of the VM. When generating the corresponding assembly code, the following optimizations are applied at the superoperator level (as directed by annotations):

- register allocation (applied only to superoperators' bytecode).
- removal of instructions rendered unnecessary by SO-level register allocation and of stack pointer updates.

3.2.1 Register allocation at superoperator level

This optimization (similar to the approach in [4]) is performed at the superoperator level (i.e. **only** on the superoperators' bytecode), in order to generate optimized assembly code corresponding to the chosen set of SOs for an application. The key idea behind this optimization is: instead of using the same temporary register(s) to load values into (as in the original scheme of VM compiler), we are using a few general-purpose registers (denoted in what follows as R1, R2, ...) to hold values loaded from local variables or results of bytecode operations, thus being able to reuse them in subsequent operations (similar to copy propagation and common subexpression elimination). To illustrate the register allocation discussed above, consider the bytecode fragment in Figure 1.

39 iload_1	*... ⟶ R1*
40 iload 6	*... ⟶ R6*
42 iadd	*R1, R6 ⟶ R8*
43 istore 8	*R8 ⟶ ...*
45 aload_0	*... ⟶ R0*
46 getfield #6	*R0 ⟶ R2*
49 iload 6	*//reuse R6*
51 iaload	*R2, R6 ⟶ R9*
52 istore 9	*R9 ⟶ ...*
54 iload 8	*//reuse R8*
56 sipush 32768	*... ⟶ R1*
59 if_icmpne 76 (+17)	*R8, R1 ⟶ ...*

Figure 1: **Annotations example**

The annotations on the right guide the register allocation process. The registers on the left-hand side of an annotation represent the "sources" of the respective instruction, while the register on the right-hand side of the annotation represents the register that will hold the result, if any. " ... " means that there is no "source" or "target", respectively.

3.2.2 Optimizations following register allocation at superoperator level

As a result of the register allocation discussed in section 3.2.1, we are able to perform the following optimizations at superoperator level:

- Removal of most stack access operations and of stack pointer updates.
- Removal of other instructions that have become unnecessary (similar to copy propagation and common subexpression elimination).

Opcode	Assembly code - normal	Assembly code - SO
iload_1	loadIntLocal(T0, 1) pushInt(T0)	loadIntLocal(R1, 1) *pushInt(R1)*⟵
iload 6	loadIntLocal(T0, 6) pushInt(T0)	loadIntLocal(R6, 6) *pushInt(R6)*⟵
iadd	popInt(T0) popInt(T1) asm.ADD(T2, T1, T0) pushInt(T2)	*popInt(R6)*⟵ *popInt(R1)*⟵ asm.ADD(R8, R1, R6) *pushInt(R8)*⟵
istore 8	popInt(T0) storeIntLocal(T0, 8)	*popInt(R8)*⟵ storeIntLocal(R8, 8)

Table 1: Comparison of assembly code generated in normal mode and in superoperator mode; removed instructions are marked with ”⟵“.

Let us show an example for 1). In Fig. 1 consider instructions from 39 to 43. Because we are using different registers to hold values loaded from local variables by instructions 39 and 40, we do not need to ”push” the values on the virtual stack anymore (as it was done initially , when the same temporary register T0 was used to load all local variables or constants). Those values will be present in R1 and R6 respectively. Since instruction 42 will use R1 and R6 to compute the sum of the values kept in them, there is no need for the initial ”popping” from the virtual stack anymore. The result of instruction 42 will be present in R8 after execution, rendering the initial ”popping” from the virtual stack of instruction 43 unnecessary. Table 1 illustrates the example.

As can be seen from Table 1, all the pushes and pops have been eliminated due to our register allocation at the SO level (deleted instructions are marked with ”⟵“).
However, sometimes even though we can eliminate a virtual stack push or pop (e.g. a load from a local variable that has been performed before and whose value has not changed since), the stack pointer still needs to be adjusted in order to keep the state of the virtual stack correct.

Another situation is when we cannot eliminate all virtual stack pushes and pops. If for example in the bytecode block analyzed before (see Table 1), the bytecode fragment would start with instruction 40 instead of 39, then in the assembly routines generated for instruction 42 we could not remove the second pop since we need to retrieve the second operand from the virtual stack.

So far we have seen that the register allocation at the superoperator level can eliminate virtual stack pushes and pops that previously were necessary in generating assembly code for bytecode instructions. Now we will see that it can also eliminate entire bytecode instructions. Take for example the bytecode fragment in Fig. 1. The value present in local variable 6 is loaded 2 times throughout the fragment by instructions 40 and 49. Between these instructions the value of local variable 6 does not change, so instead of performing a second unnecessary load we can instead just reuse the value present in R6 in instruction 51, thus eliminating instruction 49 completely. The same idea is applied in eliminating instruction 54 since the value of local variable 8 in still in register R8 as the result of instruction 42.

To summarize, by using register allocation at the superoperator level, we have generated optimized assembly code corresponding to the set of superoperators selected for each application.

3.3 Profiling method

In order to find out which basic blocks would be most profitable as superoperators, we have developed an *edge counter-based profiling method* that tells us which basic blocks are the most important in an application. The measure that we have used in determining importance is the *code coverage* of a basic block, expressed as the ratio of the product between the execution frequency of a basic block and the length of that basic block (i.e the number of bytecodes contained in the basic block), to the total executed code.

The profiling method is based on edge counter information. For each conditional branch, the number of times a conditional branch has been taken (denoted as **t**) and the number of times it has not been taken (denoted as **t’**) are recorded. Based on these numbers available for each conditional branch, our profiling algorithm that computes the *local* execution frequency of each basic block (w.r.t. the method it belongs to) is shown below.

1 Algorithm 1

- **initialize** frequency of all basic blocks to 0
- **for** the current method being examined
  - **for each** branch in the method
    - update frequency of block that ends in current branch by adding **t** + **t’**
    - **if** target block does not end in conditional branch, **then** update frequency of target block by adding **t**
    - **if** fall-through block does not end in conditional branch, **then** update frequency of fall-through block by adding **t’**
  - **endfor**
  - **for each** basic block in the method
    - **if** current block does not end in a conditional branch and this block’s closest predecessor does not end in a conditional branch either, **then** update the frequency of the current block by adding the frequency of that predecessor
  - **endfor**
- **endfor**

After the local execution frequencies of the basic blocks have been computed, we determine the *global* execution frequency of each *unique* basic block (w.r.t. the entire application). This is done by keeping a list of all the unique basic blocks in the application and their corresponding frequencies. At first, the list is empty. We process each method that was executed: if a basic block was executed and is not in the list, then we add it and its corresponding frequency to the list. Otherwise, if the basic block was executed and

is already in the list (meaning that it was also executed in another method), then only add the frequency of the basic block that is being examined to the frequency of the copy block already present in the list.

Based on the global execution frequencies of the basic blocks, we compute the code coverage of each basic block (as defined above). Next, we choose the superoperators from the top 20 basic blocks, sorted in descending order by code coverage. In the selection process we take into account, for each basic block, its potential for performance improvement, as well as the amount of VM modifications necessary to implement it as a superoperator. The selected superoperators are different for each application in the benchmark suite.

Also, to make sure that the information obtained using our profiling method is correct, we have profiled all the applications in our benchmark suite using the JProfiler tool ([17]) and found that the results obtained using JProfiler concur with the information we have obtained using our profiling method.

4. DIFFERENT APPROACHES TO SUPEROPERATOR SELECTION

In order to test and evaluate the performance impact that our superoperators have on the benchmark suite, we have investigated three different approaches by modifying the baseline VM accordingly:

- **SO-extended configuration**, in which we have added profile-selected, extended bytecode basic blocks as superoperators (new bytecode instructions).
- **PJ-extended configuration**, in which we have enabled the optimal execution of all PicoJava folding patterns detected in each benchmark (for more details see section 4.2).
- **SOPJ-extended configuration**, in which we have combined the benefits of having profile-selected, optimized superoperators with that of optimally executing all PicoJava folding patterns present outside these superoperators in the application.

The rest of this section describes the details for each approach.

4.1 SO-extended VM configuration

In this configuration we have modified the initial (unoptimized) VM baseline configuration by adding extended bytecode basic blocks as superoperators. Given that our implementation of superoperators is based on the JikesRVM infrastructure, some of the following descriptions may show some dependence on the underlying VM platform. However, we believe that the approach described is general and can be applied to any JVM. The following modifications have been made:

- We have added **new bytecode instructions** corresponding to the superoperators chosen for an application.
- Based on the current method being examined by the compiler, we insert the new bytecode instruction corresponding to the superoperator present in the current method (if any) in the structure that contains a copy of the bytecode stream.
- In the fetch-decode *switch* statement present in the compiler we have added `<case>` branches for each superoperator. These new branches control the optimized assembly code generation for each superoperator. Fig. 2(a) shows an excerpt from the *switch* statement.
- Each new case branch calls the `emit_<application_name>_<SO_id>`(...) method corresponding to superoperator `<id>` of the application currently being executed. This method will generate the optimized assembly code (as described in section 3.2) corresponding to the bytecode instructions comprised in superoperator `<id>`. Upon returning from the "emit" method, the bytecode stream "cursor" is set to point to the bytecode instruction following the end of the superoperator.
- We have placed the `emit_<application_name>_<SO_id>`(...) assembly code-generating methods in the platform-dependent (in our case PowerPC) compiler. An example of such a method is shown in Fig. 2(b).

```
while (bcodes.hasMoreBytecodes()) {
  ...
  int code = bcodes.nextInstruction();
  switch (code) {
    ...
    case JBC_iload: {
      int index = bcodes.getLocalNumber();
      if (shouldPrint) asm.noteBytecode(biStart, "iload",
      index);
      emit_iload(index);
      break;
      }
    ...
    case JBC_so1 :{
      if (shouldPrint) asm.noteBytecode(biStart, "so1");
      emit_compress_so1 (field16, field38, field22, field213);
      bcodes.reset (178);
      break;
      }
    ...
    }
  }
```

(a) Switch statement excerpt

```
protected void emit_compress_so3() {
   //iload_2 ... ⟶ R2
   loadIntLocal(R2, 2);
   //iload 5 ... ⟶ R5
   loadIntLocal(R5, 5);
   //isub R2, R5 ⟶ R1
   asm.emitSUBFC (R1, R5, R2);
   //dup ⟶ use R1 ⟶ remove instruction
   //istore_2
   storeIntLocal (R1, 2);
   //ifge 153 R1 ⟶ ...
   asm.emitADDICr (R1, R1, 0);
   genCondBranch (GE, 153);
}
```

(b) *emit_<application_name>_<SO_id>(...)* routine example

Figure 2: Superoperator code examples.

With the modifications described above, the compiler is able to recognize the superoperators as new bytecode instructions and generate the corresponding optimized assembly code by using the emit_<application_name>_<SO_id>(...) methods that we have added.

4.2 PJ-extended VM configuration

Since PicoJava folding patterns can be considered very small superoperators, we initially developed this configuration with the purpose of comparing its results to the effects of the SO-extended VM configuration on benchmarks' execution performance. In this section we will present the modifications we have made to the original VM configuration in order to detect and optimally execute *all* the PicoJava folding patterns present in the Java applications.

4.2.1 PicoJava folding patterns description

Due to the stack architecture employed by Java bytecodes, there is inherent redundancy that impacts the execution performance of the JVM. One approach to alleviate the negative effects is *instruction folding*, which turns a sequence of stack-based instructions that have true data dependency into one register-based instruction. Bytecode instruction folding was implemented along bytecode hardware translation in the PicoJava II core ([15]).

During our profiling of the benchmark suite we have noticed that some bytecode instructions that were initially classified as non-foldable appeared very often. Since we are implementing PicoJava folding patterns in software, we decided to change the folding types of *getfield, getstatic, putfield* and *putstatic* from non-foldable (NF) to one of the other 4 foldable types (similar changes in folding types of bytecodes have been done in [4]).
Based on the bytecode folding type classification, groups of bytecodes can be determined as belonging to one of the following patterns:

```
1.LV, LV, OP, MEM     e.g.  iload_1 iload_2 iadd istore 4
2.LV, LV, OP          e.g.  iconst_1 iload 6 imul
3.LV, LV, BG2         e.g. iconst_1 iload_1 if_icmplt
4.LV, OP, MEM         e.g. iload_2 iadd istore 3
5.LV, BG2             e.g. iload_1 if_icmpne
6.LV, BG1             e.g. iload_0  ifgt
7.LV, OP              e.g. iload_1 isub
8.LV, MEM             e.g. iload 4  istore 5
9.OP, MEM             e.g. iadd istore 3
```

In order to detect and execute all the PicoJava folding patterns, we have added pattern-detection code to the platform-independent baseline compiler, more precisely in the beginning of the *fetch-decode while loop* shown in Fig. 2(a), before the *switch* statement. New emit_PicoJava _pattern_<X>(...) methods that generate optimized assembly code for each one of the 9 patterns (upon pattern detection) have been also added to the compiler.

4.3 SOPJ-extended VM configuration

After exploring the previous two approaches, an interesting question that arises naturally is: what would the impact on execution performance be if we were to combine the two configurations?
Since we have classified the new bytecode instructions added for superoperators as being **non-foldable(NF)** instructions, there would be no runtime conflict between superoperators and PicoJava folding patterns if we were to merge the two approaches presented above.

Also, as we will see in the next section, by developing the SOPJ-extended configuration we have combined the benefits of having superoperators tailored on the needs of each application with the advantages of executing all "general-purpose" PicoJava folding patterns present in every application in the benchmark suite.

5. PERFORMANCE EVALUATION

5.1 JikesRVM overview

In our performance evaluation we have used the Jikes Research Virtual Machine (JikesRVM)([18]).
JikesRVM is a research platform where new bytecode compilation and execution ideas can be explored and tested. It makes for an attractive choice because it is completely implemented in Java and is very well modularized. JikesRVM provides two compilers: the baseline compiler, which simply translates bytecodes to machine code, mimicking the virtual stack behavior of the VM, and an optimizing compiler that can be used in their JIT compilation system, paired with adaptive optimization.

5.2 Performance evaluation

First, we have measured the performance of the three different approaches presented in section 4, relative to the performance achieved by the JikesRVM baseline, fast-to-develop, compiler in the "production" configuration and in non-adaptive mode. We used the `processor cycles` metric, which denotes the number of processor cycles it takes for an application to execute.
The set of "embedded-type" benchmarks used in our performance evaluation, together with a brief description for each one, is as follows:

[1] *SpecJVM98_201_compress*: Modified Lempel-Ziv method for code compression

[2] *Blowfish*: encrypts a text file using symmetric block cipher with a variable-length key

[3] *PNG*: extracts properties of a PNG image (e.g. width, depth) using the sample program `PropertiesExample.java` contained in the `sixlegs.com.png` package([19]).

[4] *G*SM: Decompresses GSM encoded files into PCM audio files

[5] *DES.encrypt*: DES-based encryption of a text file using the `bouncycastle.crypto` package. Encrypts a text file using the `DESExample.java` encryption/decryption sample program in the `BouncyCastle Crypto` package([20]).

[6] *DES.decrypt*: DES-based decryption of a text file using the `bouncycastle.crypto` package. Decrypts a text file using the `DESExample.java` encryption/decryption sample program in the `BouncyCastle Crypto` package([20]).

We have run our experiments on a 300 MHz PowerPC 750 processor with 384 MB of RAM. One of the reasons why we chose this platform to conduct our performance evaluation is that it provides hardware performance counters. The operating system is `Ubuntu Linux 5.10` for Macintosh. We have

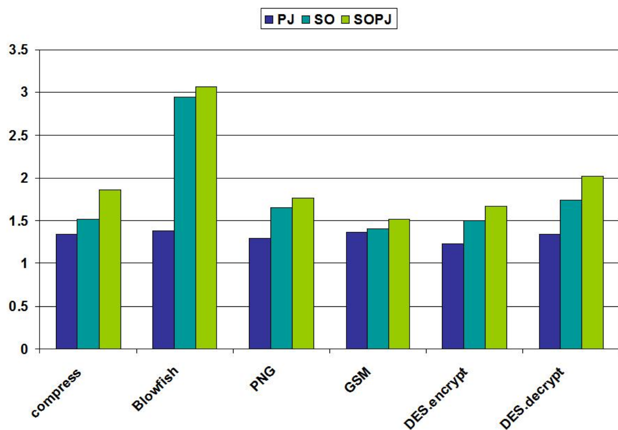

Figure 3: Speedups achieved by the three developed VM configurations over the unoptimized VM configuration

added performance counter support by modifying the 2.6.12 Linux kernel in order to include the `perfctr` library([21]). The JikesRVM version that we have used is 2.3.3, "production" configuration, cross-built using a `Linux/i386` platform, with added support for accessing the hardware performance counters through the `perfctr` library. We have introduced the specific `perfctr` support for the PowerPC750 processor to JikesRVM.

We first compare our performance results relative to the performance of the JikesRVM baseline compiler with no recompilation enabled (we called it "baseline" or "unoptimized" configuration). Fig. 3 shows the speedups of each of the PJ-extended, SO-extended and SOPJ-extended configurations over the baseline configuration. As can be seen in Fig. 3, the PJ-extended configuration yields speedups that range between a factor of 1.23 and a factor of 1.38, with an average of 1.33. We were expecting performance improvements in this range since the code coverage provided by the PicoJava patterns executed is small for all applications in the benchmark suite.

Figure 3 shows that the speedups for the SO-extended configuration range from 1.41 to 2.94, with an average of 1.79. As can be observed in Fig. 3, `Blowfish` and `DES.decrypt` are the top two benchmarks in terms of speedup. Table 2 lists the number of superoperators used by each application as new bytecode instructions, as well as the minimum size and maximum size of the superoperators (measured as number of bytecodes contained in a superoperator). As can be

	No of SOs	Min. size of SOs	Max. size of SOs
_201_compress	9	4	30
Blowfish	1	550	550
PNG	9	5	41
GSM	12	3	27
DES.encrypt	1	214	214
DES.decrypt	1	214	214

Table 2: Superoperators' characteristics

seen in Table 2, very few superoperators are being added to the applications: three of the benchmarks use only one superoperator, two of the applications have a set of 9 superoperators each, while the `GSM` benchmark has the largest number of superoperators, 12. The sizes of the superoperators vary considerably. The benchmarks with a larger set of superoperators have modestly-sized SOs (e.g. `PNG, GSM`), while the applications with a small number of superoperators have very large sizes (e.g. `DES.encrypt, Blowfish`). Overall, the size of the superoperators ranges from 3 bytecodes to 550 bytecodes.

Let us further analyze the speedups of the SO-extended configuration. From Fig. 3 we see that `Blowfish` and `DES.decrypt` have the largest performance improvements. This is due to the fact that these two benchmarks have large-sized superoperators, which create considerably more possibilities of optimizing the generated assembly code corresponding to these superoperators than in the case of modestly-sized superoperators.

There is one problem that might arise in the case of very large superoperators: are there enough registers to perform register allocation and consequently optimizations? For these two benchmarks, the superoperators have repetitive patterns (from loop unrolling), making it possible to reuse registers and also making it fairly easy to generate assembly code corresponding to the superoperators.

As can be seen in Fig. 3, the `GSM` and `_201_compress` benchmarks have reasonable speedups, but not as much as we would expect, considering the fact that they implement 9 and 12 superoperators respectively. This is due to the fact that the most important superoperators implemented for these benchmarks are modestly-sized, thus offering fewer possibilities of optimizing the generated assembly code.

The SOPJ-extended configuration is the one that yields the largest speedups, as it was expected, since in this configuration applications benefit both from the customized superoperators and from the "general-purpose" PicoJava folding patterns. The speedup ratios vary between 1.52 and 3.07, with an average of 1.98.

One might ask why the speedups for the SOPJ approach are not merely the sum of the speedups for the PJ-extended and the SO-extended configurations for each application. The answer is that many of the PicoJava folding patterns that were initially executed by the PJ-extended configuration (and contributed to the speedups achieved by that configuration) are contained in the superoperators added in the SO-extended configuration. In conclusion, the improvement in speedup for the SOPJ-extended VM over the speedup yielded by the SO-extended VM comes only from the optimal execution of PicoJava folding patterns that are not part of the selected superoperators.

So far we have compared the performance results that we have obtained to the performance of the unoptimized VM configuration. Since we want the proposed compilation system to deliver competitive performance w.r.t. state-of-the-art VMs, we will now compare the speedups achieved by the SOPJ-extended VM to the performance results obtained by `JikesRVM`'s adaptive optimization system (`AOS`[2]).

As can be seen in Fig. 4, the `compress, Blowfish` and `GSM` benchmarks, when executed using the SOPJ-extended VM, show differences in speedups of a factor of 2.89, 4.08 and 3.92, respectively, w.r.t. the speedups achieved by the `AOS` configuration. This behavior can be explained by the wide

[2] AOS denotes the JikesRVM "production" configuration with the adaptive optimization system turned on($-enable_recompilation = true$) and having the optimizing compiler as the initial compiler.

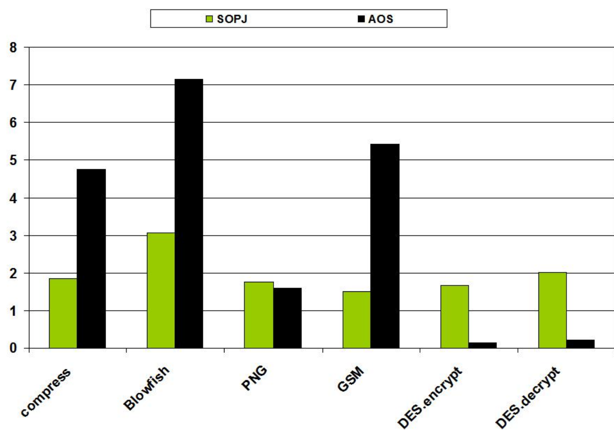

Figure 4: Comparison of speedups achieved by SOPJ and AOS configurations, relative to baseline configuration

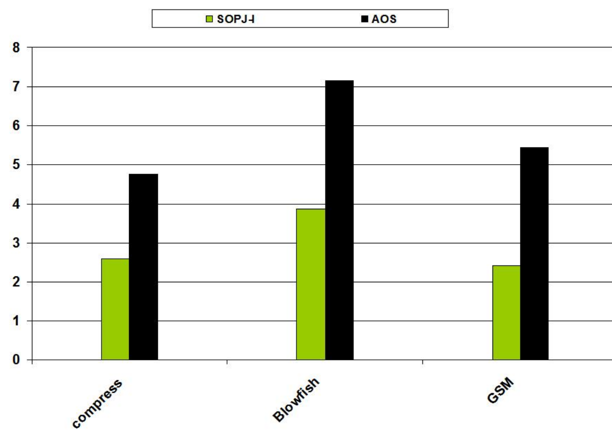

Figure 5: Comparison of speedups achieved by SOPJ-I and AOS configurations, relative to the baseline configuration

range of optimizations performed by the optimizing compiler and by the dynamic recompilation of "hot" methods employed by the AOS configuration. However, dynamic recompilation and the application of many expensive optimizations are not suitable for resource-constrained embedded processors, due to large memory consumption. Our profile-guided compilation system is much more modest w.r.t. resource consumption and thus is an appropriate alternative to a JIT/AO system in the field of resource-constrained environments (see section 5.4).

Compared to the speedups achieved in the SOPJ-extended VM, the `PNG` benchmark performs slightly worse in the `AOS` configuration, while `DES.encrypt` and `DES.decrypt` experience considerable decrease in performance by a factor of 0.15 and 0.22, respectively. This shows that for simpler benchmarks, as many embedded applications are, a few less expensive, but very targeted profile-guided optimizations lead to better performance than a JIT/AO system.

5.3 Method inlining

In order to improve the selected SOs and consequently achieve performance results closer to the `AOS` performance, we decided to perform *selective inlining* of method calls embedded in the chosen superoperators for the `compress`, `Blowfish` and `GSM` benchmarks, as directed by runtime profiling data. We chose to perform selective, profile-guided inlining only on these benchmarks since the speedups produced by the SOPJ-extended configuration for these applications are less than the `AOS` performance results. We used the `Javassist` toolkit([22]) in order to perform the high-level (bytecode) inlining of the method calls present in the original superoperators. The reason why we selected **only** the method calls present in the chosen superoperators for inlining is because runtime profiling data indicated that these method calls accounted for a considerable portion of execution time. This led to the increase in size of many of the initial superoperators, thus not only removing the overhead incurred by the (now inlined) method calls, but also exposing more possibilities of optimizing the generated assembly code for the superoperators.

We inlined 3 method calls for the `Blowfish` benchmark, 5 method calls for the `compress` benchmark and 11 method calls for the `GSM` benchmark. Most of the method calls present in the original superoperators had 0 depth (i.e. there were no other embedded method calls present). However, we did handle 4 method calls of depth 1 (present in the `GSM` benchmark) by recursively inlining bottom-up.

Figure 5 shows the speedups attained by the SOPJ-I VM, which consists of the SOPJ-extended VM with all the method calls present in the original superoperators being inlined. We can see from Figure 5 that selectively inlining the method calls and modifying the original superoperators to reflect it led to a general speedup increase. The performance differences between the proposed SO-extended VM and the AOS VM are now 2.16 for the `compress` benchmark (was 2.89 before inlining), 3.29 for the `Blowfish` benchmark (was 4.08 before inlining) and 3.01 for the `GSM` benchmark (was 3.92 before inlining). Thus we have managed to bring our performance results much closer to the `AOS` performance.

When measuring the modifications in benchmarks' size due to inlining, we found that the benchmarks had increased by only 2.9% on average (0.1KB for the `Blowfish` benchmark, 1.9KB for the `compress` benchmark and 0.5KB for the `GSM` benchmark). We conclude that performing profile-guided, selective inlining of method calls led to further considerable improvement in runtime performance, with only minimal effect on the benchmarks' size.

5.4 Runtime memory consumption

Our profile-guided compilation system represents a viable alternative to full JIT compilation for resource-constrained environments, not only in terms of **performance** (as indicated by the results presented in sections 5.2 and 5.3), but also from a **runtime memory consumption** point of view. We have measured the runtime memory usage of our SO-extended VM vs. the amount of memory needed by the AOS VM.

	Avg. RSS %	Avg. VSZ %
SO-extended VM	2.89	2.13
AOS VM	47.67	45.11

Table 3: Runtime memory consumption increase

Table 3 shows the increases in *resident set size(RSS)* and

virtual memory size(VSZ) needed by our SO-extended VM and by the AOS VM, w.r.t. the amount of runtime memory that the original (baseline) VM needs. The results presented in Table 3 represent the average of runtime memory measurements for all the applications in the benchmark suite (presented in section 5.2), over multiple executions.

We can see that full JIT compilation, paired with adaptive optimization causes a considerable increase in runtime memory consumption (namely a RSS increase of 47.67% and a VSZ increase of 45.11%), while our SO-extended compilation system achieves competitive execution performance, with only a negligible increase in runtime memory requirements (namely a RSS increase of 2.89% and a VSZ increase of 2.13%). Thus, the proposed SO-extended VM performs better than the JIT/AO VM by approximately a factor of 16, on average, w.r.t. runtime memory consumption.

6. CONCLUSIONS AND FUTURE WORK

The research presented in this paper proposes a *profile-directed, lightweight bytecode compilation system* to improve execution performance of Java applications on resource constrained embedded systems.

In order to make our system a viable alternative to state-of-the-art JIT compilers and adaptive optimization systems for resource-constrained environments, we use profile-selected, extended bytecode basic blocks as new bytecode instructions (SOs) and apply annotation-directed optimizations only to the selected superoperators. We also evaluate the impact on execution performance due to profile-directed, selective inlining of method calls embedded in the superoperators chosen for each application. In our simplified compilation system, this compensates for the adaptive inlining performed by JIT/AO systems.

Compared to a JIT/AO system, our SO-extended VM shows a difference in speedup that is within a factor of 2 for the worst performing applications. However, w.r.t. runtime memory consumption, our SO-extended VM outperforms the JIT/AO system by a factor of 16.
All of the above make the proposed compilation system a viable choice for resource-constrained devices.

For future work, since one of the most important constraints for embedded systems is low energy consumption, we are working on evaluating the energy savings provided by our simplified compilation system on a 405EP PowerPC system. From preliminary evaluations we have determined that, in our proposed compilation system, applications experience considerable reduction in the number of completed instructions, which we believe will lead to substantial reduction in energy consumption.

7. ACKNOWLEDGEMENTS

The first author would like to thank Ilya Issenin, Arun Kejariwal and Radu Cornea for their feedback and help in technical matters. The first author would like to thank Robin Garner for providing a patch that adds `perfctr` support to JikesRVM.

8. REFERENCES

[1] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[2] Yunhe Shi, David Gregg, Andrew Beatty, and M. Anton Ertl. Virtual machine showdown: stack versus registers. In *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 153–163, New York, NY, USA, 2005. ACM Press.

[3] Todd A. Proebsting. Optimizing an ansi c interpreter with superoperators. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 322–332, New York, NY, USA, 1995. ACM Press.

[4] Ana Lucia Velloso Azevedo. *Annotation-aware dynamic compilation and interpretation.* PhD thesis, 2002. Chair-Alexandru Nicolau.

[5] Ana Azevedo, Arun Kejariwal, Alex Veidenbaum, and Alexandru Nicolau. High performance annotation-aware jvm for java cards. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 52–61, New York, NY, USA, 2005. ACM Press.

[6] Ben Stephenson and Wade Holst. Multicodes: optimizing virtual machines using bytecode sequences. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 328–329, New York, NY, USA, 2003. ACM Press.

[7] Ben Stephenson and Wade Holst. Advancements in multicode optimization. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 186–187, New York, NY, USA, 2004. ACM Press.

[8] Diarmuid O'Donoghue and James F. Power. Identifying and evaluating a generic set of superinstructions for embedded java programs. In *ESA/VLSI*, pages 192–198, 2004.

[9] Kevin Casey, David Gregg, M. Anton Ertl, and Andrew Nisbet. Towards superinstructions for java interpreters. In *SCOPES*, pages 329–343, 2003.

[10] Etienne Gagnon and Laurie Hendren. Effective inline-threaded interpretation of Java bytecode using preparation sequences. In *Compiler Construction (CC '03)*, volume 2622 of *LNCS*, pages 170–184. Springer, 2003.

[11] M. Anton Ertl, Christian Thalinger, and Andreas Krall. Superinstructions and replication in the Cacao JVM interpreter. *Journal of .NET Technologies*, 4:25–32, 2006. Journal papers from *.NET Technologies 2006* conference.

[12] Michael D. Bond and Kathryn S. McKinley. Continuous path and edge profiling. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 130–140, Washington, DC, USA, 2005. IEEE Computer Society.

[13] Patrice Pominville, Feng Qian, Raja Vallée-Rai, Laurie Hendren, and Clark Verbrugge. A framework for optimizing java using attributes. In *CASCON '00: Proceedings of the 2000 conference of the Centre for Advanced Studies on Collaborative research*, page 8. IBM Press, 2000.

[14] Sun Microsystems Inc. *Picojava* I *microprocessor core architecture*, 1999. http://solutions.sun.com/embedded/databook/pdf/whitepapers/WPR-0014-01.pdf.

[15] Sun Microsystems Inc. *Picojava*-II *programmer's reference manual*, March 1999.

[16] *Standard Performance Evaluation Corporation.* www.spec.org.

[17] ej-TECHNOLOGIES. *JProfiler.* http://www.ej-technologies.com/products/jprofiler/overview.html.

[18] *Jikes Research Virtual Machine.* http://jikesrvm.sourceforge.net/.

[19] *PNG Software.* http://www.sixlegs.com/software/png/.

[20] *The Legion of the Bouncy Castle.* http://www.bouncycastle.org/.

[21] Mikael Pettersson. *Linux performance monitoring counters driver.* http://www.csd.uu.se/ mikpe/linux/perfctr/.

[22] *Javassist Toolkit.* http://www.jboss.org/products/javassist.