A Lazy Developer Approach: Building a JVM with Third Party Software

Nicolas Geoffray, Gaël Thomas, Charles Clément and Bertil Folliot
Université Pierre et Marie Curie - LIP6/CNRS/INRIA - Regal
104 avenue du Président Kennedy
75016 Paris, France
firstname.lastname@lip6.fr

ABSTRACT

The development of a complete Java Virtual Machine (JVM) implementation is a tedious process which involves knowledge in different areas: garbage collection, just in time compilation, interpretation, file parsing, data structures, etc. The result is that developing its own virtual machine requires a considerable amount of man/year. In this paper we show that one can implement a JVM with third party software and with performance comparable to industrial and top open-source JVMs. Our proof-of-concept implementation uses existing versions of a garbage collector, a just in time compiler, and the base library, and is robust enough to execute complex Java applications such as the OSGi Felix implementation and the Tomcat servlet container.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures

General Terms

Languages, Design

Keywords

Java, LLVM, JnJVM, BoehmGC, GNU Classpath

1. INTRODUCTION

The Java Virtual Machine (JVM) specification [27] describes a complete execution environment to execute Java applications. The developer has the freedom of implementation of the execution environment as long as the implementation follows the specification. For example, a developer can choose (i) interpretation, compilation or mixed interpretation/compilation of bytecodes, or (ii) the garbage collector algorithm, whether it is generational, copying, mark and sweep, incremental, etc and (iii) the base library implementation, which must follow the Sun Java base library API.

Many implementations of the JVM are available, either developed by industrials, e.g BEA's JRockit [3], IBM's J9 [7],

Sun's OpenJDK [14], or open-source, such as Cacao [4], Kaffe [11], JamVM [10], SableVM [13], JnJVM [31], Apache Harmony [2], IKVM.NET [8] or JikesRVM [17]. They all have their own way of implementing the specifications. For example JamVM is interpretation only and JikesRVM is compilation only, with different level of optimizations.

Industrial JVMs often have their own implementation of garbage collection, compilation, interpretation and class libraries. Open-source JVMs tend to all use the GNU Classpath base library implementation [6] (some JVMs are currently being ported to the newly open-sourced Sun implementation), and the Boehm garbage collector [22]. GNU Classpath and Boehm GC are popular among JVM implementations because they are virtual machine agnostic, hence they do not depend on a particular JVM implementation.

In this paper, we present the design and implementation of LadyVM[1], which follows a lazy developer approach. For most JVM components, we try to use existing, robust software, that have the following properties:

- Virtual machine agnostic. The component must not be implemented with a specific virtual machine design in mind. One should be able to use it as-is.

- Library oriented. The component must expose a clean interface to the developer.

- Large user-community. A large user-community yields a better responsiveness to bug reports.

- Performance. The component must have been used in other projects and shown that it has good performance.

The goal is to develop a full JIT based JVM with very little effort. This paper evaluates the approach by means of specification compliance, performance, interface evolution of the third-party components, and complexity of development.

With LadyVM, we want to apply experimental JVM technologies and be able to compete with existing JVMs. For example, we are developing the Java isolation API [28] on top of LadyVM. Java isolation requires extra care of memory allocation, and memory accesses, in order to execute Java applications in the same JVM. We are currently applying

---

[1]LadyVM is the port of JnJVM [31] to LLVM and Boehm.

isolation to service oriented architectures [26] and we need a JVM with performance comparable to other JVMs.

LadyVM uses three major third-party software:

- GNU Classpath [6]: LadyVM uses GNU Classpath as its base library. Until Sun open-sourced its base library, GNU Classpath was the only free and competing implementation of the Java core libraries. It has a clean interface between the library and the virtual machine. Almost all open-source JVMs use GNU Classpath today.

- Boehm Garbage Collector [22]: LadyVM uses the Boehm conservative garbage collector, which is successfully used in JVMs such as Kaffe or Cacao. Mono, an open-source CLI implementation also uses it. BoehmGC provides the allocation and finalization interface and semantics needed by the JVM specification.

- LLVM [29]: LadyVM uses the Low-Level Virtual Machine (LLVM) as a just in time compiler. LLVM is a compiler framework that performs compilation optimizations and code generation on a low-level bitcode format. LLVM is supported by Apple and has a large user-community.

LadyVM links these third-party software to implement a full Java Virtual Machine. We show that LadyVM has comparable performance with industrial JVMs such as OpenJDK for scientific applications. For general applications, like the applications in the SPEC JVM98 benchmark [19], it needs additional optimizations to fully compete with top JVMs. LadyVM is robust enough to execute complex applications such as the Felix OSGi implementation [1] and the Tomcat servlet container [9].

The remainder of the paper is organized as follows. Section 2 presents the architecture of LadyVM. Section 3 describes the interface evolution of GNU Classpath, BoehmGC and LLVM since we started LadyVM, as well as the difficulties to port LadyVM to these new interfaces. In Section 4, we present performance of LadyVM relative to other JVMs. Section 5 presents related work and Section 6 concludes the paper.

2. ARCHITECTURE OF LADYVM

LadyVM links third party-software to implement a full Java Virtual Machine that follows the JVM specification[27]. Figure 1 shows the overall architecture of LadyVM.

LadyVM performs the following operations:

- Class loading. The GNU Classpath libraries as well as the application classes are parsed and saved as metadata in LadyVM. LadyVM resolves and intialises the classes.

- JVM bytecode to LLVM IR translation. LadyVM translates the class type to LLVM type and the Java methods to LLVM methods.

- GNU Classpath interface. GNU Classpath has a virtual machine interface represented as native methods.
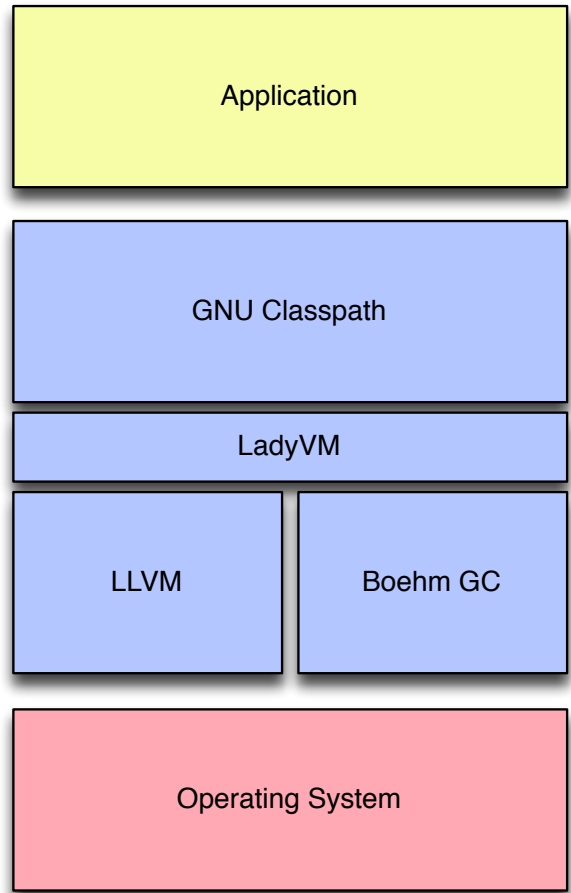

Figure 1: Basic architecture of LadyVM.

- Thread system. LadyVM provides locks for Java synchronizations.

- A function provider for the LLVM JIT. LLVM uses a callback to materialize a function when it has not been loaded. LadyVM implements this callback by either locating the function in the metadata if the function has already been loaded, or by loading its containing class.

Figure 2 shows the basic interactions between the different components of LadyVM. As of today, all the components are used as-is in LadyVM: we do not require the modification of the components source code. We achieve this independence by two means: (i) GNU Classpath and Boehm GC have been used in many virtual machine environments, hence became generic enough to be used in any environment and (ii) LLVM is developed as a compiler framework with an extensible architecture. One can load new optimization passes, provide its own memory manager or function materializing.

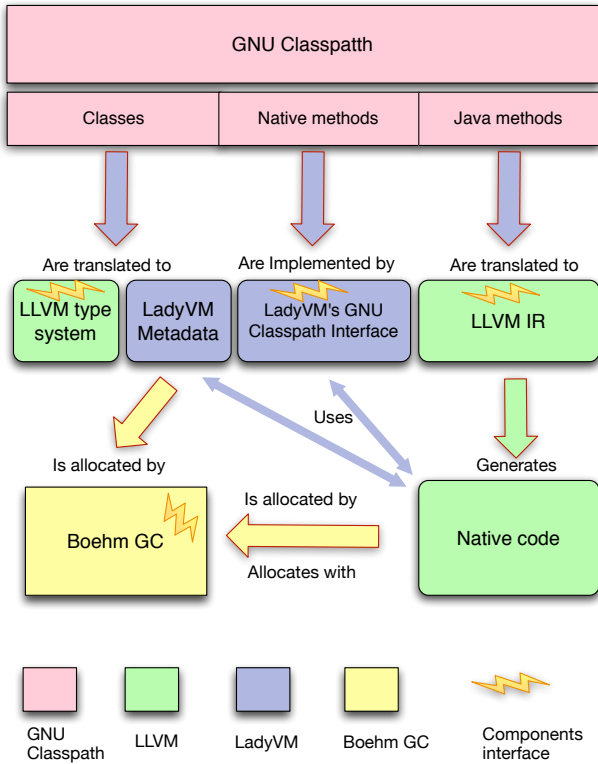LadyVM also uses two other external functionalities:

Figure 2: Interactions between LadyVM, LLVM, Boehm GC and GNU Classpath. The bolts represent the interfaces between the components and LadyVM.

- Posix threads: LadyVM uses the pthread library to map Java threads and Java synchronization primitives to native threads.

- GCC unwinding library: exceptions in LadyVM are mapped to C++ exceptions, and LLVM generates GCC-compatible exception tables. Hence LadyVM uses the GCC unwinding library to find the `catch` clause of exceptions.

2.1 Compiler

LadyVM uses LLVM as its just in time compiler. LLVM is a compiler framework that compiles a low-level instruction set either to memory for JIT or on file for static compilation. It has many compilation optimization passes, which makes it a very competing compiler, comparable to GCC's performance. Moreover, the instruction set (which is also called the intermediate representation) of LLVM is low-level enough to be the target of different languages such as C, C++, Fortran using llvm-gcc and Java with LadyVM.

LadyVM performs two translations: Java classes are transformed to LLVM types and Java methods are transformed to LLVM IR. LadyVM also implements Java-specific optimizations on the LLVM IR.

2.1.1 Type translation

Figure 3 shows the LLVM type of the the `java/lang/Object` class. The `java/lang/Object` class in GNU Classpath does not have any field. Therefore it only contains the fields of the LadyVM root object: (i) a pointer to a virtual table used for calling virtual functions, (ii) a pointer to its LadyVM internal class, used for runtime type information and (iii) a pointer to a lock for synchronizations. This layout is however subject to change for further optimizations (for example one could use a hash table for locks, or place the class pointer in the virtual table).

%struct.java_lang_Object = type { i8*, i8*, i8*}

Figure 3: LLVM type of `java/lang/Object`. The `i8*` keyword represents a pointer.

LadyVM also creates a type for the static part of a class. When a class is loaded, its static fields are all allocated in a single object which reference the fields. Figure 4 describes the process of resolving a class in LadyVM. Due to just in time resolution, all field objects are represented with the `java/lang/Object` LLVM type. The virtual table of a class is filled with function pointers or stubs when a function is not yet compiled. For simplicity reasons, a static instance inherits the `java/lang/Object` fields; its class is set to zero.

LadyVM uses the LLVM type information in three cases. First, for optimization purposes, in its type-based alias analysis. Second, to be able to perform escape analysis and allocate Java objects on stack (see Section 2.1.3). And finally, to ease the IR generation of field accesses (see Section 2.1.2).

2.1.2 Bytecode translation

All JVM bytecodes are translated to one or more LLVM instructions. Since LLVM IR is in a static single assignment (SSA) form, and the JVM bytecode is stack oriented, LadyVM creates a temporary stack when parsing bytecodes. This stack is used at compile time, not at runtime. The LLVM IR is generated by creating instructions whose operands are popped from the stack and whose result is pushed on the stack. For example the `IADD` bytecode pops two operands from the compilation stack, creates an LLVM `Add` instruction and pushes the result on the compilation stack. Figure 5 shows the process of translating the `ILOAD_0`, `IADD` and `IRETURN` in LLVM IR. Figure 6 is the translation between a Java `add` method from JVM bytecode to LLVM IR.

Most of the JVM bytecodes translate to one, two or three LLVM instructions. The notable exceptions are: field accesses , method calls, exceptions, runtime checks, synchronizations, allocations and switch instructions.

In the case of a field access, a method call, a runtime check or an allocation, the bytecode operates on a class that may have not been resolved when the bytecode is being compiled. In such case, LadyVM inserts a callback function in the LLVM IR. At runtime, the function will resolve the class and return the appropriate information (e.g. a pointer to a field or a class). The callback function is only executed once.

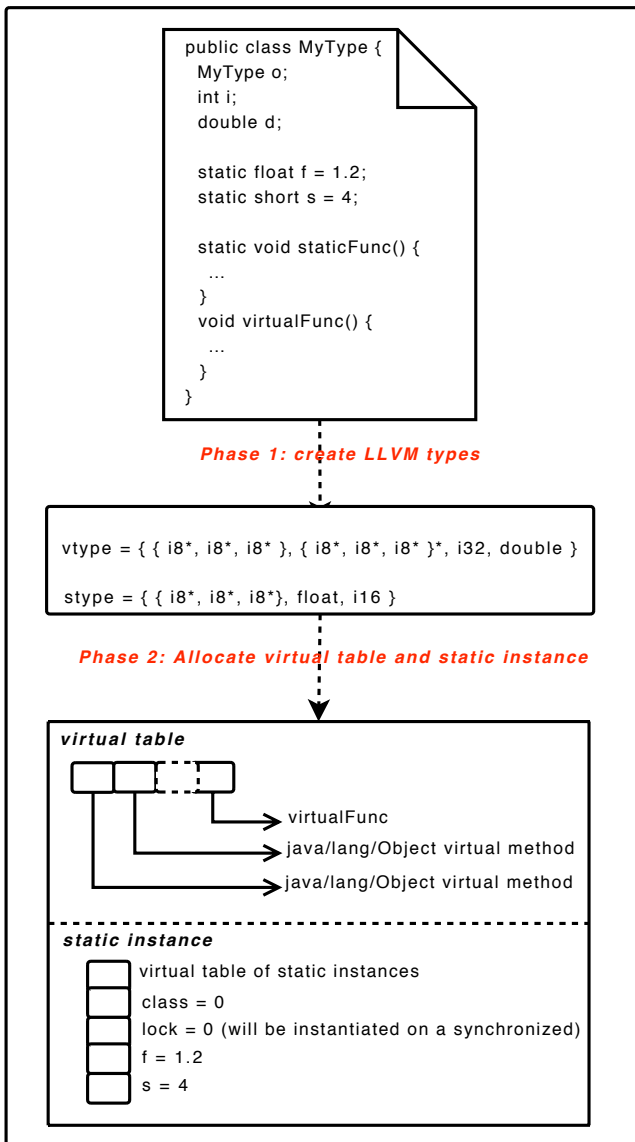For virtual field accesses, if the class of the object on which

```
public class MyType {
  MyType o;
  int i;
  double d;

  static float f = 1.2;
  static short s = 4;

  static void staticFunc() {
    ...
  }
  void virtualFunc() {
    ...
  }
}
```

Phase 1: create LLVM types

vtype = { { i8*, i8*, i8* }, { i8*, i8*, i8* }*, i32, double }

stype = { { i8*, i8*, i8*}, float, i16 }

Phase 2: Allocate virtual table and static instance

virtual table

→ virtualFunc
→ java/lang/Object virtual method
→ java/lang/Object virtual method

static instance

virtual table of static instances
class = 0
lock = 0 (will be instantiated on a synchronized)
f = 1.2
s = 4

Figure 4: Resolving a class in LadyVM

to retrieve the field is resolved, the object is casted from the default `java/lang/Object` LLVM type to its real type.

For non-virtual calls, i.e. `invokespecial` or `invokestatic`, LadyVM relies on LLVM for callbacks. If the called method has not been resolved, LLVM generates a callback which will invoke a function provider. LadyVM implements its own function provider. It is responsible for loading the class if needed, translate the JVM bytecode to LLVM IR and then return the LLVM IR to LLVM. Finally, LLVM will patch the native call instruction to call the newly resolved method instead of the callback.

For virtual, non-interface calls, i.e. `invokevirtual`, LadyVM emits LLVM instructions that loads the virtual table of the object, loads the function pointer from the virtual table at the offset of the method and calls the function. At

```
case ILOAD_0 : {
  // Push argument 0 on the compilation stack.
  push(new LoadInst(arg[0], "", currentBlock));
  break;
}

case IADD : {
  // Pop from the compilation stack and convert
  // it to a jint.
  Value* val2 = popAsInt();
  // Pop from the compilation stack and convert
  // it to a jint.
  Value* val1 = popAsInt();
  // Create the LLVM Add.
  Value* res = BinaryOperator::createAdd(val1, val2,
                            "", currentBlock);
  // Push the result on the compilation stack
  // and type it as a jint.
  push(res, AssessorDesc::dInt);

  break;
}

case IRETURN : {
  // Pop from the compilation stack and convert
  // it to a jint.
  Value* res = popAsInt();

  // Create the LLVM return instruction.
  ReturnInst::Create(res, currentBlock);
}
```

Figure 5: Translation from ILOAD_0, IADD and IRETURN bytecode to LLVM IR.

runtime, if the function has not been materialized, a stub is executed that will call the function provider of LLVM. The function provider will in turn lookup the offset of the method in the virtual table, compile the method if not already compiled and update the virtual table.

For each interface call site, i.e. `invokeinterface`, LadyVM creates a linked-list cache that records which object classes were called last. At runtime, when the object class is not referenced in the cache, LadyVM performs a full method lookup. The resulting class is then placed at the top of the list. The following calls will verify the object class with the top of the list. If they are equal, the call proceeds. This implementation of interface calls has the advantages of being compiler independent. For example, Alpern *et al.* [20] generate optimized native stubs to implement efficiently the `invokeinterface` bytecode. Since we want to rely on a generic JIT such as LLVM to generate code, we did not follow this implementation.

Synchronization bytecodes (i.e. `monitorenter` and `monitorexit`) are lowered to a thin lock implementation [21].

Switch instructions (i.e. `tableswitch` and `lookupswitch`) are decomposed in many LLVM instructions that perform the comparisons and branches.
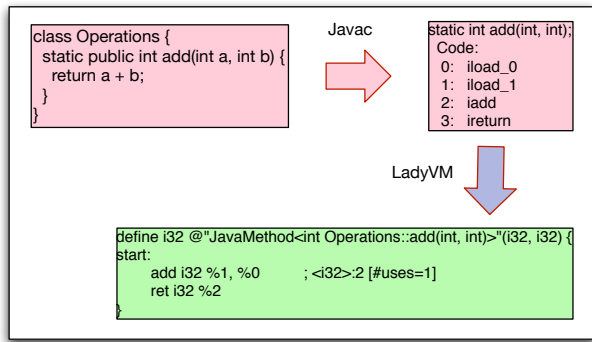
Figure 6: A Java `add` method from Java to LLVM IR.

Allocation bytecodes (i.e. `new`, `newarray`, `anewarray` and `multianewarray`) are translated to calls to the allocation functions of LadyVM. These functions will in turn load the classes if ot resolved, and invoke Boehm GC to perform the memory allocation.

2.1.3 High-level optimizations

LLVM is a compiler framework that language implementors can extend to perform language-specific optimizations, without modification of the source code of LLVM. LLVM can dynamically load new compilation passes that operate on methods, loops or modules (a module is a set of functions).

LLVM contains many compilation optimization passes. LadyVM takes advantages of these passes and always apply them to Java functions. The most relevant passes for Java code are:

- Promote memory to register: this pass will promote Java local variables to registers.

- Predicate simplifier: this pass helps to remove redundant array checks.

- Loop Invariant: this pass moves invariant code out of a loop. For example an array check can be hoisted out of a loop.

- Global Value Numbering: this pass eliminates redundant instructions.

- Constant Propagation: the `javac` compiler does a poor job for constant propagating. This pass will propagate constants inside a function.

LadyVM also extends LLVM with three new optimization passes. The first optimization performs a type based alias analysis. The second optimization loads constant values (the size of a given array, the virtual table of an object or the class of an object) only once in a function. Finally, the third optimization performs escape analysis on a single function. The escape analysis allocates objects that do not escape from the function on the stack instead of the heap.

2.1.4 Limitations

LLVM can currently only be used with a compilation-only approach. It has an interpreter, however it can not call native functions. This would prevent the use of JNI in LadyVM. Furthermore, modern JVMs often use a mixed interpretation-compilation approach, where methods are interpreted at first, and compiled when they achieve a "hotness" threshold. LLVM does not provide such a system.

Another limitation of LLVM is its lack of garbage collection support. It contains hooks to implement a garbage collector, and a starting implementation of a garbage collector. However, it has not been fully tested.

Finally, LLVM targeting mostly unsafe languages such as C or C++, does not have a type-based alias analysis (TBAA) as well as the correct infrastructure to implement it efficiently. Our implementation of TBAA is not complete, because LLVMs IR does not offer all the needed type informations.

2.2 Garbage Collector

LadyVM uses the Boehm GC as allocator and garbage collector. Metadata as well as Java objects are allocated with the Boehm GC.

The Boehm GC implements the Java finalization mechanism. Hence on each allocation, LadyVM specifies the finalizer method (or a stub to the finalizer method), which is located in the virtual table of the object to allocate. Finalization mechanisms are different depending on the language considered (C++, Java, C#). From the point of view of the garbage collector, there are mainly two algorithms: one for managed environments, and one for unmanaged environment. The former requires that the memory of an object is freed after the finalization method if the object is still unreachable. For the latter, the destructor method is always called and the object destroyed right after.

LadyVM uses two interfaces of the Boehm GC. It uses the allocation method to allocate Java objects and the finalizer registration method to inform the GC of the object's finalizer method.

2.2.1 Limitations

The Boehm GC was designed for non type-safe languages such as C or C++. The drawbacks for using the Boehm GC for a JVM is that (i) it can only be conservative, (ii) it can not be a generational GC efficiently and (iii) it can not be a copying and compacting GC. On the contrary, most modern JVMs use a generational, compacting and precise garbage collector [24].

2.3 Base Library

LadyVM uses GNU Classpath as its base library. GNU Classpath is a free implementation of the standard library of Java. Many open-source JVMs use GNU Classpath, such as JikesRVM, Cacao, Kaffe, GCJ or JamVM.

Because GNU Classpath is not tailored to any JVM implementation it must define a virtual machine interface. This interface is a set of functions that are virtual machine or operating system dependent. For virtual machine dependent

functions, the underlying JVM must implement them. For example, GNU Classpath does not have access to the layout of a runtime Java object, therefore, the `Object.getClass()` function, that returns the class of an object, has to be implemented by the JVM. GNU Classpath provides default implementations of operating system dependent functions as well as non-optimized implementations of some virtual-machine dependent functions.

LadyVM in its current state, implements 85 virtual-machine dependent functions. This is sufficient to execute standard Java benchmarks such as SpecJVM98 or the Java Grande Forum Benchmark, and complex applications such as the Felix OSGi implementation and the Tomcat servlet container.

2.3.1 Limitations
GNU Classpath follows the Java standard library API which does not have a specification. Therefore, it is not fully compatible with the Sun implementation. As of today, it implements 95% of the Java 1.5 API [6].

GNU Classpath also provides a virtual machine interface in order to be easily ported to virtual machines. However this interface limits some optimizations, if not modified.

2.4 Summary
GNU Classpath, Boehm GC and LLVM implement the desired functionalities, but impose some limitations. Table 1 summarizes these limitations.

3. INTERFACE EVOLUTION
By using third-party software to build a JVM, we face the problems of uncontrolled interface evolution. Since we are not the primary developers of these projects, we have to document ourselves and port LadyVM to new interface changes. In this Section, we present our experience on following the API changes of LLVM, BoehmGC and GNU Classpath.

3.1 LLVM Interface
LadyVM interfaces with LLVM by ways of API use, extension, and runtime hooks. LadyVM uses the LLVM API to build types and the intermediate representation.

We started the implementation of LadyVM with LLVM version 1.9. The intermediate representation API of LLVM is the most fragile API when switching to new LLVM versions. From version 1.9 to 2.3, all releases required a change to LadyVM. These changes are mostly due to new features and securing the public API.

Overall, moving LadyVM to a new LLVM release requires less than one man-day.

3.2 Boehm GC
Since the Boehm GC targets uncooperative environments such as C or C++, it does not require compilation information, nor compiler barriers or typed mallocs. Therefore LadyVM only uses two functions from Boehm GC, which are allocation and finalizer registration.

We started the project with Boehm GC version 6.8. When moving from Boehm GC 6.8 to 7.0, there was no API changes

Time (sec.)	Time %	Name of the Pass
5.84	58.7	X86 Instruction Selection
0.55	5.5	Local Register Allocator
0.37	3.4	Java bytecode to LLVM IR
0.17	1.6	X86 Machine Code Emitter
9.95		Total time

Table 2: Compilation time without optimization of a Java HelloWorld program.

for LadyVM.

3.3 GNU Classpath
GNU Classpath provides VM classes, a one-to-one mapping between a standard class (e.g. `java/lang/Class`) and a virtual machine implementation (e.g. `java/lang/VMClass`). The virtual machine class contains native functions or default implementations that virtual machines must implement or optimize.

We started the project with GNU Classpath version 0.93. LadyVM now uses the latest version, 0.97.2. Since GNU Classpath has had many releases before 0.93, the API changes between 0.93 and 0.97.2 were minimal.

GNU Classpath moved to java version 1.5, including generics, with release 0.95. Since the JVM specification did not change between version 1.4 and 1.5, the move to GNU Classpath 0.95 did not require any changes in LadyVM. The only modification we made was due to the `ldc` opcode, which loads classes since JVM 1.5.

4. RESULTS
LadyVM comprises 18k lines of C++ code for the core virtual machine code, and 4k lines of C code for the GNU Classpath interface. These numbers are equivalent to projects such as JamVM (that does not have a JIT), or IKVM.Net (which is based on Mono for GC and JIT).

For benchmarking, we use an Athlon XP 1800+ processor with $512MB$ of memory on Gentoo Linux 2.6.23.

4.1 Compilation time
Table 2 shows the time for compiling an `HelloWorld` Java class program without any optimization and Table 3 shows the time with the optimization passes turned on. We only show the most expensive and Java-relevant passes, such as the ones described in 2.1.3. Since LadyVM is a compile-only JVM, the execution of the `HelloWorld` program involves compiling many methods. These methods create the class loader, load the file and print. With GNU Classpath version 0.97.2, 536 methods have to be compiled, with a total of 26952 bytecode instructions.

When optimizations are off, most of the compilation time is spent in instruction selection, the local register allocator (which is one of the simplest in LLVM) and the live analysis of variables do not cost much. With optimizations and the linear scan register allocator (the most efficient register allocator of LLVM), instruction selection is also the

Component	Limitations
Boehm GC	Non-conservative, non copying, non-compacting.
LLVM	Compilation-only, no default implementation of Garbage Collection.
GNU Classpath	Not fully compatible with the Java base library. The virtual machine interface is not optimized.

Table 1: Limitations of the components.

Time (sec.)	Time %	Name of the Pass
3.99	32.7	X86 Instruction Selection
1.21	9.9	Global Value Numbering
0.66	5.3	Live Variable Analysis
0.65	5.3	Predicate Simplifier
0.53	4.2	Live Interval Analysis
0.45	3.6	Linear Scan Register Allocator
0.37	3.3	Java bytecode to LLVM IR
0.15	1.2	X86 Machine Code Emitter
12.21		Total time

Table 3: Compilation time with optimizations of a Java HelloWorld program.

Time (sec.)	Time %	Name of the Pass
25.8	34.3	X86 Instruction Selection
4.56	6.0	Live Variable Analysis
3.79	5.6	Linear Scan Register Allocator
3.70	4.9	Live Interval Analysis
3.50	4.6	Predicate Simplifier
2.53	3.3	Java bytecode to LLVM IR
2.43	3.2	Global Value Numbering
1.06	1.4	X86 Machine Code Emitter
75.17		Total time

Table 4: Compilation time with optimizations of Tomcat version 6.0.16.

most time-consuming pass. The linear scan register allocator needs live variable and interval analysis. In comparison, other JVMs execute the `HelloWorld` program in less than one second. OpenJDK executes the HelloWorld program in 228 milliseconds, JikesRVM in 108 milliseconds and Cacao in 279 milliseconds.

Table 4 shows the compilation time for running Tomcat 6.0.16. LadyVM boots Tomcat in 75 seconds. In comparison, OpenJDK boots Tomcat in 8 seconds.

Since the just in time compiler of LLVM is still under development (most of the current improvements in LLVM target static compilation), there has been little to no optimization on compilation time. There is no `baseline` compiler such as in the JikesRVM [17] that compiles a method quick but in an unoptimized fashion. We think that, with LLVM gaining more and more users, optimizations on compilation time will be investigated.

Modern JVMs have a mixed interpreter/compiler environment, that compiles hot methods and interprets cold methods. A profiling mechanism selects which methods must be

compiled with aggressive optimizations [30].

4.2 Performance

To evaluate LadyVM we used the Jave Grande Forum (JGF) benchmark [23] and the SPECJVM98 benchmark [19]. We run all VMs with a minimum and maximum heap size of $512MB$.

We compare LadyVM with industrial and open-source JVMs. We use Sun's OpenJDK (Java version 1.7) in server mode, IBM's J9 version 1.6.0.1, JikesRVM version 2.9.2 with the optimizing compiler as the default compiler and Cacao version 0.98. JikesRVM is the most popular open-source JVM lead by IBM, and has many contributors. Cacao is an open-source JVM focused on just in time compilation. Its developer base is equivalent to ours. JikesRVM, Cacao and LadyVM use GNU Classpath as their base library. All these JVMs perform runtime optimizations, and J9 and OpenJDK execute in a mixed interpreter and compiler environment. To limit our comparisons with steady-state performance (i.e. excluding compilation and class loading times), we iterate 10 times each benchmark in the same JVM and extract the best number.

The main limitations of LadyVM, which will explain most of the performance loss compared to other JVMs are:

1. Object creation: we use the Boehm GC allocator, which is not as fine tuned as industrial GCs for Java.

2. Exception throw: we use the unwinding library of GCC, which does not have good performance.

3. Removal of array checks: the removal of array checks in LadyVM is based on a non-optimized predicate simplifier of LLVM.

4. Escape analysis: LadyVM does not perform an interprocedural escape analysis.

Overall, LadyVM competes well with other JVMs in scientific applications (present in the JGF benchmark). On applications that benchmark all aspects of a JVM, LadyVM is outperformed by JikesRVM and Sun, but has similar performance than Cacao.

JGF has three types of benchmarks. Section 1 contains low-level operations such as mathematical operations or memory assignments. Section 2 contains CPU-intensive applications. Section 3 contains large-scale applications. On all figures, higher is better (a relative score of two means two times better than LadyVM).

Figure 7 shows a subset of the micro-benchmarks (Section 1 of JGF). We see that LadyVM has similar performance for computation and memory micro-operations than industrial JVMs or JikesRVM, but does not perform well on exceptions and allocations.
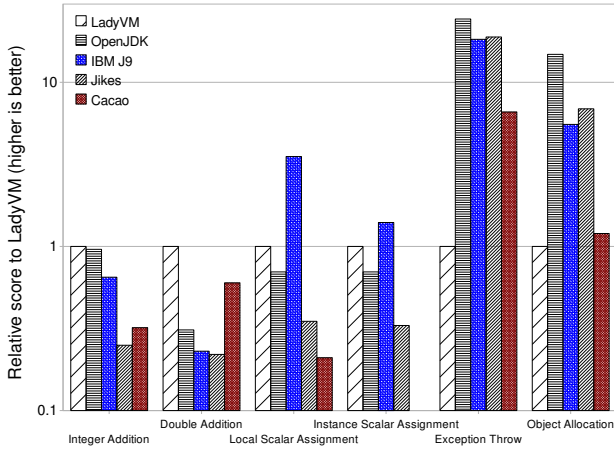

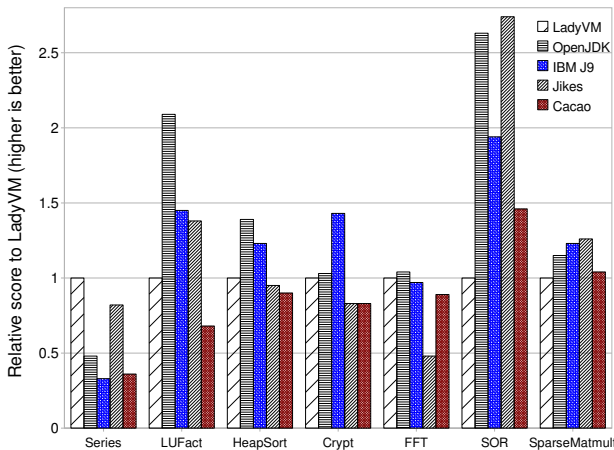
Figure 7: Section 1 of JGF (Logarithmic scale).

Figure 8: Section 2 (small input size) of JGF.

Figures 8, 9 and 10 show the results of Section 2 of JGF, applied on different input sizes (respectively small, medium and large). A score of 0 means an out of memory error. On most benchmarks LadyVM has similar performance than OpenJDK, J9 and JikesRVM, and outperforms Cacao.

The poor performance of LadyVM for the SOR benchmark is mainly due to our non-optimized array checks removal implementation, which is based on an incomplete LLVM optimization pass.

Figure 11 shows the results of Section 3 of JGF. LadyVM has poor performance with the Euler benchmark. This is due to our non-optimized escape analysis. In previous work [31], we have shown that these two benchmarks are very sensitive
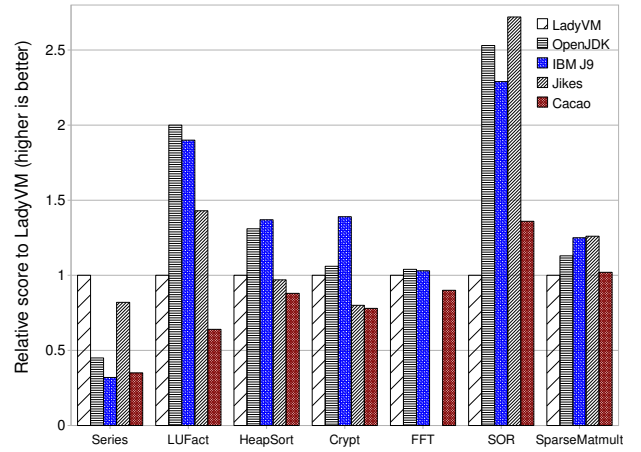

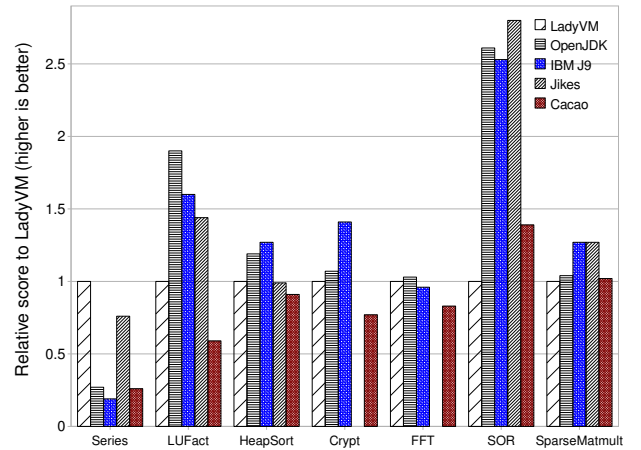**Figure 9: Section 2 (medium input size) of JGF.**

Figure 10: Section 2 (large input size) of JGF.

to a good escape analysis. An optimized escape analysis will yield better performance, as the garbage collector is less often invoked. In [31], we show that with escape analysis, the Euler benchmark goes from 1616 collections to 963.

On the MolDyn, RayTracer and AlphaBetaSearch benchmarks, LadyVM is similar in performance to JikesRVM (from 1.0 to 1.6, higher is better). OpenJDK and IBM J9 outperform the other virtual machines.

Figure 12 shows the results of the SPECJVM98 benchmark. Due to limitations in LadyVM we could not run the benchmarks following the SPEC rules: LadyVM can not currently execute the `mpegaudio` benchmark. To exclude as much as possible compilation time, for each VM we took the best time out of 10 iterations of a benchmark. The results show that LadyVM is equivalent in performance to Cacao (LadyVM and Cacao use the same garbage collector), but is outperformed by OpenJDK, J9 and JikesRVM.
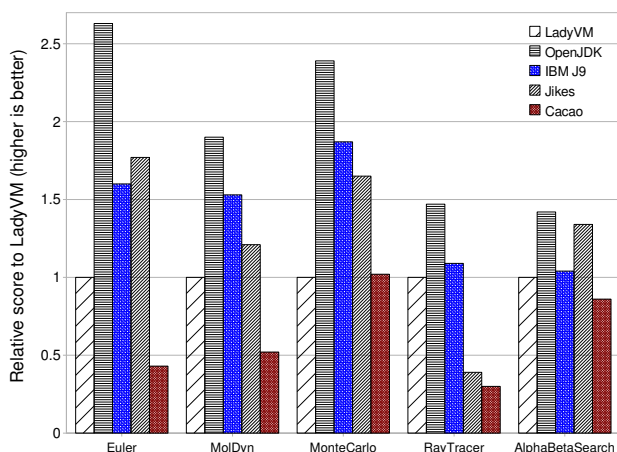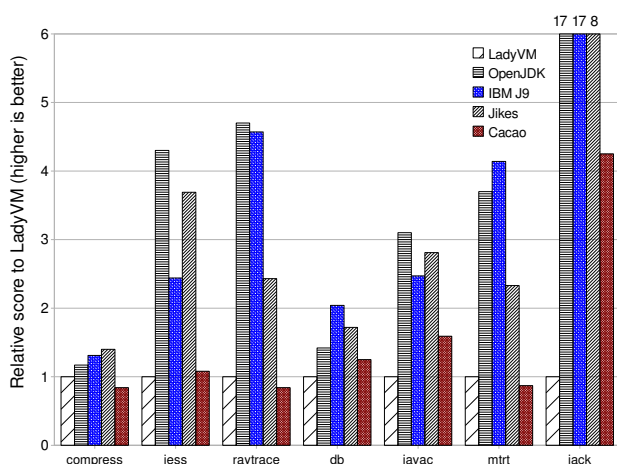
Figure 11: Section 3 of JGF

Figure 12: SPECJVM98 results

4.3 Summary

In this Section, we have shown that LadyVM has performance comparable to industrial JVMs in non memory intensive scientific applications. For memory intensive applications, LadyVM is outperformed by industrial JVMs. Moreover, the compilation time also limits the performance of LadyVM. It boots Tomcat 10 times longer than industrial JVMs. These limitations are due to the compile-only approach of LadyVM, as well as its non-generational Boehm GC.

5. RELATED WORK

As stated in introduction, there are many implementations of the Java Virtual Machine [2, 14, 7, 4, 17, 31, 10, 3, 13, 8, 11, 16]. Most research implementations reuse existing software, such as the base library with GNU Classpath and the Boehm GC. However our approach is unique in the sense that all major components are third party software with different goals than creating a JVM. Table 5 describes the components used by different JVM implementations.

Our approach is very close to IKVM.Net [8] or GCJ [16]. These projects use existing infrastructures to build a Java Virtual Machine. IKVM.Net uses Mono [18] and GCJ uses GCC [15]. However, IKVM.Net can not follow the exact semantics of JVM class loading. LadyVM follows the JVM specification [27] and is thus fully compliant with the class loading mechanism. GCJ is a ahead-of-time compiler and therefore can only interpret dynamically loaded classes. Both IKVM.Net and GCJ limitations are due to using infrastructures which are too high-level and/or not well suited.

6. CONCLUSION

We have shown that with little effort (18k lines of code), one can build a full Java Virtual Machine using third-party software. For scientific applications, the performance is equivalent to industrial JVMs. There are however performance limits with other types of applications that stress many parts of the JVM (garbage collection or exceptions). LadyVM is robust enough to execute complex applications such as the Felix OSGi implementation and the Tomcat servlet container.

By developing our own JVM in a timely fashion, we are keen to develop JVM research extensions, such as the Isolate API [28] and an application of isolates to the OSGi platform [26].

We also validated the approach of LadyVM by developing a Common Language Infrastructure [25], called N3 [12]. It uses the Boehm GC, LLVM and the pnetlib library [5].

7. AVAILABILITY

LadyVM (and N3) is publicly available on an open-source license. The project is hosted by the LLVM team on subversion.

```
svn checkout
http://llvm.org/svn/llvm-project/vmkit/trunk vmkit
```

8. REFERENCES

[1] Apache felix. http://felix.apache.org/site/index.html.
[2] Apache Harmony. harmony.apache.org.
[3] BEA JRockit. www.bea.com.
[4] Cacao JVM. www.cacaojvm.org.
[5] DotGNU portable.NET. dotgnu.org/pnet.html.
[6] The GNU Classpath Project.
    www.gnu.org/software/classpath/ classpath.html.
[7] IBM J9. www.ibm.com/developerworks/java/jdk.
[8] IKVM.Net. www.ikvm.net.
[9] Jakarta tomcat. http://jakarta.apache.org/tomcat/.
[10] JamVM. jamvm.sourceforge.net.
[11] Kaffe JVM. www.kaffe.org.
[12] N3: N3 is Not .NET. llvm.org.
[13] Sable VM. www.sablevm.org.
[14] Sun OpenJDK. openjdk.java.net.
[15] *The GNU Compiler Collection.* http://gcc.gnu.org/.
[16] *The GNU Compiler for the Java Programming Language.* http://gcc.gnu.org/java.
[17] The Jikes Research Virtual Machine.
    http://www-124.ibm.com/
    developerworks/oss/jikesrvm.
[18] The Mono Project. www.mono-project.org.

	Garbage collector	Base libraries	Compiler	Execution mode	Development type
J9 (IBM)	Internal	IBM JRE	Internal	Mixed	Closed
OpenJDK (Sun)	Internal	Sun JRE	Internal	Mixed	Open-source
JRockit (BEA)	Internal	Sun JRE	Internal	Mixed	Closed
Apache Harmony	Internal	Internal	Internal	Mixed	Open-source
JikesRVM	Internal	GNU Classpath	Internal	Compilation	Open-source
Cacao	Boehm or Internal	GNU Classpath	Internal	Compilation	Open-source
Kaffe	Boehm or Internal	GNU Classpath	Internal	Mixed	Open-source
JamVM	Internal	GNU Classpath	-	Interpretation	Open-source
SableVM	Internal	GNU Classpath	-	Interpretation	Open-source
IKVM.NET	Boehm (Mono GC)	GNU Classpath	Mini (Mono JIT)	Compilation	Open-source
GCJ	Boehm	GNU Classpath	GCC	Compilation	Open-source
LadyVM	Boehm	GNU Classpath	LLVM	Compilation	Open-source

Table 5: Comparison of virtual machine development.

[19] The SPEC JVM98 Benchmark.
http://www.spec.org/osg/jvm98/.

[20] B. Alpern, A. Cocchi, S. Fink, and D. Grove. Efficient Implementation of Java Interfaces: Invokeinterface Considered Harmless. In *Proceedings of the Object Oriented Programming, Systems, Languages, and Applications Conference*, pages 108–124, New York, NY, USA, 2001. ACM.

[21] D. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin Locks: Featherweight Synchronization for Java. In *Proceedings of the Programming Language Design and Implementation Conference*, pages 258–268, Montreal, Canada, 1998.

[22] H. Boehm, A. Demers, and S. Shenker. Mostly Parallel Garbage Collection. In *Proceedings of the Programming Language Design and Implementation Conference*, pages 157–164, Toronto, Canada, June 1991.

[23] E. P. C. Center. Java grande forum benchmark suite – version 2.0. http://www.epcc.ed.ac.uk/javagrande/, 2003.

[24] R. Dimpsey, R. Arora, and K. Kuiper. Java Server Performance: a Case Study of Building Efficient, Scalable JVMs. *IBM Systems Journal*, 39(1):151–174, 2000.

[25] ECMA International. Common Language Infrastructure (CLI), 4th Edition. Technical Report ECMA-335.

[26] N. Geoffray, G. Thomas, C. Clément, and B. Folliot. Towards a new Isolation Abstraction for OSGi. In *Proceedings of the First Workshop on Isolation and Integration in Embedded Systems (IIES 2008)*, pages 41–45, Glasgow, Scotland, UK, April 2008.

[27] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, Boston, USA, 2000.

[28] Java Community Process. JSR-121: Application Isolation API Specification. http://jcp.org/jsr/detail/121.jsp. by Sun Microsystem.

[29] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 75–86, Palo Alto, USA, March 2004. IEEE Computer Society.

[30] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. A Dynamic Optimization Framework for a Java Just-in-time Compiler. *SIGPLAN Not.*, 36(11):180–195, 2001.

[31] G. Thomas, N. Geoffray, C. Clément, and B. Folliot. Designing Highly Flexible Virtual Machines: the JnJVM Experience. In *Software: Practice and Experience*. John Wiley & Sons, Ltd., 2008.