Improving Java Virtual Machine performance with SIMD technology

Danke Xie, Chern-Hsi Jeffrey Wang and Michael Schimpf
Department of Computer Sciences and Engineering
Santa Clara University

Operating Systems (COEN283)
Dr. Ming-Hwa Wang

1. Abstract

2. Introduction

In virtual machine environment, if the host machine and the target machine use different instruction set architectures (ISA). The virtual machine hypervisor has to emulate the target machine. The traditional way of doing it is to do instruction-by-instruction interpretation, and performs the required operations on data. The performance can be 20 to 1. This technique can be greatly improved by Just-in-time compilation, which dynamically translates the to-be-executed code into machine code of the host machine. The performance can be 10 times faster than interpreter.

However, there are different issues associated with JIT compilation.

- Startup time. JIT compilation requires more time to compile and optimize the code before the program can be run. In Android phones, in Dalvik virtual machine, Google didn't use a traditional JIT compilation as used in Sun's Java virtual machine. What they did is to generate a intermediate representation (IR) that can run faster, instead of generating machine code that can directly run on host machine. The result is 2x to 5x fasters. However, another project called Dalvik Turbo claims 3.5x faster than Dalvik.
- About 80% code is executed once. So there is no need to translate and run. It is as slow as interpreting the code. So to effectively use JIT, the virtual machine needs to find code that runs for substantial amount of time, and use JIT to optimize execution of the code. One important aspect is to identify hotspots.

- Context-switching between virtual machine and translated code may incurs more overhead than traditional LRU cache replacement algorithm. Virtual-machine aware cache may help virtual machine performance.
- To avoid overhead incurred by dynamic translation, the code can also be translated by a Ahead-of-time compiler (AOTC). It has been shown to improve performance by 5% than Dalvik.
- Dynamic translation also allows the code to use new machine features (SIMD) to make the code run faster than original machine code.

In this project, we will utilize SIMD technologies and push most of the heavy-lifting to an ahead-of-time (AOT) compiler/optimizer to optimize Java code performance. In case needed, we will also modify a small-size JVM to achieve better performance.

2.1 Objective

The objective of our project is to improve the performance of Java code with various techniques, especially using the SIMD technology. We will review existing research that uses SIMD or autovectorization to optimize Java performance, and present our improvement on the current methods.

2.2 Problems Overview

In recent years, there has been a trend toward developing and running Internet and network servers using safe languages and processes-level Virtual Machine (VM)-based runtime environments. This trend is justified; VMs offer a more secure execution environment than native platforms, while they allow programs to be portable. Those facilities come at a cost: Java programs that run on a Java Virtual Machine tend to perform slower than equivalent programs written in C. The system neutrality of bytecode acts as a disadvantage where performance is concerned. This is because code optimization relies heavily on system-specific features. Since Java bytecode is system-neutral, it cannot be optimized for a specific hardware set.

2.2.1 Java bytecode interpretation is performance bottleneck

Java bytecode has the same design issue like the language itself it is designed to run on all available machines. This means it is not easy to create a machine dependent and therefore fast solution. The bytecode is a generic approach and it encompasses the common attributes of all processor. Bytecode will not use any processor extensions and specific properties which can make a assembler program very fast. Another issue is that the bytecode itself has not much improved over the last 20 years because of the compatibility issues between the various versions of the Java language and as Java developer you have also not the choice to select between some general processor architecture like CISC and RISC. All this together results in a situation in which the bytecode is the bottleneck and cannot be improved very much.

2.2.2 Improving Java performance by translation to native machine code

One approach to make Java code faster is to use a Just in Time compiler. The task for a JIT is to locate hot spots in Java which means code areas which are interpreted more often than other code fragments and to

compile this often interpreted code segments into machine language for the specific platform under which the Java program is currently running.

This approach includes some problems, it is not easy to find the code areas which are really often called and which makes it worth to translate it onetime to save execution time the next callings. Another problem is the initial overhead if you JIT translate code fragments it will results in a compiling time which makes the execution for the first time slower.

As a rule of thumb: most of the code (80 %) will be executed only one time and only 20 % of the code of a typical Java program will be executed very often. So it is very important to select the right code fragments to be faster than the only interpreted version of the Java program.

The nowadays JIT approach is more than 10 years old and still a main topic in Java, so that we can assume that the JIT technology is a mature approach to speed up Java programs.

2.2.3 Vectorization of scalar code (SIMD) - Single Instruction, Multiple Data

SIMD is a very old approach from the beginnings of computing. Texas Instruments ASC, CDC Star-100 and Cray supercomputers uses this approach, but they abandon this idea for better solutions like MIMD (Multiple Instructions, Multiple Data. The desktop computer development of the 1980's have grasp this idea and implemented it in the current processors of this time phase. The processor extensions are known as SSE, MMX, AltiVec, etc.and this extensions are used mostly for Graphics, Audio, 3d, etc.

The basic concept is, when a assembler command is already translated in the processor core it can be reused with other data and a time consuming additionally fetching of a command is not needed. This means if you have a sequence of equal assembler commands the processor can fasten this sequence. For example, if you have to display the screen and move 16 millions times data from one point to another and you use all the time the same "move" command. Then it will be much faster with a vector extension. But, you have to develop and program your code to this extension and this process is called vectorization.

2.2.5 Computational complexity of vectorization

Vectorization is still not an easy part of Software development you have to know about this kind of functionality and mostly you have to use assembler to get the most out of all the processor extensions which are available, but also the compiler creators have done a lot and tried to optimize the higher language code with the usage of vectorization code fragments. It is not easy in a compilation process to try to analyze the code and to find code sequences which are good to transform into a vectorized segment if the user has not explicit develop his code for vectorization execution. One advantage of the compiler optimization process is, that the compiler can go over the code very often and he can use as much passes as he think is useful. After a couple of passes it is possible to identify the hot spots of the source code and to use super instructions to replace the old code with the new code.

But, this is not easy in a runtime environment. The Java runtime and the JIT has not much time to decide which code should be replaced by SIMD fragments. If the computing and searching for code segments is too long the benefit is diminishing and can turn possible to a minus speed up. There is only a couple of milliseconds time to find and identify the code fragment and to replace it with the vectorize code and this is a real challenge in a

runtime environment or setup.

2.3 Why This is a Project Related to the Class

A virtual machine is an Operating System. Conceptually, similar to an operating system, it performs resource management and provides a computing platform for applications and guest OSes to run. Historically, the first virtual machine is VM/CMS, an IBM virtual machine OS on IBM mainframe System/370, System/390, zSeries, System z and compatible systems.

This research is about OS support for faster Java code execution.

2.4 Scope of Investigation

In this research, we investigate how to efficiently apply vectorization technology to Java virtual machine. It includes the methods to translate Java bytecode to compiler IR and to automatically vectorize the IR by a compiler (LLVM). The vectorized machine code generated from IR in placed in a dynamic library. The Java program (bytecode) is automatically modified to utilize the library. Previous research has used C as the source language, GCC for vectorization and CLI as the IR [1]. Due to recent development of LLVM, which is also the recipient of the 2012 Software System Award by ACM [7], we decide to use LLVM for vectorization in this research, Java as the source language, and as small-size portable Avian JVM as the virtual machine.

In this research, we focus on 1) AOT compilation and optimization, 2) automatic modification of Java bytecode to use optimized native code, 3) optimizing the interpreter by using superinstructions. We also aim to understand the strengths and limits of current vectorizers and the use of superinstructions in JVM.

3. Theoretical Bases and Literature Review

Traditional virtual machines for languages such as Java and .NET use interpreters for fast startup time. Recent virtual machines are augmented with JIT (Just-in-time) compilers for increased performance. The performance of interpreter is lower than what static and JIT compilers can achieve. The causes are inherent to the nature of interpreters. First, a program is translated into an IR used internally by the interpreter. At this point, only fast optimizations can be applied. Then, the IR is “executed”: each instruction is dispatched, and the code chunk matching its semantics is executed. Instruction dispatch is an important source of inefficiency due to the overhead introduced. It is found that the execution of a single bytecode takes from 20 to 30 machine instructions [1]. They account for loading a bytecode from memory, decoding it, and transferring the flow of control to the code that performs the appropriate function. Parameters, if any, are then read from the evaluation stack, the computation is performed, and the result is written back.

A common way for improving performance of interpreters is to reduce the number of instructions to dispatch. To this end, common sequences of instructions are gathered into single superinstructions [2].

3.1 Theoretical background to solve the problem

3.1.1 Automatic vectorization

Since most opportunities for data parallelism occur between iterations of loops, traditional vectorization techniques mainly focus on exploiting loop level data parallelism. This kind of techniques is referred to as loop-based vectorization. Another kind of vectorization is called SLP (Superword Level Parallelism) vectorization [10] [13]. It identifies groups of isomorphic instructions exposing superword level parallelism, and combines them into equivalent vector instructions. The loop-based vectorization exploits data parallelism among different executions of the same instruction, while the SLP vectorization exploits data parallelism among different instructions in the straight-line code (usually in the same basic block), so the SLP vectorization can be a complement to the loop-based vectorization.

3.1.2 SIMD support on modern processors

Single instruction, multiple data (SIMD) are a class of computers with multiple processing elements that perform the same operation on multiple data points simultaneously. These machines are able to take advantages of data level parallelism. Most modern CPU designs use SIMD instructions in order to improve the performance of multimedia use, such as adjusting the contrast of an image or the volume of an audio file.

3.2 Related research to solve the problem

In previous research, a JIT compiler is used to automatically translate the bytecode to vectorized native code [10]. The performances of some benchmarks are improved by 50-100% when running in single thread. As we will explain later, JIT may incur too much overhead for auto-vectorization.

JNI is sometimes used for optimizing Java performance, especially in the numeric field. In previous research [11], the same numeric computing routines have two implementations, one implementation in Java, and the other in C. The availability of C implementation is detected at runtime. If it is available, the C routines are used via JNI, otherwise, the Java routines are used. This research does not use an automatic vectorizer for Java.

In another research, C code is converted to a vectorized bytecode. The bytecode is converted to machine code by a JIT compiler [8]. We will further reduce the overhead by moving machine code generation to compile time. The machine code part and the Java bytecode part are linked via JNI. In yet another research, autovectorization is done in interpreter [1]. We believe the performance would be better if at lesat part of the bytecode is compiled instead of interpreted.

3.3 Our Solution: Auto-vectorizing with JNI and Superinstructions

Our approach is to improve JVM performance through auto-vectorizing compilation. We want to compile Java bytecode into a vectorized code ahead-of-time (AOT). The Java bytecode will be modified to invoke native methods via Java Native Interface (JNI). JNI enables one to write native methods to handle situations when an application cannot be written entirely in the Java programming language. The dynamic library can be generated from our platform-independent code. The performance should be competitive with native compilation.

For vectorization, we want to target parts of the code runs for substantial amount of time, such as computationally heavy loops. In these loops, we can replace multiple occurrences of the same scalar instruction with a single vector instruction that operate on multiple data elements simultaneously (SIMD). The individual data elements must be independent.

Superinstructions can also be used to execute multiple instructions with a single dispatch. They are groups of the most common instructions concatenated. One example is an instruction sequence of an iload followed by an iadd. This can be concatenated to form a single instruction iloadiadd. The compiler of the interpreter can now perform more powerful optimizations. Two stack operations can be optimized away as the local variable accessed in the iload instruction can be accessed within the instruction. Further a whole instruction dispatch is made redundant as the two instructions now form a single implementation in the interpreter [12].

3.4 Other Solutions and Why Ours Is Better

Most existing JVMs use JIT, which dynamically translates the to-be-executed code into machine code of the host machine. The performance can be 10 times faster than interpreter. However, the startup time is a concern. JIT compilation requires more time to compile and optimize the code before the program can be run. Also, about 80% code is executed once. So there is no need to translate and run. It is as slow as interpreting the code. So to effectively use JIT, the virtual machine needs to find code that runs for a substantial amount of time, hotspots, and use JIT to optimize execution of the code.

JIT compiles intermediate code into machine code for a native run while the intermediate code is executing, which may decrease an application's performance. On the other hand, AOT compilation eliminates the need for this step by performing the compilation before execution rather than during execution. AOT in most cases produces machine optimized code, just like a 'standard' native compiler. The difference is that AOT transforms the bytecode of an existing virtual machine into machine code. AOT compilers can perform complex and advanced code optimizations, which in most cases of JIT compilation will be considered much too costly.

4. Hypothesis

Based on the papers we have reviewed, we believe that using auto-vectorization from a modern compiler, Java programs can run faster on a virtual machine. We will compare the performance of computation-intensive loops with and without auto-vectorization, and verify that auto-vectorization can indeed improve Java performance. Because our method is based on preprocessing of Java classes, very little runtime overhead is expected. We will verify this is the case. Further, our method does not constrain optimized Java bytecode to a specific machine type, and it can run on different machine architectures. If time permits, we will verify this aspect, by compiling the vectorized IR to different machine codes.

5. Methodology

Our approach to the vectorization problem for Java virtual machine is to use LLVM to do the heavy-lifting -- that is, optimizing the code by vectorization. Because vectorization has a complexity of $O(n^4)$ and is not suitable for JIT [8], so we use AOT (ahead-of-time) compilation for Java bytecode. Specifically, we implement the following optimization process in Figure 5.1.

The system takes a Java source or class bytecode as input, converts the bytecode into scalar code in LLVM language (using a Java front-end) and runs LLVM optimizer (opt) to generate vectorized code in LLVM language.

Then, the center component, Java bytecode optimizer does the following processing:

a. Parses the vectorized code in LLVM, and generates a list of functions that are vectorized. We use the parser-generator ANTLR to generate an LLVM parser.
b. Reads in the original Java class file, changes the vectorized functions to a native function, and inserts the dynamic library name for the native functions. We use the ASM tool to parse and modify Java class files.

At last, we run LLVM compilers to generate the system-dependent dynamic library for the Java class. The modified Java class and dynamic library can be loaded to JVM and execute.

To achieve optimal performance, we optimizes the code at three possible optimization points (OPT1 - 3 as shown in Figure). We will try different combinations of these optimization points to achieve best performance.

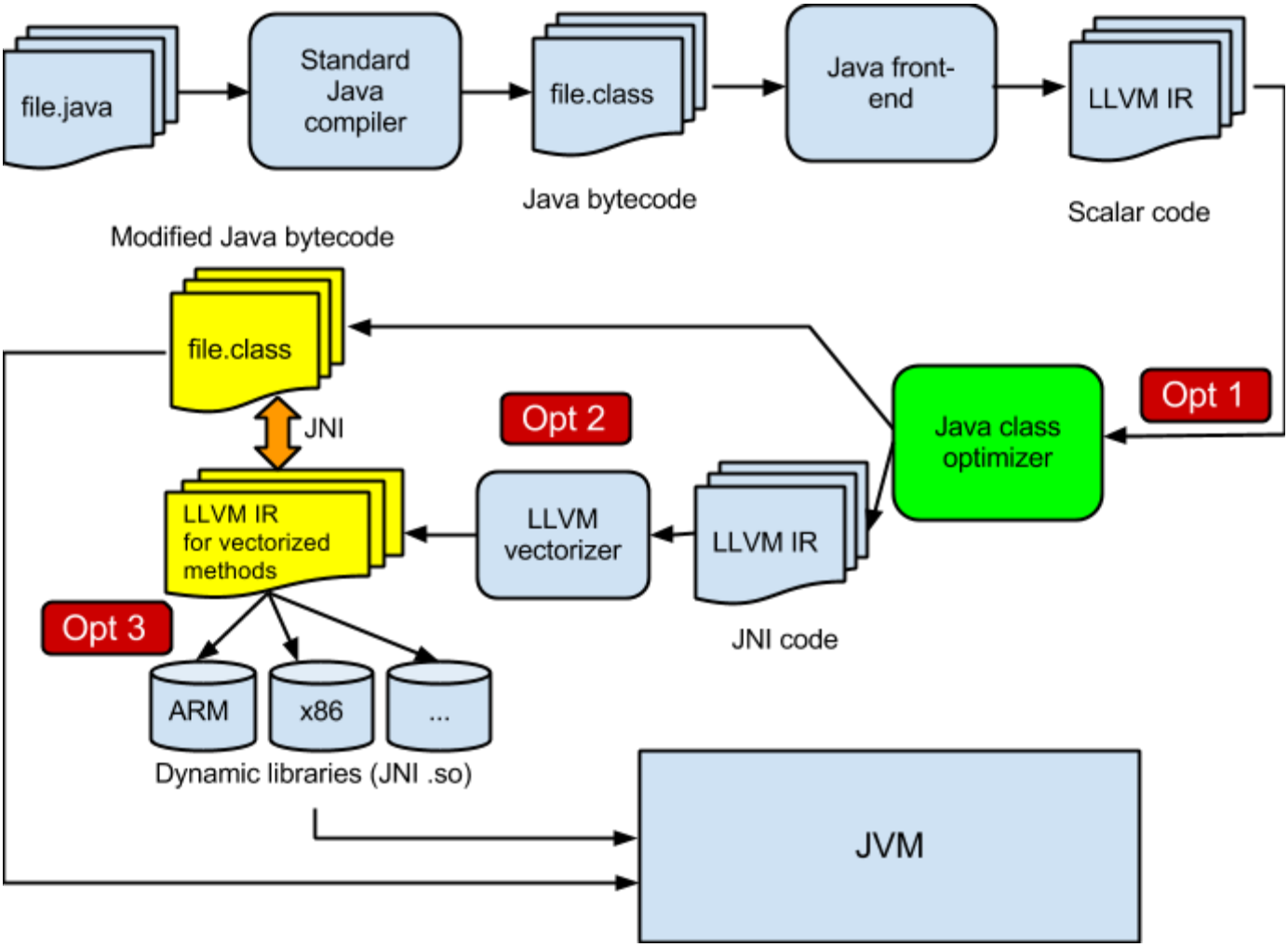


Figure 5.1: Auto-vectorization flows

5.1 Java bytecode to native vectorized code

5.1.1 Java bytecode to LLVM IR

Java bytecode does not represent vector operations. The first step into auto-vectorization is to identify an intermediate representation (IR) which supports vector semantics. Among the choices, we have GCC IR, LLVM IR and custom vectorized bytecode [8]. In this research, we choose LLVM IR, because LLVM has long supported an extensive set of vector data types and operations, has support for generating vector instructions in several backends, and contains generic lowering and scalarization code to handle code generation for operations without native support.

Some examples of LLVM IR vector operations are shown in Figure 5.2. LLVM language is based on SSA (single static assignment), where each variable is assigned only once. This makes it easy for modeling data dependency and optimization. The code shown in Figure 5.2 does the following:

1. Vector multiplication and addition
    a. Load the 2-element vector value at %addr into local variable %mul8
    b. Multiple %mul8 with %add10 and store into %mul11
    c. Add %mul11 and %add7 and store to %add12
2. Vector manipulations
    a. Cast the pointer %addr2 from double* type to <2 x double>* type (2-element vector)
    b. Insert double value %A1 into the vector at index 0
    c. Insert double value %B2 into the vector at index 1
    d. The 'shufflevector' instruction constructs a permutation of elements from two input vectors, returning a vector with the same element type as the input and length that is the same as the shuffle mask.
    e. The 'extractvalue' instruction extracts the value of a member field from an *aggregate* value.

```
%mul8 = load <2 x double>* %addr, align 8
%mul11 = fmul <2 x double> %mul8, %add10
%add12 = fadd <2 x double> %add7, %mul11
%vaddr = bitcast double* %addr2 to <2 x double>*
store <2 x double> %add12, <2 x double>* %vaddr, align 8
%Y2 = insertelement <2 x double> undef, double %A1, i32 0
%Y1 = insertelement <2 x double> %Y2, double %B2, i32 1
%Z1 = shufflevector <2 x double> %Y1, <2 x double> undef, <2 x i32> <i32 1, i32 1>
%q = extractelement <2 x double> %Z1, i32 0
```

Figure 5.2: LLVM IR vector operation examples

Although the latest LLVM package (version 3.2) still doesn't have a fully functional Java front-end, there are some possible ways to generate LLVM IR from Java bytecode.

1. Use a GCC plugin (DragonEgg) to intercept assembly code generated by GCC's Java compiler (GCJ) and emit LLVM IR.
2. Still use the Java front-end of LLVM to emit LLVM IR. However, it does not support full Java features.
3. Use the Java virtual machine VMKit based on LLVM.

In this research, we will method 1 (GCJ + DragonEgg plugin) to generate LLVM IR from Java bytecode. The pre-vectorizing IR contains no vector operations.

5.1.2 Vectorization by LLVM

LLVM has two vectorizers: the Loop Vectorizer, which operates on Loops [5], and the Basic Block Vectorizer [9], which optimizes straight-line code.

Here is some examples that show the capability of the LLVM Loop Vectorizer. For a complete discussion of LLVM vectorizer algorithms, see [5].

Loop without trip count

The Loop Vectorizer supports loops with an unknown trip count. 1) The iteration start and finish points are unknown; 2) Loops that do not start at zero; 3) Array size 'n' may not be a multiple of the vector width and the vectorizer has to execute the last few iterations as scalar code. The disadvantage is that keeping a scalar copy of the loop increases the code size.

Runtime check of pointers

Two pointers A and B may point to consecutive addresses, then it is illegal to vectorize the code because some elements of A will be written before they are read from array B. Some programmers use the 'restrict' keyword to notify the compiler that the pointers are disjointed, but the Loop Vectorizer has no way of knowing that the pointers A and B are unique. The Loop Vectorizer handles this loop by placing code that checks, at runtime, if the arrays A and B point to disjointed memory locations. If arrays A and B overlap, then the scalar version of the loop is executed.

Reductions

In this example the sum variable is used by consecutive iterations of the loop. Normally, this would prevent vectorization, but the vectorizer can detect that 'sum' is a reduction variable. The variable 'sum' becomes a vector of integers, and at the end of the loop the elements of the array are added together to create the correct result. LLVM supports a number of different reduction operations, such as addition, multiplication, XOR, AND and OR.

```
int foo(int *A, int *B, int n) {
  unsigned sum = 0;
  for (int i = 0; i < n; ++i)
    sum += A[i] + 5;
  return sum;
}
```

Straight-line code optimization

The goal of SLP vectorization (a.k.a. superword-level parallelism) is to combine similar independent instructions within simple control-flow regions into vector instructions. Memory accesses, arithmetic operations, comparison operations and some math functions can all be vectorized using this technique(subject to the capabilities of the target architecture). For example, the following function performs very similar operations on its inputs (a1, b1) and (a2, b2). The basic-block vectorizer may combine these into vector operations.

```
void foo(int a1, int a2, int b1, int b2, int *A) {
  A[0] = a1*(a1 + b1)/b1 + 50*b1/a1;
  A[1] = a2*(a2 + b2)/b2 + 50*b2/a2;
}
```

5.1.3 Conversion of Java bytecode to vectorized machine code

We will write a Java bytecode optimizer to change Java methods that can be vectorized. Since Java bytecode does not support vectorization, the vectorized code will be placed in a native method (C/C++ or ASM), invoked by the original Java method via JNI (Java-native-interface).

The following example demonstrates how JNI works.

```
package com.jniexamples;

public class Hello {
  public native void sayHi(String who, int times); // (1)

  static { System.loadLibrary("HelloImpl"); } // (2)

  public static void main (String[] args) {
    Hello hello = new Hello();
    hello.sayHi(args[0], Integer.parseInt(args[1])); // (3)
  }
}
```

Figure 5.3: An example of JNI

(1) The method sayHi can be implemented in C/C++ in separate file(s), which will be compiled into a library.
(2) The library filename will be called libHelloImpl.so (on Linux/Unix), HelloImpl.dll (on Windows) and libHelloImpl.jnilib (Mac OS/X), but when loaded in Java, the library has to be loaded as HelloImpl.
(3) The method sayHi is invoked as a Java method.

The tool *javah* can generate the header file for the native method. After compiling the source code *Hello.java*, running "*javah -jni com.jniexamples.Hello*" in the class directory will generate the header file *com_jniexamples_Hello.h*. We can then implement the *Java_com_jniexamples_Hello_sayHi* function.

The header file *com_jniexamples_Hello.h* looks like:

```
...
#include <jni.h>
...
JNIEXPORT void JNICALL Java_com_jniexamples_Hello_sayHi
  (JNIEnv *, jobject, jstring, jint);
...
```

The file Hello.c looks like:

```
#include <stdio.h>
#include "com_jniexamples_Hello.h"

JNIEXPORT void JNICALL Java_com_jniexamples_Hello_sayHi(JNIEnv *env,
jobject obj, jstring who, jint times) {
  jint i;
  jboolean iscopy;
  const char *name;
  name = (*env)->GetStringUTFChars(env, who, &iscopy);
  for (i = 0; i < times; i++) {
    printf("Hello %s\n", name);
  }
}
```

The compilation of Hello.c is system-dependent. This will create libHelloImpl.so, HelloImpl.dll, libHelloImpl.jnilib (depending on the O/S). Set LD_LIBRARY_PATH to the path where the compiled library is stored, and the Java application can be run.

To vectorize a Java class, we will do the following

1. Generate the vectorized code (LLVM IR) as in Figure 5.1
2. Scan for vectorized methods M
3. Use our Java class optimizer to generate a pair of files:
    a. Modified file.class. This file contains the original Java bytecode, but optimll
    b. ized methods M are changed to native methods, which are dynamically linked to a .so file by JNI.
    c. An LLVM IR file which can compile to the JNI library. It is generated by the steps below:
        i. Create a temporary Java file which contains the signatures of all native methods M, each with the keyword 'native'.
        ii. Run the Java tool *javah* to generate a C header file for the Java class
        iii. Generate a C stub file to implement each of these native functions. Compile the C stub file to LLVM IR, and modify the LLVM IR to insert the auto-vectorized code.
4. To deploy the optimized Java class for a specific machine type, we compile the LLVM IR to assembly code for that machine, and link into a dynamic library.
5. Place the dynamic libraries in a system-dependent directory, add the path to LD_LIBRARY_PATH
6. Execute the Java application by JVM.

5.2 Input data

We create a set of benchmark Java programs which can be vectorized. If there is time, we may use the standard LINPACK library. LINPACK is a software library for performing numerical linear algebra on digital computers.

5.3 Output data

5.3.1 System-independent vectorized code

The vectorized code is generated using our Java class optimizer described above. It is in LLVM assembly language.

5.3.2 System-dependent vectorized code

For each machine type, the LLVM IR can be compiled to machine code which possibly supports vector operations. For example, x86 includes MMX, SSE2 and SSE3 instruction sets which support vector operations. ARM supports NEON instructions.

5.3.3 Profiling and benchmarking results

We run the profiler on the pre-optimized Java code and the optimized Java code. The optimized code may incur some dynamic linking overhead but general performance is expected to exceed that of the original code.

5.3.4 Language Used

- C/C++, Java, Make, bash script

5.3.5 Tools Used

- A standard JDK (javac compiler and javah JNI header generator, JVM)
- LLVM/GCC -- Compilers with vectorization features. DragonEgg LLVM IR generator.
- ASM -- An all purpose Java bytecode manipulation and analysis framework.
- ANTLR -- A parser generator that uses LL(*) parsing. We use ANTLR to generate an LLVM assembly parser.
- Profiler: Oracle hpof (old output format)

5.4 Testing Against Hypothesis

For each Java class, we compare performance of original Java bytecode, converted native code, and vectorized code. We expect to see performance improvement in the native code and vectorized code.

The tests we perform can be summarized by the following table

1. Java bytecode without JIT
2. Java bytecode with JIT
3. Native code with vectorization

6. Implementation

6.1 Code

We used Java to implement the Java class optimizer.

6.2 Design of the Java bytecode optimizer

6.2.1 LLVM parser

The LLVM parser is used to process LLVM code, and outputs a modified LLVM code which implements the required JNI interface. The parser is generated from an ANTLR grammar file, which defines LLVM syntax. The parsed LLVM functions and variables are stored in parser data structures for easy access.

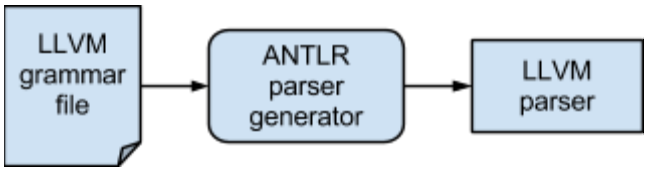


Figure 6.1 Parser generator

6.2.2 Automatic function converter

The function converter converts a Java function in LLVM language to a JNI function. The following example shows the difference.

The testSum function from Java bytecode has the following signature:

```
define i32 @_ZN10HelloWorld7testSumEJii(%struct.HelloWorld* %unnamed_arg, i32 %unnamed_arg2)
uwtable align 2 {
   ...
}
```

The testSum function in the JNI library has the following signature (see Appendix 1 for original C code and compiled LLVM IR):

```
define i32 @Java_HelloWorld_testSum(%struct.JNINativeInterface_** %env, i8* %thisobj, i32
%arg0) unnamed_addr nounwind uwtable ssp {
  ...
}
```

The function converter modifies the LLVM code from the Java form to JNI form. Though it is possible to do so without a full functioning parser, we choose to use a LLVM parser for reliability and flexibility.

After parsing, the following steps can be done to convert the functions:

1. Modify the function signature

   a. Insert `%struct.JNINativeInterface_** %env` as the first argument (% is the prefix for local variables, which can either be a type or a value).
   b. Change *this* pointer from type `%struct.HelloWorld*` to `i8*`, and rename it from `%unnamed_arg` to `%thisobj`. The variable mapping is handled by class VariableMap.
2. Scans the argument list and prepares variable mapping (TypeA idA → TypeB idB), which includes three aspects
   a. Type mapping (TypeA → TypeB)
   b. Name mapping (idA → idB)
   c. Operation mapping (e.g., array operations, idA.length → env->getIntArrayLength(idA))
3. Scan the definition of the function, and performs all necessary variable mapping, including access Array types and Object types. For example, there are 4 Java array operations that need special handling when converted to native code (See 6.2.3 for details).
   a. Get array base
   b. Get array length
   c. Read/write array element
   d. Returning modified array to Java from native code

   Mapping templates can be extended. If the function uses some data type that is not yet supported (for example, floating point array), the error is reported and the developer can extend the optimizer and re-run it. After implementing the complete JNI specification, all Java bytecode can be converted to native code.
4. Modify Java function invocation (not supported in this project).

6.2.3 Variable mapper

A variable map m is a quadruple: $m(T, id, T', id')$, where T and T' are source and destination types respectively, and id and id' are source and destination variable names respectively.

Some extra data are provided for these maps.

1. Semantic operation mapper. $f_{T,T'}$ is a map function to convert an operation on type T to a semantically equivalent operation of type T' (see examples in 6.2.3)
2. Constructors. $a_{T'}$ is an initialization function to construct the mapped variable id' of type T'.
3. Destructors. $z_{T'}$ is a cleanup function to free allocation of the mapped variable id' of type T'.

The variable mapper is referenced by the function converter to perform necessary code changes.

6.2.3 Special code recognition and code generator

To automatically convert Java array access code to native code, we need to recognize sequences of LLVM assembly instruction that perform Java array operations, and generate semantically equivalent code in the JNI environment.

These operations are recognized by predefined instruction sequence patterns (with wildcards), and converted to corresponding operations in the JNI environment. The patterns and code generation support optimization levels -O0 to -O3, and 32-bit and 64-bit machine word sizes.

a. **Array base**. Instructions accessing element 2 of Java Object (the superclass of arrays) are converted

to invocation of a native function GetIntArrayElements().

b. **Array length**. Instructions accessing element 1 of Java Object (the superclass of arrays) are converted to invocation of a native function GetIntArrayLength().
c. **Read/write array element**. These instructions are not changed but the new array base variable is used.
d. **Returning modified array** to Java from native code. New code is automatically inserted before function return (supporting multiple return points) to release arrays. An array is released with function ReleaseIntArrayElements(). After it is released, the modified array is returned to Java environment.

6.2.3 Code generation

The converted LLVM code is emitted with the following structure:

```
jni_preamble          // includes all type definitions prior to function definition
function_definition*  // functions in the JNI library
```

The jni_preamble is a piece of code we obtain by compiling a generic JNI file. The function_definition defines a function following the JNI interface, generated by the above automatic function converter.

7. Data Analysis and Discussion

7. 1 Output Generation

7.1.1 Basic timing for optimized functions

After Java functions are converted to native code and invoked via JNI, they are hypothesized to run faster. We do the following experiment to verify this hypothesis. First, we create 4 basic test functions:

1. Array sum (testArraySum)
2. SLP vectorization test (testSLPVectorize)
3. Selection sort for array (testSort)
4. Insertion sort for array (testInsertionSort)

The code for SLP vectorization test is below:

```
void testSLPVectorize(int a1, int a2, int b1, int b2, int A[]) {
   int i;
   for (i = 0; i < A.length - 1; i += 2) {
     A[i] = a1*(a1 + b1)/b1 + 50*b1/a1;
     A[i + 1] = a2*(a2 + b2)/b2 + 50*b2/a2;
   }
}
```

We used bash scripts to generate code for the following cases:

1. Standard Java bytecode (Java compiler)
2. JNI and vectorized code (our Java optimizer + LLVM)
3. JNI and non-vectorized code (ditto)

Then, we created test code to time the functions. For those that take arrays as arguments, we try different array sizes until the execution time is in the order of 100ms. Then, we used a timer with milliseconds (In Java test code) to time the functions.

7.1.2 Advanced testing, stress testing, profiling

To be sure that we are on the right track, we did very intense testing sessions to prove that our results are reliable and valid.

For that reason, we created a test suite which executes 16 different major tests in different configurations and repeated the test cases with a factor of 40 or 50. As a result, we executed a sequence of 800 test cases in 10.5 hours and 640 test cases for the last iteration in 5 hours at Monday.

Tests

1. Standard Java - with JIT
2. Standard Java - without JIT
3. Standard Java - with JIT and activated profiling
4. Standard Java - without JIT and activated profiling
5. JNI optimized version - with JIT
6. JNI optimized version - without JIT
7. JNI optimized version - with JIT and activated profiling
8. JNI optimized version - without JIT and activate profiling
9. JNI-Fast optimized version - with JIT
10. JNI-Fast optimized version - without JIT
11. JNI-Fast optimized version - with JIT and activated profiling
12. JNI-Fast optimized version - without JIT and activated profiling
13. Vectorized version - with JIT
14. Vectorized version - without JIT
15. Vectorized version - with JIT and activated profiling
16. Vectorized version - without JIT and activated profiling

Test computer setup

The test computer was a 64 bit computer with the following data:

View basic information about your computer

Windows edition

Windows 7 Enterprise

Copyright © 2009 Microsoft Corporation. All rights reserved.

Service Pack 1

System

Rating:	3.4 Windows Experience Index
Processor:	Intel(R) Core(TM)2 Duo CPU L9600 @ 2.13GHz 2.13 GHz
Installed memory (RAM):	4.00 GB (3.90 GB usable)
System type:	64-bit Operating System
Pen and Touch:	No Pen or Touch Input is available for this Display

On top of Windows 7, we used a Virtual Machine VirtualBox version 4.2.12 from Oracle which executes an Ubuntu 12.04. Linux.

To be sure that nothing disturbs the test computer, everything was deactivated: no Internet, no virus software, no screen saver or anything that could distort the test results. This was also a learning process and with every test run it improved. A sign that the computer and the environment produces distortion is when thin peaks in the test result table/graphs were viewable. The solution for the first couple of test runs was simply to delete the peaks. In the later test runs, there were no peaks because of the intense deactivation of everything that is not needed for the test.

Profiling

Profiling was a bigger issue as expected. We tested some profiling tools like JProfiler from ej-technologies and also some tools like JMeter from Apache which is more a stress testing tools.
The test setup makes a standard profiling nearly impossible because the basic idea about our project is that we outsource a Java function into a native method which is a C function in a dynamic library which is located on the computer. As a result, if our test and implementation is successfully implemented and executed, the optimized/vectorized functions simply disappears from the profiling report. This is not an error because we outsourced it and is not anymore visible and measurable. A very good outcome from this situation is that we always know that our optimized/vectorized functions are really working and that these functions has to disappear from the profiling output.

We had the following tests:
- testArraySum
- testSLPVectorize
- testSort

- testInsertionSort

All tests are optimized and vectorized except “testInsertionSort” in the last execution. This is an example of an error which could be detected through the stress tests. After we saw that this function was not converted into a native version we had investigated and corrected this error.

We included in our 16 major test cases in our test suite 8 profiling tests . The results are also included in the attachments.

For our profiler, we used hprof and especially the old profiling format. This format is very good for additional processing steps because it delivers the data as a space delimited list which could be easily imported in Excel and creates output like this:

	A	B	C	D
1	count	callee	caller	time
2	3200000	java.util.Random.nextInt()I	VectorTestStressVersion.executeTests(I)V	778236
3	160	java.lang.StringBuilder.append(Ljava/lang/String;)Ljava/lang/StringBuilder;	VectorTestStressVersion.executeTests(I)V	49
4	160	java.lang.StringBuilder.append(J)Ljava/lang/StringBuilder;	VectorTestStressVersion.executeTests(I)V	42
5	80	java.util.Random.<init>()V	VectorTestStressVersion.executeTests(I)V	43
6	40	VectorTestStressVersion.testArraySum([I)I	VectorTestStressVersion.executeTests(I)V	92425
7	40	VectorTestStressVersion.testSLPVectorize(IIII[I[I)V	VectorTestStressVersion.executeTests(I)V	85370
8	40	VectorTestStressVersion.testSort([I)V	VectorTestStressVersion.executeTests(I)V	84454
9	40	VectorTestStressVersion.testInsertionSort([I)V	VectorTestStressVersion.executeTests(I)V	34639
10	40	java.lang.StringBuilder.<init>()V	VectorTestStressVersion.executeTests(I)V	3
11	40	java.lang.StringBuilder.append(I)Ljava/lang/StringBuilder;	VectorTestStressVersion.executeTests(I)V	12
12	40	java.lang.StringBuilder.toString()Ljava/lang/String;	VectorTestStressVersion.executeTests(I)V	13
13	40	java.io.PrintStream.println(Ljava/lang/String;)V	VectorTestStressVersion.executeTests(I)V	482
14	40	VectorTestStressVersion.executeTests(I)V	VectorTestStressVersion.main([Ljava/lang/String;)V	1192869
15	4	java.lang.ClassLoader.checkPackageAccess(Ljava/lang/Class;Ljava/security/ProtectionDoma	VectorTestStressVersion.executeTests(I)V	2
16	3	java.lang.ClassLoader.loadClass(Ljava/lang/String;)Ljava/lang/Class;	VectorTestStressVersion.executeTests(I)V	8
17	1	VectorTestStressVersion.<init>()V	VectorTestStressVersion.main([Ljava/lang/String;)V	0
18	1	java.util.Random.<clinit>()V	VectorTestStressVersion.executeTests(I)V	5
19	1	VectorTestStressVersion.main([Ljava/lang/String;)V	<unknown	1192881

In this figure above (which is taken from the standard Java program without any optimization or vectorization) our four test cases are viewable.

	count	callee	caller	time
2	3200000	java.util.Random.nextInt()I	VectorTestStressVersion.executeTests(I)V	763516
3	160	java.lang.StringBuilder.append(Ljava/lang/String;)Ljava/lang/StringBuilder;	VectorTestStressVersion.executeTests(I)V	44
4	160	java.lang.StringBuilder.append(J)Ljava/lang/StringBuilder;	VectorTestStressVersion.executeTests(I)V	52
5	80	java.util.Random.<init>()V	VectorTestStressVersion.executeTests(I)V	46
6	40	VectorTestStressVersion.testInsertionSort([I)V	VectorTestStressVersion.executeTests(I)V	33958
7	40	java.lang.StringBuilder.<init>()V	VectorTestStressVersion.executeTests(I)V	5
8	40	java.lang.StringBuilder.append(I)Ljava/lang/StringBuilder;	VectorTestStressVersion.executeTests(I)V	5
9	40	java.lang.StringBuilder.toString()Ljava/lang/String;	VectorTestStressVersion.executeTests(I)V	7
10	40	java.io.PrintStream.println(Ljava/lang/String;)V	VectorTestStressVersion.executeTests(I)V	470
11	40	VectorTestStressVersion.executeTests(I)V	VectorTestStressVersion.main([Ljava/lang/String;)V	1027016
12	3	java.lang.ClassLoader.findNative(Ljava/lang/ClassLoader;Ljava/lang/String;)J	VectorTestStressVersion.executeTests(I)V	0
13	3	java.lang.ClassLoader.loadClass(Ljava/lang/String;)Ljava/lang/Class;	VectorTestStressVersion.executeTests(I)V	6
14	3	java.lang.ClassLoader.checkPackageAccess(Ljava/lang/Class;Ljava/security/ProtectionDoma	VectorTestStressVersion.executeTests(I)V	1
15	1	java.lang.ClassLoader.checkPackageAccess(Ljava/lang/Class;Ljava/security/ProtectionDoma	VectorTestStressVersion.<clinit>()V	0
16	1	java.lang.System.loadLibrary(Ljava/lang/String;)V	VectorTestStressVersion.<clinit>()V	48
17	1	VectorTestStressVersion.<clinit>()V	<unknown	48
18	1	VectorTestStressVersion.<init>()V	VectorTestStressVersion.main([Ljava/lang/String;)V	0
19	1	java.util.Random.<clinit>()V	VectorTestStressVersion.executeTests(I)V	5
20	1	VectorTestStressVersion.main([Ljava/lang/String;)V	<unknown	1027028

In this example after the first optimization you can see that the profiler can only analyze the “testInsertionSort” because the other ones were outsourced.

To activate the profiler it is only needed to add this parameter to the java interpreter call "-agentlib:hprof=cpu=old"

Measuring time

As a consequence of the profiler programs to profile native code we choose the classical way and count milliseconds before and after a test which looks like this code fragment:

```
// Test 2 - SLP
long test3Start=System.currentTimeMillis();
this.testSLPVectorize(10, 20, 30, 40, ia, ib);
long test3End=System.currentTimeMillis();
long test3dur=test3End-test3Start;
```

This segment calls testSLPVectorize(...) and simply counts the milliseconds. Every test was collected in a major test suite: VectorTestStressVersion.java

This test suite executes our four test cases:

1. testArraySum
2. testSLPVectorize
3. testSort
4. testInsertionSort

As output we generated a semicolon separated output which looks like this:

```
0;2154;2160;2608;855
1;2160;2133;2440;837
2;2116;2082;2452;860
3;2163;2093;2387;844
4;2206;2138;2467;859
5;2153;2048;2374;852
6;2141;2080;2406;860
7;2152;2071;2388;843
```

First column is the test iteration number, the next columns represents the test cases. One call of the test suite repeated 40 or 50 times. This setup makes it easy to execute our 16 major test cases with the following statements:

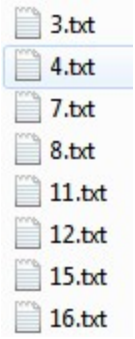
1. java VectorTestStressVersion
2. java -Djava.compiler=NONE VectorTestStressVersion
3. java -agentlib:hprof=cpu=old VectorTestStressVersion
4. java -agentlib:hprof=cpu=old -Djava.compiler=NONE VectorTestStressVersion
5. java -Djava.library.path=. VectorTestStressVersion
6. java -Djava.library.path=. -Djava.compiler=NONE VectorTestStressVersion
7. java -Djava.library.path=. -agentlib:hprof=cpu=old VectorTestStressVersion
8. java -Djava.library.path=. -agentlib:hprof=cpu=old -Djava.compiler=NONE VectorTestStressVersion
9. java -Djava.library.path=. VectorTestStressVersion
10. java -Djava.library.path=. -Djava.compiler=NONE VectorTestStressVersion

11. java -Djava.library.path=. -agentlib:hprof=cpu=old VectorTestStressVersion
12. java -Djava.library.path=. -agentlib:hprof=cpu=old -Djava.compiler=NONE VectorTestStressVersion
13. java -Djava.library.path=. VectorTestStressVersion
14. java -Djava.library.path=. -Djava.compiler=NONE VectorTestStressVersion
15. java -Djava.library.path=. -agentlib:hprof=cpu=old VectorTestStressVersion
16. java -Djava.library.path=. -agentlib:hprof=cpu=old -Djava.compiler=NONE VectorTestStressVersion

We created a very simply shell script test-sequence.sh to execute the 16 major test:

```
echo Standard Java Test

cd out-java
echo -1- java VectorTestStressVersion
java VectorTestStressVersion

echo -2- java -Djava.compiler=NONE VectorTestStressVersion
java -Djava.compiler=NONE VectorTestStressVersion

echo -3- java -agentlib:hprof=cpu=old VectorTestStressVersion
java -agentlib:hprof=cpu=old,file=3.txt VectorTestStressVersion

echo -4- java -agentlib:hprof=cpu=old -Djava.compiler=NONE VectorTestStressVersion
java -agentlib:hprof=cpu=old,file=4.txt -Djava.compiler=NONE VectorTestStressVersion

… 3 additionally blocks to cover all 16 major test cases.
```

The complete test execution creates a test result file: TestResult_Test0610.txt and additionally 8 profiling files:



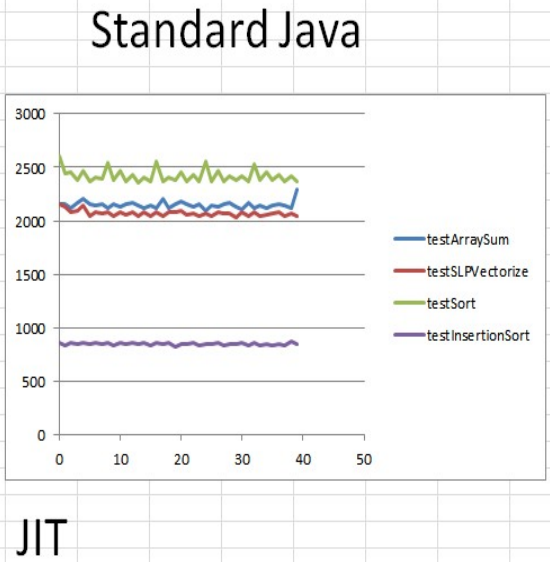
All these files were transferred to the Windows operating system and carried and collected in a major result Excel file: TestResulst061013-1.xlsx

Results of the Advanced testing, stress testing, profiling sessions

The setup produces a lot of single test data the main approach was after the collection to create one Excel with all information. This process needs a huge amount of time because it was not automatable.

-1- java VectorTestStressVersion

Test No.	testArraySum	testSLPVectorize	testSort	testInsertionSort
0	2154	2160	2608	855
1	2160	2133	2440	837
2	2116	2082	2452	860
3	2163	2093	2387	844
4	2206	2138	2467	859
5	2153	2048	2374	852
6	2141	2080	2406	860
7	2152	2071	2388	843
8	2113	2082	2538	855
9	2162	2050	2387	836
10	2134	2087	2464	860
11	2158	2062	2364	851
12	2170	2080	2433	858
13	2144	2045	2360	844
14	2118	2082	2409	857
15	2145	2041	2363	839
16	2116	2080	2552	863
17	2209	2042	2371	847
18	2114	2077	2402	866
19	2155	2082	2380	827
20	2186	2089	2460	854
21	2153	2053	2363	847
22	2126	2075	2428	863
23	2159	2050	2370	839



Average	
2,149	testArraySum
2,071	testSLPVectorize
2,421	testSort
850	testInsertionSort

There is one page for every one of the 16 major test cases. On the left side are the test samples and their results listed which is used in the graph in the middle. There is also an average on the left side and also some information which test case it is. In this example, it is a plain “Standard Java” execution with JIT-compiler.

In addition to the test overview, there are also 8 profiling test result pages in the Excel file:

count	callee	caller	time
3200000	java.util.Random.nextInt()I	VectorTestStressVersion.executeTests(I)V	778236
160	java.lang.StringBuilder.append(Ljava/lang/String;)Ljava/lang/StringBuilder;	VectorTestStressVersion.executeTests(I)V	49
160	java.lang.StringBuilder.append(J)Ljava/lang/StringBuilder;	VectorTestStressVersion.executeTests(I)V	42
80	java.util.Random.<init>()V	VectorTestStressVersion.executeTests(I)V	43
40	VectorTestStressVersion.testArraySum([I)I	VectorTestStressVersion.executeTests(I)V	92425
40	VectorTestStressVersion.testSLPVectorize(IIII[I[I)V	VectorTestStressVersion.executeTests(I)V	85370
40	VectorTestStressVersion.testSort([I)V	VectorTestStressVersion.executeTests(I)V	84454
40	VectorTestStressVersion.testInsertionSort([I)V	VectorTestStressVersion.executeTests(I)V	34639
40	java.lang.StringBuilder.<init>()V	VectorTestStressVersion.executeTests(I)V	3
40	java.lang.StringBuilder.append(I)Ljava/lang/StringBuilder;	VectorTestStressVersion.executeTests(I)V	12
40	java.lang.StringBuilder.toString()Ljava/lang/String;	VectorTestStressVersion.executeTests(I)V	13
40	java.io.PrintStream.println(Ljava/lang/String;)V	VectorTestStressVersion.executeTests(I)V	482
40	VectorTestStressVersion.executeTests(I)V	VectorTestStressVersion.main([Ljava/lang/String;)V	1192869
4	java.lang.ClassLoader.checkPackageAccess(Ljava/lang/Class;Ljava/security/ProtectionDoma	VectorTestStressVersion.executeTests(I)V	2
3	java.lang.ClassLoader.loadClass(Ljava/lang/String;)Ljava/lang/Class;	VectorTestStressVersion.executeTests(I)V	8
1	VectorTestStressVersion.<init>()V	VectorTestStressVersion.main([Ljava/lang/String;)V	0
1	java.util.Random.<clinit>()V	VectorTestStressVersion.executeTests(I)V	5
1	VectorTestStressVersion.main([Ljava/lang/String;)V	<unknown	1192881

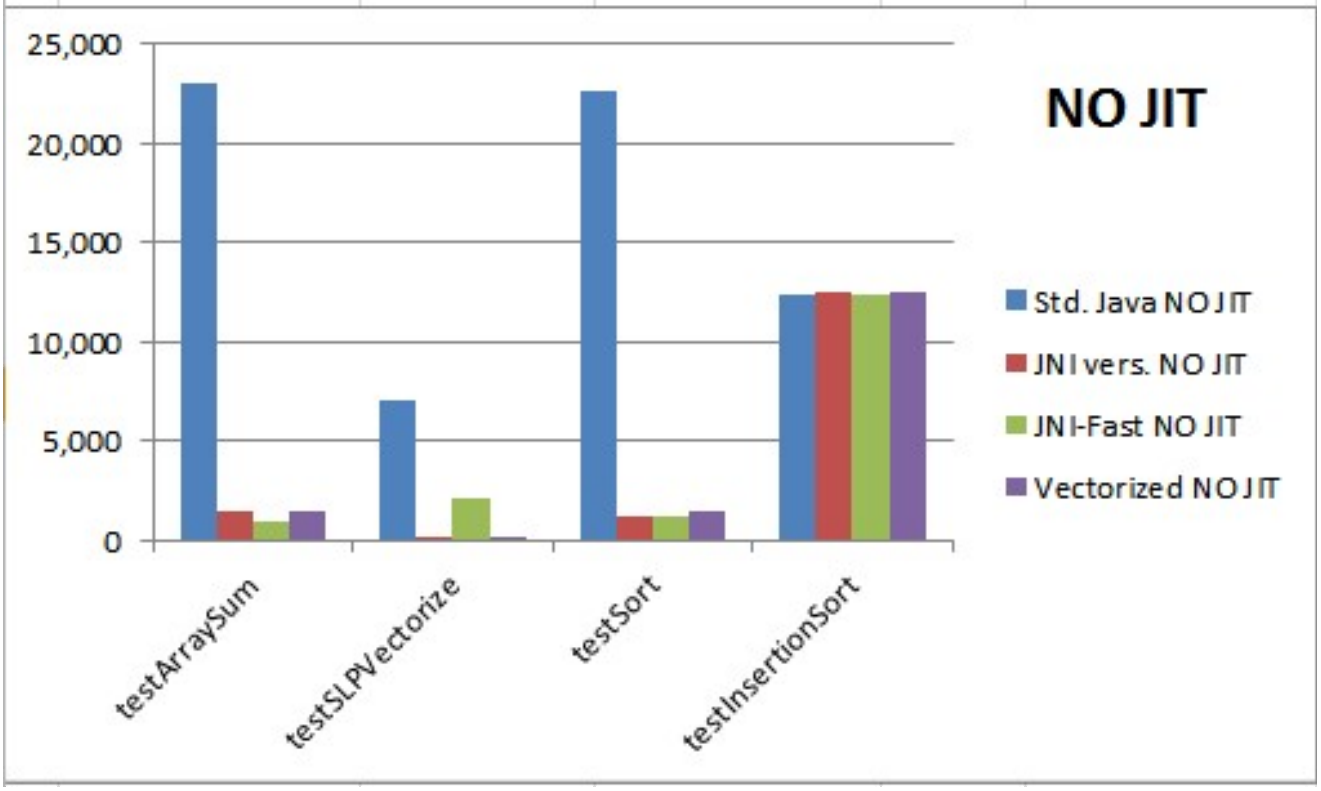
The profiling report in this example is reduced to the important items. In this example, you will see 18 from normally ca. 1,000 profiling lines, but we also included the complete profiling reports in our test data.

7.2 Output Analysis

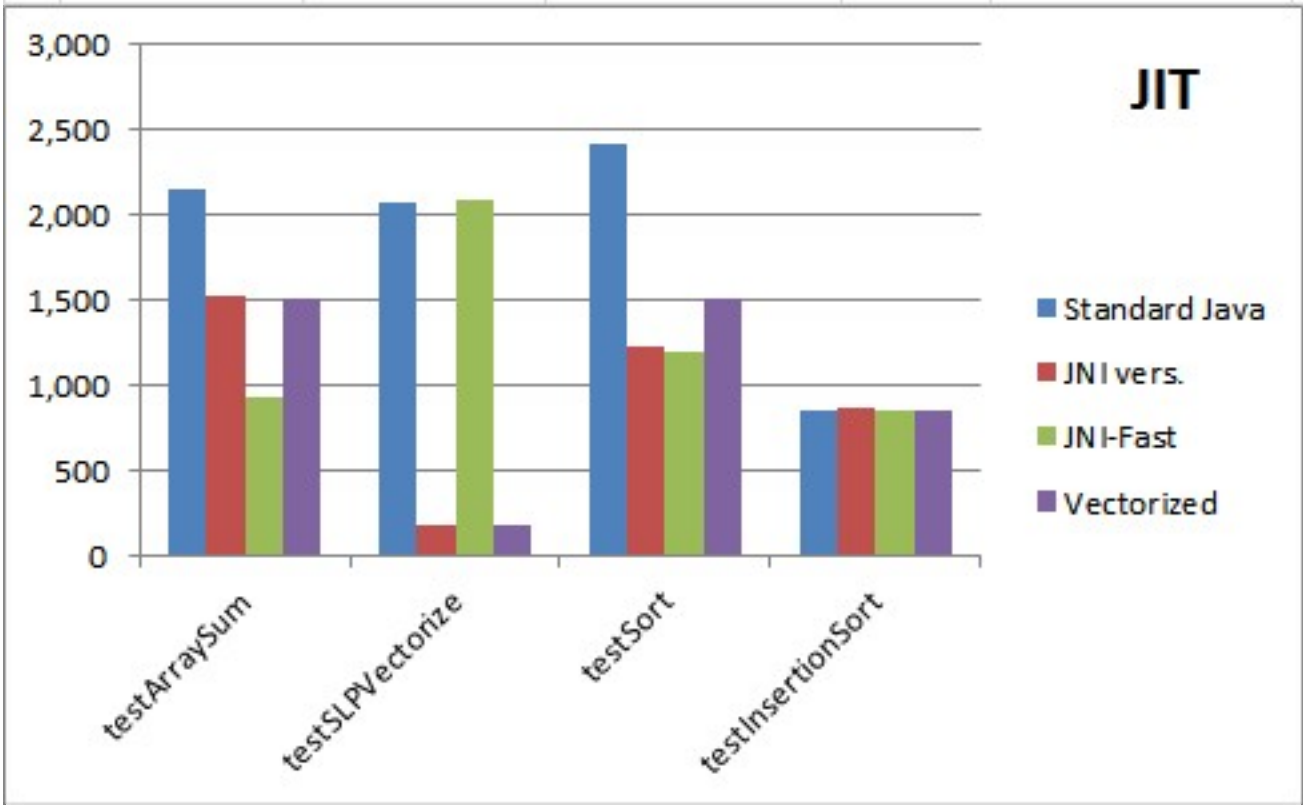
7.2.2 Basic timing for optimized functions

Stress Test Results

	Test Case	testArraySum	testSLPVectorize	testSort	testInsertionSort
2	Std. Java NO JIT	22,980	7,023	22,634	12,327
6	JNI vers. NO JIT	1,465	176	1,220	12,445
10	JNI-Fast NO JIT	905	2,116	1,209	12,415
14	Vectorized NO JIT	1,472	177	1,512	12,457



	Test Case	testArraySum	testSLPVectorize	testSort	testInsertionSort
1	Standard Java	2,149	2,071	2,421	850
5	JNI vers.	1,524	180	1,220	866
9	JNI-Fast	934	2,095	1,202	857
13	Vectorized	1,513	176	1,510	853



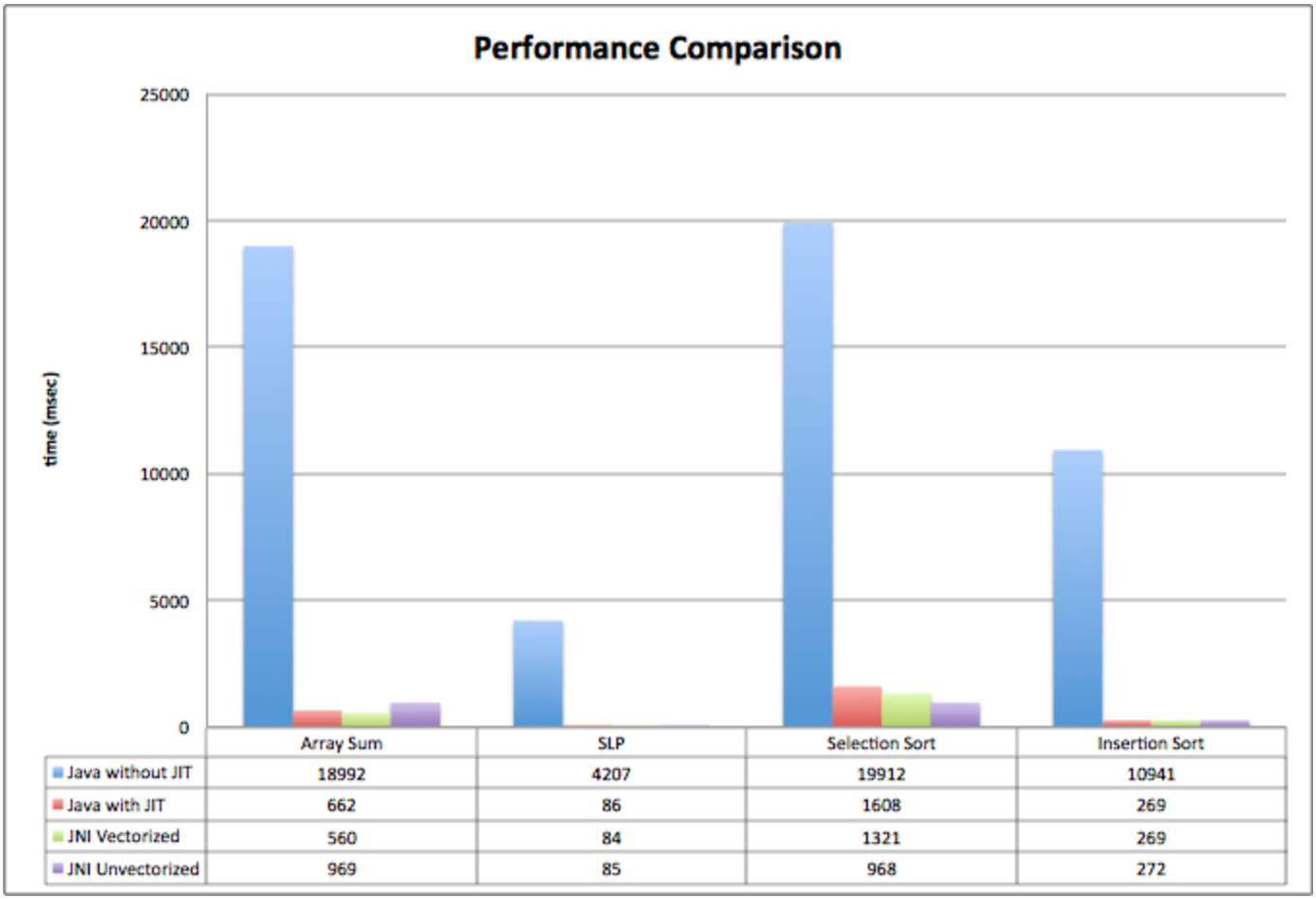


Figure 7.1 Performance comparison

Performance results were kept in the following log.txt:

```
==========================================================================================
Tue Jun 11 05:19:14 PDT 2013
==========================================================================================
Java with JIT:
662
86
1608
269

Java without JIT:
18992
4207
19912
10941

MACHINETYPE=x86_64 VECTORIZE=1 VECTOR_SIZE=4 OPT_BEFORE_JNI=0 OPT_AFTER_JNI=1
OPT_COMPILE_NATIVE=1
Results:
560
84
1321
269
```

```
MACHINETYPE=x86_64 VECTORIZE=0 VECTOR_SIZE=0 OPT_BEFORE_JNI=0 OPT_AFTER_JNI=1
OPT_COMPILE_NATIVE=1
Results:
969
85
968
272
```

7.1.4 Vector size determination

When using LLVM 'opt' tool to vectorized code, the -force-vector-width parameter needs to be specified. It controls the vectorization factor (VF), that is, how many elements are processed with one vector operation. This parameter is sensitive to performance. To determine the best vector size, we ran optimizations with various possible vector sizes, and pick the one that gives the best result.

Based on the result, we see sometimes the code performance depends on vector size (Array Sum), which means the code has been successfully vectorized. We pick the reasonable vector size (VF = 4) based on the result. It achieved about 50% savings.

The parameter -force-vector-width=0 means 'automatically' determine vector size by the optimizer. However, we found automatically determined vector size performs worse than a properly specified value (vector size 0 vs. 4).

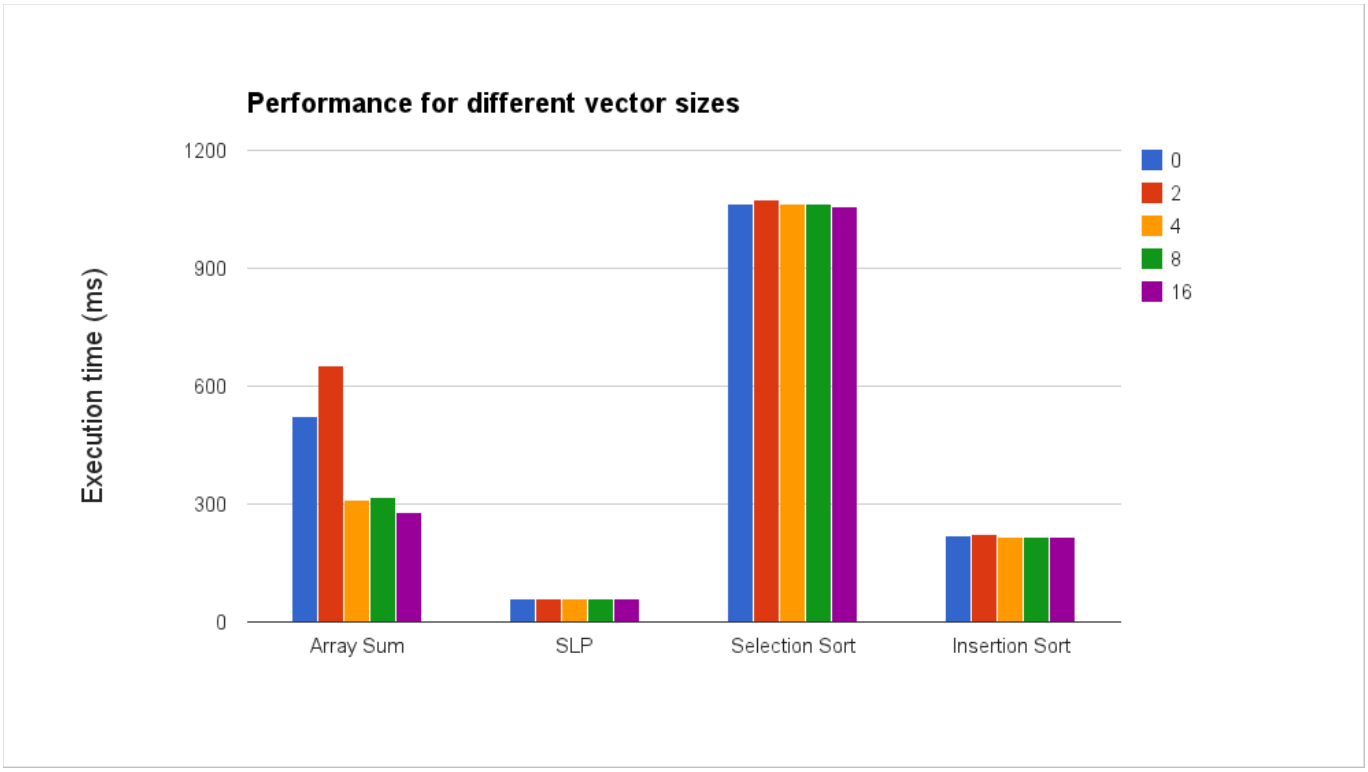


Figure 7.2 Performance as a function of vector size

7.1.5 Compilation time overheads

We know JIT may incur some compilation overhead. To find out the JIT overhead, we define it as follows

JIT overhead = (command line runtime) - (java main function runtime).

The data is shown below:

	JIT	JNI-VEC	JNI	JNI-Fast
Total runtime	2570	1965	1699	3237
Main() runtime	2511	1907	1642	3181
JIT overhead	59	48	57	56

It shows JIT compilation overhead is comparable to load time of dynamic libraries used by JNI versions.

In our solutions, we found our ATOC overhead is mainly for parsing the LLVM assembly code, which can be avoided if intermediate text files are avoided. Because such optimization is not done, the ATOC overhead is not measured yet.

7.3 Compare Output against Hypothesis

Our output has proved that we are able to improve the performance with vectorization. In this stage we have a tiny improvement, but with a little bit more investigation and more knowledge about how vectorization really works it is possible to improve the execution time remarkable.

7.4 Statistic Regression

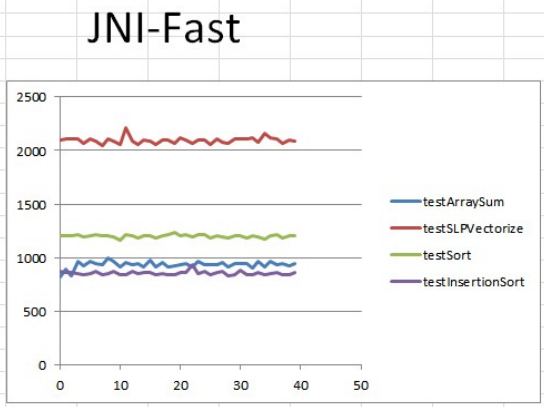
Our statistic regression is included as attachment as result Excel file:

TestResulst061013-2-ProfileDataShrink.xlsx

For every major test exists a graph (16 regressions) which are all similar to the following depiction

-9- java -Djava.library.path=. VectorTestStressVersion

Test No.	testArraySum	testSLPVectorize	testSort	testInsertionSort
0	825	2103	1209	868
1	891	2110	1202	867
2	830	2108	1208	860
3	966	2107	1211	857
4	923	2069	1193	842
5	968	2110	1209	849
6	943	2089	1211	869
7	939	2048	1201	840
8	997	2107	1202	857
9	963	2092	1199	870
10	913	2058	1169	844
11	953	2210	1211	842
12	932	2091	1206	869
13	943	2061	1180	857
14	915	2103	1206	862
15	981	2089	1201	865
16	914	2059	1183	838
17	951	2094	1210	852
18	911	2103	1219	847
19	927	2068	1237	837
20	931	2119	1210	861
21	946	2094	1215	859



Average	
934	testArraySum
2,095	testSLPVectorize
1,202	testSort
857	testInsertionSort

Every of our 16 major test cases is always executed 40 or 50 times, so that we can eliminate arbitrary deviations and distortions from the operating system or hardware to a minimum.

7.5 Discussion

The project and his test results shows us that code optimization is not an easy task. A lot of people are investing much more time to improve aspects of Java compilation and execution than we could in this short period of time. But we are able to do a tiny improvement to one aspect of the Java compilation. With the right amount of time there is much more possible. One outcome is that nobody should underestimate vector code. It is not per definition automatically faster. You have to investigate how to write the right code to make it faster.

Side effect outcomes:

- the importance of the JIT compiler. We gathered a huge amount of test data which shows that the JIT compiler makes Java program approximately more than ten times faster.
- hprof profiler: The activation of the profiler distorted our test result durations and makes the execution time ⅓ faster, but only the time sampling was faster, the real duration of a profiled Java execution was three times slower.

Another aspect of our implementation is that we make the Java platform dependent and this creates a discussion about the basic idea of Java to be platform independent. This is a kind of a contradiction if you really want faster code of a specific machine than it would be better to use directly C and add some assembler code to your program, but on the other hand if it is possible that the Java interpreter or JIT could better detect which processor is used on the machine where the code is executed he could add vector code for different processors and this would help to stay platform independent and much faster and if a processor is not offering vector extension than it would simply not be used.

8. Conclusion and Recommendations

8.1 Summary and Conclusions

Summary of findings

1. Auto-generated native code from Java can achieve 10x speed up compared to Java without JIT
2. Vectorization can be done in AOTC compilation, though current LLVM optimizer may not find the best vector size to achieve the best performance.
3. For platforms without the fastest JIT compiler, our approach can be used to achieve JIT-level performance and even better.

The conclusion is that it is possible to vectorize Java code and that it is easier as expected, but tiny test programs like ours are miles away to be an extension to a Java release. Our project shows that it is doable.

8.2 Recommendations for Future Studies

Given more time, we would like to implement function calling support to the Java code. This will give us the advantage of testing recursive algorithms such as merge sort and quick sort. Additionally, we want to be able to work with arrays of any data type. Currently, only integer arrays are supported and thus limit our testing capabilities.

Further steps would be to implement the vectorization into the Java interpreter or JIT to improve the overall speed of all Java programs, but this would be a huge project to figure out on which platform which vectorization is useful and creates not so much side effects and is stable enough that the old code base is still executable. The improvement could be enormous if vectorized code would be used on every computer where Java will executed.

Bibliography

[1] Erven Rohou, Kevin Williams, David Yuste: Vectorization technology to improve interpreter performance. TACO 9(4): 26 (2013)

[2] Ertl, M. A. and Gregg, D. 2004. Combining stack caching with dynamic superinstructions. In Proceedings of the Workshop on Interpreters, Virtual Machines and Emulators (IVME'04). ACM, New York, USA, 7–14.

[3] Auto-vectorization in GCC 4.7. URL: http://locklessinc.com/articles/vectorize/ [Retrieved 4/29/2013]

[4] Auto-vectorization (Chinese). URL: http://www.lingcc.com/2011/06/03/11619/ [Retrieved 4/29/2013]

[5] Auto-vectorization with LLVM. URL: http://llvm.org/docs/Vectorizers.html [Retrieved 4/29/2013]

[6] Seonggun Kim, Hwansoo Han: Efficient SIMD code generation for irregular kernels. PPOPP 2012: 55-64.

[7] Award Winners Made Breakthroughs in Network Efficiency, Data Mining, Education, Game Theory, Programming, and Community Problem-Solving. The Association for Computing Machinery. URL: *http://www.acm.org/press-room/news-releases/2013/acm-tech-awards-2012*. [Retrieved 4/29/2013]

[8] Dorit Nuzman, Sergei Dyshel, Erven Rohou, Ira Rosen, Kevin Williams, David Yuste, Albert Cohen, Ayal Zaks: Vapor SIMD: Auto-vectorize once, run everywhere. CGO 2011: 151-160

[9] Hal Finkel. Autovectorization with LLVM. The LLVM Compiler Infrastructure 2012 European Conference.

[10] Jiu-tao Nie, Buqi Cheng, Ligang Wang, Xiao-Feng Li, Vectorization for Java, IFIP International Conference on Network and Parallel Computing (NPC 2010), Zhengzhou, China, Sep 13-15, 2010.

[11] Douglas Lyon: "Java Optimization for Superscalar and Vector Architectures", in *Journal of Object Technology*, vol. 4, no. 2, March-April 2005, pp. 27-39.

[12] Kevin Williams, Albert Noll, Andreas Gal and David Gregg. Optimization Strategies for a Java Virtual Machine Interpreter on the Cell Broadband Engine
URL: http://people.inf.ethz.ch/anoll/publications/java_interpreter_cellbe.pdf [Retrieved 4/29/2013]

[13] Rosen, I., Nuzman, D., Zaks, A.: Loop-aware slp in gcc. In: GCC Summit. (2007) 131–142.

Appendix

1. Listing of testSum function

(a) Original C code for JNI library

```
JNIEXPORT jint JNICALL Java_HelloWorld_testSum (JNIEnv *env, jobject thisobj, jint arg0) {
{
        jint i;
        jint sum = 0;
        for (i = 0; i < arg0; i++)
                sum += i;
        return sum;
}
```

(b) Compiled LLVM IR

```
define i32 @Java_HelloWorld_testSum(%struct.JNINativeInterface_** %env, i8* %thisobj, i32
%arg0) unnamed_addr nounwind uwtable ssp {
 %env_addr = alloca %struct.JNINativeInterface_**, align 8
 %thisobj_addr = alloca i8*, align 8
  %arg0_addr = alloca i32, align 4
  %i = alloca i32
  %sum = alloca i32
  %"<retval>" = alloca i32
  %"alloca point" = bitcast i32 0 to i32
  store %struct.JNINativeInterface_** %env, %struct.JNINativeInterface_*** %env_addr, align 1
  store i8* %thisobj, i8** %thisobj_addr, align 1
  store i32 %arg0, i32* %arg0_addr, align 1
  %0 = load i32* %arg0_addr, align 4
  %"ssa point" = bitcast i32 0 to i32
  br label %"2"

"2":                                              ; preds = %entry
  br label %"4"

"3":                                              ; preds = %"4"
  %1 = add i32 %4, %3
  %2 = add i32 %3, 1
  br label %"4"

"4":                                              ; preds = %"3", %"2"
  %3 = phi i32 [ %2, %"3" ], [ 0, %"2" ]
  %4 = phi i32 [ %1, %"3" ], [ 0, %"2" ]
  %5 = icmp slt i32 %3, %0
  br i1 %5, label %"3", label %"5"

"5":                                              ; preds = %"4"
  store i32 %4, i32* %"<retval>", align 1
  br label %return

return:                                           ; preds = %"5"
  %6 = load i32* %"<retval>", align 4
  ret i32 %6
}
```

2. Listing of testSum function in LLVM IR (compiled from Java bytecode)

```
define i32 @_ZN10HelloWorld7testSumEJii(%struct.HelloWorld* %unnamed_arg, i32 %unnamed_arg2)
uwtable align 2 {
entry:
  %unnamed_arg_addr = alloca %struct.HelloWorld*, align 8
  %unnamed_arg_addr1 = alloca i32, align 4
  %"#slot#4#0" = alloca i32
  %"#slot#3#1" = alloca i32
  %"#slot#2#2" = alloca i32
  %D.250 = alloca i32
  %D.251 = alloca i32
  %"#slot#5#3" = alloca i32
  %D.253 = alloca i32
  %D.254 = alloca i32
  %D.255 = alloca i32
  %"<retval>" = alloca i32
  %"alloca point" = bitcast i32 0 to i32
  call void @llvm.dbg.declare(metadata !{%struct.HelloWorld** %unnamed_arg_addr}, metadata
!149), !dbg !150
  store %struct.HelloWorld* %unnamed_arg, %struct.HelloWorld** %unnamed_arg_addr, align 1
  call void @llvm.dbg.declare(metadata !{i32* %unnamed_arg_addr1}, metadata !151), !dbg !150
  store i32 %unnamed_arg2, i32* %unnamed_arg_addr1, align 1
  %0 = load i32* %unnamed_arg_addr1, align 4
  %"ssa point" = bitcast i32 0 to i32
  br label %"2", !dbg !150

"2":                                              ; preds = %entry
  br label %"4", !dbg !152

"3":                                              ; preds = %"4"
  %1 = add i32 %3, %4, !dbg !153
  %2 = add i32 %4, 1, !dbg !152
  br label %"4", !dbg !152

"4":                                              ; preds = %"3", %"2"
  %3 = phi i32 [ %1, %"3" ], [ 0, %"2" ], !dbg !152
  %4 = phi i32 [ %2, %"3" ], [ 0, %"2" ], !dbg !152
  %5 = icmp slt i32 %4, %0, !dbg !152
  br i1 %5, label %"3", label %"5", !dbg !152

"5":                                              ; preds = %"4"
  store i32 %4, i32* %"<retval>", align 1, !dbg !154
  br label %return, !dbg !154

return:                                           ; preds = %"5"
  %6 = load i32* %"<retval>", align 4, !dbg !154
  ret i32 %6, !dbg !154
}
```