JIT Compiler for i386

Mathias Kölsch and Kristian Kvilekval

December 15, 1998

1 Introduction

In this report we describe our implementation of a JIT compiler for Sun's JavaVirtualMachine (JVM) for the i386 instruction set. By extending and augmenting the JVM of the JDK 1.1.5 distribution, we provide an interpreter-compiler - JVM system that can execute any legal java program.

Our focus was on an extensible design to be a platform for further projects. For this worthy goal we provided for a variety of different optimizations, including delayed code generation and recompilation. Our implementation for this release was focused on the ability to compile as many bytecodes as possible and support most of the features of the original JVM.

2 JVM Interface

This section describes several interfaces interconnecting our compiler and the compiled code with the standard java interpreter. The process begins when the JVM interpreter calls the compiler. The compiler relies on certain functionality of the JVM. Once a method is compiled, it will be invoked by the interpreter using a special "invokeCompiledMethod". From within compiled code, we call back to an intermediate layer in the case of a method call. This intermediate layer is responsible for the execution of the method, and it delegates the call either to another invoker, or it restarts the original interpreter. We will discuss the interfaces in this order.

2.1 Interface: JVM interpreter to JIT compiler

The JVM was changed minimally, in order to interface to our compiler(we are called from only three places). Two of these call the compiler, which in turn determines if a method can be compiled or not. These routines then compile and link our invoker in the method block. The location of both compiler hooks is in the original default invoker for any Java method(other than native methods). (file: classruntime.c)

The third hook is in the class loader. Here, our method "jitCheckForbidCompilation" quickly examines the main characteristics of every method in the loaded class and marks it "not compilable" if it encounters one of the following:

- a try-catch block,
- floating point arguments or return types,
- method is a class initializer (not worth compiling),
- class is in a java.* or sun.* package.

(files: function call in classloader.c, function in jitruntime.c)

2.2 Interface: JIT compiler to JVM interpreter

In order to resolve field names, class constants and everything else that can be done at compile time (== class loading time == first method execution time), we use the same functions that the interpreter uses. None of these functions will invoke the compiler again, even though there might be some code executed during class initialization of a newly referenced class. (file: compiler.c)

2.3 Interface: JVM interpreter to compiled code

The invoker assigned to the methodblock for compiled code is a C function that we provide as a trampoline before we actually call the compiled machine code. In order to allow the JVM to call the code and vice versa with only minimal intermediate processing we chose to use the same activation frame model as the JVM uses. This incurs a small space overhead, plus a small time overhead necessary to initialize the frame. The trampoline sets up the execution environment and frame, and handles any synchronization and monitoring.

Compiled code uses slightly different return semantics which must be taken care of here: Instead of returning a true/false value depending on the success of the method execution as it is done for all the original method invokers, we return the actual result value from the machine code. This decision has proven very difficult to deal with, since we have to switch back and forth between those models. Even more problematic is the exception handling. See the known bugs section for more details. (file: jitruntime.c)

2.4 Interface: compiled code to JVM interpreter

The same work has to be done in reverse if we called a Java method from within compiled code. The compiler inserts a call to one of four C methods into the machine code, depending on the type of the Java method being called:

- jitInvokeStaticJavaMethod
- jitInvokeVirtualJavaMethod
- jitInvokeInterfaceJavaMethod
- jitInvokeSpecialJavaMethod

(file: jitruntime.c)

Each of these functions resolves information that is only dynamically available, again using functionality of the original JVM.

One special case should be mentioned here, the inline caching of the last lookup result for the implementation of an interface method. This is done in a very similar way to the original JVM method. There, the index into the "interface table" is stored after every invocation within the bytecode sequence for the invoke_interface opcode. We have provided a spot within the machine code where we can put an arbitrary value without interfering with the processor's interpretation of it. A "compare register eax with immediate" does exactly this, it has some space for a value-operand and it does not influence the execution path. The result of the lookup in the C function is stored one byte after the call in the machine code, while the byte immediately after the call instruction is the encoding of the compare.

2.5 Frame interface

The JVM internally uses two stacks: an operand stack for values, and a control stack which that is used by the actual interpreter. The operand stack is also used for control information completely contained within the interpreter (i.e. exception values and byte code return PCs).

Independent of the type of method invoked, we execute the individual invoker of the methodblock. This can be a

- Java method

  The invoker sets up the execution environment and the frame, and expects interpretation of the methodblock's code by the main interpretation loop in method ExecuteJava (file executeJava.c or assembly code files). Usually, the invoker is called by the interpreter, so it just returns there with a different current PC. We can't simply return since we were called by the compiled code, so we have to instantiate a new initial execution environment and frame. After this we start a new, independent interpreter. This, fortunately, is not very costly.

- Native method

  The execution of the invoker will return a valid result in JVM - style. It does not need further interpretation.

- Compiled method

  Just like in the native case, the call of our trampoline for the compiled code will return a valid result.

3 Compiler

The compiler was responsible for translating java method byte codes into executable i386 instructions to be executed immediately following the compilation. Specifically we had to allow for the compiler to be executing at any moment.

3.1 Byte code translations (compile.c, genops.c)

Byte code translation looks very similar to the JVM interpreter loop. We placed the actual generation of instructions into separate routines. These **gen** functions operate on JVM operands. The runtime pushing and poping of JVM operands generates a dependency graph between groups of generated instructions. All pushes and pops are removed by the compiler as these are unnecessary after translation. Our generator functions allow two modes of operation. The first is to generate code on the decoding of the JVM byte code. This is the default mode that we have currently tested and that our results show. The second mode just creates the graph of the gen functions and their operands. This graph can then be used at some later time to generate the code sequence after JVM operands have been mapped to machine operands. It leaves room for optimizations such as loop unrolling, dependency analysis etc.

3.2 Operand translation (joperands.c, genops.c)

There are two distinct sets of operands that are manipulated by the compiler. The first group is the JVM operands. This set includes JVM local variables and arguments, field operands and immediate values such as an address into the constant pool. The second group of operands is machine operands including immediate, register, indirect, indexed, and spill. The **gen** functions are responsible for translating any JVM operand into a machine operand if they have not already been translated by some other agent.

There are two basic routines responsible for operand translation. One makes a any operand a machine *addressable* while the other forces any type of operand into a *register*. Using the two routines, the **gen** routines can provide legal operands to the assembler level(described below).

3.3 register allocation and operand caching (joperands.c, jregister.c)

We actually implemented two completely different systems for mapping JVM operands to machine operands (i.e. registers, addresses). The first used a simple algorithm that allowed source operands to be addressable while output(destination) operands would be registers. A simple rotating register allocation scheme. Around subroutine calls from compiled code we had to save and restore the normal caller saved register for i386.

Operand caching was designed to eliminate unnecessary translation of JVM operands to machine operands. A simple table is kept mapping the last calculated use of a register to a JVM operand. The operand tranlsator routines first check this cache before generating the operator translation code.

The second system was implemented to improve on the code generation of the first system. We could produce code very fast, but reuse of computed values in registers turned out to be very difficult and error prone. Instead of searching through a cache to reduce runtime-reloading of runtime-operands, we keep track of how a machine operand is created. Further operands that map to the same Java operand just "parent" the existing one, and such that the previous computation (that changed the originally created operand into a machine addressable one) does not have to be repeated. This caching is not a trivial task and that it is difficult to keep track of which machine operands map. In other words, the semantics of an operand is much more similar to a java operand now. Instead of creating and discarding many operands during the production of machine code, the exact same operand now changes its face.

Again, the big advantage is, that it is known which registers map to operand values.

3.4 Assembler Level (jassem.c)

An assembler library was built in order that the **gen** routines and other components could generate machine code directly into memory. In order to make the programmers job easier we built the library to accept machine operand (defined above) in any machine legal combination. The assembler instruction routine, itself, would decide what sort opcodes were necessary in order to generate the instruction. This made the task of interfacing to the assembler almost the equivalent of generating textual assembler code to the assembler much more readable.

4 Future improvements

- Intermediate result caching for the first system's operand semantics; state: almost working.
- Code motion (loop invariants, especially for arrays).
- Better selection of compilable routines (we compile too many).
- Inlining of short functions.

  As we found out during the evaluation phase with SpecJava98, a lot of programs are actually written in a nice OO-style, i.e. implement the concept of information hiding quite well. There are virtually hundreds of access functions like this one:

  `public Point GetMax() { return (Max); }`

  Setting up a new frame is much more costly than just accessing a field, as can be seen from the micro benchmarks: field access and access methods. Clearly this is the most promising optimization we should look into, even though it might be not easy due to dynamic dispatching.

5 Results

5.1 Micro Benchmarks

5.1.1 compilation speed

We measured the speed of compilation for groups of bytecodes. For that purpose, we created functions that consisted mainly of those instructions. In order to be able to measure it at all, we had to put several thousand bytecodes in these functions. Depending on the size of the individual bytecodes, the code size varied between 10,000 and 25,000 bytes (for 3,000 to 12,000 bytecode-instructions).

Numbers shown are in instructions compiled per ms and bytes/ms:

Type of bytecode	instr/ms	bytes/ms
integer arithmetic:	56.0	56.4
static variable access:	32.1	87.3
member variable access:	48.5	93.0
local variable access:	61.8	164.7
method calls:	23.8	43.3
if statements:(forward gotos)	57.3	102.5
average:	46.6	91.2

The speed for generating method calls and references to other classes depends on the use of C functions provided by the JDK.

5.1.2 Execution speed

We used three different versions of the JDK 1.1.5 runtime system.

1. The interpreter provided by Sun, written in C code, compiled with `cc -Xa -xO4 -v -DTRIMMED -Di386 -DSOLARIS2`
2. The interpreter provided by Sun, written in assembly language, compiled with `cc -Xa -xO4 -v -DTRIMMED -Di386 -DSOLARIS2`
3. Our Jit compiler, in combination (for not compilable methods) with the interpreter 1), compiled with `gcc -DUSE_JIT -DTRIMMED -Di386 -DSOLARIS2`

Given are absolute times in seconds and number of individual sets of operations per millisecond. Speedup is relative to the C version of the interpreter. JVM startup times (0.6 seconds) are subtracted.

Reference loops	intr	asm	jit
empty loop:	2725	792	187
(compare, inc, goto)	1.57k	5.42k	22.9k
speedup	1.0	3.45	14.59
nested loops:	60.4	20.0	5.0
(compare, inc, goto)	1.66k	5.0k	20.0k
speedup	1.0	3.01	12.05

Variable Access	intr	asm	jit
local variables:	94.0	28.6	5.44
(loop substracted)	30.3	10.15	1.07
(assignment)	3.3k	9.85k	94.4k
speedup	1.0	2.98	28.61
field variables:	10.8	3.57	1.21
(loop substracted)	10.2	3.37	1.20
(assignment)	0.98k	2.97k	8.33k
speedup	1.0	3.03	8.50

Arithmetic	intr	asm	jit
arithmetics (a = b + c)	11.8	3.4	0.67
(- loop & assign)	8.8	2.4	0.56
(plus)	1.4k	4.17k	17.9k
speedup	1.0	2.98	12.79
arithmetics (a = b * c)	12.6	4.1	1.10
(- loop & assign)	9.6	3.1	1.0
(multiplication)	1.04k	3.23k	10.0k
speedup	1.0	3.11	9.62
arithmetics (a = b ? c)	14.9	6.8	3.45
(- loop & assign)	13.9	5.8	3.35
(modulo and division)	0.72k	1.72k	2.99k
speedup	1.0	2.39	4.15

Method invocation	intr	asm	jit
method calls (compiled)	16.6	6.1	13.9
(loop substracted)	10.2	4.3	13.8
(invoker overhead)	0.98k	2.32k	0.72k
speedup	1.0	2.37	0.73
method calls (n-inline)	25.7	10.9	14.4
(- loop & body)	13.9	7.5	13.8
(invoker overhead)	0.72k	1.33k	0.72k
speedup	1.0	1.85	1.0
access methods	35.0	15.2	32.4
(- loop & field-acc)	24.2	11.6	31.2
(method overhead)	0.41k	0.86k	0.32k
speedup	1.0	2.10	0.78

Arrays	intr	asm	jit
arrays	12.3	5.4	2.5
(- loop)	11.7	5.2	2.5
(op: a[i] = i)	0.9k	1.92k	4.0k
speedup	1.0	2.13	4.44
arrays	9.2	3.9	1.7
(- loop)	8.6	3.7	1.6
(op: a[i] = a[i-10])	1.16k	2.70k	6.25k
speedup	1.0	2.33	5.39

5.1.3 Comments

1. The Java methods we used for those benchmarks are given in the appendix.

2. The shown execution speed actually includes the speed for compilation. But the functions compiled are so small that the compilation time is a tiny fraction of the total execution time.

3. The speedup of the assembly code version is rather uniform. That is probably due to an improvement of the interpreter loop, since this would be a constant factor. This conclusion is supported by the speedup for method invocations, which is smaller by another constant factor. The reason is that certain operations have to be performed in both the C and the assembly code version: method lookup, frame setup and so on.

4. The method invocations for very small methods (getX() etc.) are faster in the interpreted versions, since the JDK interpreters inline the code in the caller. In the not-inlined case, the JIT is not faster than the C interpreter because it maintains the JavaFrame semantics.

5. Longer arithmetic expressions don't have any effect on the speedup, the different JVMs show linear relative speed behavior.

6. For arrays, we had to use a lot of memory and start the jvm with java -mx64m. The times reported in the table are user times, as determined by the shell built-in command "time". The actual time varied between a few seconds more up to eight times more. For the other benchmarks, the user time used to be about .4 to 1 second less than the total execution time. Therefore, the time reported was not subtracted by the jvm-startup time.

5.2 Spec Benchmarks

We compared our JIT performance with the standard Java interpreter.

The following table shows the result of trying the SPEC benchmarks with our compiler.

Program	intr	JIT	asm	Total
_200_check	0:01.51	0:01.49	0:01.49	
_201_compress		fail-null		
_202_jess		fail-reg		
_205_raytrace	25:18.61	22:47.20	13:48.97	
_209_db				
_213_javac		fail-gc		
_222_mpegaudio	33:51.24	39:33.26		
_227_mtrt	17:44.63	22:20.26	15:21.52	
_228_jack	17:04.00	7:53.54 → nullpointer	14:21.06	3 Parsers-gc
_999_checkit				

5.3 Known bugs

1. We do no allocation checks or null checks. They can be added easily, but they didn't contribute a lot to our goal to run as many programs as possible.

2. There is still a bug in the system that reveals itself during a garbage collection. It occurs while the GC is walking the thread stack.

3. There is one register allocation bug that occurs when requesting a special register (some instruction require specific registers). The current spill code may not always find the operand using the register and therefore refuses to spill it.

4. Exception handling is maintained through a special control variable of the ExecutionEnvironment. Additionally, the original invoker returns FALSE in case an exception might have occured. In our case, exception return handling is not done through the frame. We actually use C compiler style return handling (passing return values in a register), but we copy this to the JVM frame during the frame unwinding. In order to pass through the compiled code while doing an exception return, we forcibly unwind the compiled code frame. This is done using several i386 frame unwind instructions placed in a row. Unfortunately, proper stack unwinding may not occur if registers were allocated in the intervening frames.

   We must add exception handling to the compiled methods.

A Example compilations

We show the Java files used for the micro benchmarks and the correspond output from gdb.

```
void empty_loop() {
    int i,
      min = Integer.MIN_VALUE, // == -2147483648
      max = Integer.MAX_VALUE; // == 2147483647
    for (i = min; i < max; i ++) { // 4294967295
    }
}
```

```
0x80b9c48: pushl  %ebp
0x80b9c49: movl   %esp,%ebp
0x80b9c4b: pushl  %ebx
0x80b9c4c: pushl  %esi
0x80b9c4d: pushl  %edi
0x80b9c4e: movl   0xc(%ebp),%edi
0x80b9c51: movl   $0x80000000,0x8(%edi)
0x80b9c58: movl   $0x7fffffff,0xc(%edi)
0x80b9c5f: movl   0x8(%edi),%ecx
0x80b9c62: movl   %ecx,0x4(%edi)
0x80b9c65: jmp    0x80b9c6e
0x80b9c6a: addl   $0x1,0x4(%edi)
0x80b9c6e: movl   0x4(%edi),%edx
0x80b9c71: cmpl   %edx,0xc(%edi)
0x80b9c74: jg     0x80b9c6a
0x80b9c76: popl   %edi
0x80b9c77: popl   %esi
0x80b9c78: popl   %ebx
0x80b9c79: leave
0x80b9c7a: ret
```

```
void nested_loops() {
    for (int i = 0; i < 1000; i ++) {
      for (int j = 0; j < 1000; j ++) {
for (int k = 0; k < 100; k ++) {
}
      }
    }
}
```

```
0x80c3860: pushl  %ebp
0x80c3861: movl   %esp,%ebp
0x80c3863: pushl  %ebx
0x80c3864: pushl  %esi
0x80c3865: pushl  %edi
0x80c3866: movl   0xc(%ebp),%edi
0x80c3869: movl   $0x0,0x4(%edi)
0x80c3870: jmp    0x80c38a8
0x80c3875: movl   $0x0,0x8(%edi)
0x80c387c: jmp    0x80c389b
0x80c3881: movl   $0x0,0xc(%edi)
0x80c3888: jmp    0x80c3891
0x80c388d: addl   $0x1,0xc(%edi)
0x80c3891: cmpl   $0x64,0xc(%edi)
0x80c3895: jl     0x80c388d
0x80c3897: addl   $0x1,0x8(%edi)
0x80c389b: cmpl   $0x3e8,0x8(%edi)
0x80c38a2: jl     0x80c3881
0x80c38a4: addl   $0x1,0x4(%edi)
0x80c38a8: cmpl   $0x3e8,0x4(%edi)
0x80c38af: jl     0x80c3875
0x80c38b1: popl   %edi
0x80c38b2: popl   %esi
0x80c38b3: popl   %ebx
0x80c38b4: leave
0x80c38b5: ret
```

```
  void locals() {
    int a = 4, b = 3;
    for (int i = 0; i < 100000000; i ++) { // hundred million
      a = b;
    }
  }
```

```
0x80b95b0: pushl  %ebp
0x80b95b1: movl   %esp,%ebp
0x80b95b3: pushl  %ebx
0x80b95b4: pushl  %esi
0x80b95b5: pushl  %edi
0x80b95b6: movl   0xc(%ebp),%edi
0x80b95b9: movl   $0x4,0x4(%edi)
0x80b95c0: movl   $0x3,0x8(%edi)
0x80b95c7: movl   $0x0,0xc(%edi)
0x80b95ce: jmp    0x80b95dd
0x80b95d3: movl   0x8(%edi),%eax
0x80b95d6: movl   %eax,0x4(%edi)
0x80b95d9: addl   $0x1,0xc(%edi)
0x80b95dd: cmpl   $0xf4240,0xc(%edi)
0x80b95e4: jl     0x80b95d3
0x80b95e6: popl   %edi
0x80b95e7: popl   %esi
0x80b95e8: popl   %ebx
0x80b95e9: leave
0x80b95ea: ret
```

```
  void fields() {
    for (int i = 0; i < 10000000; i ++) { // ten million
      sa = sb;
    }
  }
```

```
0x80b9910: pushl  %ebp
0x80b9911: movl   %esp,%ebp
0x80b9913: pushl  %ebx
0x80b9914: pushl  %esi
0x80b9915: pushl  %edi
0x80b9916: movl   0xc(%ebp),%edi
0x80b9919: movl   $0x0,0x4(%edi)
0x80b9920: jmp    0x80b9939
0x80b9925: movl   $0x80c429c,%ecx
0x80b992b: movl   (%ecx),%edx
0x80b992d: movl   $0x80c4284,%ebx
0x80b9933: movl   %edx,(%ebx)
0x80b9935: addl   $0x1,0x4(%edi)
0x80b9939: cmpl   $0x186a0,0x4(%edi)
0x80b9940: jl     0x80b9925
0x80b9942: popl   %edi
0x80b9943: popl   %esi
0x80b9944: popl   %ebx
0x80b9945: leave
0x80b9946: ret
```

```
  void arithmetics() {
    int a = 0, b = 0, c = 0;
    for (int i = 0; i < 10000000; i ++) { // ten million
      a = b + c;
    }
  }
```

```
0x80b9c48: pushl  %ebp
0x80b9c49: movl   %esp,%ebp
0x80b9c4b: pushl  %ebx
0x80b9c4c: pushl  %esi
0x80b9c4d: pushl  %edi
0x80b9c4e: movl   0xc(%ebp),%edi
0x80b9c51: movl   $0x0,0x4(%edi)
0x80b9c58: movl   $0x1,0x8(%edi)
0x80b9c5f: movl   $0x2,0xc(%edi)
0x80b9c66: movl   $0x0,0x10(%edi)
0x80b9c6d: jmp    0x80b9c7f
0x80b9c72: movl   0xc(%edi),%ecx
0x80b9c75: addl   0x8(%edi),%ecx
0x80b9c78: movl   %ecx,0x4(%edi)
0x80b9c7b: addl   $0x1,0x10(%edi)
0x80b9c7f: cmpl   $0x186a0,0x10(%edi)
0x80b9c86: jl     0x80b9c72
0x80b9c88: popl   %edi
0x80b9c89: popl   %esi
0x80b9c8a: popl   %ebx
0x80b9c8b: leave
0x80b9c8c: ret
```

```
void compilable_calls() {
    for (int i = 0; i < 10000000; i ++) { // ten million
      empty();
    }
  }
```

```
0x80b9388: pushl  %ebp
0x80b9389: movl   %esp,%ebp
0x80b938b: pushl  %ebx
0x80b938c: pushl  %esi
0x80b938d: pushl  %edi
0x80b938e: movl   0xc(%ebp),%edi
0x80b9391: movl   $0x0,0x4(%edi)
0x80b9398: jmp    0x80b93b6
0x80b939d: movl   (%edi),%ecx
0x80b939f: movl   %ecx,0x30(%edi)
0x80b93a2: pushl  0x8(%ebp)
0x80b93a5: pushl  $0x80c46ec
0x80b93aa: call   0x80588b4 <jitInvokeVirtualJavaMethod>
0x80b93af: addl   $0x8,%esp
0x80b93b2: addl   $0x1,0x4(%edi)
0x80b93b6: cmpl   $0x186a0,0x4(%edi)
0x80b93bd: jl     0x80b939d
0x80b93bf: popl   %edi
0x80b93c0: popl   %esi
0x80b93c1: popl   %ebx
0x80b93c2: leave
0x80b93c3: ret
```

```
void empty() { }
```

```
0x80acff0: pushl  %ebp
0x80acff1: movl   %esp,%ebp
0x80acff3: pushl  %ebx
0x80acff4: pushl  %esi
0x80acff5: pushl  %edi
0x80acff6: movl   0xc(%ebp),%edi
0x80acff9: popl   %edi
0x80acffa: popl   %esi
0x80acffb: popl   %ebx
0x80acffc: leave
0x80acffd: ret
```

```
void not_inlinable() {
  int a, b = 0, c = 0;
  a = b + c;
}
```

```
0x80b9dd0: pushl  %ebp
0x80b9dd1: movl   %esp,%ebp
0x80b9dd3: pushl  %ebx
0x80b9dd4: pushl  %esi
0x80b9dd5: pushl  %edi
0x80b9dd6: movl   0xc(%ebp),%edi
0x80b9dd9: movl   $0x0,0x8(%edi)
0x80b9de0: movl   $0x0,0xc(%edi)
0x80b9de7: movl   0x8(%edi),%edx
0x80b9dea: addl   0xc(%edi),%edx
0x80b9ded: movl   %edx,0x4(%edi)
0x80b9df0: popl   %edi
0x80b9df1: popl   %esi
0x80b9df2: popl   %ebx
0x80b9df3: leave
0x80b9df4: ret
```

```
  void accessMethods() {
    for (int i = 0; i < 10000000; i ++) { // ten million
      setSA(getSB());
    }
  }
```

```
0x80cfe40: pushl  %ebp
0x80cfe41: movl   %esp,%ebp
0x80cfe43: pushl  %ebx
0x80cfe44: pushl  %esi
0x80cfe45: pushl  %edi
0x80cfe46: movl   0xc(%ebp),%edi
0x80cfe49: movl   $0x0,0x4(%edi)
0x80cfe50: jmp    0x80cfe86
0x80cfe55: movl   (%edi),%esi
0x80cfe57: movl   %esi,0x30(%edi)
0x80cfe5a: pushl  0x8(%ebp)
0x80cfe5d: pushl  $0x80c4410
0x80cfe62: call   0x80588b4 <jitInvokeVirtualJavaMethod>
0x80cfe67: addl   $0x8,%esp
0x80cfe6a: movl   (%edi),%ecx
0x80cfe6c: movl   %ecx,0x30(%edi)
0x80cfe6f: movl   %eax,0x34(%edi)
0x80cfe72: pushl  0x8(%ebp)
0x80cfe75: pushl  $0x80c43b4
0x80cfe7a: call   0x80588b4 <jitInvokeVirtualJavaMethod>
0x80cfe7f: addl   $0x8,%esp
0x80cfe82: addl   $0x1,0x4(%edi)
0x80cfe86: cmpl   $0x186a0,0x4(%edi)
0x80cfe8d: jl     0x80cfe55
0x80cfe8f: popl   %edi
0x80cfe90: popl   %esi
0x80cfe91: popl   %ebx
0x80cfe92: leave
0x80cfe93: ret
```

```
void setSA(int s) { sa = s; }
```

```
0x80b9ac0: pushl  %ebp
0x80b9ac1: movl   %esp,%ebp
0x80b9ac3: pushl  %ebx
0x80b9ac4: pushl  %esi
0x80b9ac5: pushl  %edi
0x80b9ac6: movl   0xc(%ebp),%edi
0x80b9ac9: movl   0x4(%edi),%esi
0x80b9acc: movl   $0x80c4284,%eax
0x80b9ad2: movl   %esi,(%eax)
0x80b9ad4: popl   %edi
0x80b9ad5: popl   %esi
0x80b9ad6: popl   %ebx
0x80b9ad7: leave
0x80b9ad8: ret
```

```
int getSB() { return sb; }
```

```
0x80ac958: pushl  %ebp
0x80ac959: movl   %esp,%ebp
0x80ac95b: pushl  %ebx
0x80ac95c: pushl  %esi
0x80ac95d: pushl  %edi
0x80ac95e: movl   0xc(%ebp),%edi
0x80ac961: movl   $0x80c429c,%edx
0x80ac967: movl   (%edx),%ebx
0x80ac969: movl   %ebx,%eax
0x80ac96b: popl   %edi
0x80ac96c: popl   %esi
0x80ac96d: popl   %ebx
0x80ac96e: leave
0x80ac96f: ret
```

```
  void arrays() {
    int[] a = new int[10000000];
    for (int i = 0; i < 10000000; i ++) { // ten million
arrays with a[i] = i;
    }
  }
```

```
0x80c3860: pushl  %ebp
0x80c3861: movl   %esp,%ebp
0x80c3863: pushl  %ebx
0x80c3864: pushl  %esi
0x80c3865: pushl  %edi
0x80c3866: movl   0xc(%ebp),%edi
0x80c3869: pushl  $0x186a0
0x80c386e: pushl  $0xa
0x80c3873: call   0x804e5d4 <ArrayAlloc>
0x80c3878: addl   $0x8,%esp
0x80c387b: movl   %eax,0x4(%edi)
0x80c387e: movl   $0xa,0x8(%edi)
0x80c3885: jmp    0x80c389c
0x80c388a: movl   0x8(%edi),%edx
0x80c388d: movl   0x4(%edi),%ebx
0x80c3890: movl   (%ebx),%ebx
0x80c3892: movl   0x8(%edi),%esi
0x80c3895: movl   %edx,(%ebx,%esi,4)
0x80c3898: addl   $0x1,0x8(%edi)
0x80c389c: cmpl   $0x186a0,0x8(%edi)
0x80c38a3: jl     0x80c388a
0x80c38a5: popl   %edi
0x80c38a6: popl   %esi
0x80c38a7: popl   %ebx
0x80c38a8: leave
0x80c38a9: ret
```

```
  void arrays() {
    int[] a = new int[10000000];
    for (int i = 0; i < 10000000; i ++) { // ten million
      arrays with a[i] = a[i - 10];
    }
  }
```

```
0x80c3cc0: pushl  %ebp
0x80c3cc1: movl   %esp,%ebp
0x80c3cc3: pushl  %ebx
0x80c3cc4: pushl  %esi
0x80c3cc5: pushl  %edi
0x80c3cc6: movl   0xc(%ebp),%edi
0x80c3cc9: pushl  $0x186a0
0x80c3cce: pushl  $0xa
0x80c3cd3: call   0x804e5d4 <ArrayAlloc>
0x80c3cd8: addl   $0x8,%esp
0x80c3cdb: movl   %eax,0x4(%edi)
0x80c3cde: movl   $0xa,0x8(%edi)
0x80c3ce5: jmp    0x80c3d07
0x80c3cea: movl   0x8(%edi),%eax
0x80c3ced: subl   $0xa,%eax
0x80c3cf0: movl   0x4(%edi),%ecx
0x80c3cf3: movl   (%ecx),%ecx
0x80c3cf5: movl   (%ecx,%eax,4),%edx
0x80c3cf8: movl   0x4(%edi),%ebx
0x80c3cfb: movl   (%ebx),%ebx
0x80c3cfd: movl   0x8(%edi),%esi
0x80c3d00: movl   %edx,(%ebx,%esi,4)
0x80c3d03: addl   $0x1,0x8(%edi)
0x80c3d07: cmpl   $0x186a0,0x8(%edi)
0x80c3d0e: jl     0x80c3cea
0x80c3d10: popl   %edi
0x80c3d11: popl   %esi
0x80c3d12: popl   %ebx
0x80c3d13: leave
0x80c3d14: ret
```

B Changed Files

Here is a listing of the files we changed from the original JDK 1.1.5 distribution. "verboseJIT" means that those files were only changed in order to get the -verboseJIT flag for the command line.

C files:

- classruntime.c
- classloader.c
- executeJava.c (verboseJIT)
- jdk/src/solaris/java/javai.c (verboseJIT)
- /share/java/runtime/jni.c (verboseJIT)
- /share/java/util/globals.c (verboseJIT)

Header files:

- jdk/src/share/java/include/jni.h (verboseJIT)
- jdk/src/share/java/include/interpreter.h (verboseJIT) Makefiles etc:
- jdk/build/solaris/java/main/GNUmakefile

C Translated Opcodes

Below is a the list of opcodes translated

aaload
aastore
aconst_null
aload
aload_0
aload_1
aload_2
aload_3
anewarray
areturn
arraylength
astore
astore_0
astore_1
astore_2
astore_3
baload
bastore
bipush
caload
castore
checkcast
dup
dup2
dup_x1
dup_x2
getfield
getstatic
goto
i2b
i2c
i2s
iadd
iaload
iand
iastore
iconst_0
iconst_1
iconst_2
iconst_3
iconst_4
iconst_5
iconst_m1
idiv
if_acmpeq
if_acmpne
if_icmpeq
if_icmpge
if_icmpgt
if_icmple
if_icmplt
if_icmpne
ifeq
ifge
ifgt
ifle
iflt
ifne
ifnonnull
ifnull
iinc
iload
iload_0
iload_1
iload_2
iload_3
imul
ineg
instanceof
invokeinterface
invokespecial
invokestatic
ior
irem
ireturn
ishl
ishr
istore
istore_0
istore_1
istore_2
istore_3
isub
iushr
ixor
new
newarray
nop
pop
pop2
return
saload
sastore
swap