Efficient JavaVM Just-in-Time Compilation

Andreas Krall
Institut für Computersprachen
Technische Universität Wien
Argentinierstraße 8
A-1040 Wien
andi@complang.tuwien.ac.at

Abstract

Conventional compilers are designed for producing highly optimized code without paying much attention to compile time. The design goals of Java just-in-time compilers are different: produce fast code at the smallest possible compile time. In this article we present a very fast algorithm for translating JavaVM byte code to high quality machine code for RISC processors. This algorithm handles combines instructions, does copy elimination and coalescing and does register allocation. It comprises three passes: basic block determination, stack analysis and register preallocation, final register allocation and machine code generation. This algorithm replaces an older one in the CACAO JavaVM implementation reducing the compile time by a factor of seven and producing slightly faster machine code. The speedup comes mainly from following simplifications: fixed assignment of registers at basic block boundaries, simple register allocator, better exception handling, better memory management and fine tuning the implementation. The CACAO system is currently faster than every JavaVM implementation for the Alpha processor and generates machine code for all used methods of the javac compiler and its libraries in 60 milliseconds on an Alpha workstation.

1 Introduction

Java's [2] success as a programming language results from its role as an Internet programming language. The basis for this success is the machine-independent distribution format of programs with the Java virtual machine [12]. The standard interpretive implementation of the Java virtual machine makes execution of programs slow. This does not matter if small applications are executed in a browser, but becomes intolerable if big applications are executed. There are two solutions to solve this problem:

- specialized JavaVM processors,
- compilation of byte code to the native code of a standard processor.

SUN took both paths and is developing both Java processors and native code compilers. In our CACAO system we chose to go for native code compilation since it is more portable and gives more opportunities for improving the execution speed. Compiling to native code can be done in two different ways: compilation of the complete program in advance or compilation on demand of only the methods which are executed (just in time compiler, JIT). The CACAO system [10] uses a JIT compiler and is freely available via the world wide web.

1.1 Previous Work

The idea of machine independent program representations is quite old and goes back to the year 1960 [14]. An intermediate language UNCOL (UNiversal Computer Oriented Language) was proposed for use in compilers to reduce the development effort of compiling many different languages to many different architectures. The design of the JavaVM has been strongly influenced by P code, the abstract machine used by many Pascal implementations [13]. P code is well known from its use in the UCSD Pascal system. There have even been efforts to develop microprocessors which execute P code directly.

The Amsterdam compiler kit [16] [15] uses a stack oriented intermediate language. This language has been designed for fast compilers which emit efficient code. The intermediate representation of the Gardens Point compiler project is also based on a stack machine called *Dcode* [8]. *Dcode* was influenced by Pascal P code. Both *Dcode* interpreters and code generators for different architectures exist.

The problems of compiling a stack oriented abstract machine code to native code are well known from the program-

ming language Forth. In his thesis [5] and in [7] Ertl describes RAFTS, a Forth system that generates native code at run time. Translating the stack operations to native code is done by translating the operations back to expressions represented as directed acyclic graphs as an intermediate step. In [6] he translates Forth to native code using C as an intermediate language. In this system the stack slots are translated to local variables of a function. Optimization and code generation are performed by the C compiler.

The first implementations of JIT compilers became available last year for the browsers from Netscape and Microsoft on PCs. They were followed by Symantec's development environment. Recently SUN released a JIT compiler for the Sparc and PowerPC processors. Silicon Graphics developed a JIT compiler for the MIPS processor and recently Digital released a JIT for the Alpha processor.

A public domain JIT compiler for several architectures is the kaffe system developed by Tim Wilkinson (`http://www.kaffe.org/`). For all the above mentioned systems, no publicly available description of the compilation techniques exists.

The translation scheme of the *Caffeine* system is described in [9]. It supports both a simple translation scheme which emulates the stack architecture and a more sophisticated one which eliminates the stack completely and uses registers instead. *Caffeine* is not intended as a JIT compiler. It compiles a complete program in advance. DAISY (Dynamically Architected Instruction Set from Yorktown) is a VLIW architecture developed at IBM for fast execution of PowerPC, S/390 and JavaVM code. Compatibility with different old architectures is achieved by using a JIT compilation technique. The JIT compilation scheme for the JavaVM is described in [4].

Adl-Tabatabai and others [1] describe a fast and effective code generation system for a JIT compiler. This compiler does optimizations like bound check elimination, common subexpression elimination and two kinds of register allocation, a simple one and a global priority based one. The results show that for most benchmark programs the complex register allocator and the subexpression eliminator incur to much overhead which does not pay back at run time.

2 Translation of stack code to register code

The JavaVM is a typed stack architecture [12]. There are different instructions for integer, long integer, floating point and address types. The main instruction set consists of arithmetic/logical and load/store/constant instructions. All these instructions either work directly on the stack or move values between the stack and local variables. There are special instructions for array access and for accessing the fields of objects (memory access), for method invocation, and for type checking.

The architecture of a RISC processor is completely different from the stack architecture of the JavaVM. RISC processors have large sets of registers. They execute arithmetic and logic operations only on values which are held in registers. Load and store instructions are provided to move data between memory and registers. Local variables of methods usually reside in registers and are saved in memory only during a method call or if there are too few registers.

2.1 Machine code translation examples

The example expression `a = b - c * d` would be translated by an optimizing C compiler to the following two Alpha instructions (the variables `a`, `b`, `c` and `d` reside in registers):

```
MULL c,d,tmp0  ; tmp0 = c * d
SUBL b,tmp0,a  ; a = b - tmp0
```

If JavaVM code is translated to machine code, the stack is eliminated and the stack slots are represented by temporary variables usually residing in registers. A naive translation of the previous example would result in the following Alpha instructions:

```
iload b   -->  MOVE b,t0
iload c   -->  MOVE c,t1
iload d   -->  MOVE d,t2
imul      -->  MULL t1,t2,t1
isub      -->  SUBL t0,t1,t0
istore a  -->  MOVE t0,a
```

The problems of translating JavaVM code to machine code are primarily the elimination of the unnecessary copy instructions and finding an efficient register allocation algorithm. A common but expensive technique is to do the naive translation and use an additional pass for copy elimination and coalescing.

2.2 The old translation scheme

The old CACAO compiler did the translation to machine code in four steps. First, basic blocks were determined. Then, the JavaVM was translated into a register oriented intermediate representation, the registers were allocated, and finally machine code was generated. The intermediate representation was oriented towards a RISC architecture target and assumed that all operands reside in registers (assuming an unlimited number of pseudo registers). The intermediate instructions contained a `MOVE` instruction for register moves, `OP1`, `OP2` and `OP3` instructions for the arithmetic/logical operations, a `MEM` instruction for accessing the fields of objects, `BRA` instructions and special instructions for method invocation (`METHOD`). Two special instructions (`ACTIVATE` and `DROP`) maintained live range information for the register allocator.

The second pass of the compiler translates each JavaVM `load` or `store` instruction into a corresponding intermediate code `MOVE` instruction using a new register as the destination register in the case of a `load`. Always using a new register yields code in a similar form to static single assignment form [3], which is commonly used for compiler optimizations. A JavaVM `iadd` instruction is translated into an `OP2` instruction, again using a new destination register.

This naive translation scheme would generate many `MOVE` instructions. Therefore `MOVE` instructions are generated lazily. The translator keeps lists which track which registers should contain the same values (that are registers which are just copies of another register). Instead of generating a `MOVE` instruction, the translator enters the register into a copy list. If the translator should later generate a `DROP` instruction, it deletes the register from the list.

When at control flow joins the register lists did not match, the corresponding `MOVE` instruction had to be generated. But for most joins the stack, and therefore the register lists, are empty or else the registers are compatible. Furthermore the register allocator tries to assign the same hardware register to the same stack slots so that `MOVE` instructions can be eliminated.

2.3 Old register allocation

For a just-in-time compiler expensive register allocation algorithms, like graph coloring, cannot be used. We therefore designed a simple and fast scheme. There are two different sets of registers: registers for stack slots and registers for local variables. First, registers for stack slots are assigned. Afterwards, the remaining registers are assigned to the local variables which are active in the whole method.

All registers are assigned to a CPU register at the beginning of a basic block. An existing allocation is left unchanged. The allocator scans the instructions and, for each instruction which activates a register and to which no CPU register has been assigned, a new CPU register is selected. If the allocator has run out of CPU registers, the register is spilled to memory. There exist some conventions for the assignment of registers when calling methods. To prevent unnecessary copy instructions at a method call prior to the allocation pass, pseudo registers which are method parameters or return values are assigned the correct register (precoloring).

2.4 Problems of the old scheme

The old compiler used a lot of doubly linked lists and allocated every object explicitly. So a large amount of memory was used and a large percentage of the compile time was spent in object allocation. It had to do four passes over the code and there were examples in our applications where the compiler took up to fifty percent of the total run time. So we searched for improvements and designed a new translation algorithm.

3 The new translation algorithm

The new translation algorithm can get by with three passes. The first pass determines basic blocks and builds a representation of the JavaVM instructions which is faster to decode. The second pass analyses the stack and generates a static stack structure. During stack analysis variable dependencies are tracked and register requirements are computed. In the final pass register allocation of temporary registers is combined with machine code generation.

The new compiler computes the exact number of objects needed or computes an upper bound and allocates the memory for the necessary temporary data structures in three big blocks (the basic block array, the instruction array and the stack array). Eliminating all the double linked lists also reduced the memory requirements by a factor of five.

3.1 Basic block determination

The first pass scans the JavaVM instructions, determines the basic blocks and generates an array of instructions which has fixed size and is easier to decode in the following passes. Each instruction contains the opcode, two operands and a pointer to the static stack structure after the instruction (see next sections). The different opcodes of JavaVM instructions which fold operands into the opcode are represented by just one opcode in the instruction array.

3.2 Basic block interfacing convention

The handling of control flow joins was quite complicated in the old compiler. We therefore introduced a fixed interface at basic block boundaries. Every stack slot at a basic block boundary is assigned a fixed interface register. The stack analysis pass determines the type of the register and if it has to be saved across method invocations. To enlarge the size of basic blocks method invocations do not end basic blocks. To guide our compiler design we did some static analysis on a large application written in Java: the javac compiler and the libraries it uses. Table 1 shows that in more than 93% of the cases the stack is empty at basic block boundaries and that the maximal stack depth is 6. Using this data it becomes clear that the old join handling did not improve the quality of the machine code.

3.3 Copy elimination

To eliminate unnecessary copies loading of values is delayed until the instruction is reached which consumes the

stack depth	0	1	2	3	4	5	6	>6
occurrences	7930	258	136	112	36	8	3	0

Table 1. distribution of stack depth at block boundary

value. To compute the information the run time stack is simulated at compile time. Instead of values the compile time stack contains the type of the value, if a local variable was loaded to a stack location and similar information. Adl-Tabatabai [1] used a dynamic stack which is changed at every instruction. A dynamic stack only gives the possibility to move information from earlier instructions to later instructions. We use a static stack structure which enables information flow in both directions.

Fig. 1 shows our instruction and stack representation. An instruction has a reference to the stack before the instruction and the stack after the instruction. The stack is represented as a linked list. The two stacks can be seen as the source and destination operands of an instruction. In the implementation only the destination stack is stored, the source stack is the destination of stack of the previous instruction.

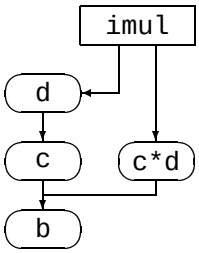

Figure 1. instruction and stack representation

This representation can easily be used for copy elimination. Each stack element not only contains the type of the stack slot but also the local variable number of which it is a copy, the argument number if it is an argument, the interface register number if it is an interface. Load (push the content of a variable onto the stack) and store instructions do no generate a copy machine instruction if the stack slot contains the same local variable. Generated machine instructions for arithmetic operations directly use the local variables as their operands.

There are some pitfalls with this scheme. Take the example of fig. 2. The stack bottom contains the local variable `a`. The instruction `istore a` will write a new value for `a` and will make a later use of this variable invalid. To avoid this we have to copy the local variable to a stack variable. An important decision is at which position the copy instruction should be inserted. Since there is a high number of `dup` instructions in Java programs (around 4%) and it is possible that a local variable resides in memory, the copy should be done with the `load` instruction. Since the stack is represented as a linked list only the destination stack has to be checked for occurrences of the offending variable and these occurrences are replaced by a stack variable.

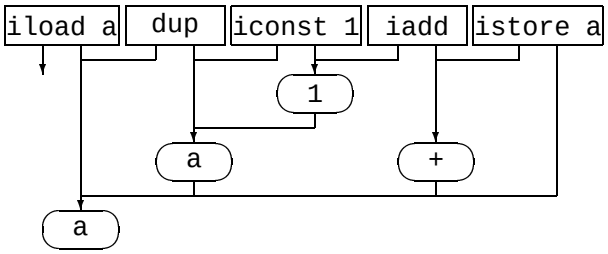

Figure 2. anti dependence

To answer the question of how often this could happen and how expensive the stack search is, we analyzed again the javac compiler. In more than 98% of the cases the stack is empty (see table 2). In only 0.2% of the cases the stack depth is higher than 1 and the biggest stack depth is 3.

stack depth	0	1	2	3	>3
occurrences	2167	31	1	3	0

Table 2. distribution of `store` stack depth

To avoid copy instructions when executing a `store` it is necessary to connect the creation of a value with the store which consumes it. In that case a `store` not only can conflict with copies of a local variable which result from `load` instructions before the creator of the value, but also with `load` and `store` instructions which exist between the creation of value and the `store`. In fig. 3 the `iload a` instruction conflicts with the `istore a` instruction.

The anti dependences are detected by checking the stack locations of the previous instructions for conflicts. Since the stack locations are allocated as one big array just the stack elements which have a higher index than the current stack element have to be checked. Table 3 gives the distribution of the distance between the creation of the value and the corresponding store. In 86% of the cases the distance is one.

The output dependences are checked by storing the instruction number of the last store in each local variable. If

chain length	1	2	3	4	5	6	7	8	9	>9
occurrences	1892	62	23	62	30	11	41	9	7	65

Table 3. distribution of creator-store distances

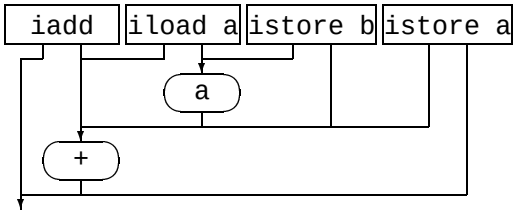

Figure 3. anti dependence

a store conflicts due to dependences the creator places the value in a stack register. Additional dependences arise because of exceptions. The exception mechanism in Java is precise. Therefore `store` instructions are not allowed to be executed before an exception raising instruction. This is checked easily by remembering the last instruction which could raise an exception. In methods which contain no exception handler this conflict can be safely ignored because no exception handler can have access to these variables.

3.4 Register allocation

Expensive register allocation algorithms are neither suitable nor necessary. The javac compiler does a coloring of the local variables and assigns the same number to variables which are not active at the same time. The stack variables have implicitly encoded their live ranges. When a value is pushed, the live range start. When a value is popped, the live range ends.

Complications arise only with stack manipulation instructions like `dup` and `swap`. We flag therefore the first creation of a stack variable and mark a duplicated one as a copy. The register used for this variable can be reused only after the last copy is popped.

During stack analysis stack variables are marked which have to survive a method invocation. These stack variables and local variables are assigned callee saved registers. If there are not enough registers available, these variables are allocated in memory.

Efficient implementation of method invocation is crucial to the performance of Java. Therefore, we preallocate the argument registers and the return value in a similar way as we handle store instructions. Input arguments (in Java input arguments are the first variables) for leaf procedures (and input arguments for processors with register windows) are preassigned, too.

3.5 Instruction combining

Together with stack analysis we combine constant loading instructions with selected instructions which are following immediately. In the class of combinable instructions are add, subtract, multiply and divide instructions, logical and shift instructions and compare/branch instructions. During code generation the constant is checked if it lies in the range for immediate operands of the target architecture and appropriate code is generated.

The old translator expanded some complex instructions into multiple instructions to avoid complex instructions in the later passes. One of such instructions was the expansion of the `lookup` instruction in a series of load constant and compare and branch instructions. Since the constants are usually quite small this unnecessarily increased the size of the intermediate representation and the final code. The new compiler delays the expansion into multiple instructions to the code generation pass which reduces all representations and speeds up the compilation.

3.6 Example

Fig. 4 shows the intermediate representation and stack information as produced by the compiler for debugging purposes. The `Local Table` gives the types and register assignment for the local variables. The Java compiler reuses the same local variable slot for different local variables if there life ranges do not overlap. In this example the variable slot 3 is even used for local variables of different types (integer and address). The JIT-compiler assigned the saved register 12 to this variable.

One interface register is used in this example entering the basic block with label `L004`. At the entry of the basic block the interface register has to be copied to the argument register `A00`. This is one of the rare cases where a more sophisticated coalescing algorithm could have allocated an argument register for the interface.

The combining of a constant with an arithmetic instruction happens at instruction 2 and 3. Since the instructions are allocated in an array the empty slot has to be filled with a `NOP` instruction. The `ADDCONSTANT` instruction already has the local variable `L02` as destination, an information which comes from the later `ISTORE` at number 4. Similarly the `INVOKESTATIC` at number 31 has marked all its operands as arguments. In this example all copies (beside the one to the interface register) have been eliminated.

	sieve	JavaLex	javac	espresso	Toba	java_cup
run time on 21164A 600MHz (in seconds)						
CACAO old total	1.120	0.720	1.336	0.858	1.208	0.398
load	0.040	0.067	0.224	0.141	0.068	0.077
compile	0.022	0.116	0.343	0.235	0.139	0.196
run	1.058	0.537	0.769	0.481	1.000	0.125
CACAO new total	0.902	0.522	0.925	0.614	0.982	0.218
load	0.040	0.067	0.223	0.141	0.068	0.077
compile	0.004	0.018	0.060	0.050	0.019	0.026
run	0.858	0.437	0.642	0.423	0.895	0.115
speedup						
speedup total old/new	1.24	1.38	1.44	1.40	1.23	1.82
speedup compile old/new	7.33	6.44	5.62	4.70	7.31	7.53
number of compiled JavaVM instructions						
	2514	13412	34759	27281	14430	17489
number of cycles per compiled JavaVM instruction						
	955	805	1035	1099	790	891

Table 4. comparison between old and new compiler

3.7 Complexity of the algorithm

The complexity of the algorithm is mostly linear with respect to the number of instructions and the number of local variables plus the number of stack slots. There are only a small number of spots where it is not linear.

- At the begin of a basic block the stack has to be copied to separate the stacks of different basic blocks. Table 1 shows that the stack at the boundary of a basic block is in most cases zero. Therefore, this copying does not influence the linear performance of the algorithm.
- A store has to check for a later use of the same variable. Table 2 shows that this is not a problem, too.
- A store additionally has to check for the previous use of the same variable between creation of the value and the store. The distances between the creation and the use are small (in most case only 1) as shown by table 3.

Compiling javac 29% of the compile time are spent in parsing and basic block determination, 18% in stack analysis, 16% in register allocation and 37% in machine code generation.

4 Results

To evaluate the differences between the old and the new compiler we used six different programs: sieve is a Java implementation of the well known prime number generation program, JavaLex is a scanner generator, javac is the Java compiler from sun, espresso is a compiler for an enhanced Java dialect, Toba is a system which translates Java class files to C and java_cup is a parser generator. As input data for javac, espresso and Toba we used all source files of Toba (18 files).

Table 4 shows the total run time, the load time, the compile time and the run time for the old and the new system on an Alpha workstation with a 600Mhz 21164a processor. The new compiler is between 5 and 7 times faster than the old compiler. The new system also has some improvements in the code generation and uses a hardware null pointer check ([11]). Both improvements together speed up the new sytem between 23% and 82%. On average only 800 to 1100 cycles are needed to compile one JavaVM instruction. A profiler which assumes that all memory accesses go to the first level cache computed 423 cycles per compiled JavaVM instruction.

To evaluate the performance of CACAO we compared it with Sun's JDK and with kaffe version 0.8 (see section 1.1). We also got access to a beta version of Digitals JIT compiler. Due to problems with the monitor implementation ([11]) this compiler gives very bad results for javac and similar programs (three times slower than the JDK), but produced effiecient code for the sieve benchmark.

Table 5 gives the run times for all these systems on an ALPHA workstation with a 300MHz 21064a processor. The CACAO system is between 3 and 5 times faster than the kaffe system and twice as fast as the Digital JIT compiler.

	sieve	JavaLex	javac	espresso	Toba	java_cup
run time on 21064A 300MHz (in seconds)						
JDK	83.2	29.8	18.5	8.7	32.1	3.5
Digital JIT	6.27	84.4	47.6	14.1	-	9.8
kaffe	9.14	9.9	17.8	12.5	-	2.98
CACAO old	4.80	2.65	4.74	3.17	4.58	1.52
CACAO new	3.87	1.92	3.29	2.26	3.72	0.83
speedup with respect to interpreter						
speedup JDK/DEC-JIT	13.3	0.35	0.38	0.48	-	0.36
speedup JDK/kaffe	9.10	3.01	1.04	0.7	-	1.17
speedup JDK/CACAO old	17.3	11.24	3.90	2.74	7.01	2.30
speedup JDK/CACAO new	21.5	15.52	5.62	3.85	8.62	4.22

Table 5. comparison between JDK, Digital JIT, kaffe and CACAO

5 Conclusion and further work

We presented an efficient algorithm for translating the JavaVM to efficient native code for RISC processors. This new algorithm is about seven times faster than the compiler used before. CACAO executes Java programs up to 5 times faster than other JIT compilers. CACAO can be obtained via the world wide web at `http://www.complang.tuwien.ac.at/java/cacao/`. Currently additional code generators for the Sparc, MIPS and PowerPC processors are being developed. We are working to integrate bound check removal, instruction scheduling and method inlining.

Acknowledgement

We express our thanks to Manfred Brockhaus, David Gregg and Anton Ertl for their comments on earlier drafts of this paper.

References

[1] A.-R. Adl-Tabatabai, M. Ciernak, G.-Y. Lueh, V. M. Parikh, and J. M. Stichnoth. Fast, effective code generation in a just-in-time Java compiler. In *Conference on Programming Language Design and Implementation*, volume 33(6) of *SIGPLAN*, page to appear, Montreal, 1998. ACM.

[2] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, 1996.

[3] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control flow graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[4] K. Ebcioğlu, E. Altman, and E. Hokenek. A Java ILP machine based on fast dynamic compilation. In *MASCOTS'97 - International Workshop on Security and Efficiency Aspects of Java*, 1997.

[5] M. A. Ertl. *Implementation of Stack-Based Languages on Register Machines*. PhD thesis, Technische Universität Wien, April 1996.

[6] M. A. Ertl and M. Maierhofer. Translating Forth to native C. In *EuroForth '95*, 1995.

[7] M. A. Ertl and C. Pirker. The structure of a Forth native code compiler. In *EuroForth '97 Conference Proceedings*, pages 107–116, 1997.

[8] K. J. Gough. Multi-language, multi-target compiler development: Evolution of the Gardens Point compiler project. In H. Mössenböck, editor, *JMLC'97 –Joint Modular Languages Conference*, Linz, 1997. LNCS 1204.

[9] C.-H. A. Hsieh, J. C. Gyllenhaal, and W. W. Hwu. Java bytecode to native code translation: The Caffeine prototype and preliminary results. In *29th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'29)*, 1996.

[10] A. Krall and R. Grafl. CACAO – a 64 bit JavaVM just-in-time compiler. *Concurrency: Practice and Experience*, 9(11):1017–1030, 1997.

[11] A. Krall and M. Probst. Monitors and exceptions: How to implement Java efficiently. In S. Hassanzadeh and K. Schauser, editors, *ACM 1998 Workshop on Java for High-Performance Computing*, pages 15–24, Palo Alto, March 1998. ACM.

[12] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[13] S. Pemberton and M. C. Daniels. *Pascal Implementation, The P4 Compiler*. Ellis Horwood, 1982.

[14] T. B. Steel. A first version of UNCOL. In *Proceedings of the Western Joint IRE-AIEE-ACM Computer Conference*, pages 371 – 377, 1961.

[15] A. S. Tanenbaum, M. F. Kaashoek, K. G. Langendoen, and C. J. H. Jacobs. The design of very fast portable compilers. *ACM SIGPLAN Notices*, 24(11):125–131, Nov. 1989.

[16] A. S. Tanenbaum, H. van Staveren, E. G. Keizer, and J. W. Stevenson. A practical tool kit for making portable compilers. *Communications of the ACM*, 16(9):654–660, September 1983.

```
java.io.ByteArrayOutputStream.write (int)void

Local Table:
     0:                (addr) S15
     1:    (int) S14
     2:    (int) S13
     3:    (int) S12   (addr) S12

Interface Table:
     0:    (int) T24

[                 L00]        0  ALOAD         0
[                 T23]        1  GETFIELD      16
[                 L02]        2  IADDCONST     1
[                 L02]        3  NOP
[                    ]        4  ISTORE        2
[                 L02]        5  ILOAD         2
[             L00 L02]        6  ALOAD         0
[             T23 L02]        7  GETFIELD      8
[             T23 L02]        8  ARRAYLENGTH
[                    ]        9  IF_ICMPLE     L005
                                ...............

[                    ]       18  IF_ICMPLT     L003
[                    ] L002:
[                 I00]       19  ILOAD         3
[                 I00]       20  GOTO          L004
[                    ] L003:
[                 I00]       21  ILOAD         2
[                 A00] L004:
[                 L03]       22  BUILTIN1      newarray_byte
[                    ]       23  ASTORE        3
[                 L00]       24  ALOAD         0
[                 A00]       25  GETFIELD      8
[             A01 A00]       26  ICONST        0
[         A02 A01 A00]       27  ALOAD         3
[     A03 A02 A01 A00]       28  ICONST        0
[ L00 A03 A02 A01 A00]       29  ALOAD         0
[ A04 A03 A02 A01 A00]       30  GETFIELD      16
[                    ]       31  INVOKESTATIC  java/lang/System.arraycopy
[                 L00]       32  ALOAD         0
[             L03 L00]       33  ALOAD         3
[                    ]       34  PUTFIELD      8
[                    ] L005:
                                ...............

[                    ]       45  RETURN
```

Figure 4. Example: intermediate instructions and stack contents