Advanced Material: Compiling C and Interpreting Java

This section gives a brief overview of how the C compiler works and how Java is executed. Because the compiler will significantly affect the performance of a computer, understanding compiler technology today is critical to understanding performance. Keep in mind that the subject of compiler construction is usually taught in a one- or two-semester course, so our introduction will necessarily only touch on the basics.

The second part of this section, starting on page 2.15-15, is for readers interested in seeing how an objected-oriented language like Java executes on the LEGv8 architecture. It shows the Java bytecodes used for interpretation and the LEGv8 code for the Java version of some of the C segments in prior sections, including Bubble Sort. It covers both the Java virtual machine and just-in-time (JIT) compilers.

Compiling C

This first part of the section introduces the internal **anatomy** of a compiler. To start, Figure 2.15.1 shows the structure of recent compilers, and we describe the optimizations in the order of the passes of that structure.

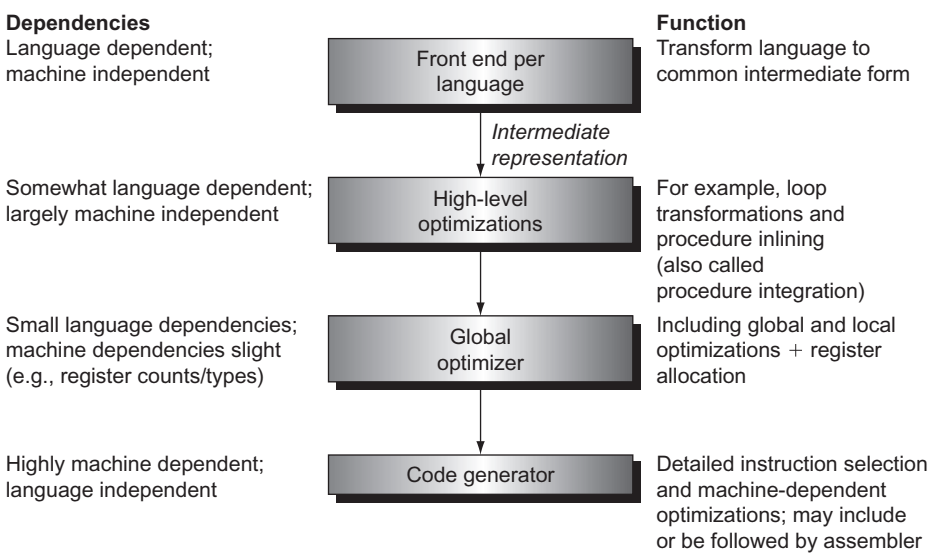


FIGURE 2.15.1 The structure of a modern optimizing compiler consists of a number of passes or phases. Logically, each pass can be thought of as running to completion before the next occurs. In practice, some passes may handle one procedure at a time, essentially interleaving with another pass.

To illustrate the concepts in this part of this section, we will use the C version of a *while* loop from page 95:

```
while (save[i] == k)
  i += 1;
```

The Front End

The function of the front end is to read in a source program; check the syntax and semantics; and translate the source program to an intermediate form that interprets most of the language-specific operation of the program. As we will see, intermediate forms are usually simple, and some are, in fact, similar to the Java bytecodes (see Figure 2.15.8).

The front end is typically broken into four separate functions:

1. *Scanning* reads in individual characters and creates a string of tokens. Examples of *tokens* are reserved words, names, operators, and punctuation symbols. In the above example, the token sequence is `while`, `(`, `save`, `[`, `i`, `]`, `==`, `k`, `)`, `i`, `+=`, `1`. A word like `while` is recognized as a reserved word in C, but `save`, `i`, and `j` are recognized as names, and `1` is recognized as a number.

2. *Parsing* takes the token stream, ensures the syntax is correct, and produces an *abstract syntax tree*, which is a representation of the syntactic structure of the program. Figure 2.15.2 shows what the abstract syntax tree might look like for this program fragment.

3. *Semantic analysis* takes the abstract syntax tree and checks the program for semantic correctness. Semantic checks normally ensure that variables and types are properly declared and that the types of operators and objects match, a step called *type checking*. During this process, a symbol table representing all the named objects—classes, variables, and functions—is usually created and used to type-check the program.

4. *Generation of the intermediate representation* (IR) takes the symbol table and the abstract syntax tree and generates the intermediate representation that is the output of the front end. Intermediate representations usually use simple operations on a small set of primitive types, such as integers, characters, and reals. Java bytecodes represent one type of intermediate form. In modern compilers, the most common intermediate form looks much like the LEGv8 instruction set but with an infinite number of virtual registers; later, we describe how to map these virtual registers to a finite set of real registers. Figure 2.15.3 shows how our example might be represented in such an intermediate form.

The intermediate form specifies the functionality of the program in a manner independent of the original source. After this front end has created the intermediate form, the remaining passes are largely language independent.

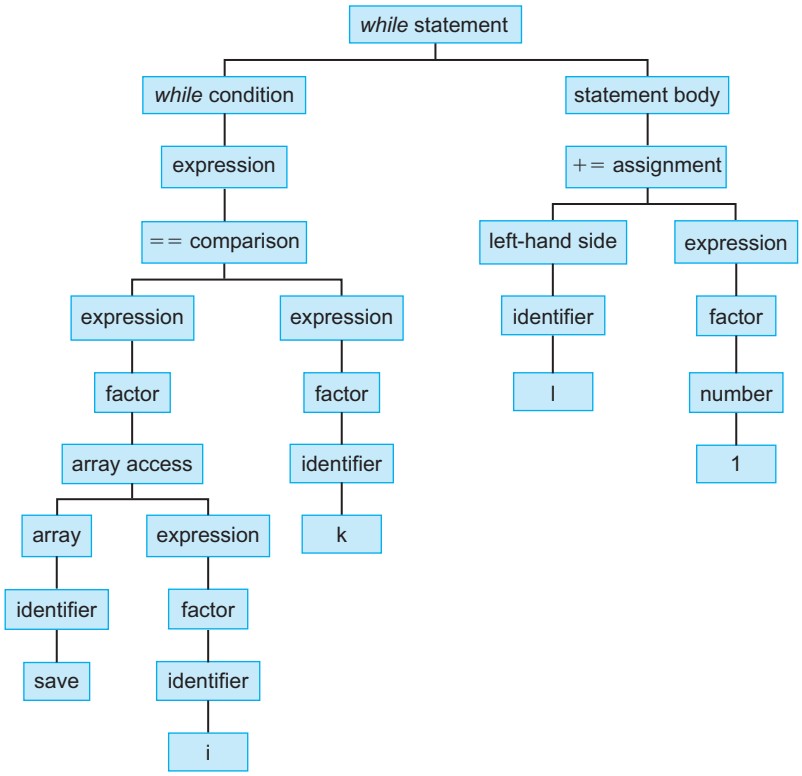


FIGURE 2.15.2 An abstract syntax tree for the *while* example. The roots of the tree consist of the informational tokens such as numbers and names. Long chains of straight-line descendents are often omitted in constructing the tree.

High-Level Optimizations

High-level optimizations are transformations that are done at something close to the source level.

The most common high-level transformation is probably *procedure inlining*, which replaces a call to a function by the body of the function, substituting the caller's arguments for the procedure's parameters. Other high-level optimizations involve loop transformations that can reduce loop overhead, improve memory access, and exploit the hardware more effectively. For example, in loops that execute many iterations, such as those traditionally controlled by a *for* statement, the optimization of **loop-unrolling** is often useful. Loop-unrolling involves taking a loop, replicating the body multiple times, and executing the transformed loop fewer times. Loop-unrolling reduces the loop overhead and provides opportunities for many other optimizations. Other types of high-level transformations include

loop-unrolling
A technique to get more performance from loops that access arrays, in which multiple copies of the loop body are made and instructions from different iterations are scheduled together.

```
      # comments are written like this--source code often included
      # while (save[i] == k)
loop: LDAR1,save#loads the starting address of save into
      R1
      LDUR R2,i
      MUL R3,R2,8 # Multiply R2 by 8
      ADD R4,R3,R1
      LDUR R5,[R4,0] # load save[i]
      LDUR R6,k
      CMP R5,R6
      B.NE endwhileloop
      # i += 1
      LDUR R6, i
      ADDI R7,R6,1 # increment
      STUR R7,i
      B loop # next iteration
      endwhileloop:
```

FIGURE 2.15.3 The *while* loop example is shown using a typical intermediate representation. In practice, the names `save`, `i`, and `k` would be replaced by some sort of address, such as a reference to either the local stack pointer or a global pointer, and an offset, similar to the way `save[i]` is accessed. Note that the format of the LEGv8 instructions is different from the rest of the chapter, because they represent intermediate representations here using `RXX` notation for registers.

sophisticated loop transformations such as interchanging nested loops and blocking loops to obtain better memory behavior; see Chapter 5 for examples.

Local and Global Optimizations

Within the pass dedicated to local and global optimization, three classes of optimization are performed:

1. *Local optimization* works within a single basic block. A local optimization pass is often run as a precursor and successor to global optimization to "clean up" the code before and after global optimization.
2. *Global optimization* works across multiple basic blocks; we will see an example of this shortly.
3. Global *register allocation* allocates variables to registers for regions of the code. Register allocation is crucial to getting good performance in modern processors.

Several optimizations are performed both locally and globally, including common subexpression elimination, constant propagation, copy propagation, dead store elimination, and strength reduction. Let's look at some simple examples of these optimizations.

Common subexpression elimination finds multiple instances of the same expression and replaces the second one by a reference to the first. Consider, for example, a code segment to add 4 to an array element:

```
x[i] = x[i] + 4
```

The address calculation for `x[i]` occurs twice and is identical since neither the starting address of `x` nor the value of `i` changes. Thus, the calculation can be reused. Let's look at the intermediate code for this fragment, since it allows several other optimizations to be performed. The unoptimized intermediate code is on the left. On the right is the optimized code, using common subexpression elimination to replace the second address calculation with the first. Note that the register allocation has not yet occurred, so the compiler is using virtual register numbers like `R100` here.

```
// x[i] + 4
LDA R100,x
LDUR R101,i
MUL R102,R101,8
ADD R103,R100,R102
LDUR R104, [R103, #0]
//
ADD R105, R104,4
LDUR R107,i
MULL R108,R107,8
ADD R109,R106,R107
STUR R105,[R109, #0]
```

```
// x[i] + 4
LDA R100,x
LDUR R101,i
LSL R102,R101,#3
ADD R103,R100,R102
LDUR R104, [R103, #0]
// value of x[i] is in R104
ADD R105, R104,4
STUR R105, [R103, #0]
```

If the same optimization were possible across two basic blocks, it would then be an instance of *global common subexpression elimination.*

Let's consider some of the other optimizations:

- *Strength reduction* replaces complex operations by simpler ones and can be applied to this code segment, replacing the MULT by a shift left.
- *Constant propagation* and its sibling *constant folding* find constants in code and propagate them, collapsing constant values whenever possible.
- *Copy propagation* propagates values that are simple copies, eliminating the need to reload values and possibly enabling other optimizations, such as common subexpression elimination.
- *Dead store elimination* finds stores to values that are not used again and eliminates the store; its "cousin" is *dead code elimination*, which finds unused code—code that cannot affect the result of the program—and eliminates it. With the heavy use of macros, templates, and the similar techniques designed to reuse code in high-level languages, dead code occurs surprisingly often.

Compilers must be *conservative*. The first task of a compiler is to produce correct code; its second task is usually to produce fast code, although other factors, such as code size, may sometimes be important as well. Code that is fast but incorrect—for any possible combination of inputs—is simply wrong. Thus, when we say a compiler is "conservative," we mean that it performs an optimization only if it knows with 100% certainty that, no matter what the inputs, the code will perform as the user wrote it. Since most compilers translate and optimize one function or procedure at a time, most compilers, especially at lower optimization levels, assume the worst about function calls and about their own parameters.

Understanding Program Performance

Programmers concerned about the performance of critical loops, especially in real-time or embedded applications, can find themselves staring at the assembly language produced by a compiler and wondering why the compiler failed to perform some global optimization or to allocate a variable to a register throughout a loop. The answer often lies in the dictate that the compiler be conservative. The opportunity for improving the code may seem obvious to the programmer, but then the programmer often has knowledge that the compiler does not have, such as the absence of aliasing between two pointers or the absence of side effects by a function call. The compiler may indeed be able to perform the transformation with a little help, which could eliminate the worst-case behavior that it must assume. This insight also illustrates an important observation: programmers who use pointers to try to improve performance in accessing variables, especially pointers to values on the stack that also have names as variables or as elements of arrays, are likely to disable many compiler optimizations. The result is that the lower-level pointer code may run no better, or perhaps even worse, than the higher-level code optimized by the compiler.

Global Code Optimizations

Many global code optimizations have the same aims as those used in the local case, including common subexpression elimination, constant propagation, copy propagation, and dead store and dead code elimination.

There are two other important global optimizations: code motion and induction variable elimination. Both are loop optimizations; that is, they are aimed at code in loops. *Code motion* finds code that is loop invariant: a particular piece of code computes the same value on every iteration of the loop and, hence, may be computed once outside the loop. *Induction variable elimination* is a combination of transformations that reduce overhead on indexing arrays, essentially replacing array indexing with pointer accesses. Rather than examine induction variable elimination in depth, we point the reader to Section 2.14, which compares the use of array indexing and pointers; for most loops, a modern optimizing compiler can perform the transformation from the more obvious array code to the faster pointer code.

Implementing Local Optimizations

Local optimizations are implemented on basic blocks by scanning the basic block in instruction execution order, looking for optimization opportunities. In the assignment statement example on page 2.15-6, the duplication of the entire address calculation is recognized by a series of sequential passes over the code. Here is how the process might proceed, including a description of the checks that are needed:

1. Determine that the two `LDA` operations return the same result by observing that the operand `x` is the same and that the value of its address has not been changed between the two `LDA` operations.
2. Replace all uses of `R106` in the basic block by `R101`.
3. Observe that `i` cannot change between the two `LDURs` that reference it. So replace all uses of `R107` with `R101`.
4. Observe that the `MUL` instructions now have the same input operands, so that `R108` may be replaced by `R102`.
5. Observe that now the two `ADD` instructions have identical input operands (`R100` and `R102`), so replace the `R109` with `R103`.
6. Use dead store code elimination to delete the second set of `LDA`, `LDUR`, `MUL`, and `ADD` instructions since their results are unused.

Throughout this process, we need to know when two instances of an operand have the same value. This is easy to determine when they refer to virtual registers, since our intermediate representation uses such registers only once, but the problem can be trickier when the operands are variables in memory, even though we are only considering references within a basic block.

It is reasonably easy for the compiler to make the common subexpression elimination determination in a conservative fashion in this case; as we will see in the next subsection, this is more difficult when branches intervene.

Implementing Global Optimizations

To understand the challenge of implementing global optimizations, let's consider a few examples:

- Consider the case of an opportunity for common subexpression elimination, say, of an IR statement like `ADD Rx, R20, R50`. To determine whether two such statements compute the same value, we must determine whether the values of `R20` and `R50` are identical in the two statements. In practice, this means that the values of `R20` and `R50` have not changed between the first statement and the second. For a single basic block, this is easy to decide; it is more difficult for a more complex program structure involving multiple basic blocks and branches.
- Consider the second `LDUR` of `i` into `R107` within the earlier example: how do we know whether its value is used again? If we consider only a single basic

block, and we know that all uses of R107 are within that block, it is easy to see. As optimization proceeds, however, common subexpression elimination and copy propagation may create other uses of a value. Determining that a value is unused and the code is dead is more difficult in the case of multiple basic blocks.

- Finally, consider the load of k in our loop, which is a candidate for code motion. In this simple example, we might argue that it is easy to see that k is not changed in the loop and is, hence, loop invariant. Imagine, however, a more complex loop with multiple nestings and *if* statements within the body. Determining that the load of k is loop invariant is harder in such a case.

The information we need to perform these global optimizations is similar: we need to know where each operand in an IR statement could have been changed or *defined* (use-definition information). The dual of this information is also needed: that is, finding all the uses of that changed operand (definition-use information). *Data flow analysis* obtains both types of information.

Global optimizations and data flow analysis operate on a *control flow graph*, where the nodes represent basic blocks and the arcs represent control flow between basic blocks. Figure 2.15.4 shows the control flow graph for our simple loop example, with one important transformation introduced. We describe the transformation in the caption, but see if you can discover it, and why it was done, on your own!

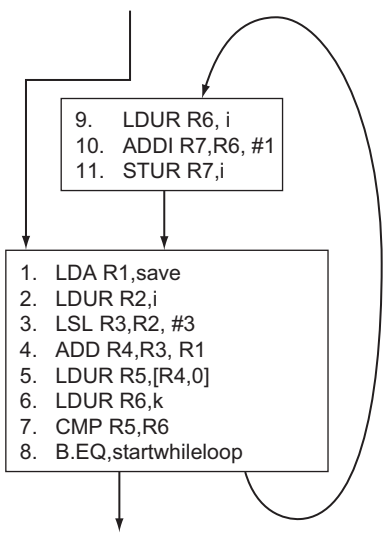


FIGURE 2.15.4 A control flow graph for the *while* loop example. Each node represents a basic block, which terminates with a branch or by sequential fall-through into another basic block that is also the target of a branch. The IR statements have been numbered for ease in referring to them. The important transformation performed was to move the *while* test and conditional branch to the end. This eliminates the unconditional branch that was formerly inside the loop and places it before the loop. This transformation is so important that many compilers do it during the generation of the IR. The MUL was also replaced with ("strength-reduced to") an SLL.

Suppose we have computed the use-definition information for the control flow graph in Figure 2.15.4. How does this information allow us to perform code motion? Consider IR statements number 1 and 6: in both cases, the use-definition information tells us that there are no definitions (changes) of the operands of these statements within the loop. Thus, these IR statements can be moved outside the loop. Notice that if the `LDA` of `save` and the `LDUR` of `k` are executed once, just prior to the loop entrance, the computational effect is the same, but the program now runs faster since these two statements are outside the loop. In contrast, consider IR statement 2, which loads the value of `i`. The definitions of `i` that affect this statement are both outside the loop, where `i` is initially defined, and inside the loop in statement 10 where it is stored. Hence, this statement is not loop invariant.

Figure 2.15.5 shows the code after performing both code motion and induction variable elimination, which simplifies the address calculation. The variable `i` can still be register allocated, eliminating the need to load and store it every time, and we will see how this is done in the next subsection.

Before we turn to register allocation, we need to mention a caveat that also illustrates the complexity and difficulty of optimizers. Remember that the compiler must be cautious. To be conservative, a compiler must consider the following question: Is there *any way* that the variable `k` could possibly ever change in this loop? Unfortunately, there is one way. Suppose that the variable `k` and the variable `i` actually refer to the same memory location, which could happen if they were accessed by pointers or reference parameters.

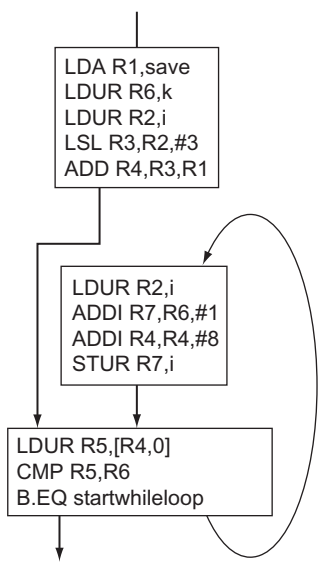


FIGURE 2.15.5 The control flow graph showing the representation of the *while* loop example after code motion and induction variable elimination. The number of instructions in the inner loop has been reduced from 11 to 7.

I am sure that many readers are saying, "Well, that would certainly be a stupid piece of code!" Alas, this response is not open to the compiler, which must translate the code as it is written. Recall too that the aliasing information must also be conservative; thus, compilers often find themselves negating optimization opportunities because of a possible alias that exists in one place in the code or because of incomplete information about aliasing.

Register Allocation

Register allocation is perhaps the most important optimization for modern load-store architectures. Eliminating a load or a store gets rid of an instruction. Furthermore, register allocation enhances the value of other optimizations, such as common subexpression elimination. Fortunately, the trend toward larger register counts in modern architectures has made register allocation simpler and more effective. Register allocation is done on both a local basis and a global basis, that is, across multiple basic blocks but within a single function. Local register allocation is usually done late in compilation, as the final code is generated. Our focus here is on the more challenging and more opportunistic global register allocation.

Modern global register allocation uses a region-based approach, where a region (sometimes called a *live range*) represents a section of code during which a particular variable could be allocated to a particular register. How is a region selected? The process is iterative:

1. Choose a definition (change) of a variable in a given basic block; add that block to the region.
2. Find any uses of that definition, which is a data flow analysis problem; add any basic blocks that contain such uses, as well as any basic block that the value passes through to reach a use, to the region.
3. Find any other definitions that also can affect a use found in the previous step and add the basic blocks containing those definitions, as well as the blocks the definitions pass through to reach a use, to the region.
4. Repeat steps 2 and 3 using the definitions discovered in step 3 until convergence.

The set of basic blocks found by this technique has a special property: if the designated variable is allocated to a register in all these basic blocks, then there is no need for loading and storing the variable.

Modern global register allocators start by constructing the regions for every virtual register in a function. Once the regions are constructed, the key question is how to allocate a register to each region: the challenge is that certain regions overlap and may not use the same register. Regions that do not overlap (i.e., share no common basic blocks) can share the same register. One way to record the interference among regions is with an *interference graph*, where each node represents a region, and the arcs between nodes represent that the regions have some basic blocks in common.

Once an interference graph has been constructed, the problem of allocating registers is equivalent to a famous problem called *graph coloring:* find a color for each node in a graph such that no two adjacent nodes have the same color. If the number of colors equals the number of registers, then coloring an interference graph is equivalent to allocating a register for each region! This insight was the initial motivation for the allocation method now known as region-based allocation, but originally called the graph-coloring approach. Figure 2.15.6 shows the flow graph representation of the *while* loop example after register allocation.

What happens if the graph cannot be colored using the number of registers available? The allocator must spill registers until it can complete the coloring. By doing the coloring based on a priority function that takes into account the number of memory references saved and the cost of tying up the register, the allocator attempts to avoid spilling for the most important candidates.

Spilling is equivalent to splitting up a region (or live range); if the region is split, fewer other regions will interfere with the two separate nodes representing the original region. A process of splitting regions and successive coloring is used to allow the allocation process to complete, at which point all candidates will have been allocated a register. Of course, whenever a region is split, loads and stores must be introduced to get the value from memory or to store it there. The location chosen to split a region must balance the cost of the loads and stores that must be introduced against the advantage of freeing up a register and reducing the number of interferences.

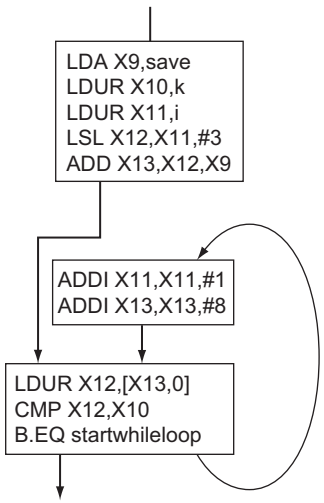


FIGURE 2.15.6 The control flow graph showing the representation of the *while* loop example after code motion and induction variable elimination and register allocation, using the LEGv8 register names. The number of IR statements in the inner loop has now dropped to only five from seven before register allocation and 11 before any global optimizations. The value of `i` resides in `X11` at the end of the loop and may need to be stored eventually to maintain the program semantics. If `i` were unused after the loop, not only could the store be avoided, but also the increment inside the loop could be eliminated!

Modern register allocators are incredibly effective in using the large register counts available in modern processors. In many programs, the effectiveness of register allocation is limited not by the availability of registers but by the possibilities of aliasing that cause the compiler to be conservative in its choice of candidates.

Code Generation

The final steps of the compiler are code generation and assembly. Most compilers do not use a stand-alone assembler that accepts assembly language source code; to save time, they instead perform most of the same functions: filling in symbolic values and generating the binary code as the last stage of code generation.

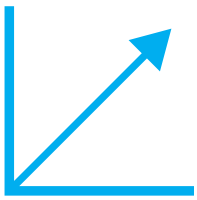
In modern processors, code generation is reasonably straightforward, since the simple architectures make the choice of instruction relatively obvious. Code generation is more complex for the more complicated architectures, such as the x86, since multiple IR instructions may collapse into a single machine instruction. In modern compilers, this compilation process uses pattern matching with either a tree-based pattern matcher or a pattern matcher driven by a parser.

During code generation, the final stages of machine-dependent optimization are also performed. These include some constant folding optimizations, as well as localized instruction scheduling (see Chapter 4).

Optimization Summary

Figure 2.15.7 gives examples of typical optimizations, and the last column indicates where the optimization is performed in the gcc compiler. It is sometimes difficult to separate some of the simpler optimizations—local and processor-dependent optimizations—from transformations done in the code generator, and some optimizations are done multiple times, especially local optimizations, which may be performed before and after global optimization as well as during code generation.

Hardware/ Software Interface



Today, essentially all programming for desktop and server applications is done in high-level languages, as is most programming for embedded applications. This development means that since most instructions executed are the output of a compiler, an instruction set architecture is mainly a compiler target. With **Moore's Law** comes the temptation of adding sophisticated operations in an instruction set. The challenge is that they may not exactly match what the compiler needs to produce or may be so general that they aren't fast. For example, consider special loop instructions found in some computers. Suppose that instead of decrementing by one, the compiler wanted to increment by four, or instead of branching on not equal zero, the compiler wanted to branch if the index was less than or equal to the limit. The loop instruction may be a mismatch. When faced with such objections,

the instruction set designer might next generalize the operation, adding another operand to specify the increment and perhaps an option on which branch condition to use. Then the danger is that a common case, say, incrementing by one, will be slower than a sequence of simple operations.

Elaboration Some more sophisticated compilers, and many research compilers, use an analysis technique called *interprocedural analysis* to obtain more information about functions and how they are called. Interprocedural analysis attempts to discover what properties remain true across a function call. For example, we might discover that a function call can never change any global variables, which might be useful in optimizing a loop that calls such a function. Such information is called *may-information* or *flow-insensitive information* and can be obtained reasonably efficiently, although analyzing a call to a function F requires analyzing all the functions that F calls, which makes the process somewhat time consuming for large programs. A more costly property to discover is that a function *must* always change some variable; such information is called *must-information* or *flow-sensitive information*. Recall the dictate to be conservative: may-information can never be used as must-information—just because a function *may* change a variable does not mean that it *must* change it. It is conservative, however, to use the negation of may-information, so the compiler can rely on the fact that a function *will* never change a variable in optimizations around the call site of that function.

Optimization name	Explanation	gcc level
High level	*At or near the source level; processor independent*	
Procedure integration	Replace procedure call by procedure body	O3
Local	*Within straight-line code*	
Common subexpression elimination	Replace two instances of the same computation by single copy	O1
Constant propagation	Replace all instances of a variable that is assigned a constant with the constant	O1
Stack height reduction	Rearrange expression tree to minimize resources needed for expression evaluation	O1
Global	*Across a branch*	
Global common subexpression elimination	Same as local, but this version crosses branches	O2
Copy propagation	Replace all instances of a variable *A* that has been assigned *X* (i.e., *A* = *X*) with *X*	O2
Code motion	Remove code from a loop that computes the same value each iteration of the loop	O2
Induction variable elimination	Simplify/eliminate array addressing calculations within loops	O2
Processor dependent	*Depends on processor knowledge*	
Strength reduction	Many examples; replace multiply by a constant with shifts	O1
Pipeline scheduling	Reorder instructions to improve pipeline performance	O1
Branch offset optimization	Choose the shortest branch displacement that reaches target	O1

FIGURE 2.15.7 Major types of optimizations and explanation of each class. The third column shows when these occur at different levels of optimization in gcc. The GNU organization calls the three optimization levels medium (O1), full (O2), and full with integration of small procedures (O3).

One of the most important uses of interprocedural analysis is to obtain so-called alias information. An *alias* occurs when two names may designate the same variable. For example, it is quite helpful to know that two pointers passed to a function may never designate the same variable. Alias information is usually flow-insensitive and must be used conservatively.

Interpreting Java

This second part of the section is for readers interested in seeing how an **object-oriented language** like Java executes on an LEGv8 architecture. It shows the Java bytecodes used for interpretation and the LEGv8 code for the Java version of some of the C segments in prior sections, including Bubble Sort.

object-oriented language
A programming language that is oriented around objects rather than actions, or data versus logic.

Let's quickly review the Java lingo to make sure we are all on the same page. The big idea of object-oriented programming is for programmers to think in terms of abstract objects, and operations are associated with each *type* of object. New types can often be thought of as refinements to existing types, and so the new types use some operations for the existing types without change. The hope is that the programmer thinks at a higher level, and that code can be reused more readily if the programmer implements the common operations on many different types.

This different perspective led to a different set of terms. The type of an object is a *class*, which is the definition of a new data type together with the operations that are defined to work on that data type. A particular object is then an *instance* of a class, and creating an object from a class is called *instantiation*. The operations in a class are called *methods*, which are similar to C procedures. Rather than call a procedure as in C, you *invoke* a method in Java. The other members of a class are *fields*, which correspond to variables in C. Variables inside objects are called *instance fields*. Rather than access a structure with a pointer, Java uses an *object reference* to access an object. The syntax for method invocation is `x.y`, where `x` is an object reference and `y` is the method name.

The parent–child relationship between older and newer classes is captured by the verb "extends": a child class *extends* (or subclasses) a parent class. The child class typically will redefine some of the methods found in the parent to match the new data type. Some methods work fine, and the child class *inherits* those methods.

To reduce the number of errors associated with pointers and explicit memory deallocation, Java automatically frees unused storage, using a separate garbage collector that frees memory when it is full. Hence, `new` creates a new instance of a dynamic object on the heap, but there is no `free` in Java. Java also requires array bounds to be checked at runtime to catch another class of errors that can occur in C programs.

Interpretation

As mentioned before, Java programs are distributed as Java bytecodes, and the Java Virtual Machine (JVM) executes Java byte codes. The JVM understands a binary format called the *class file* format. A class file is a stream of bytes for a single class, containing a table of valid methods with their bytecodes, a pool of constants that acts in part as a symbol table, and other information such as the parent class of this class.

When the JVM is first started, it looks for the class method `main`. To start any Java class, the JVM dynamically loads, links, and initializes a class. The JVM loads a class by first finding the binary representation of the proper class (class file) and then creating a class from that binary representation. Linking combines the class into the runtime state of the JVM so that it can be executed. Finally, it executes the class initialization method that is included in every class.

Figure 2.15.8 shows Java bytecodes and their corresponding LEGv8 instructions, illustrating five major differences between the two:

1. To simplify compilation, Java uses a stack instead of registers for operands. Operands are pushed on the stack, operated on, and then popped off the stack.
2. The designers of the JVM were concerned about code size, so bytecodes vary in length between one and five bytes, versus the four-byte, fixed-size LEGv8 instructions. To save space, the JVM even has redundant instructions of varying lengths whose only difference is size of the immediate. This decision illustrates a code size variation of our third design principle: make the common case *small*.
3. The JVM has safety features embedded in the architecture. For example, array data transfer instructions check to be sure that the first operand is a reference and that the second index operand is within bounds.
4. To allow garbage collectors to find all live pointers, the JVM uses different instructions to operate on addresses versus integers so that the JVM can know what operands contain addresses. LEGv8 generally lumps integers and addresses together.
5. Finally, unlike LEGv8, Java bytecodes include Java-specific instructions that perform complex operations, like allocating an array on the heap or invoking a method.

Category	Operation	Java bytecode	Size (bits)	ARMv8 instr.	Meaning
Arithmetic	add	iadd	8	ADD	NOS=TOS+NOS; pop
	subtract	isub	8	SUB	NOS=TOS–NOS; pop
	increment	iinc I8a I8b	8	ADDI	Frame[I8a]= Frame[I8a] + I8b
Data transfer	load local integer/address	iload I8/aload I8	16	LDUR	TOS=Frame[I8]
	load local integer/address	iload_/aload_{0,1,2,3}	8	LDUR	TOS=Frame[{0,1,2,3}]
	store local integer/address	istore I8/astore I8	16	STUR	Frame[I8]=TOS; pop
	load integer/address from array	iaload/aaload	8	LDUR	NOS=*NOS[TOS]; pop
	store integer/address into array	iastore/aastore	8	STUR	*NNOS[NOS]=TOS; pop2
	load half from array	saload	8	LDURH	NOS=*NOS[TOS]; pop
	store half into array	sastore	8	STURH	*NNOS[NOS]=TOS; pop2
	load byte from array	baload	8	LDURB	NOS=*NOS[TOS]; pop
	store byte into array	bastore	8	STURB	*NNOS[NOS]=TOS; pop2
	load immediate	bipush I8, sipush I16	16, 24	ADDI	push; TOS=I8 or I16
	load immediate	iconst_{–1,0,1,2,3,4,5}	8	ADDI	push; TOS={–1,0,1,2,3,4,5}
Logical	and	iand	8	AND	NOS=TOS&NOS; pop
	or	ior	8	ORR	NOS=TOS|NOS; pop
	shift left	ishl	8	LSL	NOS=NOS << TOS; pop
	shift right	iushr	8	LSR	NOS=NOS >> TOS; pop
Conditional branch	branch on equal	if_icompeq I16	24	CBZ	if TOS == NOS, go to I16; pop2
	branch on not equal	if_icompne I16	24	CBNZ	if TOS != NOS, go to I16; pop2
	compare	if_icomp{lt,le,gt,ge} I16	24	CMP	if TOS {<,<=,>,>=} NOS, go to I16; pop2
Unconditional jump	jump	goto I16	24	B	go to I16
	return	ret, ireturn	8	BR	
	jump to subroutine	jsr I16	24	BL	go to I16; push; TOS=PC+3
Stack management	remove from stack	pop, pop2	8		pop, pop2
	duplicate on stack	dup	8		push; TOS=NOS
	swap top 2 positions on stack	swap	8		T=NOS; NOS=TOS; TOS=T
Safety check	check for null reference	ifnull I16, ifnotnull I16	24		if TOS {==,!=} null, go to I16
	get length of array	arraylength	8		push; TOS = length of array
	check if object a type	instanceof I16	24		TOS = 1 if TOS matches type of Const[I16]; TOS = 0 otherwise
Invocation	invoke method	invokevirtual I16	24		Invoke method in Const[I16], dispatching on type
Allocation	create new class instance	new I16	24		Allocate object type Const[I16] on heap
	create new array	newarray I16	24		Allocate array type Const[I16] on heap

FIGURE 2.15.8 Java bytecode architecture versus LEGv8. Although many bytecodes are simple, those in the last half-dozen rows above are complex and specific to Java. Bytecodes are one to five bytes in length, hence their name. The Java mnemonics uses the prefix `i` for 32-bit integer, `a` for reference (address), `s` for 16-bit integers (short), and `b` for 8-bit bytes. We use `I8` for an 8-bit constant and `I16` for a 16-bit constant. LEGv8 uses registers for operands, but the JVM uses a stack. The compiler knows the maximum size of the operand stack for each method and simply allocates space for it in the current frame. Here is the notation in the Meaning column: `TOS`: top of stack; `NOS`: next position below `TOS`; `NNOS`: next position below `NOS`; `pop`: remove `TOS`; `pop2`: remove `TOS` and `NOS`; and `push`: add a position to the stack. `*NOS` and `*NNOS` mean access the memory location pointed to by the address in the stack at those positions. `Const[]` refers to the runtime constant pool of a class created by the JVM, and `Frame[]` refers to the variables of the local method frame. The only missing LEGv8 instructions from Figure 2.1 (or LEGv8 pseudoinstructions) are `FOR`, `ANDI`, `ORRI`, `CMPI`, `MOVK` and `MOVZ`. The missing Java bytecodes from Figure 2.1 are a few arithmetic and logical operators, some tricky stack management, compares to 0 and branch, support for branch tables, type conversions, more variations of the complex, Java-specific instructions plus operations on floating-point data, 64-bit integers (longs), and 16-bit characters.

Compiling a *while* Loop in Java Using Bytecodes

EXAMPLE

Compile the *while* loop from page 95, this time using Java bytecodes:

```
while (save[i] == k)
  i += 1;
```

Assume that `i`, `k`, and `save` are the first three local variables. Show the addresses of the bytecodes. The LEGv8 version of the C loop in Figure 2.15.3 took seven instructions and 28 bytes. How big is the bytecode version?

ANSWER

The first step is to put the array reference in `save` on the stack:

```
0 aload_3 // Push local variable 3 (save[]) onto stack
```

This 1-byte instruction informs the JVM that an address in local variable 3 is being put on the stack. The 0 on the left of this instruction is the byte address of this first instruction; bytecodes for each method start at 0. The next step is to put the index on the stack:

```
1 iload_1 // Push local variable 1 (i) onto stack
```

Like the prior instruction, this 1-byte instruction is a short version of a more general instruction that takes 2 bytes to load a local variable onto the stack. The next instruction is to get the value from the array element:

```
2 iaload // Put array element (save[i]) onto stack
```

This 1-byte instruction checks the prior two operands, pops them off the stack, and then puts the value of the desired array element onto the new top of the stack. Next, we place `k` on the stack:

```
3 iload_2 // Push local variable 2 (k) onto stack
```

We are now ready for the *while* test:

```
4 if_icompne, Exit // Compare and exit if not equal
```

This 3-byte instruction compares the top two elements of the stack, pops them off the stack, and branches if they are not equal. We are finally prepared for the body of the loop:

```
7 iinc, 1, 1 // Increment local variable 1 by 1 (i+=1)
```

This unusual 3-byte instruction increments a local variable by 1 without using the operand stack, an optimization that again saves space. Finally, we return to the top of the loop with a 3-byte branch:

```
10 go to 0 // Go to top of Loop (byte address 0)
```

Thus, the bytecode version takes seven instructions and 13 bytes, almost half the size of the LEGv8 C code. (As before, we can optimize this code to branch less.)

Compiling for Java

Since Java is derived from C and Java has the same built-in types as C, the assignment statement examples in Sections 2.2 to 2.6 are the same in Java as they are in C. The same is true for the *if* statement example in Section 2.7.

The Java version of the *while* loop is different, however. The designers of C leave it up to the programmers to be sure that their code does not exceed the array bounds. The designers of Java wanted to catch array bound bugs, and thus require the compiler to check for such violations. To check bounds, the compiler needs to know what they are. Java includes an extra doubleword in every array that holds the upper bound. The lower bound is defined as 0.

Compiling a *while* Loop in Java

EXAMPLE

Modify the LEGv8 code for the *while* loop on page 95 to include the array bounds checks that are required by Java. Assume that the length of the array is located just before the first element of the array.

ANSWER

Let's assume that Java arrays reserved the first two doublewords of arrays before the data start. We'll see the use of the first doubleword soon, but the second doubleword has the array length. Before we enter the loop, let's load the length of the array into a temporary register:

```
LDUR  X11, [X25, #8]    // Temp reg X11 = length of
array save
```

Before we multiply `i` by 8, we must test to see if it's less than 0 or greater than the last element of the array. The first step is to check if `i` is less than 0:

```
Loop: CMP  X22, XZR // Test if i < 0
B.LT IndexOutOfBounds      // if i<0, goto Error
```

Since the array starts at 0, the index of the last array element is one less than the length of the array. Thus, the test of the upper array bound is to be sure that `i` is

less than the length of the array. Thus, the second step is to branch to an error if it's greater than or equal to `length`.

```
CMP  X22,X11     // compare i to length
B.GE,IndexOutOfBounds    //if i>=length, goto Error
```

The next two lines of the LEGv8 *while* loop are unchanged from the C version:

```
LSL  X10,X22, #3    // Temp reg X10 = 8 * i
ADD  X10, X10,X25   // X10 = address of save[i]
```

We need to account for the first 16 bytes of an array that are reserved in Java. We do that by changing the address field of the load from 0 to 16:

```
LDUR X9, [X10,#16]  // Temp reg X9 = save[i]
```

The rest of the LEGv8 code from the C *while* loop is fine as is:

```
SUB   X11,X9,X24    // X11 = save[i] - k
CBNZ  X11 Exit      // go to Exit if save[i] ≠ (X11≠0)
ADD   X22,X22,1     // i = i + 1
B     Loop          // go to Loop
Exit:
```

(See the exercises for an optimization of this sequence.)

Invoking Methods in Java

The compiler picks the appropriate method depending on the type of object. In a few cases, it is unambiguous, and the method can be invoked with no more overhead than a C procedure. In general, however, the compiler knows only that a given variable contains a pointer to an object that belongs to some subtype of a general class. Since it doesn't know at compile time which subclass the object is, and thus which method should be invoked, the compiler will generate code that first tests to be sure the pointer isn't null and then uses the code to load a pointer to a table with all the legal methods for that type. The first doubleword of the object has the method table address, which is why Java arrays reserve two doublewords. Let's say it's using the fifth method that was declared for that class. (The method order is the same for all subclasses.) The compiler then takes the fifth address from that table and invokes the method at that address.

The cost of object orientation in general is that method invocation takes five steps:

1. A conditional branch to be sure that the pointer to the object is valid;
2. A load to get the address of the table of available methods;
3. Another load to get the address of the proper method;

4. Placing a return address into the return register; and finally
5. A branch register to invoke the method.

A Sort Example in Java

Figure 2.15.9 shows the Java version of exchange sort. A simple difference is that there is no need to pass the length of the array as a separate parameter, since Java arrays include their length: `v.length` denotes the length of `v`.

A more significant difference is that Java methods are prepended with keywords not found in the C procedures. The `sort` method is declared `public static` while `swap` is declared `protected static`. **Public** means that `sort` can be invoked from any other method, while **protected** means `swap` can only be called by other methods within the same **package** and from methods within derived classes. A **static method** is another name for a class method—methods that perform class-wide operations and do not apply to an individual object. Static methods are essentially the same as C procedures.

This straightforward translation from C into static methods means there is no ambiguity on method invocation, and so it can be just as efficient as C. It also is limited to sorting integers, which means a different sort has to be written for each data type.

To demonstrate the object orientation of Java, Figure 2.15.10 shows the new version with the changes highlighted. First, we declare `v` to be of the type `Comparable` and replace `v[j] > v[j + 1]` with an invocation of `compareTo`. By changing `v` to this new class, we can use this code to `sort` many data types.

public A Java keyword that allows a method to be invoked by any other method.

protected A Java key word that restricts invocation of a method to other methods in that package.

package Basically a directory that contains a group of related classes.

static method A method that applies to the whole class rather than to an individual object. It is unrelated to static in C.

```
public class sort {
   public static void sort (int[] v) {
      for (int i = 0; i < v.length; i += 1) {
         for (int j = i - 1; j >= 0 && v[j] > v[j + 1]; j -= 1) {
               swap(v, j);
         }
   }

   protected static void swap(int[] v, int k) {
      int temp = v[k];
      v[k] = v[k+1];
      v[k+1] = temp;
   }}
```

FIGURE 2.15.9 An initial Java procedure that performs a sort on the array v. Changes from Figures 2.24 and 2.26 are highlighted.

```
public class sort {
   public static void sort (Comparable[] v) {
      for (int i = 0; i < v.length; i += 1) {
        for (int j = i - 1; j >= 0 && v[j].compareTo(v[j + 1]);j -= 1) {

                    swap(v, j);
          }
   }

   protected static void swap(Comparable[] v, int k) {
      Comparable temp = v[k];
      v[k] = v[k+1];
      v[k+1] = temp;
   }}
public class Comparable {
      public int(compareTo (int x)
      { return value - x; }
      public int value;
}
```

FIGURE 2.15.10 A revised Java procedure that sorts on the array v that can take on more types. Changes from Figure 2.15.9 are highlighted.

The method `compareTo` compares two elements and returns a value greater than 0 if the parameter is larger than the object, 0 if it is equal, and a negative number if it is smaller than the object. These two changes generalize the code so it can sort integers, characters, strings, and so on, if there are subclasses of `Comparable` with each of these types and if there is a version of `compareTo` for each type. For pedagogic purposes, we redefine the class `Comparable` and the method `compareTo` here to compare integers. The actual definition of `Comparable` in the Java library is considerably different.

Starting from the LEGv8 code that we generated for C, we show what changes we made to create the LEGv8 code for Java.

For `swap`, the only significant differences are that we must check to be sure the object reference is not null and that each array reference is within bounds. The first test checks that the address in the first parameter is not zero:

```
swap: CBZ  X0,NullPointer       // if X0==0,goto Error
```

Next, we load the length of v into a register and check that index k is OK.

```
LDUR X11,[X0,#8]        // Temp reg X11 = length of array v
CMP X1,XZR              // Compare k to 0
B.LT IndexOutOfBounds   // if k < 0, goto Error
CMP X1,X11              // Compare k to length
B.GE IndexOutOfBounds   // if k >= length, goto Error
```

This check is followed by a check that k+1 is within bounds.

```
ADDI X10,X1,#1          // Temp reg X10 = k+1
CMP X10,XZR             // Compare k+1 to 0
B.LT IndexOutOfBounds   // if k+1 < 0, goto Error
CMP X10,X11             // Compare k+1 to length
B.GE IndexOutOfBounds   // if k+1 >= length, goto Error
```

Figure 2.15.11 highlights the extra LEGv8 instructions in swap that a Java compiler might produce. We again must adjust the offset in the load and store to account for two doublewords reserved for the method table and length.

Figure 2.15.12 shows the method body for those new instructions for sort. (We can take the saving, restoring, and return from Figure 2.28.)

The first test is again to make sure the pointer to v is not null:

```
CBZ  X0,NullPointer    // if X0==0,goto Error
```

Bounds check

```
swap: CBZ   X0, NullPointer          # if X0==0,goto Error
      LDUR  X10, [X0,-8]             # Temp reg X10 = length of array v
      CMP   X1, XZR                  # Test if 1 if k < 0
      B.LT  IndexOutOfBounds         # if k < 0,goto Error
      CMP   X1, X10                  # Test if k >= length
      B.GT  IndexOutOfBounds         # if k >= length,goto Error
      ADDI  X9, X1, 1                # Temp reg X9 = k+1
      CMP   X9, XZR                  # Test if k+1 < 0
      B.LT  IndexOutOfBounds         # if k+1 < 0,goto Error
      CMP   X9, X10                  # Test if k+1 >= length
      B.GT  IndexOutOfBounds         # if k+1 >= length,goto Error
```

Method body

```
      LSL       X10, X1,3          # reg X10 = k * 8
      ADD       X10, X0,X10        # reg X10 = v + (k * 8)
                                   # reg X10 has the address of v[k]
      LDUR      X9, [X10,0]        # reg X9 (temp) = v[k]
      LDUR      X11,[X10,8]        # reg X11 = v[k + 1]
                                   # refers to next element of v
      STUR      X11,[X10,0]        # v[k] = reg X11
      STUR      X9, [X10,8]        # v[k+1] = reg X9 (temp)
```

Procedure return

```
      BR        X10                # return to calling routine
```

FIGURE 2.15.11 LEGv8 assembly code of the procedure Swap in Figure 2.24.

Method body				
Move parameters		MOV	X21, X0	# copy parameter X0 into X21
Test ptr null		CBZ	X0, NullPointer	# if X0==0,goto Error
Get array length		LDUR	X22, [X0,8]	# X22 = length of array v
Outer loop	for1tst:	MOV CMP B.GT	X19, XZR X19, X1t0, exit1	# i = 0 # test if X19 ≥ X1 (i ≥ n) # go to exit1 if X19 ≥ X1 (i ≥ n)
Inner loop start	for2tst:	SUBI CMP B.LT	X20, X19, 1 X20, XZR exit2	# j = i - 1 # Test if X20 < 0 (j < 0) # go to exit2 if X20 < 0 (j < 0)
Test if j too big		CMP B.GT	X20, X22 IndexOutOfBounds	# Test if j >= length # if j >= length, goto Error
Get v[j]		LSL ADD LDUR	X10, X20, 3 X11, X0, X10 X12, [X11,0]	# reg X10 = j * 8 # reg X11 = v + (j * 8) # reg X12 = v[j]
Test if j+1 < 0 or if j+1 too big		ADDI CMP B.LT CMP B.GT	X10, X20, 1 X10, XZR IndexOutOfBounds X10, X22 IndexOutOfBounds	# Temp reg X10 = j+1 # Test if j+1 < 0 # if j+1 < 0, goto Error # Test if j+1 >= length # if j+1 >= length, goto Error
Get v[j+1]		LDUR	X13, [X11,8]	# reg X13 = v[j + 1]
Load method table		LDUR	X14, [X0,0]	# X14 = address of method table
Get method addr		LDUR	X14, [X14,16]	# X14 = address of first method
Pass parameters		MOV MOV	X0, X21 X1, X20	# 1st parameter of compareTo is v[j] # 2nd param. of compareTo is v[j+1]
Set return addr		LDA	X30,L1	# load return address
Call indirectly		BR	X14	# call code for compareTo
Test if should skip swap	L1:	CMP B.LE	X12, X13 exit2	# compare X12 to X13 # go to exit2 if X12 ≤ X13
Pass parameters and call swap		MOV MOV BL	X0, X21 X1, X20 swap	# 1st parameter of swap is v # 2nd parameter of swap is j # swap code shown in Figure 2.34
Inner loop end		SUBI B	X20, X20, 1 for2tst	# j -= 1 # jump to test of inner loop
Outer loop	exit2:	ADDI B	X19, X19, 1 for1tst	# i += 1 # jump to test of outer loop

FIGURE 2.15.12 LEGv8 assembly version of the method body of the Java version of `sort`. The new code is highlighted in this figure. We must still add the code to save and restore registers and the return from the LEGv8 code found in Figure 2.27. To keep the code similar to that figure, we load `v.length` into `X22` instead of into a temporary register. To reduce the number of lines of code, we make the simplifying assumption that `compareTo` is a leaf procedure and we do not need to push registers to be saved on the stack.

Next, we load the length of the array (we use register `X22` to keep it similar to the code for the C version of swap):

```
LDUR     X22, [X0, #8]      //X22 = length of array v
```

Now we must ensure that the index is within bounds. Since the first test of the inner loop is to test if `j` is negative, we can skip that initial bound test. That leaves the test for too big:

```
CMP       X20,X22          // compare j to length
B.GE,IndexOutOfBounds      //if j > = length, goto Error
```

The code for testing `j + 1` is quite similar to the code for checking `k + 1` in `swap`, so we skip it here.

The key difference is the invocation of `compareTo`. We first load the address of the table of legal methods, which we assume is two doublewords before the beginning of the array:

```
LDUR   X14, [X0,#0]     // X14 = address of method table
```

Given the address of the method table for this object, we then get the desired method. Let's assume `compareTo` is the third method in the `Comparable` class. To pick the address of the third method, we load that address into a temporary register:

```
LDUR  X14, [X14, #16]   // X14 = address of third method
```

We are now ready to call `compareTo`. The next step is to save the necessary registers on the stack. Fortunately, we don't need the temporary registers or argument registers after the method invocation, so there is nothing to save. Thus, we simply pass the parameters for `compareTo`:

```
MOV X0, X12    // 1st parameter of compareTo is v[j]
MOV X1, X13    // 2nd parameter of compareTo is v[j+1]
```

Since we are using a branch register to invoke `compareTo`, we need to pass the return address explicitly. We use the pseudoinstruction load address (`LDA`) and label where we want to return, and then do the indirect branch:

```
LDA  X30,L1    // load return address
BR  X14        // to code for compareTo
```

The method returns, with `X6` determining which of the two elements is larger. If `X6 > 0`, then `v[j] >v[j+1]`, and we need to `swap`. Thus, to skip the `swap`, we need to test if `X6` $\leq$ `0`. We also need to include the label for the return address:

```
L1: CMP X6, XZR    // test if X6 ≤ 0
B.LE exit2         // go to exit2 if v[j] ≤ v[j+1]
```

The LEGv8 code for `compareTo` is left as an exercise.

Hardware/ Software Interface

The main changes for the Java versions of `sort` and `swap` are testing for null object references and index out-of-bounds errors, and the extra method invocation to give a more general compare. This method invocation is more expensive than a C procedure call, since it requires a load, a conditional branch, a pair of chained loads, and an indirect branch. As we see in Chapter 4, dependent loads and indirect branches can be relatively slow on modern processors. The increasing popularity of Java suggests that many programmers today are willing to leverage the high performance of modern processors to pay for error checking and code reuse.

Elaboration Although we test each reference to j and $j + 1$ to be sure that these indices are within bounds, an assembly language programmer might look at the code and reason as follows:

1. The inner *for* loop is only executed if $j \leq 0$ and since $j + 1 > j$, there is no need to test $j + 1$ to see if it is less than 0.
2. Since i takes on the values, 0, 1, 2, …, (data.length − 1) and since j takes on the values i − 1, i − 2, …, 2, 1, 0, there is no need to test if $j \leq$ data.length since the largest value j can be is data.length − 2.
3. Following the same reasoning, there is no need to test whether $j + 1 \leq$ data.length since the largest value of j+1 is data.length − 1.

There are coding tricks in Chapter 2 and superscalar execution in Chapter 4 that lower the effective cost of such bounds checking, but only high optimizing compilers can reason this way. Note that if the compiler inlined the swap method into sort, many checks would be unnecessary.

Elaboration Look carefully at the code for swap in Figure 2.15.11. See anything wrong in the code, or at least in the explanation of how the code works? It implicitly assumes that each `Comparable` element in `v` is 8 bytes long. Surely, you need much more than 8 bytes for a complex subclass of `Comparable`, which could contain any number of fields. Surprisingly, this code does work, because an important property of Java's semantics forces the use of the same, small representation for all variables, fields, and array elements that belong to `Comparable` or its subclasses.

Java types are divided into *primitive types*—the predefined types for numbers, characters, and Booleans—and *reference types*—the built-in classes like String, user-defined classes, and arrays. Values of reference types are pointers (also called *references*) to anonymous objects that are themselves allocated in the heap. For the programmer, this means that assigning one variable to another does not create a new object, but instead makes both variables refer to the same object. Because these objects are anonymous, and programs therefore have no way to refer to them directly, a program must use indirection through a variable to read or write any objects' fields (variables). Thus, because the data structure allocated for the array `v` consists entirely of pointers, it is safe to assume they are all the same size, and the same swapping code works for all of `Comparable`'s subtypes.

To write sorting and swapping functions for arrays of primitive types requires that we write new versions of the functions, one for each type. This replication is for two reasons. First, primitive type values do not include the references to dispatching tables that we used on `Comparables` to determine at runtime how to compare values. Second, primitive values come in different sizes: 1, 2, 4, or 8 bytes.

The pervasive use of pointers in Java is elegant in its consistency, with the penalty being a level of indirection and a requirement that objects be allocated on the heap. Furthermore, in any language where the lifetimes of the heap-allocated anonymous objects are independent of the lifetimes of the named variables, fields, and array elements that reference them, programmers must deal with the problem of deciding when it is safe to deallocate heap-allocated storage. Java's designers chose to use

garbage collection. Of course, use of garbage collection rather than explicit user memory management also improves program safety.

C++ provides an interesting contrast. Although programmers can write essentially the same pointer-manipulating solution in C++, there is another option. In C++, programmers can elect to forgo the level of indirection and directly manipulate an array of objects, rather than an array of pointers to those objects. To do so, C++ programmers would typically use the template capability, which allows a class or function to be parameterized by the *type* of data on which it acts. Templates, however, are compiled using the equivalent of macro expansion. That is, if we declared an instance of sort capable of sorting types X and Y, C++ would create two copies of the code for the class: one for sort<X> and one for sort<Y>, each specialized accordingly. This solution increases code size in exchange for making comparison faster (since the function calls would not be indirect, and might even be subject to inline expansion). Of course, the speed advantage would be canceled if swapping the objects required moving large amounts of data instead of just single pointers. As always, the best design depends on the details of the problem.