Compiling High-Level Languages for Embedded Systems - a top down approach

Sudhindra Rao

05/26/04

Abstract

Embedded systems have disparate requirements with respect to computing ability, power, size and connectivity with the world. The reducing size of the embedded systems with increasing demands on performance impose the requirements of limited code size. The availability of limited power also limits the amount of memory that can be used with an embedded processor. This further restricts the low-end embedded systems for power efficiency as well as code size and performance. In contrast the software design has been moving towards the object oriented technology due to its manageability and ease of maintenance. But the OO design often tends to be expensive with regards to size and performance. Hence the designer has to deal with challenges in meeting these conflicting demands. I present here a survey of current research which closes the gap between the software design and hardware restrictions. At the end of this paper it would be clear how the current research in Java virtual function resolution, opcode optimization, code compression and Virtual Machine Optimization can lead to a complete and small OO design.

1 Introduction

Embedded systems serve specific requirements of computing in a specified environment with limitations on power, size, operation etc. Embedded systems used in handhelds, palm devices have the restriction of being small in size as well as operating on limited battery power. They have a flexibility of having source to recharge the power. These restrictions of power and size are more stringent on tiny processors used in sensor networks. The operation power is limited by the miniature size, as is the memory. This limited memory space and power constrains the search space for compilation as well as limits the size of the code generated. Compilation efforts need to be targeted to optimize based on the power constraints as well as the code size.

Object oriented (OO) software is becoming omnipresent with the tools and technology available to support it. Also the elegance of design and management over the lifetime of the product make it the most attractive approach to software design. Although that is the case the OO design tends to pack lot of extra information not exactly required to run the code efficiently. This extra information includes package information, abstract class information with complete virtual tables, etc. Information such as this is imperative if the code is designed to be portable but in case of embedded systems where we target only a single processor at run-time, this becomes an overhead. Software designers constrained by the size of the embedded system often shy away from using OO designs and hence end up with low-level esoteric code, which is a management nightmare.
I analyze this problem with reference to the recent developments with focus on embedded systems and propose a top down approach for a multi-level compilation of this code design. I focus on Java as it is the programming language that has found wide acceptance among designers as a clean and compact alternative to C++. Java also has the additional benefits of interoperability and since it was originally designed for portability on embedded systems it is the ideal candidate for study.

This study is organized as follows. Section 2 presents the related work as a background to this study. Section 3 presents a problem definition of the OO design for embedded systems and uses the techniques to design a compiler. This compiler design is then tested against using the benchmarks in section 4.

2 Related Work

I present the related work for compiling high-level languages. We discuss virtual function optimizations, Java virtual machine optimizations and a few low-level optimizations in the following sub-sections.

2.1 Virtual Function Optimization

In embedded systems reducing the execution time overhead has been received a lot of attention. Even with programmers specifying virtual functions explicitly, the execution time overhead of virtual function calls in C++ has been measured to be as high as 40% [1]. In addition to this, as programmers use a truly object-oriented design, use of virtual functions increases. The costs associated with developing software are so high that the performance penalty of virtual functions is not enough incentive to design without using them. This only implies that, better compilers are needed to reduce the overhead due to virtual function calls.

Three static analysis algorithms, called Unique Name, Class Hierarchy Analysis and Rapid Type Analysis for reducing virtual call overhead are presented in [1]. Some key ideas are worth discussing here.

When a function has a unique signature (Unique Name), the virtual call is replaced with a direct call. Unique signature has the advantage that it does not require access to source code and can optimize virtual calls in library code. However, when used at link-time, Unique signature operates on object code, which inhibits optimizations such as inlining.

For a typical case, by combining the static information with the class hierarchy, one can determine that there are no derived classes of B, so that the only possible target is direct and need not be resolved at runtime. Class Hierarchy Analysis is more powerful than Unique Name for two reasons: it uses static information, and it can ignore identically-named functions in unrelated classes. Class Hierarchy Analysis must have the complete program available for analysis.

Rapid Type Analysis builds the set of possible instantiated types optimistically: it initially assumes that no functions except *main* are called and that no objects are instantiated, and therefore no virtual call to any target functions. It traverses the call graph created by Class Hierarchy Analysis starting at *main*. Virtual call sites are initially ignored. When a constructor for an object is found to be callable, all the virtual method calls are considered. The live portion of the call graph and the set of instantiated classes grows with each iteration iteratively in an interde-
Rapid Type Analysis inherits the limitations and benefits of Class Hierarchy Analysis: it must analyze the complete program. Like CHA, RTA is flow-insensitive and does not keep per-statement information, making it very fast [1].

A calculation of the direct cost of virtual function calls can be found in [2]. The authors suggested optimizations in case of single inheritance hierarchies which occur more often than multiple inheritance.

The run-time selection of a target procedure given a message name and the receiver type is called as dynamic dispatch [2]. Compared to a subroutine call in a procedural language, a direct cost or an indirect cost, may be associated to each dispatch.

The direct cost of dynamic dispatch consists of the time spent computing the target function as a function of the run-time receiver class and the selector. The ideal dispatch technique would find the target in zero cycles, as if the message send was a direct procedure call. The indirect cost arises from optimizations that cannot be performed because the target of a call is unknown at compile time. Many standard optimizations such as interprocedural analysis require a static call graph, and many intraprocedural optimizations are ineffective for the small function bodies present in object-oriented programs. Thus the presence of dynamic dispatch hinders optimization, and consequently, the resulting program will run more slowly.

The virtual function calls are typically resolved using the following type of code [2] (Figure 1).

1: load [object_reg + VFTOffset], ftable_reg
2: load [ftable_reg + deltaOffset], delta_reg
3: load [ftable_reg + selectorOffset], method_reg
4: add object_reg, delta_reg, object_reg
5: call method_reg

In case of single inheritance the operation 4 has no effect. It would be convenient if we could avoid executing
these useless operations that cause these indirect costs. Unfortunately, at compile time, the exact class of the receiver is unknown. However, the receiver’s virtual function table (VFT), which stores the offset values, “knows” the exact class (since the class definition has complete information of the type of object). The trick is to perform the receiver address adjustment only after the virtual function table entry is loaded. In the GNU GCC thunk implementation, the virtual function table entry contains the address of a parameterless procedure (a thunk), that adjusts the receiver address and then calls the correct target function [2]. Instead of always loading the offset value and adding it to the this pointer, the operation only happens when the offset is known to be non-zero. Since multiple inheritance occurs much less frequently than single inheritance, this strategy will save two instructions for most virtual function calls. Therefore, barring instruction scheduling effects, thunks would be at least as efficient as standard virtual function tables.

In a previous study the authors of [2] approximated the dispatch cost of several techniques by analyzing the call sequence carefully and describing their cost as a function of load latency and branch penalty, taking into account superscalar instruction issue. However, this approximation (e.g., 2(load latency)L + (branch penalty)B + 1 for VFT dispatch) is only an upper bound on the true cost, and the actual cost might be lower.

Branch Target Buffers (BTB) affect the cost of the VFT dispatch sequence: if the virtual call was executed previously, is still cached in the BTB, and invokes the same function as in the previous execution, the branch penalty is avoided, reducing the sequence cost to 2L + 1.

While all of the processor features discussed above improve performance on average, they also increase the variability of an instructions cost since it depends not only on the instruction itself (or the instruction and its inputs), but also on the surrounding code. Most processors sold today (e.g., the Intel Pentium and Pentium Pro processors, as well as virtually all RISC processors introduced since 1995) incorporate several or all of these features. As a result, it is hard to predict how expensive the average C++ virtual function call is on a current-generation PC or workstation.

First, the median dispatch overheads were observed to be (5.2% for the standard benchmarks and 13.7% for the all-virtual versions). These can be used as bounds on the dispatch performance. Any further improvement must come from other optimizations such as customization or inlining [2]. Given that better optimizing compilers are possible, it is inappropriate for programmers to compromise the structure of their programs to avoid dispatch.

As shown in [2], the relative overhead will increase as processors issue more instructions per cycle. At an issue width of 16, the median overhead increases by about 26%. Future processors might also have longer load latencies, further increasing dispatch cost. General compiler optimizations may also influence dispatch performance. For programs with unpredictable VFTs the dispatch cost could further increase of wide issue processors. On average, thunks remove a fourth of the overhead associated with the standard implementation of virtual function calls. As thunks remove a data dependency chain that inhibits instruction level parallelism better results could be obtained.

C++ allows a programmer to declare a method non-virtual. This informs the compiler that no subclass will override the method, and hence allowing the compiler to implement invocations of the method as direct procedure calls. However, the C++ programmer must make explicit decisions of which methods need to be virtual, making the programming process more difficult. When developing a reusable framework, the framework designer must make decisions to allow or disallow operations to be overridable by clients of the framework. The decisions made
by the framework designer more often than not fall short of the needs of the client program; in particular, a well-written highly-extensible framework will often provide flexibility that goes unused for any particular application, incurring an unnecessary run-time performance cost. In contrast, class hierarchy analysis is automatic and adapts to the particular framework/client combination being optimized. If extensions to the class hierarchy are made that require a non-virtual function to become overloaded and dynamically dispatched, the source program requires modifications. This can be particularly difficult in the presence of separately-developed frameworks which offer restricted access to clients. Class hierarchy analysis as shown in Figure 2, as an automatic mechanism, requires no source-level modifications [3]. A function may need to be virtual, because it has multiple implementations that need to be selected dynamically, but within some particular 2 subtree of the inheritance graph, there will be only one implementation that applies. In the example above, the \textit{m} method must be declared virtual, since there are several implementations, but there is only one version of \textit{m} that is called from \textit{F} or any of its subclasses. Class hierarchy analysis can identify when a virtual function “reverts” to a non-virtual one with a single implementation for a particular class subtree, enabling better optimization. In general, it is always the case that a message sent to the receiver of a method defined in a leaf class will have only one target implementation and hence can be implemented as a direct procedure call, regardless of whether or not the target method is declared virtual.
2.2 Optimizing the Java Virtual Machine (JVM)

The Java language, has been enjoying widespread use in many application domain and is by design also meant to be used in embedded systems. Specific APIs such as the JavaCard and EmbeddedJava specifications are available. This use of a standard format allows any third-party developed services to be installed on any Java-compatible embedded system thus keeping software portable. As mentioned above low-end embedded systems, have strong restrictions on the amount of available memory, that limit the size of applications that they can run. Memory is scarce due to: production costs(packing more bits costs more); power consumption must be minimized(more bits mean more area hence more energy used); and available physical space is limited. Thus embedded applications must consume as little memory as possible, including the space taken up by the program code itself. There is always a trade-off between the amount of space and the features of the embedded system.

The following issues arise when using Java for embedded systems [4]. In the Java class format, the constant pool occupies most of the space. The bytecode instructions only contribute about 18% of the total size. The size of a class file is only important while transporting the file over a network. In case of low-end embedded systems, only the memory footprint of the loaded program matters. Hence low-end embedded system, the constant pool is either completely removed (when dynamic loading is not needed) or reduced using ad hoc techniques. The research in [4] estimates that the bytecode accounts for roughly 75% of the memory footprint in a system.

In a low-end embedded system, there may not be sufficient memory to decompress even a single method, since only Kilo bytes of memory would be available. The time taken to uncompress such a segment of code might exceed time constraints defined by the application domain. Authors of [4] propose to factorize recurring instruction sequences into new instructions. This factorization allows more concise programs to run on a Java Virtual Machine (JVM) extended to support new instructions.

Although the Java bytecode instruction set was designed with embedded systems in mind, standard Java class files produced by compilers such as Sun’s javac compiler are not intended for use on such systems; debugging information and names of internal(private) identifiers are, for example, included by default [4]. Although these are easily stripped from class files, too much precious space is still taken up by names that are not needed during execution. For this reason, it is natural for a low-end embedded system to use its own internal space-efficient representation.

However, given the limited memory resources of low-end embedded systems, it is not possible to decompress each method as it is invoked. Stream compression can be applied individually on each basic block of the code. The instructions in a basic block are used sequentially and hence can be decompressed on-the-fly by a modified JVM, without having to store them to RAM. Here the tradeoff is between the performance and the code size. A significant time overhead will be associated with decompressing each basic block, slowing down the overall speed of the system to an unacceptable degree. However not all “wire” code compression techniques are applicable to interpretable code.

A simple way of eliminating code redundancy is to create methods that store repeated instruction sequences. Each original sequence of instructions is replaced by a call to such a method. This method extends the instruction set of the virtual machine with the new frequently used instructions. Due to large number of embedded platforms,
JVMs for embedded systems are proprietary and are as a rule written specifically for, and manually optimized to, each system. Hence as long as the changes are minimal and systematic, and as long as the JVM is still able to run standard Java bytecode, such changes can be easily added to the JVM.

Having said that, adding a fixed set of new instructions is be a nontrivial change that would significantly increase the size of the JVM [4]. Also, if the new instructions are specific to a given program, then they would have to be replaced if a different program is to be used. The trick is to extend the virtual machine to read new instruction definitions from the CAP(converted applet - a file format downloadable to JavaCard) [4] file. These macro instruction definitions consist of bytecode instructions to replace common instruction sequences in the code. Any instruction not in the standard instruction set is assumed to be a programmable instruction, defined by a table specific to the program being interpreted. The macro instructions can be stored in the run-time system, with very little memory overhead. With this approach, the number of new instructions is limited only by the number of instructions not used in the standard instruction set.

These requirements of the embedded systems for size as well as performance are difficult for the programmer to track and reason about, especially in the face of dynamic allocation and garbage collection. A novel pointer analysis method to help resolve this impasse is under development [5]. They have shown memory-usage reductions of up to 54% by applying object structure optimizations, whole program bitwidth analyses, and a new “mostly-N field” analysis. These are wholly-automated techniques to decompose objects and precisely control the bitwidth and presence of object fields, without burdening the programmer with explicit annotations or violating the modularity of the programmers view of the code. As a simple example, consider a tagged union of object types, with an integer field as a discriminator. Each possible value in the union is associated with a specific integer-valued tag. Without disturbing the programmers view of the construct, a bitwidth analysis can determine the exact bitwidth needed for the discriminator field. With further analysis the compiler can remove redundant integer tags. These and other low-level size transformations often violate code modularity by requiring whole-program knowledge, and are often maintenance nightmares; doing them automatically at compile-time solves the problem and spares the programmer.

Many functions are called just once, so reduced paging could pay for their interpretation overhead [6]. For example, the CPU is idle for most of the time during paging, so compressing pages can increase total performance even though the CPU must decompress or interpret the page contents.

When transmission is a bottleneck, we want the best possible compression, and we can afford to expand the compressed program before executing. We call such codes “wire” codes because a wire is the bottleneck. When memory is a bottleneck, the code is at least seldom used code must be stored and interpreted in compressed form. If some code must be compiled to run fast enough, the JIT (just in time) compilation speed is required to be high. When both transmission and memory are bottlenecks, it may make sense to decompress a wire code into a compressed interpretable form [6].

In programs, one important class of streams can be separated by patternizing the input. Programs are scanned to detect patterns and replace them with simpler, smaller opcode-operand combinations. [6] describe an operand specialization and opcode combination to yield a dense, randomly addressable program representation called BRISC. Since BRISC is required to be interpretable, it is constrained to ensure that instructions occur on byte boundaries. Hence, where the split-stream compression techniques described above would use 2-3 bits per opcode, BRISC
will always use 8 or 16 bits per opcode. To make up for the increased size of its opcodes, BRISC packs more information into each opcode. It does so through operand specialization and opcode combination.

The compiler optimization problem here is to compress virtual machine (VM) code, and to determine how to generate compact automata that accurately predict the next VM operator or operand based on the current context, so that tokens common in the current context can be given the shortest encodings. An example of the BRISC representation is given as follows. Say, \( \text{ld} \text{iuw n0, } 4(sp) \) is the most frequently occurring input instruction it makes sense to add to the dictionary of possible instruction patterns some of the specialized forms of this instruction. By doing so, we avoid explicitly representing common operands such as \( n0 \) or 4.

Opcode combination captures common code generation idioms. For example, data movement instructions such as \( \text{ld} \text{iuw} \) and \( \text{mov} \text{i} \) frequently occur to set up parameters before call instructions. This results in a quantized version of the tree construction. The results of optimizing and generating BRISC code as presented in [6] are:

<table>
<thead>
<tr>
<th>Abstract machine variant</th>
<th>Compressed size/native size</th>
</tr>
</thead>
<tbody>
<tr>
<td>RISC</td>
<td>0.54</td>
</tr>
<tr>
<td>minus immediates</td>
<td>0.56</td>
</tr>
<tr>
<td>minus register-displacement</td>
<td>0.57</td>
</tr>
<tr>
<td>minus both</td>
<td>0.59</td>
</tr>
</tbody>
</table>

These results suggest that a minimal abstract machine compresses nearly as well as one with typical ad hoc features for making programs smaller.

3 Building an optimization technique for embedded systems

With the background of Section 2, we can proceed to implement a compiler for embedded systems. The task is to implement a compiler for Java to produce size limited code for embedded systems. Before compilation the target hardware is known to some extent. This means that rough constraints on the target hardware have been determined as well as the future progression of the hardware has been foreseen. This compiled code would satisfy the constraints on the current system but will also be manageable and expandable. So that similar code could be reused after design change or also be ported to another product. The current target hardware will be only a minimum requirement and would not constrain the applications in any other way. As defined in the JVM specification [7] all computation in the JVM centers on the stack. Because the JVM has no registers for storing arbitrary values, everything must be pushed onto the stack before it can be used in a calculation. Bytecode instructions therefore operate primarily on the stack. For example, in the above bytecode sequence a local variable is multiplied by two by first pushing the local variable onto the stack with the \( \text{iload} \_0 \) instruction, then pushing two onto the stack with \( \text{iconst} \_2 \). After both integers have been pushed onto the stack, the \( \text{imul} \) instruction effectively pops the two integers off the stack, multiplies them, and pushes the result back onto the stack. The result is popped off the top of the stack and stored back to the local variable by the \( \text{istore} \_0 \) instruction. The JVM was designed as a stack based machine rather than a register-based machine to facilitate efficient implementation on register-poor architectures such as the
Intel 486. This feature of Java works to our benefit, which further relaxes our constraint of target machine. Our optimization can now focus only on the JVM optimization. With this brief definition of the problem we go ahead and apply the techniques discussed above. The problem is divided into two parts:

- Compilation of OO code to generate size-limited Java VM code
- Compilation and optimization of the JVM for the target machine

The software application is designed and developed using standard Java tools and techniques. While designing the OO code the regular design techniques for embedded systems are used. The hierarchical class design includes abstract classes and derived classes for implementing specific features. We implement the templates (implemented as abstract classes) in a special way as explained later.

### 3.1 Virtual Function Optimisations

The first stage of compilation performs all the virtual function optimizations as described in [2] and [1]. The Class Hierarchy analysis yields the cost of the virtual functions and also resolves method selectors. Thunks are used as described in [2] to implement efficient redirection of calls. All the abstract classes that are not instantiated in the hierarchy are considered as templates. They are handled as templates and the virtual function implementations are inlined. This is an elegant solution to the problem of unused class definitions appearing as overhead in the program class file.

**Proposition**: Only abstract classes used in any context undergo the following transformation. For example, consider a List abstract class that is unused and does not appear in the compiled code. Whereas a Vector class which has been used at least once is converted to a concrete instantiation (acting like a template) and inlined during use. All this can be easily done since in case of most of the single hierarchy classes the class type is known before instantiation.

### 3.2 Pattern Identification

The optimizer further identifies patterns in the class file and replaces frequently used operations with special tags. A Huffman code frequency based tagging of the class file can reduce the code size and its portability. The research in [8] details the JEPES platform design takes these techniques a step further. The JEPES proceeds as follows.

The JEPES compiler uses a context-insensitive, whole program data flow analysis which incorporates constant propagation, sign analysis, CHA, and escape analysis [8]. The use of values through object fields is also traced by the analysis, but only on a per-class basis. The analysis results are used to perform standard optimizations such as constant folding, branch prediction, etc., as well more specific optimizations such as virtual dispatch elimination, inlining, stack allocation of objects and elimination of object field operations that manipulate constant values. After dead code elimination, all fields, methods and classes that are detected as being unused are eliminated from the program. The lack of pointers in Java facilitates safely performing aggressive optimizations that would be difficult to perform in languages such as C or C++. The objects that can be completely eliminated from the program are referred to as ghost objects. **Ghost objects** can be used in the program without any overhead. The compiler lets
the designer define macros. The body of a macro is ignored by the compiler and can thus contain arbitrary Java code; this feature is used where the native code needs to be implemented in pure Java. Defining stack allocation as a per-class attribute is coarse-grained, and a per-allocation-site granularity is probably be more appropriate in some cases.

The statistics [9] collected on the JDK base libraries revealed that the opcodes and the operands contribute equally to the size of the code array. The authors separate the opcodes from the operands and apply different techniques to each group. Reconstructing the original code from the two compressed groups is trivial since each opcode is followed by a specified number of operands.

3.3 Opcode optimization [9]

Every opcode is represented by a single byte inside the class file, allowing up to 256 different opcodes. But only 204 of these opcodes are used by the Java Virtual Machine Specification. Moreover, the frequencies with which the opcodes appear inside class files have a very uneven distribution.

The Huffman algorithm reduces the number of bits required to represent the most frequent opcodes and increase the number of bits for the infrequent opcodes. Another the observation is that some opcodes are very likely to occur in pairs. Different patterns that exist in the code generated by Java compilers, such as javac can be used to the advantage. Each opcode can be represented as a unique state in a Markov state model. A sequence of opcodes can then be represented by the first state and a sequence of state transitions. Custom opcodes are used to represent commonly occurring sequences of real opcodes.

3.4 Operand optimization [9]

Every opcode is followed by a predefined number of operands. The operands that follow an opcode represent a specific type of information. These can help identify opcode groups based on the type of operands they have. The majority of the operands are one of the following types: Constant Pool Index and Method Offset.

3.5 Discussion

Applying the above optimizations to the virtual machine code can reduce the size of the code. Further the optimizations discussed in 2.2 will lead to optimal Java Virtual machines for the target embedded systems.

Dividing the problem in this manner assists in separating the design problems and still maintain manageability. The optimization problems also could be combined so that dead code can be identified at all the levels of code generation. The constraints on the small size VM can be applied to optimize the VM code. Some parts of the bytecode if detected as not least frequently used can be used to improve the VM representation. This iterative optimization will lead to overall code size reduction.
4 Experiments and Testing

The compiler is tested by using the frequently used programs on embedded systems. We test by using MediaBench [10] and some custom programs. These custom programs have the unique quality of being portable on different systems. They are scalable applications that can be used on different devices like a personal computer, a laptop, a PDA and a cellphone. The applications that fall into this category are games (like pacman, snake, spelling bee), contacts/calendar application which can manage up to a predefined number of items and an operating system with similar capabilities of scheduling. We compare the code size generated by these applications using the compiler designed in this research, with those generated by the compilers available in the market like gcc. This comparison of size is a good indicator of optimizer performance. Also since the optimizer is being targeted for different platforms the effect on its performance can be studied. It should be seen from these experiments that code size constrained code could be used as the preliminary step for designing low-end embedded systems using Java. Since the optimizations can be constrained by the different requirements of the target hardware and the application, the optimizations are retargetable by design. Hence this particular compilation approach should yield a scalable code depending on the constraints.

5 Acknowledgement

This paper has been written as a survey of the current research in the area of Compiler Optimizations. It would not have been possible without the research presented to propose a compiler optimization. I would like to thank the respective authors for their work.

References


