Improving Mobile Program Performance Through the Use of a Hybrid Intermediate Representation

Chandra Krintz
Computer Science Department
University of California, Santa Barbara

Abstract

We present a novel transfer format for mobile programs that is a hybrid of two existing formats: bytecode and SafeTSA. Java bytecode offers a compact representation and ease of interpretation (fast-compilation); SafeTSA offers amenability to optimization. We use program profiling to guide format selection at the method-level. Methods deemed "hot" are those for which optimization should be expended and as such, are encoded using the SafeTSA format. All other methods are encoded using bytecode. Our hybrid format exploits the benefits of each constituent format to reduce compilation, execution, and transfer overhead.

1 Introduction

The Internet is a constantly changing set of high-performance computational, communication, and storage devices, the aggregation of which offers tremendous performance potential. Users of the Internet can employ the vast processing power that is available using a programming methodology called *Mobile Computing* in which code is transferred from where it is stored to where it is executed automatically by the execution environment. Currently, however, the utility of mobile computing is limited by poor performance. This is due to three primary sources of overhead introduced during mobile program execution:

- *Transfer delay:* the transfer time between the source and destination machines for all non-local code and data. Internet performance can be highly variable across links as well as for the same link making it extremely difficult to maintain acceptable transfer times.
- *Compilation delay:* the translation time from transfer format (source, intermediate form, etc.) to native code of the target architecture. Optimization time (if any) contributes to this overhead.
- *Execution delay:* the opportunity cost (loss of optimization potential) introduced by the use of dynamic compilation and optimization time. It has proven to be extremely difficult to achieve efficient execution without introducing significant compilation delay.

These overheads are experienced by the executing program as long startup times, intermittent interruption, and slow execution speeds. The degree to which transfer, compilation, and execution delay impact performance is dictated by the choice of mobile program transfer format.

Java bytecode [16] is the most pervasive mobile program transfer format in use today. It has prospered, among other reasons, since it is highly portable and enables type-safe execution. Bytecode is an architecture-independent intermediate format that, once written, can execute on any machine on which there is an execution environment (a Java Virtual Machine (JVM) [16]). The bytecode format is a 0-address, or stack-machine, representation in which operations are performed using a single data structure called a stack.

The use of the bytecode format as well as other similar formats, e.g., the Common Intermediate Language (CIL) [8] from the Microsoft .Net Framework [13], introduces significant compilation and execution delay. Since most of the hardware architectures in use today implement a register machine model, bytecode programs must be translated from the stack model to the register model. This is not a straight-forward transformation and requires abstract execution and aggressive optimization to enable high-performance. Since the compilation and optimization of mobile Java programs occurs *while* the program is running, significant overhead is required to achieve high-performance. Existing Java execution environments that implement highly optimizing compiler and adaptive optimization technology, have yet to enable the high-performance of similar C and Fortran programs [2, 7].

An alternative transfer format, called SafeTSA [3], is more amenable to optimization. SafeTSA implements a virtual register model (as opposed to the stack-based model used in bytecode) and encodes instructions using an extension to the Static Single Assignment (SSA) [9, 6] form, an intermediate form used extensively in *static*, highly-optimizing compilers. The use of an SSA-like mobile transfer format has the potential of enabling optimized execution time with reduced compilation delay. However, SafeTSA is larger than bytecode (introducing transfer delay) and it cannot be directly interpreted (fast-compiled) as bytecode can. Fast-compilation is vital for effective adaptive optimization of mobile applications [14, 2, 7].

Both bytecode and SafeTSA must trade off one source of mobile program overhead for another: bytecode trades off compilation overhead for transfer delay; SafeTSA trades off transfer and compilation delay for execution performance. **With this work, we present a novel mobile program format that uses *both* bytecode and SafeTSA that exploits the benefits yet avoids the overheads of each**.

2 Hybrid Intermediate Representation for Mobile Java Programs

To reduce the transfer, compilation, and execution delay that is imposed on mobile Java programs, we present a novel intermediate representation for mobile Java programs. Our format is a *hybrid* of two existing formats: Java bytecode and SafeTSA. In this section, we describe each of these formats then show how they can be combined to improve mobile program performance.

2.1 Java Bytecode

The Java transfer format, bytecode, is the most common mobile program format currently used. It is a 0-address, or stack-machine, representation in which operations are performed using a single stack data structure. The stack format is extremely simple; We can easily implement interpreters (and fast-compilers) as many have [12, 1, 2, 17]. Method code in this format are very compact since multiple operations are implicitly encoded into a single instruction. However, access to values is restricted to the top of the stack. For example, the add instruction indicates that the top two values on the stack should be extracted (popped), moving all other stack values up. The values should then be added and the result should be placed on the top of the stack (pushed), moving all other values down. Ease in interpretation (fast translation for adaptive compilation) and compactness (fast transfer) are vital to mobile program performance.

However, bytecode is not amenable to optimization. It requires considerable effort (and hence compilation delay) to convert methods to an intermediate form that can be used directly by an optimizing compiler. Difficulties arise, among other reasons, since the stack model does not directly map to the register model used by commonly available architectures. In addition, the stack model serializes computation and prevents reuse of values (since only the value at top of the stack is accessible at any given time).

2.2 SafeTSA

A mobile transfer format proposed recently, called SafeTSA, replaces the stack-based model with a virtual register model. The format uses an extension to the Static Single Assignment [9, 6] (SSA) form, an intermediate representation used extensively in *static*, highly-optimizing compilers. SSA is a representation in which each variable value is assigned a name. SSA simplifies optimization by limiting the number of reaching definitions (an assignment to a variable name that *reaches* a program point at which there is a use of that variable name) and enabling efficient data-flow algorithms to be used for optimization. Since SafeTSA is an extension of SSA form, SafeTSA programs need not be converted to SSA form by a compilation system. That is, optimization algorithms can operate directly on the transfer format. As such SafeTSA has the potential of enabling optimized execution time with reduced compilation delay.

However, an SSA representation of a program is significantly larger than the original program [9]. This is reflected in the size of SafeTSA programs. In [3], the authors present reductions in SafeTSA program size over bytecode, however, SafeTSA is compressed and the bytecode programs are not. SafeTSA is encoded using a prefix encoding that is similar to Huffman encoding with equal symbol probabilities; the instruction stream is organized to improve the compression ratio of the algorithm [3, 10]. In the programs we studied (for which results are presented below), SafeTSA is 3 times larger than compressed bytecode.

Another limitation of SafeTSA is that it cannot be directly interpreted or quickly translated. There is no direct mapping between a *phi-node* in SSA form (used for join points in the control flow graph [9]) and an instruction in any popular architecture instruction set. SafeTSA therefore, requires translation of *phi-nodes* prior to interpretation or fast-compilation. As a result, SafeTSA programs cannot be adaptively optimized as effectively as bytecode programs. Adaptive optimization is a popular dynamic optimization strategy (described below) that reduces compilation overhead while enabling highly optimized execution speeds by combining *both* fast-compilation and aggressive optimization.

2.3 Combining Java Bytecode and SafeTSA to Form a Hybrid Representation

The use of Java bytecode as an intermediate format, imposes optimization and execution delay yet enables compactness, and direct, fast, unoptimized compilation for use in adaptive optimization. Use of the SafeTSA format introduces transfer and compilation overhead (for fast compilation) but enables improved execution performance. We believe that by combining these two formats into a *hybrid* representation, we can exploit the benefits that both enable yet avoid much of the overhead that is introduced by the use of each.

To enable this, we incorporate many of the ideas from existing adaptive optimization systems [2, 7, 12, 11, 15, 17]. Adaptive optimization is a technique in which a fast, non-optimizing compiler (or interpreter) is used to compile a method the first time it is invoked. Instrumentation is inserted into the method and on-line measurements are made to determine when program execution characteristics warrant optimization. When a threshold for a method is reached, e.g., due to number of method invocations or the amount of time spent in a method, an optimizing compiler is used to re-compile the method using various levels of optimization (or just a single level in some systems). Such systems are called *adaptive optimization systems* since they use optimization to enable program performance to adapt as program execution behavior changes. Adaptive systems reduce compilation overhead since optimization is only performed as deemed necessary by the measurement system.

In our hybrid model, we encode programs at the *method-level* with either bytecode or SafeTSA. We use bytecode for its compactness and direct interpretation (fast-compilation) to reduce transfer delay and to enable adaptive optimization. We use SafeTSA for its amenability to optimization to reduce compilation delay and to improve execution speeds. In concurrent work [4], researchers experiment with combining SafeTSA and bytecode in programs at the *class level* to reduce compilation delay. To our knowledge, our hybrid approach is the first to propose such encoding at the method-level to reduce compilation, execution, and transfer delay. Another distinguishing feature of our hybrid model is that we incorporate off-line profile information, much like that performed by adaptive optimization systems *on-line*, to guide selection of method-level encoding.

We generate a profile of the dynamic characteristics of each program, *off-line*. We insert instrumentation into programs and monitor their execution. From this information we can identify popular, or *hot*, methods. *Hot* methods are those that account for the majority of the program execution time. Since most of the

Benchmark	OptCT (Secs)	OptCT (No SSA_build)	OptTT (Secs)	BaseCT (Secs)	BaseTT (Secs)	Method Count	Methods Execd	Hot Methods
DB (209)	2.00	1.48	19.53	0.00	22.91	34	27	8
Jack (228)	6.60	5.02	8.85	0.04	9.16	315	265	10
Javac (213)	13.28	9.50	13.98	0.08	12.12	1190	740	36
Jess (202)	7.04	5.70	7.29	0.04	8.82	690	412	29
Average	7.23	5.42	12.41	0.04	13.25	557	361	21

Table 1: Benchmark Statistics Using JikesRVM.

execution time is spent in these methods, we postulate that these are the methods that should be optimized. We use the JikesRVM Controller and sampling system [5] to generate our profiles.

We encode *hot* methods in the SafeTSA format. All other methods are implemented with bytecode. During execution (using JikesRVM), the dynamic compilation system determines which format a method is in using bytecode annotation [14]. A bit associated with each method is set in the class file if a method is encoded using SafeTSA. An optimizing compiler is used to optimize an annotated method (in SafeTSA). All other methods (in bytecode) are fast-compiled (without optimization) or interpreted.

2.4 Discussion of Hybrid Performance Potential

Currently, there is no freely available Java compilation system that has a SafeTSA front-end; we plan to implement one as part of this continuing project. In addition, SafeTSA tools are not freely available. To enable evaluation of our hybrid format, we assume that SafeTSA performance characteristics are similar to those of bytecode executed using the JikesRVM optimizing that (at Level 2) performs SSA-based optimizations. To evaluate the performance potential of our hybrid format, we extrapolate the performance potential of our hybrid technique from actual JikesRVM performance measurements.

We measured compilation (optimization) and execution time for four SpecJVM programs using the JikesRVM compilation system on a 2GHz x86 machine with 512KB of memory and Debian Linux version 2.4.16. The measurements are shown in Table 1. The first column of data is the time for optimized JikesRVM compilation. All times in the table are in seconds. The second column is column 1 without the time required for building the SSA form in JikesRVM. On average, SSA-form construction accounts for 20% of the total compilation time by the optimizing compiler. The third column of data is the total number of seconds for both compilation and execution. The fourth and fifth columns are the same as the first and third, only for the JikesRVM fast (baseline) compiler. The sixth column shows the total number of static methods in the program; the seventh column indicates the number of these methods that are executed. The last column shows the number of methods that our profile indicates are *hot*. More information about how *hot* methods are identified can be found in [5].

To estimate the performance potential of our hybrid model, we make several assumptions using these measurements. First, we assume that SafeTSA compilation time is the same as that for the JikesRVM optimization without the overhead of converting the JikesRVM high-level intermediate representation to SSA-form (column 2 in Table 1). Actual results for this comparison support this assumption [4], in which compilation time for class files in SafeTSA is 20% faster than when they are in bytecode format. Second, we assume that SafeTSA execution speed is similar to that for the code generated by the JikesRVM optimizing compiler (Level 2).

To compute the performance potential of programs in our hybrid format, we measured the time required for JikesRVM execution of the benchmarks during which only the hot methods are optimized. Figure 1 indicates the performance potential of our hybrid model on compilation and execution delay. The first bar for each benchmark is the total time (compilation plus execution time) for fast-compiled, unoptimized

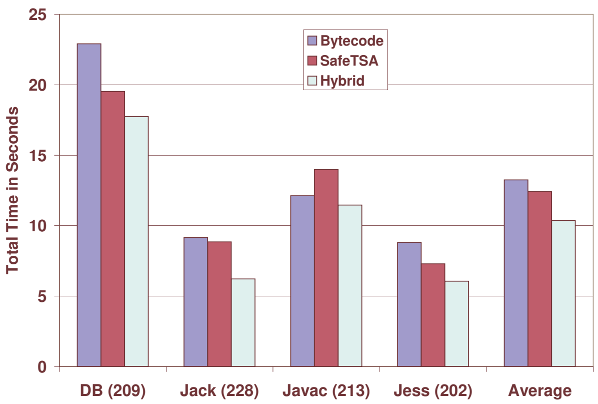

Figure 1: Performance Potential of the Hybrid Model on Total Time (compilation plus execution delay).

bytecode programs. Times (y-axis) are in seconds. The second bar is the total time for SafeTSA (level 2 optimized bytecode programs). The last bar is the performance potential of our hybrid model: hot methods, in SafeTSA, are optimized and cold methods, in bytecode, are fast compiled. The right-most set of results indicate the average performance potential. On average, the hybrid model reduces total time (compilation plus execution delay) by 28% over unoptimized bytecode and by 20% over optimized SafeTSA.

We also considered the potential effect of our hybrid model on transfer delay. These transfer characteristics are shown in Table 2. Column 1 of data shows the size of the benchmarks in bytecode format. All sizes are in kilobytes (KB). Column 2 shows the size of the benchmarks archived using the UNIX *tar* utility and compressed using UNIX gzip compression utility. The third column shows the size of SafeTSA programs. This number is extrapolated from the results reported in [3]; the authors present results that indicate that SafeTSA programs are 26% smaller than bytecode files on average. The final column indicates the performance potential of our hybrid model on transfer delay. For these values we computed the size using compressed bytecode for cold methods and SafeTSA for hot methods. On average, the hybrid model has the potential to reduce transfer size over SafeTSA programs by over 250%.

3 Conclusion

We present an alternative transfer format for mobile programs that is a hybrid of two existing formats: Java bytecode and SafeTSA. Java bytecode offers a compact representation and ease of interpretation (fast or non-optimized compilation) and SafeTSA offers amenability to optimization. Using our hybrid model, class files are encoded in SafeTSA or bytecode at the *method-level*. We select the SafeTSA format for methods that warrant optimization overhead to improve execution performance, i.e., are *hot*. We measure hotness using off-line profiling. We encode *hot* methods with SafeTSA and all others with bytecode. A dynamic compilation system then optimizes only SafeTSA methods and fast-compiles bytecode methods. Our preliminary evaluation of the performance potential of this hybrid model indicates that it can reduce both compilation and execution delay as well as transfer delay.

References

[1] A. Adl-Tabatabai, M. Cierniak, G. Lueh, V. Parikh, and J. Stichnoth. Fast,Effective Code Generation in a Just-In-Time Java Compiler. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, October 2000.

	Size in KB			
Benchmark	Bytecode	Compressed Bytecode	SafeTSA	Hybrid
DB (209)	20.00	5.41	14.76	7.61
Jack (228)	170.00	50.74	125.46	53.11
Javac (213)	2480.00	732.19	1830.24	765.41
Jess (202)	500.00	83.99	369.00	95.97
Average	792.50	218.09	584.87	230.53

Table 2: Benchmark Size Statistics and Performance Potential of the Hybrid Model on Transfer Delay.

[2] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P.Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1), 2000.

[3] W. Amme, N. Dalton, J. Ronne, and M. Franz. SafeTSA: A Type Safe and Referentially Secure Mobile-Code Representation Based on Static Single Assignment Form. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 137–147, June 2001.

[4] W. Amme, J. von Ronne, and M. Franz. Using the safetsa representation to boost the performance of an existing java virtual machine. Technical Report UC Irvine 06/02, University of California, Irvine, 2002.

[5] M. Arnold, S.J. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive optimization in the jalaepeño jvm. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, October 2000.

[6] J. Choi, R. Cytron, and J. Ferrante. Automatic Construction of Sparse Data Flow Evaluation Graphs. pages 55–66, January 1991.

[7] M. Cierniak, G. Lueh, and J. Stichnoth. Practicing JUDO: Java Under Dynamic Optimizations. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, October 2000.

[8] ECMA standardization of the Common Language Infrastructure. `http://msdn.microsoft.com/net/ecma/`.

[9] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[10] Personal communcation with Michael Franz, author of [3]. `http://www.ics.uci.edu/~franz/`.

[11] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. Eggers. Dyc: An expressive annotation–directed dynamic compiler for c. Technical Report Tech Report UW-CSE-97-03-03, University of Washington, 2000.

[12] The Java Hotspot performance engine architecture.

[13] Microsoft Inc. Microsoft Explorer. `http://www.microsoft.com/net/`.

[14] C. Krintz and B. Calder. Using Annotation to Reduce Dynamic Optimization Time. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, October 1998.

[15] C. Krintz, D. Grove, V. Sarkar, and B. Calder. Reducing the Overhead of Dynamic Compilation. *Software—Practice and Experience*, 31(8):717–738, 2001.

[16] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.

[17] M. Plezbert and R. Cytron. Does just in time = better late than never? In *Proceedings of the SIGPLAN'97 Conference on Programming Language Design and Implementation*, January 1997.