Beyond Java: An Infrastructure for High-Performance Mobile Code on the World Wide Web

Michael Franz
Department of Information and Computer Science
University of California
Irvine, CA 92697-3425
franz@uci.edu

Abstract: We are building an infrastructure for the platform-independent distribution and execution of high-performance mobile code as a future Internet technology to complement and perhaps eventually succeed Java. Key to our architecture is a representation for mobile code that is based on adaptive compression of syntax trees. Not only is this representation more than twice as dense as Java byte-codes, but it also encodes semantic information on a much higher level. Unlike linear abstract-machine representations such as p-code and Java byte-codes, our format preserves structural information that is directly beneficial for advanced code optimizations.

Our architecture provides fast on-the-fly native-code generation at load time. To increase performance further, a low-priority compilation thread continually re-optimizes the already executing software base in the background. Since this is strictly a re-compilation of already existing code, and since it occurs completely in the background, speed is not critical, so that aggressive, albeit slow, optimization techniques can be employed. Upon completion, the previously executing version of the code is supplanted on-the-fly and re-optimization starts over.

Our technology is being made available under the name "Juice", in the form of plug-in extensions for the Netscape Navigator and Microsoft Internet Explorer families of WWW browsers. Each plug-in contains an on-the-fly code-generator that translates Juice-applets into the native code of the target machine. As far as end-users are concerned, there is no discernible difference between Java-applets and Juice-applets, once that the plug-in has been installed, although the underlying technology is very different. The two kinds of applets can coexist on the same WWW page, and even interact with each other through the browser's API. Our work not only demonstrates that executable content need not necessarily be tied to Java technology, but also suggests how Java can be complemented by alternative solutions, and potentially be displaced by something better.

1. Introduction

One of the most beneficial aspects of the rapid expansion of the Internet is that it is driving the deployment of "open" software standards. We are currently witnessing the introduction of a first suite of interoperability standards that is already having far-reaching influences on software architecture, as it simultaneously also marks the transition to a *component model* of software. The new standards, such as *CORBA* (Object Management Group), *COM/OLE* (Microsoft), and *SOM/OpenDoc* (Apple Computer, IBM, Novell), enable software components to interoperate seamlessly, even when they run on different hardware platforms and have been implemented by different manufacturers. Over time, the monolithic application programs of the past will be supplanted by societies of interoperating, but autonomous, components.

It is only logical that the next development step will lead to even further "open-ness", not only freeing components from all dependence upon particular hardware architectures, but also giving them the autonomy to migrate among machines. Instead of executing complex transactions with a distant server by "remote control" over slow communication links, software systems will then be able to send self-contained mobile agents to a server that complete the transactions autonomously on the user's behalf. The inclusion of *executable content* into electronic documents on the World Wide Web already gives us a preview of how powerful the concept of mobile code is, despite the fact that so far only a unidirectional flow of mobile programs from server to client is supported. Distributed systems that are based on freely-moving agents will be even more powerful.

In order to transfer a mobile program between computers based on different processor architectures, some translation of its representation has to occur at some point, unless the mobile program exists in multiple execution formats simultaneously. Although the latter approach seems feasible in the current context of software distribution via CD-ROM, its limits will soon become apparent when low-bandwidth wireless connectivity becomes pervasive. Hence, a compact *universal* representation for mobile code is required. The search for such a universal representation is the subject of much current research [Engler 1996, Inferno, Lindholm et al. 1996], including recent work of the author [Franz & Kistler 1996, Kistler & Franz 1997].

Although Sun Microsystems' *Java* technology is now the de-facto standard for portable "applets" distributed across the Internet, it remains surprisingly simple to provide alternatives to this platform, even within the context of commercial browser software. We have created such an alternative to the Java platform and named it *Juice*. Juice is an extension of the author's earlier research on portable code and on-the-fly code generation[1] [Franz & Ludwig 1991, Franz 1994a, Franz 1994b]. Our current work is significant on two accounts: First, Juice's portability scheme is technologically more advanced than Java's and may lead the way to future mobile-code architectures. Second, the mere existence of Juice demonstrates that Java can be complemented by alternative technologies (and potentially be gradually displaced by something better) with far less effort than most people seem to assume. In fact, once that Juice has been installed on a machine, end-users need not be concerned at all whether the portable software they are using is based on Juice or on Java. In light of this, we question whether the current level of investment in Java technology is justified, in as far as it is based on the assumption that Java has no alternatives.

In the following, we swiftly introduce the mobile code format upon which all of our work is based. We then give an overview of our run-time architecture, which not only provides on-the-fly code generation, but also dynamic code re-optimization in the background. Finally, we report on the current state of our implementation, specifically the availability of an integrated authoring and execution environment for Juice components, and of a family of plug-in extensions for two popular commercial WWW browsers that enable these browsers to execute Juice-based content.

2. An Effective Representation for Mobile Code

Our mobile-code architecture is based on a software distribution format called *slim binaries* [Franz & Kistler 1996] that constitutes a radical departure from traditional software-portability solutions. Unlike the common approach of representing mobile programs as instruction sequences for a virtual machine, an approach taken both with p-code [Nori et al. 1976] as well as with Java byte-code [Lindholm et al. 1996], the slim binary format is instead based on *adaptive compression of syntax trees* [Franz 1994a]. When compiling a source program into a slim binary, it is first translated into a tree-shaped intermediate data structure in memory that abstractly describes the semantic actions of the program (e.g., "add result of left sub-tree to result of right sub-tree"). This data structure is then compressed by identifying and merging isomorphic sub-trees, turning the tree into a directed acyclic graph with shared sub-trees (for example, all occurrences of "x + y" in the program could be mapped onto a single sub-tree that represents the sum of "x" and "y"). The linearized form of this graph constitutes the slim binary format.

In the actual implementation, tree compression and linearization are performed concurrently, using a variant of the classic LZW data-compression algorithm [Welch 1984]. Unlike the general-purpose compression technique described by Welch, however, our algorithm is able to exploit domain knowledge about the internal structure of the syntax tree being compressed. Consequently, it is able to achieve much higher information densities [Fig. 1]. We know of no conventional data-compression algorithm, regardless of whether applied to source code or to object code (for any architecture, including the Java virtual machine), that can yield a program representation as dense as the slim binary format.

[1] note that this earlier work on mobile code predates Java by several years

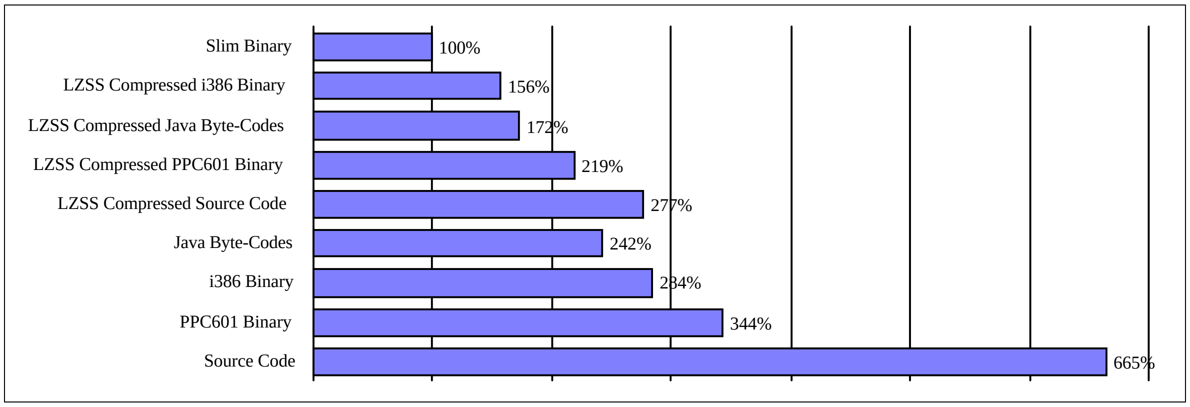

Figure 1: Relative Size of a Representative Program Suite in Various Formats

The compactness of the slim binary format may soon become a major advantage, as many network connections in the near future will be wireless and consequently be restricted to small bandwidths. In such wireless networks, raw throughput rather than network latency again becomes the main bottleneck. We also note that one could abandon native object code altogether in favor of a machine-independent code format if the portable code would not only *run* as fast as native code, but also *start up* just as quickly (implying that there would be no discernible delay for native-code translation). As the author has shown in previous work, this becomes possible if the portable software distribution format is so dense that the additional computational effort required for just-in-time code generation can be compensated entirely by reduced I/O overhead due to much smaller “object files” [Franz 1994a, Franz 1994b, Franz 1997a].

Compactness does come at a small price: since isomorphic sub-trees have been merged during encoding, a program represented in the slim binary format cannot simply be interpreted byte-by-byte. Conversely, the individual symbols in an abstract-machine representation such as Java byte-codes are self-contained, permitting random access to the instruction stream as required for interpreted execution. However, in exchange for giving up the possibility of interpretation, which by its inherent lack of run-time performance is limited to low-end applications anyway, the slim binary format confers a further important advantage:

It turns out that the tree-shaped program representation from which the slim binary format is generated (and which is re-created in memory when a slim binary file is decoded) is an almost perfect input for an optimizing code generator. The slim binary format preserves structural information such as control flow and variable scope that is lost in the transition to linear representations such as Java byte-codes. In order to perform code generation with advanced optimizations from a byte-code representation, a time-consuming pre-processing step is needed to re-create the lost structural information. This is not necessary with slim binaries. A similar argument applies with respect to *code verification*: analyzing a mobile program for violation of type and scoping rules is much simpler when the program has a tree-based representation than it is with a linear byte-code sequence.

3. A Run-Time Architecture Featuring Dynamic Re-Optimization

We are developing a run-time architecture in which the capability of generating executable code from a portable intermediate representation is a *central function of the operating system itself* [Franz 1997b]. It thereby becomes possible to perform advanced optimizations that transcend the boundaries between individual portable components, as well as the boundary between user-level and system-level code.

Consider a scenario in which a user downloads several portable components from various Internet sites during a single computing session. Every time that such a component is downloaded, it is translated on-the-fly into the native code of the target machine so that it will execute efficiently. This “just-in-time” translation is able to achieve remarkable speed-up factors when compared to interpreted execution, but it still cannot extract the theoretically achievable optimum performance from the system as a whole. This is because every component has been compiled and optimized individually, rather than in the context of all other components in the system.

In order to achieve even better performance, one would have to perform inter-component optimizations. Examples of such optimizations are procedure inlining across component boundaries, inter-procedural register allocation, and global cache coordination. However, since the set of participating components is open-ended and the user has the option of interactively adding further components at any time, it is of course impossible to perform these optimizations statically. Unfortunately, the principle of dynamic composability that fundamentally underlies open, component-based systems runs counter to the needs of optimizing compilers. The problem is compounded further by the fact that component-based systems are often made out of a relatively large numbers of relatively small parts.

There is, however, a solution: at any given time, the set of *currently* active components is well known. Hence, a globally optimized version of the system can in fact be constructed, except that this has to be done at run-time and that its validity extends only until the user adds the next component. This leads to the key idea of our run-time architecture: to perform the translation from the slim binary distribution format into executable code not just once, but to do so continually, constructing a *series* of globally cross-optimized code images in memory, each of which encompasses all of the currently loaded components. Whenever such a cross-optimized image has been constructed, it supersedes the previously executing version of the same code, i.e. the new code image is “hot-swapped” into the operational state while the previous one is discarded. At the same time, construction of yet another code image is initiated. We call this iterative process *re-optimization*, and it is performed with low priority in the background.

Since re-optimization occurs in the background while an alternate version of the same software is already executing in the foreground, it is largely irrelevant how long this process takes. This means that far more aggressive optimization strategies can be employed than would be possible in an interactive context. Further, because re-optimization occurs at run-time, “live” execution-profile data can be taken into account for certain optimizations [Ingalls 1971, Hansen 1974, Chang et al. 1991]. This is why our model is continuous: although re-optimization would strictly be necessary only whenever new components are added to the system, usage patterns among the existing components still shift over time. Re-optimization at regular intervals makes it possible to take these shifts into account as well. Our system bases each new code image on dynamic profiling data collected just moments earlier, and hence can provide a level of fine-tuning that is not possible with statically-compiled code.

This leaves the question of what happens when a new component is added interactively to the running system. Clearly, one cannot wait for the completion of a full re-optimization cycle of the whole system before the new component can be used. This problem is taken care of by a second operational mode of our code generator: besides being able to generate high-quality optimized code in the background, it also has a “burst” mode in which compilation speed is put ahead of code quality so that execution can commence immediately. Using this “burst” mode, each new component is translated into native code as a stand-alone piece of code not cross-optimized with the rest of the system. For a short while, it will then execute at less than optimum performance. Upon the next re-optimization cycle, it will automatically be integrated with the remaining system and henceforth run more efficiently.

4. Our Prototype Implementation

Our work has originated and continues to evolve in the context of the *Oberon System* [Wirth & Gutknecht 1989, Wirth & Gutknecht 1992]. Oberon constitutes a highly dynamic software environment in which executing code can be extended by further functionality at run-time. The unit of extensibility in Oberon is the *module*; modules are composed, compiled and distributed separately of each other. Oberon is programmed in a language of the same name [Wirth 1988], a direct successor of Pascal and Modula-2. The Oberon System is available on a wide variety of platforms [Franz 1993, Brandis et al. 1995].

For all practical purposes, Oberon’s modules supply exactly the functionality that is required for modeling mobile components. Modules provide encapsulation, their interfaces are type-checked at compilation time and again during linking, and they are an esthetically pleasing language construct. The only feature that we have recently added to the original language definition is a scheme for the globally unique naming of qualified identifiers. Hence, when we have been talking about “components” above, we were referring to Oberon modules.

We have already come quite far in deploying the ideas described above in a broader sense than merely implementing them in a research prototype. The current Oberon software distribution [Oberon] uses the architecture-neutral slim binary format to represent object code across a variety of processors. Our on-the-fly code generators have turned out to be so reliable that the provision of native binaries could be discontinued altogether, resulting in a significantly reduced maintenance overhead for the distribution package. Currently, our implementations for Apple Macintosh on both the *MC680x0* and the *PowerPC* platforms (native on each) and for the *i80x86* platform under Microsoft Windows 95 all share the identical object modules, except for a small machine-specific core that incorporates the respective dynamic code generators and a minimal amount of "glue" to interface with the respective host operating systems.

The latest release of the Oberon software distribution additionally contains an authoring kit for our Juice mobile-component architecture. The main difference between ordinary Oberon modules and Juice components is that they are based on different sets of libraries. The Juice API is smaller than Oberon's, and modeled after Netscape's Java-Applet-API. Components that are based on this reduced system interface cannot only be executed within the Oberon environment, but also within the *Netscape Navigator* and *Microsoft Internet Explorer* families of WWW browsers, both on the Macintosh (PowerPC) and Microsoft Windows (i80x86) platforms. Hence, by choosing the optional Juice API rather than Oberon's standard libraries, developers of Oberon-based components can address a much larger potential market.

In order to enable Juice components to execute within the aforementioned WWW browsers, we supply a set of platform-specific plug-ins [Juice]. Each plug-in contains a dynamic code-generator that translates the slim binary representation into the native code of the respective target architecture (PowerPC or Intel 80x86). This translation occurs before the applet is started, using the aforementioned "burst mode" of code generation. It is fast enough not to be noticed under normal circumstances, and the resulting code quality is comparable to the current generation of just-in-time Java compilers. Unlike our Oberon-based research platform, our Juice plug-ins do not yet provide background re-optimization and the additional performance gains that come with it. However, we plan to periodically incorporate our research results into Juice.

Juice differs considerably from Java, yet from the web-browsing end-user's perspective, there is no obvious difference between Java and Juice applets. We claim that this is important, because it shows that Java can be complemented by alternative technologies in a user-transparent manner. In the long run, the choice of a particular mobile-code solution may often simply be a matter of personal taste, rather than a technological necessity. Luckily, it is the applet developer that needs to make this choice; the end user need not know any of it as multiple mobile-code technologies, such as Java and Juice, can happily coexist, even on the same web page.

5. Conclusion and Outlook

Mobile code for the Internet need not necessarily be tied to Java technology. In this paper, we have presented various aspects of a mobile-code infrastructure that differs from Java on several key accounts. Not only is our implementation a test-bed for novel code-representation and dynamic-compilation techniques, but it also confirms the suitability of the existing browser plug-in mechanism for supporting alternative software portability solutions.

As our implementation demonstrates, the plug-in mechanism can even be utilized to provide on-the-fly native-code generation, enabling alternative portability schemes to compete head-on with Java in terms of execution speed. Using plug-in extensions for the most popular browsers, many mobile-code formats could potentially be introduced side-by-side over time, *gradually* reducing Java's pre-eminence rather than having to displace it abruptly. This would make the eventual migration path from Java to a successor standard at the end of Java's life-cycle much less painful than most people anticipate now. The same strategy could also be employed to simultaneously support several mutually incompatible enhancements of the original Java standard.

We contend that dynamic code generation technology is reaching a level of maturity that it will soon be relatively inexpensive to support multiple software distribution formats concurrently. It will then become less important how much "market share" any incumbent software distribution format such as Java byte-codes or Intel binary code already owns. In order to be commercially successful, future software distribution formats will have to mimic Java as far as providing architecture neutrality and safety, but further considerations such as code density will surely gain in importance. Some future formats, for instance, will be more narrowly targeted towards particular application domains. In this larger context, the current enthusiasm surrounding Java may soon appear to have been somewhat overblown.

Acknowledgement

The author gratefully acknowledges Thomas Kistler as the co-author of the Juice system. Part of this work is being funded by the National Science Foundation under grant CCR-9701400.

References

[Brandis et al. 1995] M. Brandis, R. Crelier, M. Franz, and J. Templ (1995); "The Oberon System Family"; *Software-Practice and Experience*, 25:12, 1331-1366.

[Chang et al. 1991] P. P. Chang, S. A. Mahlke, and W. W. Hwu (1991); "Using Profile Information to Assist Classic Code Optimizations"; *Software–Practice and Experience*, 21:12, 1301-1321.

[Engler 1996] D. R. Engler (1996); "Vcode: A Retargetable, Extensible, Very Fast Dynamic Code Generation System"; *Proceedings of the ACM Sigplan '96 Conference on Programming Language Design and Implementation*, published as *ACM Sigplan Notices*, 31:5, 160-170.

[Franz 1993] M. Franz (1993); "Emulating an Operating System on Top of Another"; *Software-Practice and Experience*, 23:6, 677-692.

[Franz 1994a] M. Franz (1994); *Code-Generation On-the-Fly: A Key to Portable Software*; Doctoral Dissertation No. 10497, ETH Zürich, simultaneously published by Verlag der Fachvereine, Zürich, ISBN 3-7281-2115-0.

[Franz 1994b] M. Franz (1994); "Technological Steps toward a Software Component Industry"; in J. Gutknecht (Ed.), *Programming Languages and System Architectures*, Springer Lecture Notes in Computer Science, No. 782, 259-281.

[Franz 1997a] M. Franz (1997); "Dynamic Linking of Software Components"; *IEEE Computer*, 30:3, 74-81.

[Franz 1997b] M. Franz (1997); "Run-Time Code Generation as a Central System Service"; in *The Sixth Workshop on Hot Topics in Operating Systems (HotOS-VI)*, IEEE Computer Society Press, ISBN 0-8186-7834-8, 112-117.

[Franz & Kistler 1996] M. Franz and T. Kistler (1996); "Slim Binaries"; *Communications of the ACM*, to appear; also available as Technical Report No. 96-24, Department of Information and Computer Science, University of California, Irvine.

[Franz & Ludwig 1991] M. Franz and S. Ludwig (1991); "Portability Redefined"; in *Proceedings of the Second International Modula-2 Conference*, Loughborough, England.

[Juice] M. Franz and T. Kistler; *Juice*; http://www.ics.uci.edu/~juice.

[Hansen 1974] G. J. Hansen (1974); *Adaptive Systems for the Dynamic Run-Time Optimization of Programs* (Doctoral Dissertation); Department of Computer Science, Carnegie-Mellon University.

[Ingalls 1971] D. Ingalls (1971); "The Execution Time Profile as a Programming Tool"; *Design and Optimization of Compilers*, Prentice-Hall.

[Oberon] Department of Information and Computer Science, University of California at Irvine; *Oberon Software Distribution*; http://www.ics.uci.edu/~oberon.

[Kistler & Franz 1997] T. Kistler and M. Franz (1997); "A Tree-Based Alternative to Java Byte-Codes"; *Proceedings of the International Workshop on Security and Efficiency Aspects of Java*, Eilat, Israel.

[Lindholm et al. 1996] T. Lindholm, F. Yellin, B. Joy, and K. Walrath (1996); *The Java Virtual Machine Specification*; Addison-Wesley.

[Inferno] Lucent Technologies Inc.; *Inferno*; http://plan9.bell-labs.com/inferno/.

[Nori et al. 1976] K. V. Nori, U. Amman, K. Jensen, H. H. Nägeli and Ch. Jacobi (1976); "Pascal-P Implementation Notes"; in D.W. Barron (Ed.); *Pascal: The Language and its Implementation*; Wiley, Chichester.

[Welch 1984] T. A. Welch (1984); "A Technique for High-Performance Data Compression"; *IEEE Computer*, 17:6, 8-19.

[Wirth & Gutknecht 1989] N. Wirth and J. Gutknecht (1989); "The Oberon System"; *Software-Practice and Experience*, 19:9, 857-893.

[Wirth & Gutknecht 1992] N. Wirth and J. Gutknecht (1992); *Project Oberon: The Design of an Operating System and Compiler*; Addison-Wesley.

[Wirth 1988] N. Wirth (1988); "The Programming Language Oberon"; *Software-Practice and Experience*, 18:7, 671-690.