Slim Binaries

The traditional path to software portability among various hardware platforms takes a new turn with the use of slim binaries.

The power of computers has increased dramatically over the past 20 years. Not only has the performance of processors risen continuously from one generation to the next and from architecture to architecture, but the interval between these performance steps has also been shrinking steadily. Unfortunately, computer users often cannot take immediate advantage of these improvements, as they are working with software optimized for earlier processor generations. (Consider 16-bit software running on 32-bit processors). The heightened frequency of new processor releases makes it increasingly difficult for software suppliers to furnish adequate program updates in a timely manner. Even then, software is usually tailored only toward the most common implementation of an architecture. For cost reasons and to avoid user confusion, manufacturers do not ordinarily provide separate versions of their programs for different members of a processor family.

At present, we are witnessing a potentially more disruptive transition than the mere appearance of yet another generation of an established processor architecture. Major computer manufacturers are embracing RISC as their new technology platform, although they have a large installed base of CISC machines with a substantial software legacy. In order to offer existing customers a smooth upgrade path to the new RISC architectures, these manufacturers often provide software emulation of a previous CISC platform on their new hardware, or even full dynamic translation from an old instruction set to a new one.

Emulation solves the problem of backward-compatibility, but keeps the software tied to the old architecture. Software developers wishing to give users of the new architecture the maximum possible performance without alienating those who haven't made the transition yet will have to provide their products in both formats simultaneously. Having multiple versions of a program, however, irritates users. It is also discouraged by hardware manufacturers, who want to create the illusion of a unified vendor-specific platform.

This illusion can be conveyed by the provision of *fat binaries*, multiple versions of the same program within a single object file. Providing users with a fat binary containing executable code for different hardware architectures enables them to substitute computers at their leisure among the supported processor families without having to worry about software compatibility.

Unfortunately, this convenience comes at a price. As their name suggests, fat binaries are larger than

Michael Franz and Thomas Kistler

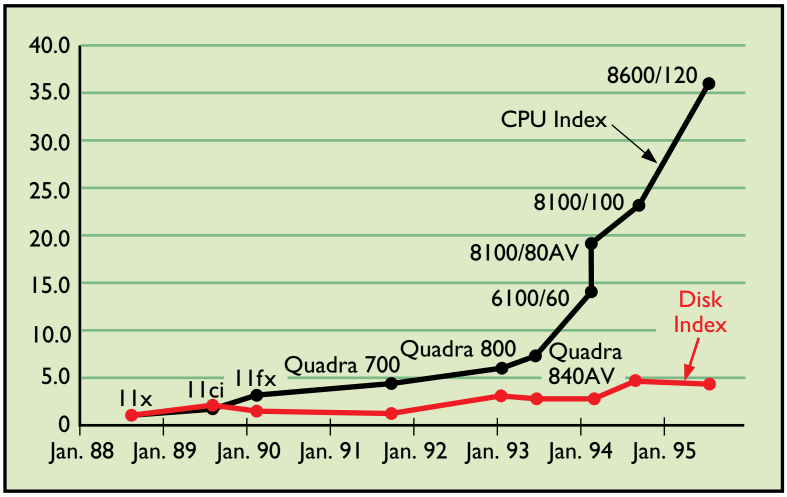

Figure 1. Performance comparison of Macintosh computer models

their counterparts that run on only one architecture. They are also harder to manufacture, as they require the simultaneous operation of multiple compilers. And since each fat binary usually contains only a single code image per supported architecture, rather than separate versions for different implementations of each architecture, fat binaries still do not solve the problem of intra-architectural hardware variation.

We have implemented a system taking a wholly different approach that we call "slim binaries." The object files in our system do not really contain native code for any particular processor at all, but a highly compact, architecture-neutral intermediate program representation. The major achievement of our implementation is that it manages to generate native code of high quality on-the-fly from this intermediate representation, fast enough that it can compete with the loading of compiled code from a fat binary. Slim binaries not only solve the problem of compatibility between different architectures, they also allow to fine-tune the object code towards the specific processor and operating system version that it will run on.

The Importance of Being Slim

As strange as it may sound at first, the key to the effectiveness of slim binaries is indeed their slim-ness. Performance data for off-the-shelf components from major hardware manufacturers suggests the input and output speed of computer memories and peripheral devices is not growing at the same rate as the raw computational power of microprocessors (Figure 1). This is shifting the economics of a variety of computing tasks in such a way that it is becoming increasingly more efficient to recalculate certain intermediate results than to off-load them to secondary storage and read them back later. The central claim of this article is that code generation belongs to the class of computing tasks that exhibit this behavior.

Slim-binary encoding is based on the observation that different parts of a program are often similar to each other. For example, in typical programs there are often procedures that are called over and over with practically identical parameter lists. We exploit these similarities by use of a *predictive compression* algorithm that allows to encode recurring subexpressions in a program space efficiently while facilitating also fast decoding with simultaneous code-generation. Our compression scheme is based on adaptive methods such as LZW [10], but has been tailored towards encoding abstract syntax trees rather than character streams. It takes advantage of the limited scope of variables in programming languages, which allows to deterministically prune entries from the compression dictionary, and uses prediction heuristics to achieve a denser encoding.

Adaptive compression schemes encode their input using an evolving vocabulary. In our encoding, the vocabulary initially consists of a small number of primitive operations (such as *assignment, addition,* and *multiplication*), and of the data items appearing in the program being processed (such as *integer i* and *procedure P*). Translation of the source code into the portable intermediate representation is a two-step process (Figure 2). First, the source program is parsed and an abstract syntax tree and a symbol table are constructed. If the program contains syntax or type errors (including illegal uses of items imported from external libraries), they are discovered during this phase. After successful completion of the parsing phase, the symbol table is written to the slim binary file. It is later required for placing the initial data symbols into the vocabulary of the decoder, and for supplying type information to the code generator.

The abstract syntax tree is then traversed and encoded into a stream of symbols from the evolving vocabulary. The encoder processes whole subtrees of the abstract syntax tree at a time; these roughly correspond to statements on the level of the source language. For each of the subtrees, it searches the current vocabulary to find a sequence of symbols that expresses the same meaning. For example, the procedure call *P(i + 1)* can be represented by a combination of the operation symbols *procedure call* and *addition*, and the data symbols *procedure P, variable i*, and *constant 1.*

After encoding a subexpression, the vocabulary is updated using adaptation and prediction heuristics. Further symbols describing variations of the expression just encoded are added to the vocabulary, and symbols referring to closed lexical scopes are removed from it. For example, after encoding the expression *i + 1*, the special symbols *i-plus-something* and *something-plus-one* might be added. Suppose that further along in the encoding process the similar expression *i + j* were encountered, this could then be represented using only two symbols, namely *i-plus-something* and *j*. This is more space efficient, provided the new symbol *i-plus-something* takes up less space than the two previous symbols *i* and *plus*. Using prediction heuristics, one might also add *i-minus-something* and *something-minus-one* to the vocabulary, speculating on symmetry in the program. This decision could also be made dependent on earlier observations about symmetry during the ongoing encoding task.

In our implementation, compression lowers the input/output overhead by so much that it can almost compensate for the extra time that is needed for on-the-fly code generation. The load-time code-generators that we have constructed are exceptionally fast, orders of magnitude faster than the traditional compilers and linkers they supplant. Again, this is partly a result of reduced input/output costs. Unlike ordinary compilers and linkers, our code-generating loader never needs to generate an output file, while file writing is often even slower than reading.

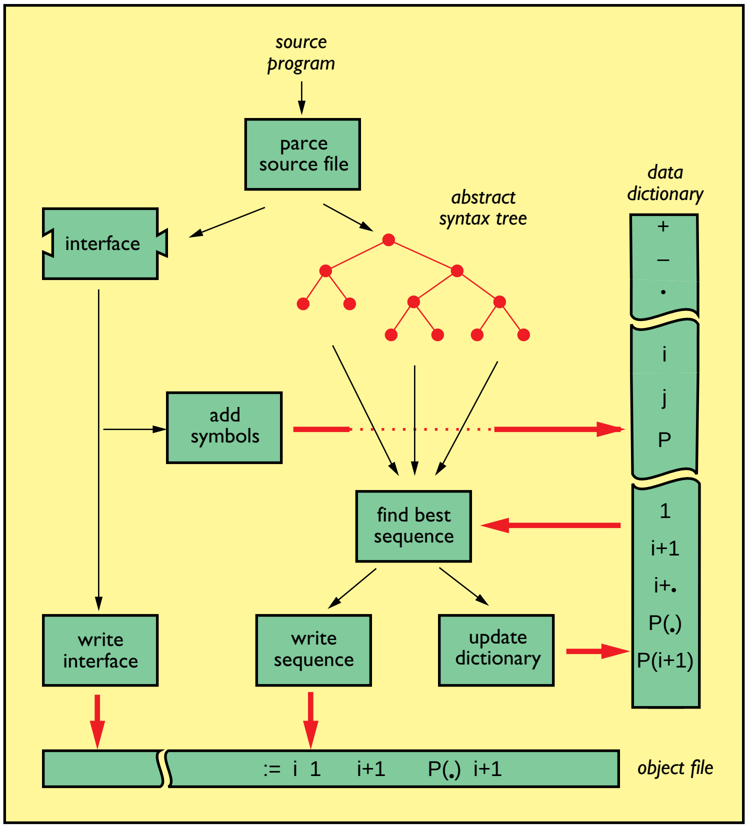

Figure 2. Translation from source code into a slim binary

Software Modules

We have integrated slim binaries into an environment that supports software modules with type-safe separate compilation and dynamic loading. A module is an encapsulated unit of software that interacts with other modules through some well-defined interfaces. Only the interface part of a module is visible from the outside; the rest is considered private and protected from accidental misuse. By crafting interfaces carefully, one can often even guarantee module invariants. Programming languages such as Mesa, Modula-2, Ada, and their descendants provide direct language support for modules.

A module *imports* functionality from the interfaces of other modules and *exports* its own interface. This leads to a hierarchical module-ordering, where the intermediate levels simultaneously serve as libraries to higher-level client modules and as clients to lower-level library modules. In this model, the operating system can be represented as a collection of modules that are situated at the bottom of the module hierarchy. User programs extend this pre-existing hierarchy by adding modules at the top.

The import/export relationships between modules are resolved in a process called "binding" or "linking," which in many modern systems takes place only at the time of loading. It is then called "dynamic loading," which usually also implies that at most one copy of any library module exists in memory at any one time, although several client modules (which might even be part of different application programs) may be using it concurrently. Another implication of dynamic loading is that individual modules are procured independently. One can upgrade an existing module simply by replacing it with another one that fulfills the same interface.

Our implementation provides users with this plug-and-play functionality of software modules in a run-time environment based on the Oberon System [12]. Programmers create software modules using the Oberon programming language [11] and pass them through a compiler that creates slim binaries as its output. The resulting object-modules are then immediately dynamically loadable on any of the supported architectures; generation of the final executable code

Slim binaries not only solve the problem of compatibility between different architectures, they also allow to fine-tune the object code towards the specific processor and operating system version that it will run on.

occurs transparently as part of the dynamic loading process and is invisible to users.

In our system, compilation and dynamic loading are fully type-safe: illegal uses of imported items are detected at compilation (encoding) time, and import/export relationships are again verified at dynamic loading (code generation) time. We use a combination of the methods described by Franz [5] and Crelier [3] to describe intermodule links within slim binaries, yielding a highly flexible link format that permits local library interfaces to contain slight variations across different target platforms. It also allows the addition of functionality to an existing software module without affecting any of its dependents, that is, no recompilations are ever necessary unless a change in a library actually *invalidates* a client (for example, by removing a function called by the client, or by changing its result type).

Extensible Systems

While the original idea of dynamic linking had been to factor out common functions so that they could be shared among several application programs, extensible systems take this idea one step further, allowing the addition of further modules at the top of a module hierarchy at run-time. Application programs can thereby be augmented by additional functionality in reaction to user needs. For example, viewing a certain multimedia document on the World-Wide Web might require the support of a specific video data-format. In an extensible system, a software module providing such support can be downloaded and activated without users ever noticing that the capabilities of their Web browser had increased.

Not only does our implementation provide this kind of run-time extensibility transcending even machine-architecture boundaries, it does so in a type-safe manner. The Oberon programming language, on which our system is based, provides explicit support for extensibility by way of a type extension (subtyping) mechanism. Type extension enables the definition of new data types with extended functionality that are backward-compatible with the data types of the original application. The ability to mechanically exclude type-errors greatly simplifies the construction of extensible systems, which unlike traditional software systems have no final form and cannot be subjected to final total analysis.

There has been considerable popular interest in extensible software systems lately, as general-purpose operating systems are moving forward to embrace dynamic linking and compound-document architectures. A compound document is a container that integrates various forms of user data seamlessly, such as text, graphics, and multimedia. These various kinds of content are supported by independent, dynamically loadable content editors (“applets”) that cooperate in such a way they appear to the end-user as a single unified application. The user’s software environment can be augmented by such applets at run-time when required.

Hence, instead of a single software package that does everything, users obtain only those components they actually need. This is ideal as long as users are authoring new documents all the time, but what happens if a user wants to work with an already existing document? How does he or she find out which components are needed, and where to obtain these components?

Possible solutions, that could also be combined with each other, include a basic extensible system that could provide a special fall-back mode to handle cases of missing components; for instance, by representing the corresponding parts of the document by non-functional outlines or bitmaps. Beyond that, the system could attempt to download missing components from a remote server; this would, of course, require a network connection. And, even more unusual at first sight, the software components themselves could be included within the document, leading to fully self-contained *active documents.*

Slim binaries facilitate elegant implementations of the latter two solutions. Since they are independent of the target architecture on which they will eventually run, they require only a single version of each component to be stored on a central server for downloading-on-demand scenarios. Further, their compactness makes downloading particularly efficient. Both of these qualities also make the inclusion of software components inside of documents practicable. In fact, since documents form the basis of information interchange, it is entirely feasible to distribute whole application packages as documents that represent the corresponding user interface. In the user's view, the application *becomes* the user interface!

A further potential application area of slim binaries is the encoding of intelligent agents—software entities that move from machine to machine and perform local computing tasks at different sites. The slim binary format offers the necessary target-machine independence for this purpose while simultaneously providing a flexible linking scheme that accommodates minor variations in library interfaces without jeopardizing type-safety when mobile agents call such libraries. Moreover, the small size of slim binaries makes the format particularly attractive when network transfer of agent code is required.

Results

Slim binaries have been available for MC68020-based Apple Macintosh computers since late 1993. Originally a by-product of the first author's doctoral-dissertation work [7], the slim binary format meanwhile has become a core technology of ETH's popular MacOberon software distribution. MacOberon is a package that emulates the Oberon operating environment on the Macintosh platform [6]. The complete package consists of a core system, incorporating the central functions such as memory and file management and a module loader, as well as a suite of application modules that can be dynamically linked and run from within this environment.

Before the arrival of slim binaries, we used to maintain two separate versions of MacOberon, one each for MC68020- and PowerPC-based Macintoshes. Replacing their respective suites of natively compiled application modules by a single set of slim-binary encoded modules has reduced the total size of our software distribution quite dramatically (Figure 3). Although it required the addition of a dedicated code-generating loader to each of the two core systems, the additional space required for these code-generating loaders was insignificant compared to the savings of not having to duplicate all of the application modules. Almost paradoxically, we were able to hide the distinction between the two kinds of Macintosh versions from users altogether by embedding their two core systems into a single fat binary.

On-the-fly code generation turned out to be so reliable that the provision of native binaries could be discontinued altogether in MacOberon, resulting in significantly reduced maintenance overhead for the distribution package.

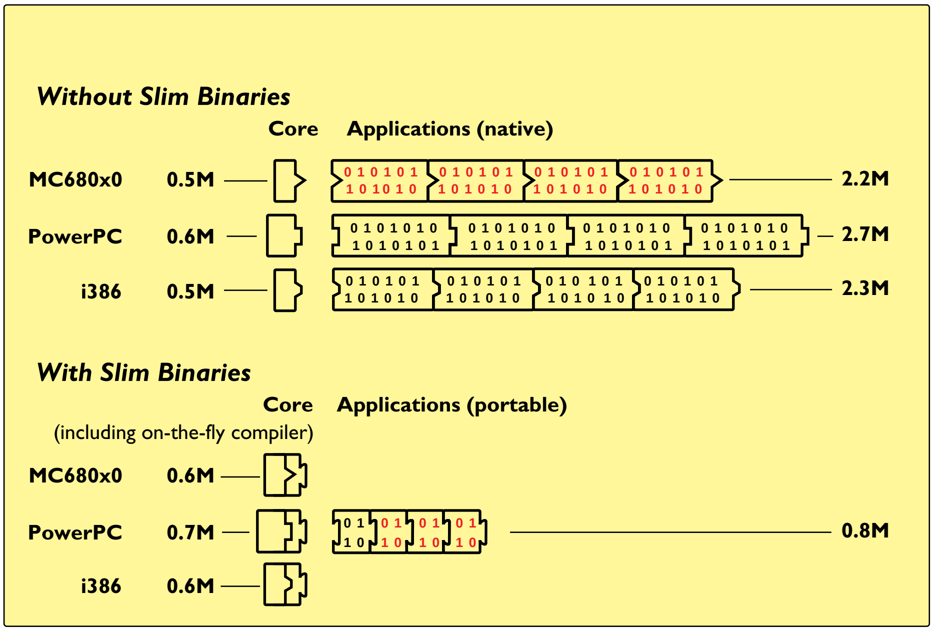

Figure 3. Influence of slim binaries on the size of our software distribution

The economies of size and maintenance effort effected by our use of slim binaries don't end with MacOberon. They are easily multiplied by adding further code-generating loaders. A third of these for the i80x86 architecture is already available, enabling WinOberon users (on the Microsoft Windows 95 platform) to use modules from the Macintosh distribution as if they contained native Intel code. Of course, it is also possible to generate slim binaries on any of the supported architectures, as the compiler that produces them is itself part of the portable application suite.

Hence, slim binaries provide seamless cross-platform portability. By removing the need for multiple compiled versions of a program, they can also significantly reduce the overall space required for storing software that needs to run on several different hard-

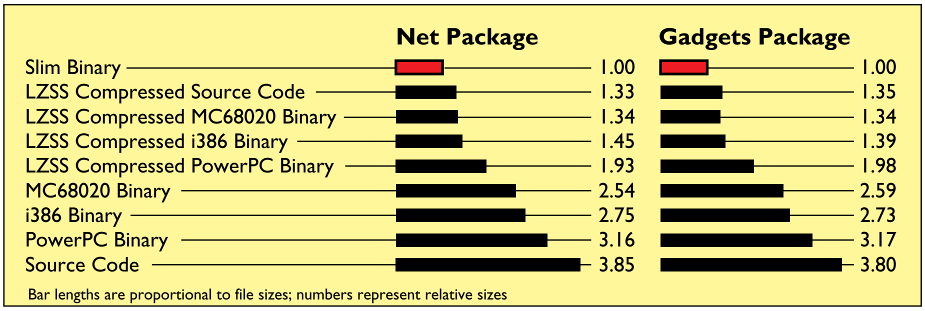

Figure 4. Size comparison between different representations of the same program suite

ware platforms. But interestingly enough, the representation even reduces the space requirements for computer programs on an absolute scale. Slim binaries are not only more compact than ordinary compiled code, but also surpass the information density effected by popular variants of the LZW [10] compression scheme when applied to either object code or to source code after the removal of program comments.

We have summarized these results (Figure 4) for two program suites from ETH's Oberon software distribution: the Net Package includes a Web browser, a Telnet application with VT100 emulation, an email program based on POP/SMTP, an Internet News reader, and applications supporting the gopher, finger, and ftp protocols. The size of this package on the native platforms is around 500-600KB of object code. The Gadgets Package on the other hand comprises the graphical user interface tool-kit of Oberon and is on the order of 2MB in size.

The initial claim of this article was that reducing the size of object files allows to exploit the different growth rates of processor power relative to storage speeds, so that on-the-fly code generation becomes viable. Our measurements (Figure 5) compare our code-generating loader to its respective native counterparts on a variety of hardware platforms (the same as listed in Figure 1) introduced over a six-year period. Our benchmark measures the time required for loading all of the applications in the Net Package; the values referring to slim binaries additionally include the time of generating the appropriate executable code.

These results suggest that the overhead required for on-the-fly code generation is largely a function of processor power, and substantiate our initial claim that code-generation should be considered an I/O-bound process. Of course, as processors become more complex, the techniques required to generate good code for them also tend to be more elaborate. It is an open question whether the speed of processors will grow faster than the complexity of generating adequate code for them, but we are confident this is the case. The code quality achieved by our current generation of code-generating loaders is comparable to that of ETH's latest generation of Oberon compilers; the two build on a common family of compiler back-ends [1].

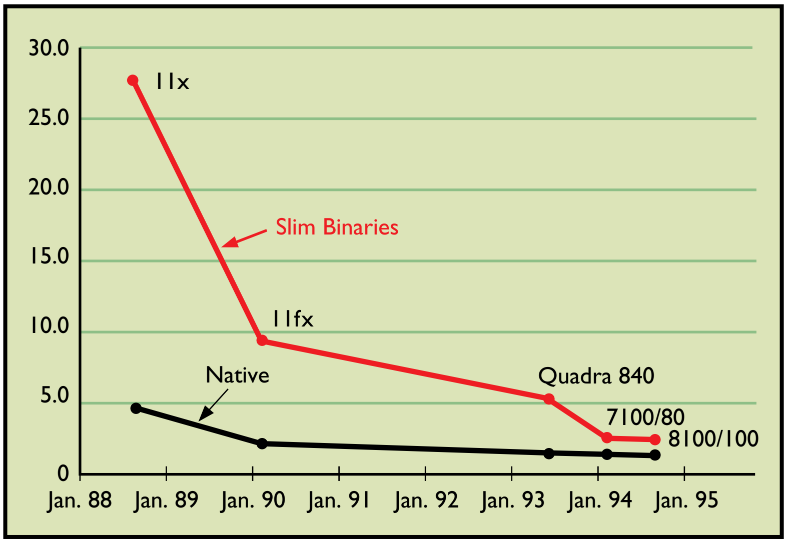

Figure 5. Time required for loading all of the applications in the Net Package

It is also important to note that the *absolute delay* an interactive user experiences when code is generated dynamically is more important than the *relative speed* in comparison to traditional loading. On the fastest computers of our benchmarks, it takes about two seconds to simultaneously load all of the applications contained in our Net Package from slim binaries. Although this is still almost twice as much as required for native binaries, the extra second is within the range that we have found users to be willing to tolerate. In return for a minimally increased application-startup time, they gain the benefit of cross-platform portability without sacrificing any run-time efficiency.

Further, typical users of our system do not start all of the applications in the Net Package at the same time. Quite the opposite. Due to the extensible, modular structure of our system, the incremental workload of on-the-fly code generation is usually quite small. Most of the applications are structured in such a manner that seldom-used functions are implemented separately and linked dynamically only when needed. Moreover, there are many modules that are shared among different applications and need to be loaded only once. Hence, the effective throughput demanded of our on-the-fly code generator is much smaller than might be expected when extrapolating from systems based on statically linked application programs.

Related Work

On-the-fly code generation has traditionally been used for improving the performance of dynamically typed object-oriented languages. Implementations such as Deutsch and Schiffmann's Smalltalk-80 [4] and Chambers and Ungar's Self [2] have benefited from type information available at run-time that, in the absence of static typing, could not have been extracted from source code.

The use of a machine-independent intermediate representation for achieving portability is also an old concept. The idea has recently undergone a renaissance, as exemplified by the ANDF project of the Open Software Foundation [9], Sun Microsystems' Java [8], and our own work [7]. Of these projects, the OSF-ANDF project is the most conventional, as it never seems to have contemplated code generation on-the-fly; the available literature treats code generation strictly as an off-line process.

The Java project at Sun Microsystems has recently gained considerable popularity and industry support. Early implementations of Java were based on the interpreted execution of the intermediate representation, but now just-in-time compilers are appearing. These compilers operate similarly to our own on-the-fly compiler, except they compile individual procedures at the time of invocation rather than whole modules at the time of loading. The granularity of our approach is better suited for generating optimized code.

The tree-based encoding used inside of slim binaries is markedly different from the virtual-machine representation on which Java is based [8]. It has the apparent disadvantage that it cannot simply be interpreted byte-by-byte. Rather, every symbol in a slim binary describes a subtree of an abstract syntax tree in terms of all the subtrees that precede it in the file. Constructing a decoder for slim binaries is so complex that it would be relatively pointless to attach a simple interpreter at its output. Instead, all of our implementations are integrating the decoding step with on-the-fly code generation. The two can be interweaved elegantly.

On the other hand, a tree-based encoding has several important advantages over a linear stream of byte-codes for some virtual machine. It preserves the control-flow structure of the original program, which makes it easier to subsequently generate high-quality native code on-the-fly. For example, modern processors have several functional units that require a certain instruction mix in order to operate at top speed. By reordering certain mutually independent instructions, a better instruction mix may be achieved. A tree-based encoding maintains the notion of a basic block and makes it relatively easy to decide if two instructions are mutually independent. In effect, the tree structure needs to be laboriously reconstructed from a linear virtual-machine representation before comparable optimizations can be performed.

Further, our tree-based encoding has an advantage, the effects of which we are only just beginning to contemplate. We have reason to believe that our approach avoids many of the security issues that are difficult to solve in Java, or in any virtual-machine representation for that matter. As explained earlier, we represent a program by a stream of symbols from an evolving vocabulary. By the very definition of the encoding scheme, this vocabulary at all times contains only those data-reference symbols (for example, procedures and variables) that can be accessed legally at the current position in the program. As stated, this dictionary pruning was introduced originally to minimize the size of the dictionary and achieve a denser encoding. It does, however, also make it impossible to construct, even by hand, a slim binary that violates the lexical scoping rules of our source language.

Java's byte-code instructions are at a much lower semantic level. Although it is possible to verify that a given byte-code sequence doesn't perform any illegal action, this requires data-flow analysis and a partial repetition of the compiler's integrity checks. In effect, the slim binary format makes this particular data-flow analysis unnecessary. It also allows for highly efficient load-time integrity checking, as every node in the encoded abstract syntax tree is fully typed.

Outlook

Our present implementation of slim binaries is restrictive as it supports exactly one source language, Oberon. In this respect, our system presently doesn't do much better than abstract-machine-based portability schemes in which the instruction set of the virtual machine is explicitly crafted to support a particular source language. While we do not foresee any difficulties in encoding syntax trees for other languages, possibly even using the identical format, the suitability for other languages has yet to be established by an actual implementation.

The main thrust of our current research is focused on improving code quality. Our implementations so far are all based on ETH's well-established family of compiler back-ends that produce high quality code comparable to that of straightforward commercial compilers [1]. On the newer RISC architectures, however, these back-ends cannot compete with highly optimizing compilers. Of further concern to our particular application of load-time code generation is the fact that optimizers for certain RISC architectures may have vastly different run-time characteristics than the compilers we have been using so far.

Consequently, we are now pursuing a two-tier strategy of code generation. Rather than compiling every module exactly once when it is loaded and then leaving it alone, we use a background process executing only during idle cycles that keeps compiling the already loaded modules over and over. Since this is strictly a re-compilation of already functioning modules, and since it occurs completely in the background, this process can be as slow as it needs to be, allowing the use of far more aggressive, albeit slower, optimization techniques than would be tolerable in an interactive compiler. When background code-generation has completed, the code-images of the re-generated modules are substituted for their older counterparts.

Periodic re-optimization of already executing code allows to fine-tune the code-generator's output beyond the level generally achievable by static compilation. Not only does it enable run-time profiling data from the current execution to drive the next iteration of code optimization, it also makes it possible to cross-optimize application programs and their dynamically loaded extensions and libraries. We are currently experimenting with global optimization techniques pioneered by incremental compilers and link-time optimizers. Among them are register allocation and code inlining across module boundaries, global instruction scheduling, and cache optimization. Extensible systems present new challenges to these old problems, as no closed analysis is possible due to the fact that further modules can be added to the module graph at any time.

We are also working on other aspects of highly dynamic extensible systems, specifically the problems posed by run-time code-generation in heterogeneous, distributed environments connected over unsecure data links. Our primary concern in this area at the moment is security, as we are devising ways in which an extensible system can protect itself from malicious migrating objects attempting to cause damage.

Finally, we are currently developing a family of plug-in extensions to the Netscape Navigator and Microsoft Internet Explorer Web browsers that provide slim-binary support even outside of the Oberon environment. Code-named "Juice," we have implemented two prototypes for the Macintosh and Windows platforms that support interactive, slim-binary-encoded applets embedded within Web pages. Each plug-in contains an on-the-fly code generator that is activated by the browser upon downloading of the applet. Similar in functionality, but not in the underlying technology, Juice is intended to complement Java. The two kinds of applets can live on the same page and communicate with each other through the browser. The Juice plug-ins can be downloaded from our Web site.

While the road still left to travel is long, it seems as if slim binaries and related approaches have finally brought us a step closer to the dream of mass-produced software components envisioned by MacIlroy so many years ago.

The greater part of the research described in this article was carried out while both authors were at Institut für Computersysteme, ETH Zürich, Switzerland.

References

1. Brandis, M., Crelier, R., Franz, M., and Templ, J. The Oberon system family. *Softw.-Pract. Exp. 25*, 12 (1995), pp. 1331–1366.
2. Chambers, C., Ungar, D., and Lee, E. An efficient implementation of SELF, a dynamically typed object-oriented language based on prototypes. In *Proceedings of the OOPSLA '89 Conference*. ACM, New York, pp. 49–70.
3. Crelier, R. Extending module interfaces without invalidating clients. *Struct. Pro. 16*, 1 (1996), pp. 49–62.
4. Deutsch, L.P., and Schiffmann, A.M. Efficient implementation of the Smalltalk-80 system. *Conference Record of the 11th Annual ACM Symposium on Principles of Programming Languages*. (1984, Salt Lake City), pp. 297–302.
5. Franz, M. The case for universal symbol files. *Struct. Prog. 14*, 3 (1993) pp. 136–147.
6. Franz, M. Emulating an operating system on top of another. *Softw.-Pract. Exp. 23*, 6 (1993), pp. 677–692.
7. Franz, M. Code-Generation On-the-Fly: A Key to Portable Software (Doctoral Dissertation No. 10497). ETH, Zurich published by Verlag der Fachvereine, Zürich, 1994.
8. Lindholm, T., Yellin, F., Joy, B., and Walrath, K. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, Mass. 1996.
9. Open Software Foundation. OSF Architecture-Neutral Distribution Format Rationale, 1991.
10. Welch, T.A. A technique for high-performance data compression. *IEEE Comp. 17*, 6 (1984), pp. 8–19.
11. Wirth, N. The programming language Oberon. *Softw.-Pract. Exper. 18*, 7 (1988), pp. 671–690.
12. Wirth, N., and Gutknecht, J. The Oberon *System. Softw.-Pract. Exp. 19*, 9 (1989) pp. 857–893.

Further information is available from: http://www.ics.uci.edu/~franz/SlimBinaries.html

Michael Franz is an assistant professor in the Department of Information and Computer Science at the University of California at Irvine.

Thomas Kistler is a Ph.D. student in the Department of Information and Computer Science at the University of California at Irvine.