

Adaptive Compression of Syntax Trees and Iterative Dynamic Code Optimization: Two Basic Technologies for Mobile-Object Systems

Michael Franz

Department of Information and Computer Science

University of California

Irvine, CA 92697-3425

Abstract

We are designing and implementing a flexible infrastructure for mobile-object systems. Two fundamental innovations distinguish our architecture from other proposed solutions. First, our representation of mobile code 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 than linear abstract-machine representations such as p-code or Java byte-codes. The extra structural information that is contained in our mobile-code format is directly beneficial for advanced code optimizations.

Second, our architecture achieves superior run-time performance by integrating the activity of generating executable code into the operating system itself. Rather than being an auxiliary function performed off-line by a stand-alone compiler, code generation constitutes a central, indispensable service in our system. Our integral code generator has two distinct modes of operation: instantaneous load-time translation and continuous dynamic re-optimization.

In contrast to just-in-time compilers that translate individual procedures on a call-by-call basis, our system's integral code-generator translates complete code-closures in a single burst during loading. This has the apparent disadvantage that it introduces a minor delay prior to the start of execution. As a consequence, to some extent we have to favor compilation speed over code quality at load time.

But then, the second operation mode of our embedded code generator soon corrects this shortcoming. Central to our run-time architecture is a thread of activity that continually optimizes all of the already executing software 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 same code is supplanted by the newly generated one and re-optimization starts over. By constructing globally optimized code-images from mobile software components, our architecture is able to reconcile dynamic composability with the run-time efficiency of monolithic applications.

1. Introduction

This chapter introduces two new basic technologies for mobile object systems: a highly compact representation for portable code and a run-time architecture that reconciles the traditionally conflicting goals of modularity and performance through the use of a dynamic code optimizer embedded within the operating system. The two technologies are complementary to each other: while the high semantic level of the machine-independent representation makes it particularly well suited for supporting code optimizations, its conciseness accelerates I/O operations, partially compensating for the effort of load-time code generation. The unusual compactness of object files also facilitates their retention in a memory cache, leading to more efficient optimization cycles later.

In the following, we give an overview of each of the two technologies, pointing to further publications and on-line documentation where appropriate. The *Slim Binary* mobile-code format [FK96] is the outgrowth of the author's doctoral dissertation work [Fra94a, Fra94b], while the system architecture incorporating dynamic optimization is ongoing research at UC Irvine. We also report on the current state of our implementation, specifically the availability of an integrated authoring and execution environment [Oberon] for mobile software components that is based on the Oberon System [WG89, WG92], and a family of plug-in extensions [Juice] for the *Netscape Navigator* and *Microsoft Internet Explorer* World Wide Web browsers that recreate this execution environment so that Oberon-based components can be used within these browsers.

2. Representing Mobile Code

In the course of the past six years, the author has designed and successfully implemented a portability scheme for modular software that is based on dynamic code generation at load time [FL91, Fra94a, Fra94b, FK96]. At the core of a suite of implementations is a machine-independent program representation called *slim binaries*¹. This representation is based on adaptive compression of syntax trees and achieves exceptional information densities. For example, it is more than twice as dense as Java byte-codes. In fact, the author knows of no denser program representation; standard data compression algorithms such as LZW [Wel84] applied to either source code or object code (for any architecture, including the Java virtual machine) perform significantly worse than our dedicated syntax-tree-directed method (Figure 1).

¹ The name “slim binary” was deliberately chosen to contrast “fat binary”, which has been used to describe object files that contain multiple instruction sequences for different target architectures. Slim binaries provide a similar functionality as fat binaries, namely the ability to be executed on more than one hardware architecture, but they consume only a fraction of the storage space.

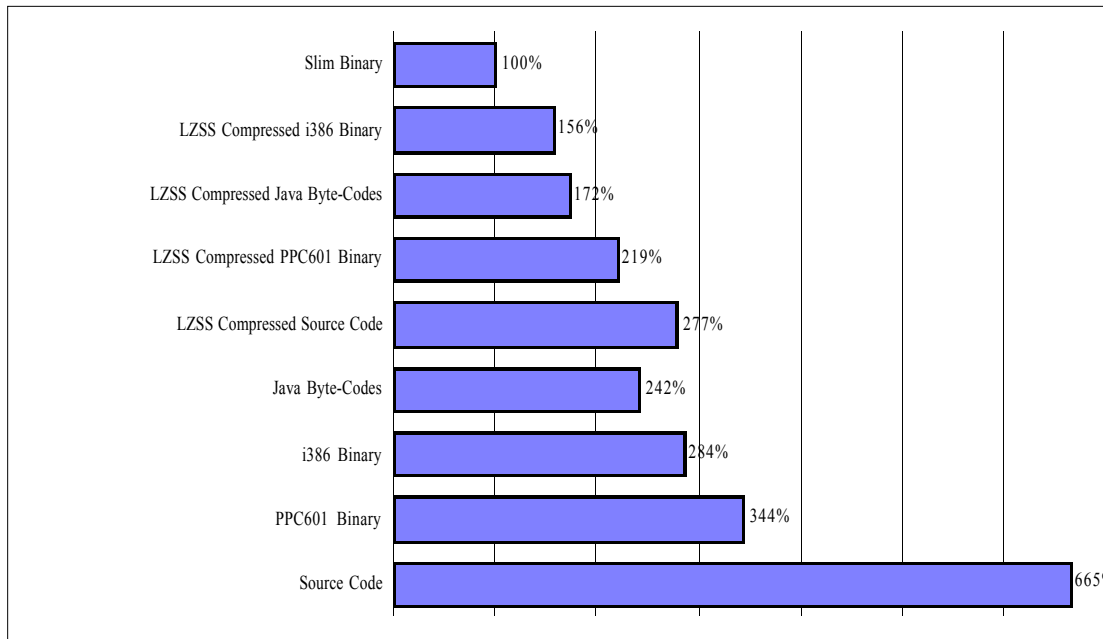


Figure 1: Relative Size of Representative Program Suite in Various Formats

Reducing the network-transfer time of mobile components (by a factor of more than two in comparison to Java byte-codes, for example) is an advantage that should not be underestimated, considering that 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.

Taking the *Oberon System* developed by Niklaus Wirth and Jürg Gutknecht [WG89, WG92] as a starting point, the author and his graduate students have implemented a family of run-time extensible systems [Oberon] in which the slim binary format is used instead of native object code, yielding seamless cross-platform code portability. Currently, we support three different architectures: *MC68020* and *PowerPC* under the Macintosh operating system, and *i386* under Microsoft Windows 95. In all three implementations, native object code is generated on-the-fly at load time from the slim binary representation, in a manner that is transparent to the user. From the user's perspective, "object files" simply become portable. The implemented systems support dynamic linking (with dynamic code generation) of extension code into running applications and the target-machine independent use of mobile code distributed as a slim binary across a network.

In the systems we have implemented, the time required for reading "object files" is reduced dramatically due to the compactness of the slim binary representation. This applies to code read from an external storage medium as well as to code received over a network. The time that has been saved due to reduced input overhead is then instead

spent on dynamic code generation at load time, making the implemented scheme almost as fast as traditional program loading.

The interesting aspect of trading reduced input/output overhead for a greater amount of processing at load time is that hardware technology is currently evolving in its favor. Raw processor power is growing much more rapidly than the speed of input and output operations (Figure 2). Any computer application that reduces its input/output overhead at the expense of additional computations can benefit from this effect in the long run, even if the immediate performance gain doesn't seem to reward an increased algorithm complexity. Code generation happens to be a particularly good example, because processor instruction sets are not optimized for information density but have other constraints such as regularity and ease of decoding. Hence, object files are usually much larger than they need to be.

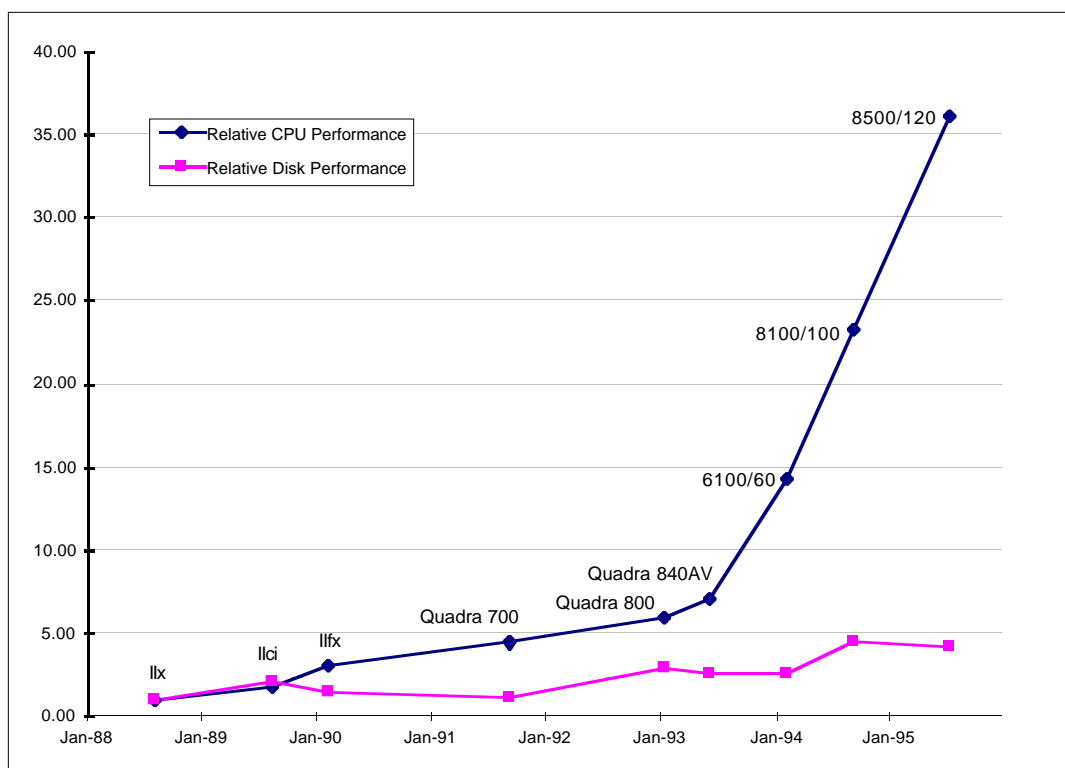


Figure 2: Different Growth Rates of Processor Power vs. I/O Speed for Different Models of the Apple Macintosh Computer Family (as Measured by the Tool Speedometer 4.02)

The process illustrated in Figure 2 has had the effect that dynamic code-generation at load-time has now become practical and will continue to increase its appeal. As Figure 3 illustrates, the additional cost of using slim binaries rather than native code has plunged dramatically over the last 6 years as the performance gap between processors and storage

has widened. Figure 3 compares the times required for reading and dynamically compiling from slim binaries all of the applications in a large representative suite of Internet applications (a *WWW* browser, a *Telnet* application with *VT100* emulation, an electronic mail system, and further tools) versus simply reading pre-fabricated executables of the same applications.

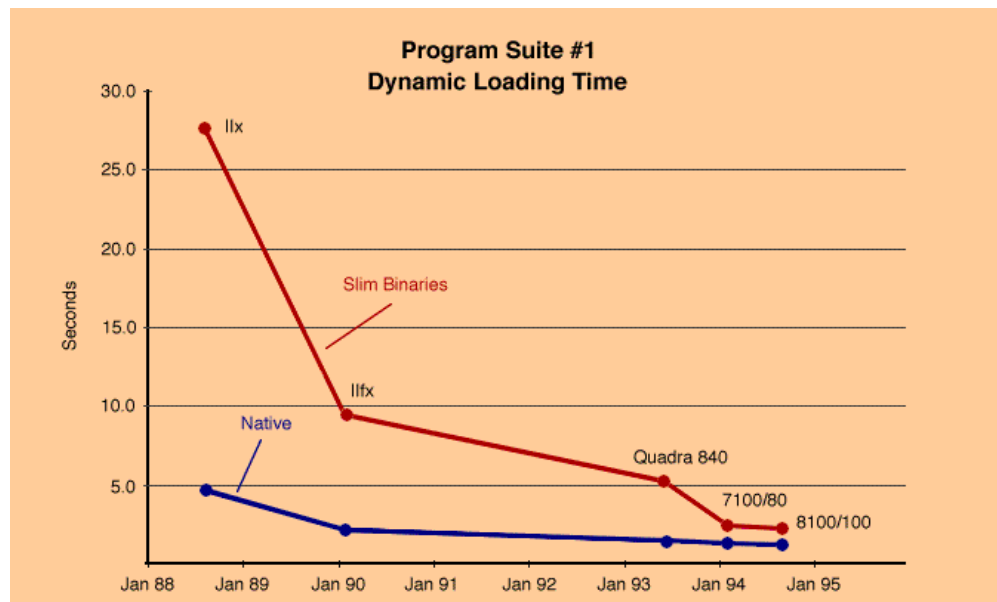


Figure 3: Time Required for Loading Representative Program Suite

Of course, as processors become more complex, the techniques required to generate good code for them tend to be more elaborate also. It is still an open question whether the speed of processors will grow faster than the complexity of generating adequate code for them. However, as described in the next section, the inclusion of dynamic re-optimization in our architecture makes this question largely irrelevant.

It should also be noted that the *absolute delay* that 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 the benchmark suite from slim binaries. Although this is still almost twice as much as is 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; all code generation occurs strictly before the execution commences.

Further, typical users of our system do *not* start all of the applications in the program suite 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.

2.1 The Slim Binary Representation: Some Technical Details

Unlike other program representations that have been proposed for achieving software portability, such as p-code [NAJ76] or Java byte-codes [LYJ96], the slim binary format is based on adaptive compression of syntax trees, and not on a virtual-machine representation. Every symbol in a slim-binary encoding describes a sub-tree of an abstract syntax tree in terms of all the sub-trees that precede it. During the encoding of a program, more and more such sub-trees are added to its slim-binary representation, steadily evolving the “vocabulary” that is used in the encoding of subsequent program sections.

The key idea behind this encoding is the observation that different parts of a program are often similar to each other. For example, in typical programs there are often procedures that get 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 sub-expressions in a program space-efficiently while facilitating also time-efficient decoding with simultaneous code-generation. Our compression scheme is based on adaptive methods such as LZW [Wel84] but has been tailored towards encoding abstract syntax trees rather than character streams. It also 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 4). 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

After encoding a sub-expression, 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 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 that 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 session.

2.2 Advantages and Disadvantages of Slim Binaries

Using a tree-based, nonlinear representation as a software distribution format has the apparent disadvantage that the portable code cannot simply be interpreted byte-by-byte. The semantics of any particular symbol in a slim-binary-encoded instruction stream are revealed only after all symbols preceding it have been decoded. Conversely, the individual symbols in an abstract-machine representation 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, we gain several important advantages in addition to the extreme compactness already mentioned.

The tree-based nature of our distribution format constitutes a considerable advantage when the eventual target machine has a super-scalar architecture requiring advanced optimizations. Many modern code-optimizations rely on structural information that, although readily available at the syntax-tree level, is lost in the transition to linear intermediate representations such as Java byte-codes, and whose reconstruction is difficult. For example, the code for a processor that provides multiple functional units can often be improved by reordering the individual instructions so that the functional units are operating in parallel (instruction scheduling). Two instructions in the program can be exchanged if they are functionally independent of each other, and if no branch originates or terminates between them. The tree-based slim binary representation preserves the control-flow data required for this and related optimizations, giving it an edge over linear representations that require an additional time-consuming pre-processing step for extracting the needed structural information. In order to achieve the same efficiency with byte-codes, these would have to be instrumented with hints about block boundaries [Han74], which is inelegant and space-consuming.

One might argue that the presence of this extra semantic information also makes the *reverse engineering* of slim binaries easier, exposing the trade secrets of software developers. It is true that any intermediate format that preserves the abstract structure of programs can be reverse-engineered to produce a “shrouded” source program, i.e. one that contains no meaningful internal identifiers [Mac93]. However, with current technology, reverse-engineering to a similar degree is possible also from binary code and from linear byte-codes. Many of the algorithms that have been developed for object-level code-optimization [DF84, Dav86] are useful for these purposes. Moreover, the statement [DRA93] is probably correct that portable formats are such attractive targets to reverse-engineer that suitable tools will become available eventually, regardless of how difficult it is to produce such tools. It would, therefore, not make much sense to jeopardize the advantages of slim binaries in an attempt to make reverse-engineering more difficult.

As a further advantage of our mobile-code representation, unlike most linear representations, every node in a slim-binary encoded syntax tree is strongly typed, and all variable references are accompanied by symbolic scope information. This simplifies the task of *code verification*. The problem with mobile code is that it may turn out to be malicious or faulty and thereby compromise the integrity of the host system. For example, variables in private scopes must not be accessed from the outside, but a mobile program may have been generated by a rogue compiler that explicitly allows these illegal accesses. Hence, incoming code must be analyzed for violation of type and scoping rules. For our tree-based representation, this analysis is almost trivial; for linear code, it is not.

3. A Run-Time Architecture based on Dynamic Re-Optimization

The *granularity of code generation* has a profound influence on the quality of the resulting object code. In general, large pieces of code provide more opportunities for optimizations than small ones. Traditional compilers limit their search for possible optimizations to the individual compilation unit: a recent study by Aigner and Hölzle [AH96] demonstrates how the execution times of several benchmark programs can be improved considerably simply by combining all of the source files into one large file prior to compilation. More sophisticated link-time optimization strategies, such as the Titan/Mahler system [Wal92], partially overcome this effect, as they are able to perform certain inter-module optimizations on already compiled code. However, even these advanced schemes still assume a context of static linking and are unable to optimize calls to dynamic link libraries.

As an illustration of the underlying problem, consider an application program A that calls a routine R . The compiler may want to inline R at a specific call site, replacing the call to R by an instance of R 's body into which the actual parameters have been hard-

coded. Since the body of R can then be optimized in the context of A , this might lead to significant further simplifications. However, none of this is possible if R is implemented in an external dynamic-link library, because the library may be changed independently of the application program.

Hence, the needs of optimizing compilers run counter to the principle of dynamic composability that fundamentally underlies mobile-object systems. These are usually made out of a large number of relatively small components. Consequently, they have to pay a performance penalty for their added flexibility, as optimizations such as procedure inlining and inter-procedural register allocation can usually not be performed across component boundaries at reasonable cost. Note that the technique of *just-in-time* compilation that is currently gaining in popularity is compounding the problem, as it compiles methods individually as they are called.

Our system is able to overcome these limitations. The key idea is to perform the translation from the slim binary distribution format into executable code not just once, but to do so continually. When a piece of mobile code is initially activated in our system, its slim-binary representation is translated into native code a single burst, putting compilation speed ahead of code quality so that execution can commence immediately. However, the resulting code will usually not be executed for long. Immediately after a component has become active, its code becomes a candidate for *re-optimization*.

In our architecture, code generation is provided as a *central system service*. A low-priority thread of control uses the idle time of the machine to perpetually *integrate* all components loaded at that moment, recompiling the already executing code base again and again in the background into fully optimized, quasi-monolithic code images. Whenever such an image has been constructed, it supersedes the previously executing version of the same code and re-compilation commences again (Figure 5).

Since a single code image is being constructed out of a large number of individually-distributed parts, code optimizations that transcend the boundaries of these parts can be put to use without limitation. Furthermore, all of this occurs in the background while an alternate version of the same software is already executing in the foreground, so that the speed of re-compilation is not critical. This means that far more aggressive optimization strategies can be employed than would be possible in an interactive context. Run-time profiling data can also be exploited during re-compilation [Ing71, Han74, CMH91] so that successive iterations yield better and better code.

Hence, iterative dynamic re-compilation can provide the run-time efficiency of a globally optimized monolithic application in the context of mobile objects, or even surpass it due to the fine-tuning that profiling makes possible. It leads to a new execution model that combines the advantages of conventional application programs with those of dynamically configurable component-based systems. We call this model *quasi-monolithic*, since it exhibits most of the characteristics of a monolithic application. Unlike monolithic applications, however, a quasi-monolithic executable is extensible and

can be augmented at any time by further components that are linked to the monolithic core. Eventually, the components “outside” of the monolithic core will get drawn inside it, as the run-time code generator integrates them on successive iteration cycles.

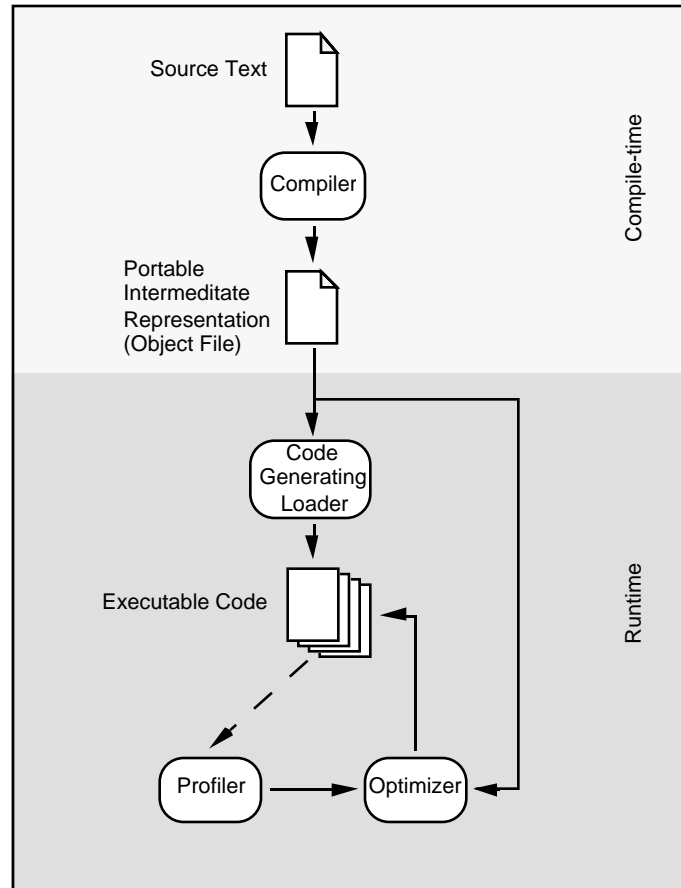


Figure 5: Schematic Overview of the Run-Time Architecture

3.1 Quasi-Monolithic System Architectures: Further Considerations

Run-time re-optimization affords not only a profitable utilization of a processor’s idle cycles, but is also an ideal task to be handled by a temporarily unused processor in a multi-processor system. Such a re-optimization cycle requires no synchronization with the rest of the system until it is completed. This applies even to memory accesses: a run-time code generator takes a series of (possibly cached) “portable object files” as its input and produces a completely new code image as its output; it need not have access to the currently executing object code and can make a copy of any profiling data prior to

beginning its cycle. Hence, run-time code generation, if executed concurrently with application programs by a dedicated processor, could be implemented in such a way that it doesn't slow down the remaining processors (although we haven't done this yet). As processor costs decline, it may become perfectly viable to add a further processor to a computer system specifically for handling dynamic re-optimization. This may in fact be more cost-effective than attempting true multiprocessing when the problem set is not well suited for parallelization.

There is, however, still a price to be paid for the increased performance of quasi-monolithic code: inlining, loop unrolling and other optimizations such as *customization* [CU89] can lead to a much larger memory requirement for the whole system. This behavior is likely to be more pronounced for well-structured modular applications, in which all common functions have been "factored out", than for application programs that are not so well-structured, since dynamically compiling a collection of modules into a quasi-monolith has the effect of "multiplying out" many of the functions that were previously factored. Hence, while programs that make pervasive use of shared dynamically-linked libraries are expected to benefit most from dynamic re-optimization, they are also likely to demonstrate the downside of the technique most clearly. While we contest that the rapidly diminishing cost of memory is a small price to pay for the advantage of achieving high performance and modularity simultaneously, one of the goals of our research is to develop techniques for limiting the memory requirements of our method. It should be possible to eventually develop good heuristics for this purpose based on the dynamic history of recompilation effects and the available run-time profiling data.

We expect that integral run-time code-generation and re-optimization will become a common feature in component-based systems. Already today, operating-system manufacturers are beginning to support for software-portability solutions such as the Java Virtual Machine [LYJ96]. Initially, this support will come in the form of just-in-time compilers that translate portable program representations into the native code of a target machine. To attain further benefits, however, the operating system's code itself will need to be accessible to the run-time code generator, so that all-encompassing quasi-monolithic code images can be created. We trust that this additional step will eventually also be taken in mainstream operating systems, as it promises better performance for the small and rapidly diminishing price of greater memory consumption.

4. Current State of the Implementation

The work described in this chapter has originated and continues to evolve in the context of the *Oberon System* [WG89, WG92]. 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 [Wir88] and has been ported to a wide variety of platforms [Fra93, BCF95].

For all practical purposes, Oberon's modules supply exactly the functionality that is required for modeling mobile objects. 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 globally unique naming of qualified identifiers. Hence, when we have been talking about "components", or "objects" above, we were referring to Oberon modules.

We have already come quite far in deploying the ideas described here 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 tiny amount of "glue" to interface with the respective host operating systems.

The latest release of this distribution also contains an authoring kit for creating mobile components that are based on a reduced system interface modeled after the Java-Applet-API. We have created *plug-in extensions* for the *Netscape Navigator* and *Microsoft Internet Explorer* families of WWW browsers, again both for the Macintosh (PowerPC) and Microsoft Windows (i80x86) platforms, that implement runtime environments providing this API in conjunction with on-the-fly code generation. Users are able to create components within the Oberon environment that can then be run not only within Oberon, but also from within the third-party browsers. We call our mobile-component architecture "Juice" [Juice], as it complements Java in many ways. The two kinds of mobile components can live on the same page and communicate with each other through the browser's API. Our implementations of Oberon and the Juice plug-in modules can be freely downloaded from our Internet site [Oberon, Juice].

5. Summary and Conclusion

We have presented two new technologies that we believe to be vital for future mobile-object systems:

- The *slim binary* machine-independent software distribution format is extremely dense, which will be an important benefit when wireless connectivity becomes pervasive. Already today, its tree structure presents advantages over linear byte-code solutions when advanced optimizations are required.
- The quasi-monolithic run-time model of code execution, made possible by iterative dynamic code optimization, may very well become a necessity as the average unit of distributed code gets smaller and smaller. Only by constructing all-encompassing fully cross-optimized code images at run-time will it be possible to exploit the full power of future superscalar processors for such software.

Hardware is currently developing in favor of these two techniques, as wireless communications are becoming a reality while processor power is increasing rapidly, making dynamic code generation practical. Cheaper processors also mean that eventually a dedicated additional processor could be employed to perform background re-optimization of code ultimately executed by a primary processor. This utilization of extra processors might in fact often be more cost-effective than attempting genuine multiprocessing.

Acknowledgment

The Oberon System has turned out to be a stable foundation for projects far beyond its original scope. The author gratefully acknowledges the original creators of Oberon, Niklaus Wirth and Jürg Gutknecht, and the co-author of the Oberon/Juice software distribution, Thomas Kistler. Thanks also go to Martin Burtscher and Markus Dätwyler for collaborating on the system's implementation.

References

- [AH96] G. Aigner and U. Hölzle; "Eliminating Virtual Function Calls in C++ Programs"; *ECOOP'96 Conference Proceedings*, published as *Springer Lecture Notes in Computer Science*, No. 1098, 142-166; 1996.

- [BCF95] M. Brandis, R. Crelier, M. Franz, and J. Templ; “The Oberon System Family”; *Software–Practice and Experience*, 25:12, 1331-1366; 1995.
- [CMH91] P. P. Chang, S. A. Mahlke, and W. W. Hwu; “Using Profile Information to Assist Classic Code Optimizations”; *Software–Practice and Experience*, 21:12, 1301-1321; 1991.
- [CU89] C. Chambers and D. Ungar; “Customization: Optimizing Compiler Technology for SELF, a Dynamically-Typed Object-Oriented Programming Language”; *Proceedings of the ACM Sigplan ‘89 Conference Programming Language Design and Implementation*, published as *Sigplan Notices*, 24:7, 146-160; 1989.
- [Dav86] J. W. Davidson; “A Retargetable Instruction Reorganizer;” *Proceedings of the ACM Sigplan ‘86 Symposium on Compiler Construction*, Palo Alto, California, 234-241; 1986.
- [DF84] J. W. Davidson and C. W. Fraser; “Code Selection through Object Code Optimization”; *ACM Transactions on Programming Languages and Systems*, 6:4, 505-526; 1984.
- [DRA93] United Kingdom Defence Research Agency; *Frequently Asked Questions about ANDF, Issue 1.1*; June 1993.
- [FK96] M. Franz and T. Kistler; “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; 1996.
- [Juice] M. Franz and T. Kistler; *Juice*; <http://www.ics.uci.edu/~juice>.
- [FL91] M. Franz and S. Ludwig; “Portability Redefined”; in *Proceedings of the Second International Modula-2 Conference*, Loughborough, England; September 1991.
- [Fra93] M. Franz; “Emulating an Operating System on Top of Another”; *Software–Practice and Experience*, 23:6, 677-692; 1993.
- [Fra94a] M. Franz; *Code-Generation On-the-Fly: A Key to Portable Software*; Doctoral Dissertation No. 10497, ETH Zurich, simultaneously published by Verlag der Fachvereine, Zürich, ISBN 3-7281-2115-0; 1994.
- [Fra94b] M. Franz; “Technological Steps toward a Software Component Industry”; in *Programming Languages and System Architectures*, Springer Lecture Notes in Computer Science, No. 782, 259-281; 1994.

- [Han74] G. J. Hansen; *Adaptive Systems for the Dynamic Run-Time Optimization of Programs* (Doctoral Dissertation); Department of Computer Science, Carnegie-Mellon University; 1974.
- [Ing71] D. Ingalls; “The Execution Time Profile as a Programming Tool”; *Design and Optimization of Compilers*, Prentice-Hall; 1971.
- [Juice] M. Franz and T. Kistler; *Juice*; <http://www.ics.uci.edu/~juice>.
- [LYJ96] T. Lindholm, F. Yellin, B. Joy, and K. Walrath; *The Java Virtual Machine Specification*; Addison-Wesley; 1996.
- [Mac93] S. Macrakis; *Protecting Source Code with ANDF*; Open Software Foundation Research Institute; June 1993.
- [NAJ76] K. V. Nori, U. Amman, K. Jensen, H. H. Nägeli and C. Jacobi; “Pascal-P Implementation Notes”; in D.W. Barron, editor; *Pascal: The Language and its Implementation*; Wiley, Chichester; 1981.
- [Oberon] Institut für Computersysteme, ETH Zurich, and Department of Information and Computer Science, University of California at Irvine; *Oberon Software Distribution*; <http://www-cs.inf.ethz.ch/Oberon.html> or <http://www.ics.uci.edu/~oberon>.
- [Wal92] D. W. Wall; “Experience with a Software-Defined Machine Architecture”; *ACM Transactions on Programming Languages and Systems*, 14:3, 299-338; 1992.
- [Wel84] T. A. Welch; “A Technique for High-Performance Data Compression”; *IEEE Computer*, 17:6, 8-19; 1984.
- [WG89] N. Wirth and J. Gutknecht; “The Oberon System”; *Software-Practice and Experience*, 19:9, 857-893; 1989.
- [WG92] N. Wirth and J. Gutknecht; *Project Oberon: The Design of an Operating System and Compiler*; Addison-Wesley; 1992.
- [Wir88] N. Wirth; “The Programming Language Oberon”; *Software-Practice and Experience*, 18:7, 671-690; 1988.