Feature

Dynamic Linking of Software Components

Traditionally, dynamic linkers merely combined previously compiled pieces of code. Faster processors are now making outright code generation at load time practical, leading to cross-platform portability at very little extra cost.

Michael Franz
University of California, Irvine

In recent years operating systems have again begun to link software libraries to client programs dynamically at execution time. The concepts underlying dynamic linking date back at least to the Multics system[1] of the mid-1960s and have now been reintroduced in some of the most popular workstation and PC operating systems. Operating systems based on modular languages such as Mesa,[2] Modula-2, and Oberon[3] have offered similar capabilities for well over a decade. Indeed, the concept of dynamic linking reached maturity through these modular operating systems, which largely laid the groundwork for dynamic linking's resurgence.

Modular systems are written in languages that blur the distinction between libraries and application programs. In these languages, all software consists of modules arranged in a (usually acyclic) dependence hierarchy. Individual modules serve as libraries when they are referenced by clients higher up in the hierarchy and as clients when they reference modules farther down. Consequently, as Figure 1 shows, the hierarchy's inner nodes serve simultaneously as libraries and as clients.

In such a system, the operating system is itself represented as a collection of modules at the bottom of the hierarchy. Operating system and application program modules have no intrinsic difference, and the boundary between them is purely conceptual. Because only those parts of the operating system that the application requires at the moment need be present in memory, dynamic linking lets us build powerful systems that consume relatively little storage.

Already powerful by itself, dynamic linking is even more useful when it is combined with a language that directly supports software system extensibility. An extensible system can evolve without requiring changes in any of its original parts. Combining extensibility with dynamic linking leads to highly flexible systems that use computing resources efficiently because they don't duplicate functionality. Niklaus Wirth recently presented a compelling case for such runtime-extensible software systems.[4]

While the central idea behind dynamic linking is quite straightforward, it can be implemented through a surprising variety of strategies. In this article I contrast three simple dynamic linking schemes with two much more elaborate strategies. The latter represent a workload shift from the compiler to the dynamic linker, just as dynamic linking itself represents a shift that moves the functions of a separate linker into the loader. Because the new techniques I describe promise the profound additional benefit of cross-platform portability, they will most likely displace the currently popular linking-loader approach.

MODULES

A module is a collection of constant, type, variable, and procedure declarations encapsulated behind a rigid interface. A module's designer can control which of its features appear in the interface by *exporting* them selectively. Features that are not exported cannot be accessed from outside the module and are therefore protected from accidental misuse. In this way, a module can guarantee its invariants. For example, a module may export an abstract data type and a set of procedures operating on it while hiding the type's internal structure and the actual procedure implementations.

0018-9162/97/$10.00 © 1997 IEEE

Modules may of course *import* as well. An intermodule relationship is established when a client module (higher in the module hierarchy) imports a feature from the interface of a library module (lower in the hierarchy). In principle, every module may serve simultaneously as a library to higher level clients and as a client of lower level libraries. There are no fundamental restrictions on the number of clients a library can support simultaneously or on the number of libraries a client may access.

Intermodule references established by import/export relationships are resolved in a process separate from compilation called binding, or linking. In modular systems, linking occurs at the time of loading and is therefore invisible to the user. Such dynamic linking usually implies that at most one copy of any library module exists in memory at a given time, although several client modules may use it concurrently. In operating systems that offer only offline static linking, each application program must contain a private copy of every library it uses. (The exception is a kernel library, whose services are accessed via supervisor calls and are hence "linked" during compilation.) Linking is recursive; linking a library module entails repeating the process for all of the modules imported by that module.

The original idea behind dynamic linking was to factor out common functions so that they could be shared among several applications. Extensible systems take this idea a step further, allowing more modules to be added to the system even at the top of the module hierarchy. Applications can thereby be augmented by additional functionality at runtime; the extra modules register their presence with the original application, which can then use them without an explicit import relationship. Instead, they are activated by indirect procedure *up-call* or *callback* mechanisms. Providing extensibility in a type-safe manner requires a language that explicitly supports this capability. Oberon's[3] type extension mechanism[5] exemplifies this kind of support. Type extension enables the definition of new data types with extended functionality that are nevertheless backward-compatible with the original application's data types.

A modular system's constituent modules are kept distinct at all times; they serve as the units of compilation and are compiled separately. *Separate compilation*[6,7] usually implies that the compiler verifies the consistency of every use of an imported item with its declaration in the originating module. This contrasts with the more primitive *independent compilation*, in which type errors occurring across module boundaries can be detected only at link time, if at all.

Since no static linking step is required, modules remain separate until the time of loading. This enables them to be maintained individually, and the corresponding object files can be distributed and reused independently of each other. Thus, library modules can be replaced at any time by upgraded but otherwise equivalent modules providing the same interface, considerably simplifying library management.

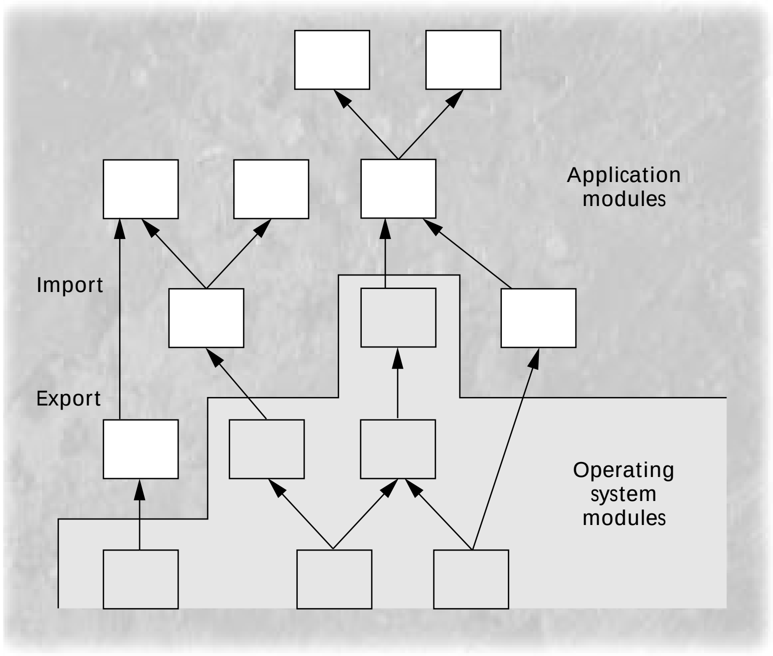


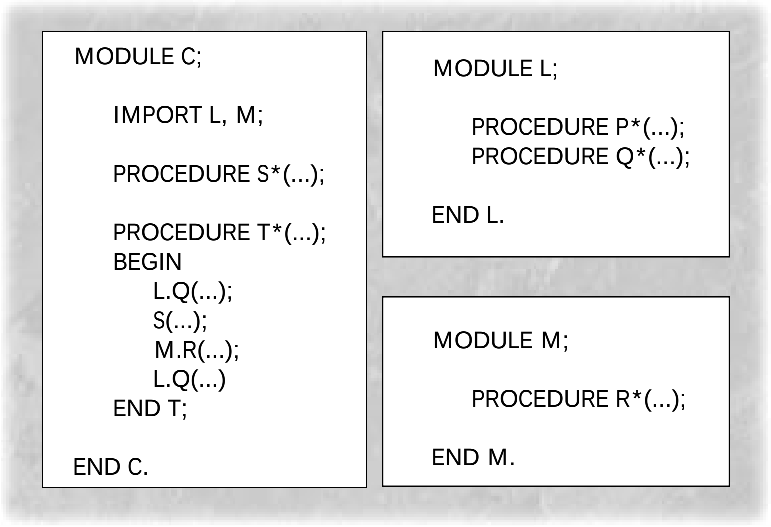
Figure 1. Hierarchical structure of a modular system. Inner nodes can serve as both libraries and clients.

LINKING MECHANISMS

To bind client and library modules, the linker must match every imported client feature with an exported library feature. If this matching fails, the modules cannot be used together and linking must be aborted. Such link-time failures are rare, occurring only if the library's interface was changed after the client was compiled. Otherwise, checks during separate compilation have already established that the client is compatible with all of the libraries it imports.

Hence, rather than verifying every single import/export relationship again, which is complex and time-consuming, the linker must verify only that nothing imported from a library has changed since the client was compiled. The simplest way to achieve this is to include with every object file the time stamps of all imported libraries; the linker can then compare just the time stamps.

Fingerprints provide even greater flexibility than time stamps.[8] A fingerprint is a hash value computed from a library interface; the probability of two interfaces yielding the same fingerprint is very small if an intelligent hash function is chosen. An object file importing an interface with a certain fingerprint can then be assumed to be linkable with any library having the same fingerprint, no matter when and on what machine the library was compiled. Going a step further, every library item can be given its own finger-



```
MODULE C;

    IMPORT L, M;

    PROCEDURE S*(...);

    PROCEDURE T*(...);
    BEGIN
        L.Q(...);
        S(...);
        M.R(...);
        L.Q(...)
    END T;

END C.
```

```
MODULE L;

    PROCEDURE P*(...);
    PROCEDURE Q*(...);

END L.
```

```
MODULE M;

    PROCEDURE R*(...);

END M.
```

Figure 2. Example modules with import/export relationships. Module C imports the two libraries L and M while exporting procedures S and T to other client modules. Module L exports procedures P and Q, and module M exports procedure R.

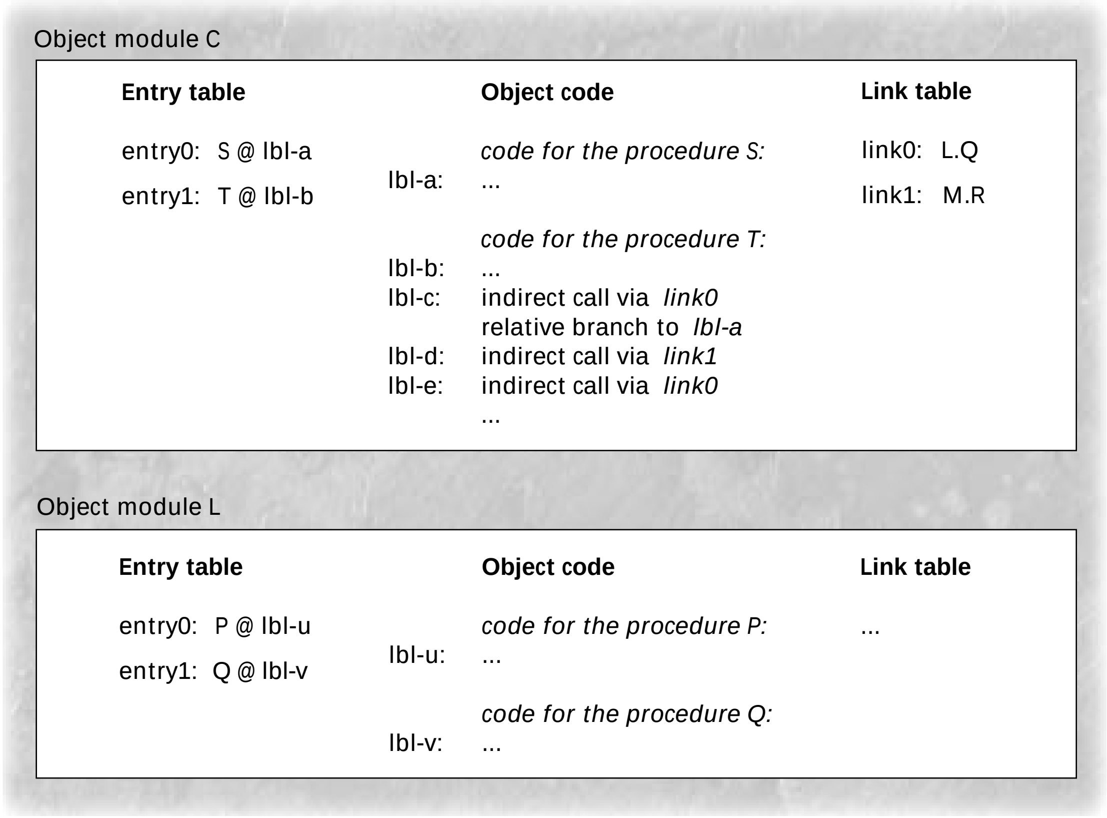
print,[9] thus allowing them to be added, removed, or changed individually without invalidating clients that do not refer directly to the affected items.

In discussing the three example modules in Figure 2, I use Oberon's syntax, in which every export of an identifier is denoted by a trailing asterisk. Our sample module C (as in client) imports the two libraries L and M (actually, it imports their interfaces) and simultaneously exports two procedures S and T for use in other client modules. Module L exports two procedures and module M exports just one.

To enable linking, every object file needs two directories that reveal to the linker the intermodular references within the compiled code. The first of these directories, the entry table, describes where the module's various exported features can be found inside its object code. The second directory, the link table, describes which features from other modules are actually accessed by the module. The corresponding object files for modules C and L might appear as in Figure 3.

Runtime table lookup

To link module C to the two library modules it imports, the linker simply replaces the symbolic references found in C's link table with the actual memory positions of the corresponding procedures. For example, the link-table entry at link0 would be replaced by the runtime position of lbl-v (module L's entry for procedure Q). Since all external calls contained within the object code are indirect and go through the link table, under this linking scheme all



```
Object module C

    Entry table                       Object code                        Link table

    entry0:  S @ lbl-a                code for the procedure S:          link0:  L.Q
                             lbl-a:   ...
    entry1:  T @ lbl-b                                                   link1:  M.R

                                      code for the procedure T:
                             lbl-b:   ...
                             lbl-c:   indirect call via  link0
                                      relative branch to  lbl-a
                             lbl-d:   indirect call via  link1
                             lbl-e:   indirect call via  link0
                                      ...

Object module L

    Entry table                       Object code                        Link table

    entry0:  P @ lbl-u                code for the procedure P:          ...
                             lbl-u:   ...
    entry1:  Q @ lbl-v

                                      code for the procedure Q:
                             lbl-v:   ...
```

Figure 3. Object files with two directories that reveal to the linker the intermodular references within the compiled code.

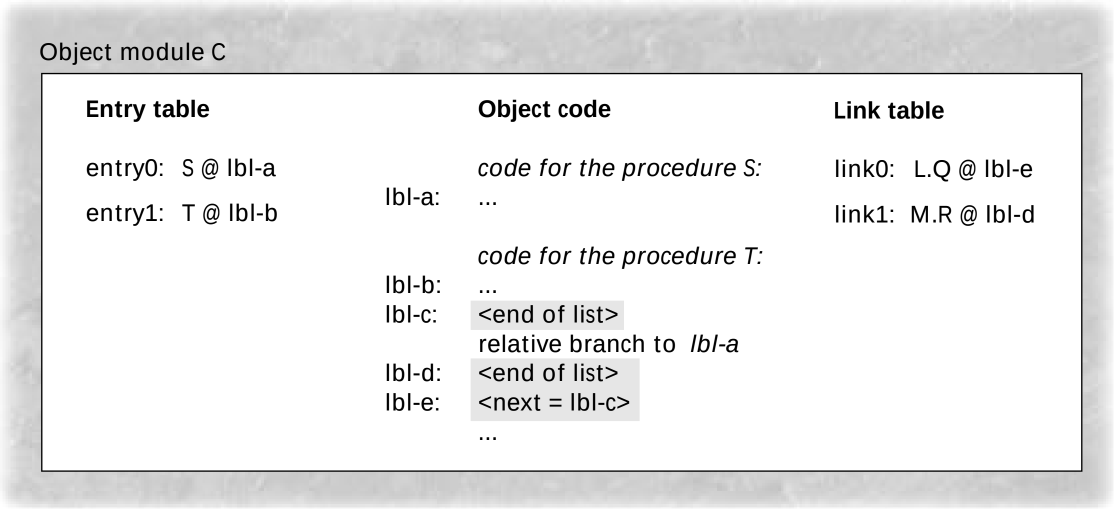


Figure 4. Object file from the previous figure with code fix-up information (shaded area) for a linking loader.

modules remain freely relocatable in memory. If a module were to be moved to another address later, only the link-table entries of its clients and its own entry table would need updating. (If the language provides indirect procedure activation via procedure variables, matters get more complicated.)

Of even greater appeal to the software developer is the ability to change a module's implementation without having to recompile its clients. All of the linking schemes discussed in this article provide this capability. This is especially attractive because changing a procedure's implementation in a module might alter its length and thereby affect the starting addresses of procedures that follow it in the object code. With dynamic linking, this does not affect client modules because intermodule links refer only to entry numbers, while the actual starting addresses are obtained from libraries' entry tables.

Load-time code modification

The simple linking process described above requires an extra memory indirection for every external procedure call. The indirection is used to obtain the final destination address from the link table. Unless specific hardware support is provided, as in the National Semiconductor 32x32 series of microprocessors, programmers usually want to avoid this cost. As a result, many linkers modify the object code at the call site through the strategy of load-time modification, resulting in *direct* calls to external routines**.**

To modify external references embedded within the object code, the linker needs to know where in the code such references occur. This extra information can be added to the link table, increasing its size considerably. Now, instead of an entry for every unique external item, the link table needs a separate entry for every external access. Fortunately, most of this extra information can be conveniently stored within the object code itself, making good use of the code locations that will later be overwritten by the linker.

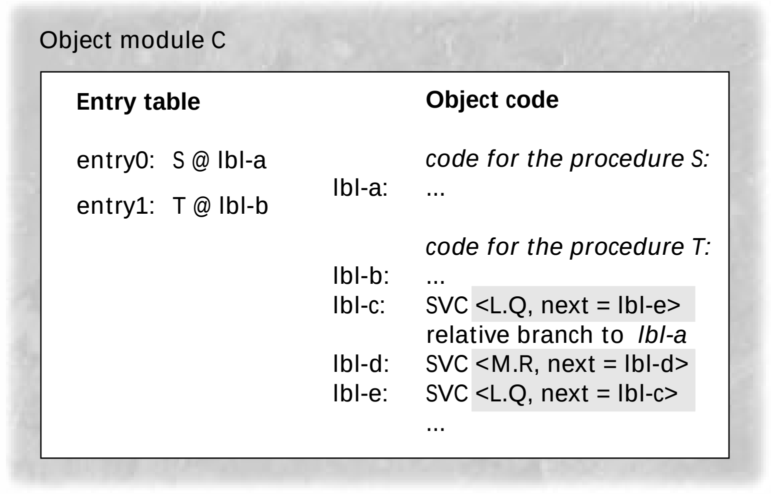
Figure 4 shows the object file from the previous example (Figure 3), adapted for use with a linker that updates the object code directly at load time. All external references within the object code remain unresolved; we use these placeholder locations to join all references to the same external item in a linear list that runs backward through the code. The start of every such list is recorded in the appropriate link table.

Linking then proceeds as follows: After reading a module into memory, the linking loader traverses the elements of the module's link table. For each imported item, the corresponding entry in the originating module is inspected to obtain the target address. Then the linear list of references to this item is traversed, starting at the position mentioned in the link table, and the correct opcode-address combination is inserted at every location in the list.

In our example, the linker obtains the final address `finadr` of the procedure Q from the entry table of module L. The link table states that fix-up for this external address needs to commence at lbl-e. The linker retrieves the location of the next fix-up, lbl-c, from the object code at location lbl-e and then inserts a `call finadr` instruction in its place. This is iterated until a special marker is found in the next-fix-up-location chain, indicating that the end of the list has been reached.

Runtime code modification

A module's external references needn't all be replaced immediately at load time. Replacement may be deferred until the need arises, a strategy called *demand linking*, or *lazy linking*. The idea is simple. In



```
Object module C

Entry table                          Object code

entry0:  S @ lbl-a                   code for the procedure S:
                             lbl-a:  ...
entry1:  T @ lbl-b
                                     code for the procedure T:
                             lbl-b:  ...
                             lbl-c:  SVC <L.Q, next = lbl-e>
                                     relative branch to  lbl-a
                             lbl-d:  SVC <M.R, next = lbl-d>
                             lbl-e:  SVC <L.Q, next = lbl-c>
                                     ...
```

Figure 5. The example object file with code fix-up information (shaded area) for a lazy linker.

the original object file, every code location that forms part of an external reference is replaced by a special supervisor call instruction, followed by information for the linker. At runtime, execution of the special supervisor call passes control to the linker, which determines the actual address of the external reference and then overwrites the <*supervisor instruction, linker information*> sequence in the object code by an actual call (or, in the case of a variable reference, a load or store instruction). The call (or load or store) is then carried out. Upon the next execution, the destination address will be reached directly, without intermittent activation of the linker.

Figure 5 shows our example module again, with appropriate modifications for lazy linking. As in the previous example, all references to the same external item are joined, except that now the list is circular, since we cannot know which of the references will be encountered first during program execution. Linking the different uses of the same item together in this manner lets us save some supervisor-call and table-lookup overhead.

For example, execution of the procedure T proceeds to the supervisor instruction at lbl-c. The linker is called and determines that this is an external call to L.Q and that a further call to the same procedure occurs at lbl-e. It obtains the final address from the entry table of module L and then overwrites the supervisor call instructions at lbl-c and lbl-e by a direct call to `finadr`.

Lazy linking is attractive because it minimizes the work of the linker: The only external references linked are those actually encountered at least once during execution. The actual loading of an imported module can even be delayed until it is referenced for the first time. Unfortunately, however, every linking step requires an explicit synchronization of the processor's instruction cache, which on many processors invalidates not only the overwritten supervisor instruction but also a whole region of cache entries close to it. The constantly rising costs associated with cache misses as processors evolve are diminishing the attraction of lazy linking.

Load-time code generation

Although compiling, linking, and loading have traditionally been performed by independent entities, they are really only different aspects of a single problem. All three activities must be performed, in a fixed sequence, on software written in a higher level language, before the software can be executed. However, no rule dictates that these actions cannot immediately follow each other or cannot be executed by fewer than three separate programs.

Dynamic linkers combine the linking phase and the loading phase into a single integrated step, leading to enhanced flexibility. On the downside, more work is required at the time-critical moment of loading—that is, while an interactive user is waiting. However, hardware has become so powerful that the cost of dynamic linking is no longer significant. In fact, processors are now fast enough that further functionality can be migrated downstream from the compiler toward the point of execution. For example, compilers often consist of two parts, a front end that processes the source program and a back end, or code generator, that creates executable code. If a compiler's back end can be integrated into the loader, it is no longer necessary to perform fix-up operations on "unfinished" code; instead, all final addresses can be inserted directly.

Unfortunately, code generation is not a trivial task. The very existence of compilers often seems justified only because most programs are compiled far less often than they are executed. Hence, compilation effort is worthwhile because of the benefit that comes with repeated execution.

Until quite recently, it was universally accepted that the effort required for compilation is so great that it must be performed offline. An experimental system providing load-time on-the-fly code generation[10,11] has invalidated this assumption. This project's success is founded on the insight that raw processor power is growing much more rapidly than the speed of I/O operations, as Figure 6 illustrates. The experimental system makes load-time code generation practical by decreasing the amount of data that must be transferred from external storage. The use of a highly compact intermediate program representation ("slim binaries") instead of native object files speeds up the I/O component of loading dramatically. The time saved is then spent on code generation.

Any application that reduces its I/O overhead at the expense of additional computation can benefit from the effect illustrated in Figure 6, even if the immediate

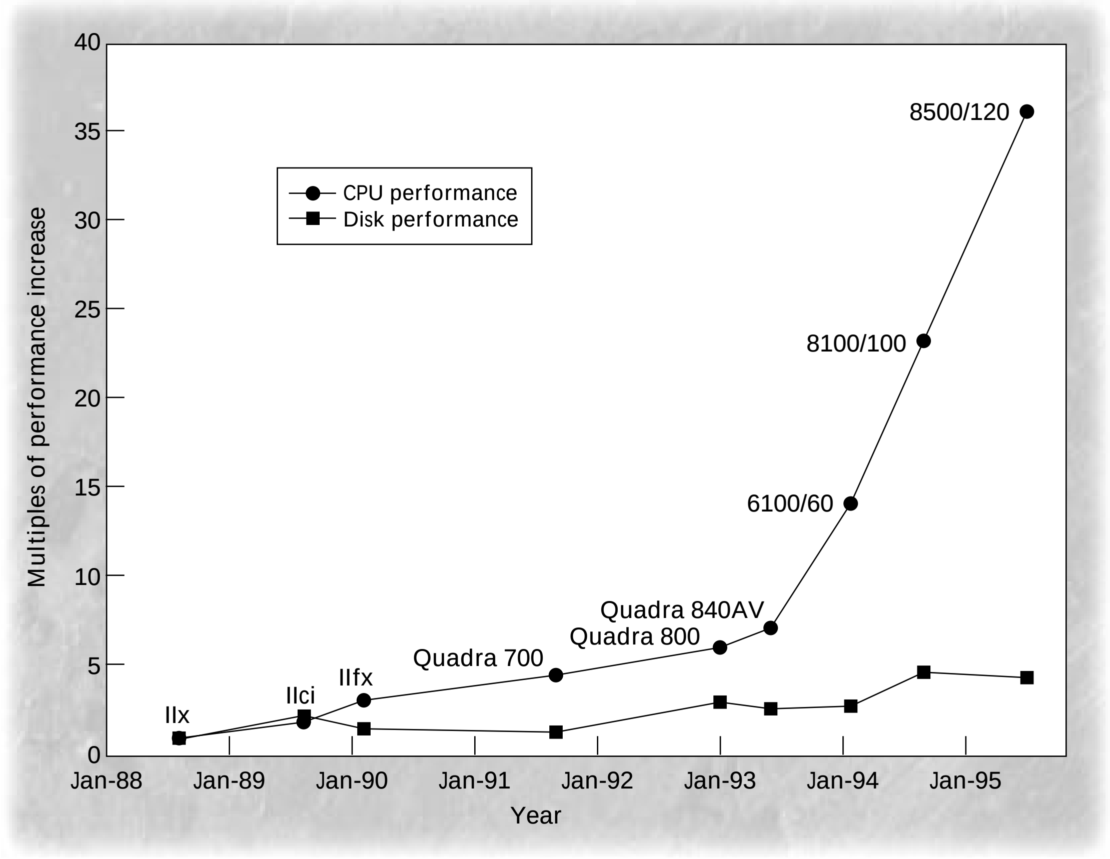


Figure 6. Processor power is growing much faster than I/O speed. This example is for various models of the Apple Macintosh as measured by the tool Speedometer 4.02.

performance gain doesn't seem to reward increased algorithm complexity. Code generation is a particularly good example because processor instruction sets are not optimized for information density, since they have other constraints such as regularity and ease of decoding. Hence, object files are usually much larger than necessary.

The slim-binary representation[10,11] is typically two and a half to three times more compact than object code for common microprocessors and 30 percent more compact than object code after the application of popular data-compression schemes. On modern hardware, the resulting speedup of file input is sufficient to offset much of the additional effort of on-the-fly code generation, bringing load-time code generation within the performance range of ordinary dynamic loading.

For example, Thomas Kistler and I recently reported[11] on a comprehensive package of networking tools comprising a Web browser, a Telnet application with VT100 emulation, POP/SMPT-Mail, News, Finger, FTP, and Gopher applications. This package, compiled into PowerPC binary code, is 603 Kbytes; the slim-binary version of the identical program suite requires only 191 Kbytes. Loading the complete program suite from PowerPC binaries onto a Macintosh 8100/100 requires about 1.1 seconds, while about 2.1 seconds are needed to load from slim binaries—including code generation time. This extra second that an interactive user must wait before these seven applications start may still seem substantial. However, users normally don't start all of these applications at the same time; furthermore, the additional cost of dynamic code generation promises to come down as the gap between processor power and storage speed widens.

Slim binaries have an advantage besides taking up less space: They are target-machine independent. The implementation Kistler and I described[11] currently supports the i80x86, MC680x0, and PowerPC architectures, using the identical object-file format. Once compiled into the slim-binary representation, modules can be used transparently on any of these three architectures as if they had been compiled into the appropriate native code.

Target-machine independence simplifies library module maintenance and thereby lowers software production cost. Further, load-time code generation enables the executable code to be truly optimized for the specific processor and operating system it will run on, not just for an architecture in general. This is a welcome and timely innovation: The appeal of traditional binary compatibility is waning because each implementation of a processor architecture requires

Table 1. Comparison of dynamic linking mechanisms modular systems.

Strategy	Load-time overhead	Runtime overhead
Runtime table lookup	Constant	O(No. of encountered references)
Load-time code modification	O*(No. of external references)	None
Runtime code modification	None	Eventually none
Load-time code generation	O(program length)	None
Full load-time compilation	O(source length)	None

***O represents order of magnitude**

a distinct instruction schedule to realize its full performance potential.

It is also easier to generate good object code for modular programs when code is generated at load time. A code-generating linker can perform intermodule optimizations such as register and cache coordination and procedure inlining, effectively constructing a large monolithic application in memory but without the usual drawbacks of monolithic programs. Furthermore, since the object code can be re-created from the intermediate representation at any time, the code of all loaded modules can be constantly reoptimized during idle cycles, guided by runtime profiling data.

Full load-time compilation

The line of thought that started with a simple table-based linking mechanism and then led us through a series of successively more complex linking schemes logically culminates in the idea of full compilation from source code at load time. A practical implementation would of course be based on some form of compressed source code in which keywords and identifiers have been replaced by tokens to reduce I/O overhead.

Unfortunately, full load-time compilation has several drawbacks that make it less attractive than load-time code generation. First and foremost, most software developers would balk at exposing their intellectual property. In principle, any algorithm comprehensible to a machine can be reverse-engineered, but the idea that compilation is a one-way transformation (along with the legal protection of copyright law) seems to make programmers feel more secure. Any method of program distribution based on source code doesn't stand a chance in today's marketplace.

Moreover, full dynamic compilation also faces technical obstacles. First of all, code generation accounts for only part of the compilation cost. Similar effort is spent in the source text scanner and parser and in the symbol table handler. Although tokenization would eliminate the need for a scanner, the added expense of a full programming-language parser seems uncalled for when an easily decodable format can be chosen to drive a load-time code generator. For example, the slim-binary format Kistler and I used has been designed specifically to serve as an input to code generation and permits interspersing of decoding and code generation.

Last but not least, the use of source text as an input to the loader might lead to compilation errors at load time, thereby exasperating users. The only way to avoid such errors would be to check every program at tokenization time. Effectively, every program would then be parsed twice, once during tokenization and a second time during loading.

COMPARISON AND OUTLOOK

Table 1 summarizes the characteristic costs of all five dynamic linking schemes. The two promising "heavyweight" strategies of load-time code generation and full load-time compilation have not yet been studied extensively, since the computing power that makes them feasible is only just becoming available. Most current systems that support dynamic linking use a linking loader that modifies the object code at load time, prior to execution. The two other "lightweight" approaches discussed here present problems: Using indirect calls with runtime table lookup entails runtime overhead, and runtime code modification leads to frequent invalidation of the instruction cache.

Lazy linking (runtime code modification) has the most interesting characteristics. It requires no load-time overhead and after an initial start-up period generates no runtime overhead. Its main drawback is that the processor's instruction cache must be flushed when code has been modified. When a program's control flow reaches a previously untouched region, a formidable amount of link activity, associated with numerous instruction-cache reloads, may suddenly occur, causing unexpected and disruptive delays for interactive users.

Interestingly, a related scheme has recently been proposed to accelerate execution of the Java Virtual Machine.[12] *Just-in-time compilation* effectively combines lazy linking with on-the-fly code generation on a procedure-by-procedure basis. Hence, when a reference to a previously unseen procedure is encountered, the Java just-in-time compiler initiates a compilation of the called procedure and modifies the calling site to point to the compiled version directly.

In contrast, load-time code generators like the one Kistler and I implemented translate complete modules into native code at once, and at a time when users are already expecting delays. Once a module has been loaded, its code executes with the full speed of compiled code and incurs no further overhead. Also, if code optimization is anticipated, modules are better suited than procedures as the unit of code generation.

Load-time code generation is approaching the speed of traditional loading not because conventional compilers and offline linkers have accelerated very much but because I/O overhead is becoming the main factor influencing loading speed. Hence, smaller object formats such as slim binaries lead to faster loading times, regardless of the computational effort required to decode them. Furthermore, unlike a code-generating loader, traditional compilers and linkers must generate output files, which consumes a lot of time. They must also consult interface files describing imported libraries. In contrast, in a dynamic code-generation system, every portable object file needs to be read only once. Loading is recursive, so that library modules are always loaded before their clients. Interface information about libraries can then be retained in memory and used in the compilation of clients, so that no additional file accesses are required.

Modular systems are staging a comeback in the form of extensible architectures based on the compound-document metaphor. A *compound document* is a container that seamlessly integrates various forms of user data, such as text, graphics, and multimedia. This varied content is supported by independent, dynamically loadable content editors (applets) that cooperate in such a way that the end user sees them as a single unified application. Load-time code generation lets systems run these applets at full speed without unexpected interruptions and without the performance penalties of interpreted execution.

As general-purpose operating systems move forward to embrace dynamic linking and compound-document architectures, the technologies chosen for linking components will play a pivotal role in the systems' long-term commercial success. ❖

Acknowledgments

My work on load-time code generation is an extension of earlier research begun at the Institut für Computersysteme, ETH Zurich, Switzerland. Thanks are due to Niklaus Wirth for being a competent advisor to the earlier work and an enlightened mentor ever since.

References

1. F.J. Corbato and V.A. Vyssotsky, "Introduction and Overview of the Multics System," *Proc. AFIPS Fall Joint Computer Conf.*, 1965, pp. 185-196.
2. J.G. Mitchell, W. Maybury, and R. Sweet, *Mesa Language Manual*, Version 5.0, Tech. Report No. CSL-79-3, Xerox Corp., Palo Alto Research Center, Palo Alto, Calif., 1979.
3. N. Wirth, "The Programming Language Oberon," *Software-Practice and Experience*, No. 7, 1988, pp. 671-690.
4. N. Wirth, "A Plea for Lean Software," *Computer*, Feb. 1995, pp. 64-68.
5. N. Wirth, "Type Extensions," *ACM Trans. Programming Languages and Systems*, No. 2, 1988, pp. 204-214.
6. D.G. Foster, "Separate Compilation in a Modula-2 Compiler," *Software-Practice and Experience*, No. 2, 1986, pp. 101-106.
7. J. Gutknecht, "Separate Compilation in Modula-2: An Approach to Efficient Symbol Files," *IEEE Software*, Nov. 1986, pp. 29-38.
8. M. Franz, "The Case for Universal Symbol Files," *Structured Programming*, No. 3, 1993, pp. 136-147.
9. R. Crelier, "Extending Module Interfaces without Invalidating Clients," *Structured Programming*, No. 1, 1996, pp. 49-62.
10. M. Franz, *Code-Generation On-the-Fly: A Key to Portable Software*, doctoral dissertation, ETH Zurich, Verlag der Fachvereine, Zurich, 1994.
11. M. Franz and T. Kistler, *Slim Binaries*, Tech. Report No. 96-24, Dept. of Information and Computer Science, Univ. of California, Irvine, 1996.
12. T. Lindholm et al., *The Java Virtual Machine Specification*, Addison-Wesley, Reading, Mass., 1996.

Michael Franz is an assistant professor in the Department of Information and Computer Science at the University of California, Irvine. His research interests are programming languages and their implementation; extensible, component-based software systems; machine-independent program representations, portable software that migrates across computer networks, and on-the-fly native-code generation; and iterative profile-guided program optimization at runtime. He received a Dr. sc. techn. degree in computer science and a Dipl. Informatik-Ing. degree, both from ETH Zurich.

Franz can be reached at the University of California, Irvine, Dept. of Information and Computer Science, Irvine, CA 92697-3425; franz@uci.edu; http://www.ics.uci.edu/~franz.