Dynamic Linking of Software Components

Michael Franz

Technical Report 96-26
Department of Information and Computer Science
University of California, Irvine, CA 92717-3425

27th June 1996

Dynamic Linking of Software Components

Michael Franz
Department of Information and Computer Science
University of California at Irvine

Abstract

This paper examines different strategies for linking separately-compiled software modules together at load time. Two of the strategies, load-time code generation and load-time compilation have only recently become practical on account of faster processors. It is likely that one of these new techniques will displace the currently popular linking-loader approach, as it promises the profound additional benefit of cross-platform portability. As general-purpose operating systems move forward to embrace dynamic linking and compound-document architectures, the technologies they choose for linking components will play a pivotal role for their long-term commercial success.

Keywords: dynamic linking, modular systems, on-the-fly code generation, software portability, compound-document architectures

1 Introduction

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 [CV65] and have gained new life with their re-introduction into some of the most popular workstation and personal computer operating systems. Operating systems based on modular languages, such as *Mesa* [MMS79], *Modula-2*, and *Oberon* [Wir88a], have offered similar capabilities for well over a decade. It is in the context of these latter, modular operating systems that the concept of dynamic linking has reached maturity. In fact, the renewed general popularity of dynamic linking should probably be attributed to the groundwork laid by these modular systems.

Modular systems are written in programming languages that weaken the distinction between libraries and application programs. In these languages, all software is composed of *modules* that are arranged into a (usually acyclic) dependence hierarchy. The individual modules in this hierarchy serve as libraries when they are referenced by clients "higher up" in the module dependence graph, and as clients when they, in turn, reference other modules "further down". Consequently, the inner nodes of this hierarchy serve simultaneously as libraries and as clients (Figure 1).

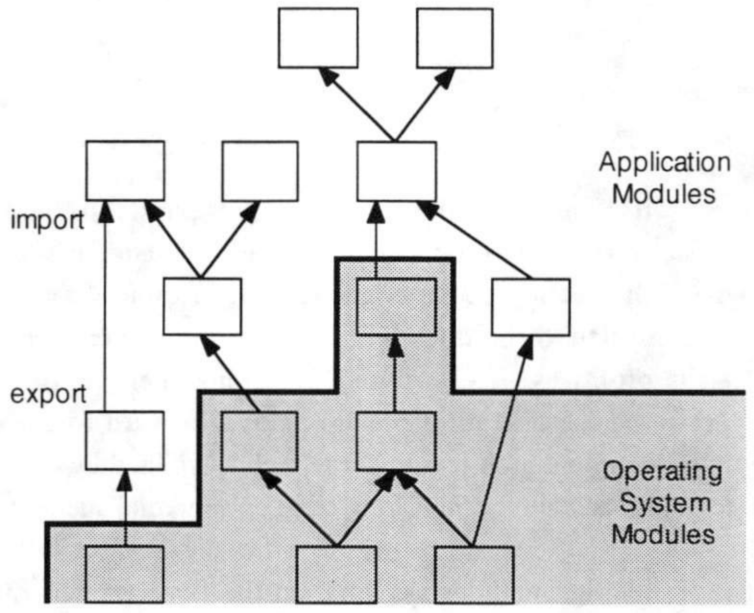


Figure 1: Hierarchical Structure of a Modular System

In such a modular system, the operating system is itself represented as a collection of modules at the "bottom" of a hierarchy that is extended "upwards" by application programs. There is no intrinsic difference between the modules of the operating system and those that are part of application programs; the boundary between the two is purely conceptual. The availability of dynamic linking makes it possible to build powerful systems consuming relatively little storage, as only those parts of the operating system need to be present in memory at any specific moment that are just then required by application programs.

Already a powerful tool by itself, the technique of dynamic linking becomes even more useful when it is combined with a language such as Oberon [Wir88a] that directly supports *extensibility* of software systems. An extensible system is one that can evolve at some later time without requiring changes in any of the original parts. Combining extensibility with dynamic linking leads to highly flexible systems that often excel in their economic use of computing resources because they avoid duplication of functionality. In a recent article, Wirth [Wir95] compellingly presents the case in favor of such run-time-extensible software systems.

This paper explores different strategies for implementing dynamic linking in modular systems. While the central idea behind dynamic linking is in itself quite straightforward, there is a surprising variety of implementation options. In the following, we contrast simpler linking schemes, such as adding an indirection to every external call, with much more elaborate strategies, such as load-time code generation. The latter represents a shift of workload from the compiler to the dynamic linker, just as dynamic linking in itself represents such a shift that moves the functions of a separate linker into the loader. On account of faster hardware, it is now becoming practicable to perform compiler-related tasks at load-time, and this provides numerous benefits.

2 Modules

A module is a collection of constant-, type-, variable-, and procedure declarations that are encapsulated behind a rigid interface. The designer of a module can control which of its features appear in the interface, by *exporting* them selectively. Features that are not exported cannot be accessed from the outside of 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.

The opposite of the *export* operation is appropriately called *import*. An intermodule relationship is established when a client module (at a higher level in the module hierarchy) imports a feature from the interface of a library module (at a lower level in the module hierarchy). In principle, every module may serve a dual function, acting as a library to higher-level clients while being a client of simpler libraries at the same time. There are no fundamental restrictions to the number of clients that a library can support simultaneously, or the number of libraries that a client may access.

The intermodule references established by import/export relationships are resolved in a process separate from compilation, called *binding* or *linking*. In modular systems, linking takes place at the time of loading and is therefore invisible to the user (*dynamic linking*). This usually also implies that at most one copy of any library module exists in memory at any one time, although several client modules may be using it concurrently. In operating systems that offer static linking only, each application program has to contain a private copy of every library it uses, with the exception of a special *system* or *kernel* library, the services of which are accessed via supervisor calls and hence are "linked" already during compilation. Linking is a recursive process; linking of a library module entails repetition of the process for all of the modules imported by it.

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 even *at the top* of the module hierarchy. Application programs can thereby be augmented by additional functionality at run-time; the extra modules register their presence to the original application and are then usable by it without an explicit import relationship. Instead, they are activated by indirect procedure "up-call" or "call-back" mechanisms. To provide this capability in a type-safe manner, a language with explicit support for extensibility is required. An example for this kind of support is the *type extension* mechanism [Wir88b] of the programming language *Oberon* [Wir88a]. Type extension enables the definition of new data types with extended functionality that are still backward-compatible with the data types of the original application.

Modular systems keep their constituent modules distinct at all times. First, modules serve as the units of compilation and are compiled separately from each other. The term "separate compilation" [Fos86, Gut86] usually implies that the compiler verifies the consistency of every use of an imported item with its declaration in the originating module. This is in contrast to "independent compilation", in which type errors that occur across module boundaries can be detected only at link time, if at all.

But even after compilation, the modules out of which modular systems are composed remain separate until they are dynamically linked together during the loading process. As a consequence of being kept apart, the various modules can be maintained individually and the corresponding object files distributed and re-used independently of each other. Since this enables library modules to be replaced at any time by equivalent ones providing the same interface, the management of libraries is simplified considerably.

3 The Linking Process

To bind two modules (hereafter identified as *library* and *client*) together, the linker needs to match every imported feature of the client with an exported feature of the library. If this matching process fails, then the two modules cannot be used together and the linking process must be aborted. However, such link-time failures are rare and can happen only if the library's interface was changed after the compilation of the client. Otherwise, the checks that occurred during separate compilation have already established that the client is compatible with all of the libraries it imports.

Hence, rather than again verifying every single import-export relationship, which is complex and time-consuming, the linker really only needs to verify that nothing imported from a library has changed since the compilation of the client. The simplest way of achieving this is by including with every object-file the time-stamps of all imported libraries, and having the linker compare these time-stamps.

More flexibility is gained by replacing time-stamps with *fingerprints* [Fra93]. A fingerprint is a hash value computed from a library interface, the idea being that the probability of two different interfaces yielding the same fingerprint is very small. An object file importing an interface with a certain fingerprint can then be assumed to be linkable with any library that has the same fingerprint, no matter when and on what machine the library was compiled. The idea of fingerprinting can be taken even further by providing every library *item* with its own individual fingerprint [Cre95]. Library items can then be added, removed, or changed individually, without invalidating clients that do not refer to the affected items directly.

The following discussion is based on the three example modules shown in Figure 2. We use the syntax of the programming language Oberon [Wir88a], 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*, and simultaneously exports two procedures *S* and *T* for use in further client modules. One of the libraries exports two procedures, and the other exports just one procedure.

```
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

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

Object Module C

```
Entries                          Object Code                       Links

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

```
Entries                          Object Code                       Links

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 Link Information

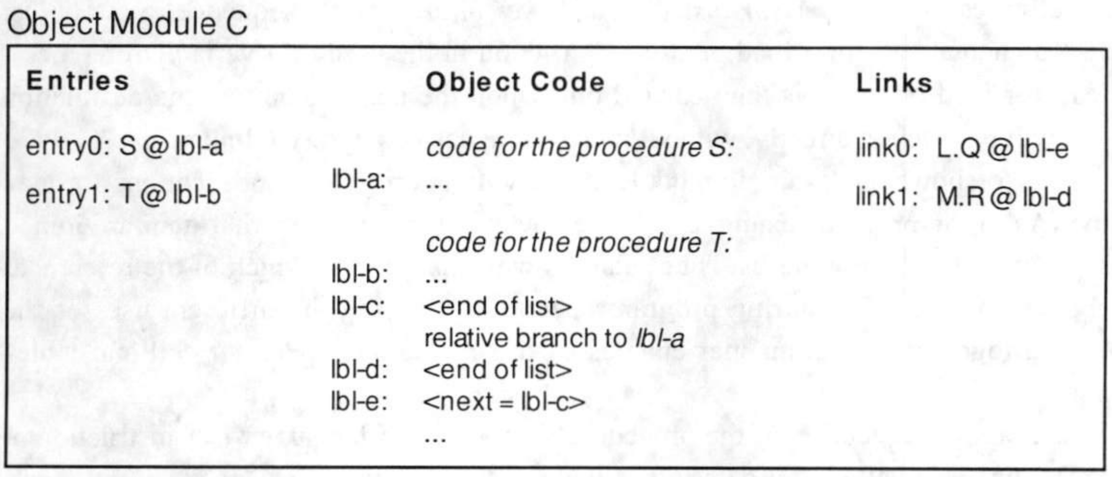
To link the module *C* from our example to the two library modules it imports, the linker merely needs to replace the symbolic references found in the link table of *C* by the actual positions in memory at which the corresponding procedures are located. For example, the link-table entry at *link0* would be replaced by the run-time position of *lbl-v*. Since all external calls contained within the object code are indirect ones that go through the link table, all modules remain freely relocatable in memory under this linking scheme. 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 to be updated. (If the programming 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 necessitating a recompilation of its clients. All of the linking schemes discussed in this paper provide this capability. Changing the implementation of a certain procedure 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 has no effect on client modules as they refer only to entry numbers, while the actual starting addresses are obtained from the entry tables of libraries.

3.1 Load-Time Code Modification

The simple linking process described above requires an extra memory indirection for every external procedure call 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 consequence, many linkers modify the object code at the call site, 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. Instead of an entry for every unique external *item* as before, the link table now 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.



```
Object Module C
Entries                          Object Code                        Links

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

                                 code for the procedure T:
                          lbl-b: ...
                          lbl-c: <end of list>
                                 relative branch to lbl-a
                          lbl-d: <end of list>
                          lbl-e: <next = lbl-c>
                                 ...
```

Figure 4: Object File with Code-Fix-Up Information for a Linking Loader

Figure 4 shows the object file from the previous example, adapted for use with a linker that updates the object code directly. All external references within the object code remain unresolved; we use these "place-holder" locations to join all references to the same external item in a linear list that runs backwards through the code. The start of every such list is recorded in the link table.

Linking then proceeds as follows: After reading a module into memory, the linking loader traverses the elements of its 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.

3.2 Run-Time Code Modification

Instead of replacing all external references of a module immediately at the time of loading, one may also defer this action until the need arises (*demand linking* or *lazy linking*). The idea is simple: in the original object file, every code-location that forms a part of an external reference is replaced by a special supervisor call instruction, followed by information for the linker. At run-time, 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 a load or store instruction in the case of a variable reference). 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 together, except that now the list is circular, as 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 enables us to 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 *finadr* 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*.

Object Module C

```
Entries                              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: Object File with Code-Fix-Up Information for a Lazy Linker

Lazy linking seems attractive, as it minimizes the work of the linker: only those external references are linked that are actually encountered at least once during execution. One can even delay the physical loading of an imported module 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 a whole region of cache entries close to it. Since the costs associated with cache-misses are constantly rising as processors evolve, this is diminishing the attraction of lazy linking.

3.3 Load-Time Code Generation

Although the actions of compiling, linking, and loading have traditionally been performed by independent entities, they are really only different aspects of a single problem. All three activities need to be performed, in a fixed sequence, on source code, before it can be executed on a computer. There is, however, no rule that prescribes that these actions cannot immediately follow each other, or cannot be executed by fewer than three separate programs.

Dynamic linkers combine the linking and loading phases into a single integrated step, leading to enhanced flexibility. On the downside, more work is required at a time-critical moment, i.e., 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 towards 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 the back-end of a compiler can be integrated into the loader, there is no longer a need to perform fix-up operations on "unfinished" code, but all final addresses can be inserted directly.

Unfortunately, it is a well-known fact that code generation is not a trivial task. The very existence of compilers often seems justified only because most programs are compiled far less frequently than they are executed. Hence, it is worthwhile to invest effort into compilation, because the benefit will be repeated. Until quite recently, it has been universally accepted that the effort required for compilation is so great that it needs to be performed off-line.

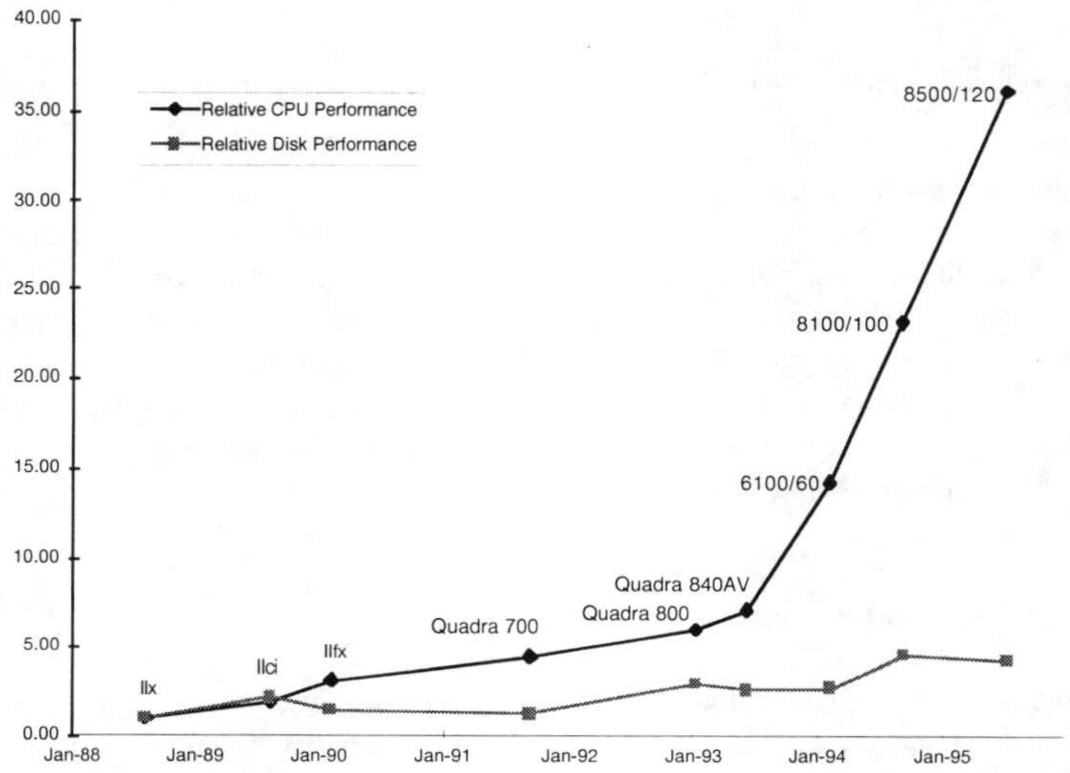


Figure 6: Different Growth Rates of Processor Power vs. I/O Speed for Different Models of the Apple Macintosh Computer Family (as Measured by the Tool Speedometer 4.02)

An experimental system providing load-time "on-the-fly" code generation [Fra94, FK96] has demonstrated that this assumption no longer holds. The success of this project is founded on the insight that raw processor power is growing much more rapidly than the speed of input and output operations (Figure 6). The experimental system makes load-time code generation practicable by decreasing the amount of data that needs to 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 computer application that reduces its input/output overhead at the expense of additional computations can benefit from the effect illustrated in Figure 6, even if the immediate performance gain doesn't seem to reward an increased algorithm complexity. Code generation is 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.

The slim binary representation described in [Fra94, FK96] is typically 2.5 - 3 times more compact than object code for common microprocessors. Even when compared to popular data-compression schemes applied to object code, density is over 30% higher. On modern hardware, the resulting speed-up of file input is sufficient to counterbalance 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, Franz and Kistler [FK96] report of a comprehensive package of networking tools, comprising a *WWW browser*, a *Telnet* application with VT100 emulation, *POP/SMPT-Mail*, *News*, *Finger*, *FTP*, and *Gopher* applications. The size of this package compiled into PowerPC binary code is 603 Kilobytes, while the slim binary version of the identical program suite requires only 191 Kilobytes. Loading the complete program suite from PowerPC binaries on a *Macintosh 8100/100* requires about 1.1 seconds, while it requires about 2.1 seconds to load from slim binaries - including the time required for code generation.

This extra second that an interactive user has to wait before his applications (all 7 of them) start may still seem substantial, although the additional cost of dynamic code-generation promises to come down as the gap between processor power and storage speed widens. But slim binaries have an additional advantage besides taking up less space: they are *target-machine independent*. The implementation described in [FK96] 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 is an important advantage. It simplifies the maintenance of module libraries and thereby lowers the cost of software production. Further, load-time code generation enables the executable code to be truly optimized for the *specific* processor and operating system on which it will run, and not just for an architecture in general. This is a welcome innovation at a time in which the appeal of traditional binary compatibility is waning as each implementation of a processor architecture requires a distinct instruction schedule to realize its full performance potential.

It is also easier to generate good object code for modular programs when this is done 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, it is possible to constantly re-optimize the code of all loaded modules during idle cycles, guided by run-time profiling data.

3.4 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 had been replaced by tokens, to reduce input/output overhead.

Unfortunately, however, there are several drawbacks associated with full load-time compilation, making it a less attractive approach than load-time code generation. First and foremost, most software developers would frown at the idea of baring their intellectual property so openly. While it is true that in principle any algorithm comprehensible to some machine can be reverse-engineered, the idea that compilation is a one-way transformation (along with the legal protection afforded by copyright law) seems to provide an important feeling of security to most programmers. Any method of program distribution that is openly based on source code does not stand a chance in today's marketplace.

But there are also technical arguments that make full dynamic compilation appear less practical. First of all, code generation accounts for only part of the cost of compilation. Similar effort is spent in the source text scanner and parser, and in the symbol table handler. While tokenization would eradicate the need for a scanner, the added cost of a full programming language parser seems inappropriate when on the other hand an easily decodable format can be chosen to drive a load-time code generator. For example, the slim binary format used in [FK96] has been designed specifically to be well-suited 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 the occurrence of compilation errors at load time, placing users in truly awkward situations. The only way to avoid this 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.

4 Comparison and Outlook

Most current systems that support dynamic linking employ a linking loader that *modifies the object code at load time* prior to execution. Among the two other "lightweight" approaches that we have presented, using *indirect calls* is disliked because it has a run-time overhead associated with it, and *run-time code modification* is unpopular because it leads to frequent invalidation of the instruction cache. The 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. Figure 7 summarizes the characteristic costs of all five schemes.

Strategy	*Load-Time Overhead*	*Run-Time Overhead*
Indirect Calls	constant	O(#encountered references)
Load-Time Code Modification	O(#external references)	none
Run-Time Code Modification	none	eventually none
Load-Time Code-Generation	O(program length)	none
Full Load-Time Compilation	O(source length)	none

Figure 7: Comparison of Dynamic Linking Mechanisms for Modular Systems

Of these schemes, run-time code modification ("lazy linking") has the most interesting characteristics. It requires no load-time overhead, and after an initial startup period generates no run-time overhead. As mentioned before, the main problem of this technique is that the processor's instruction cache needs to be flushed when code has been modified. When the control flow in a program reaches a previously untouched region, there may suddenly occur a formidable amount of link activity, associated with numerous re-loads of the instruction cache, causing disruptive delays for interactive users at unexpected points in time.

Interestingly, a related scheme has recently been proposed to accelerate the execution of the Java Virtual Machine [LYJ96]. Named *just-in-time compilation*, it effectively combines lazy linking with on-the-fly code generation on a procedure-by-procedure basis. Encountering a reference to a previously unseen procedure under this scheme not only modifies the calling site to point to the callee directly, but also initiates a compilation of the called procedure.

In contrast, load-time code generators such as the one implemented by Franz and Kistler [FK96] translate complete modules into native code at once, and at points in time at which 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. Modules are also better suited than procedures as the unit of code generation if code optimization is anticipated.

Load-time code generation is approaching the speed of traditional loading. Although this may at first seem surprising, since conventional compilers and off-line linkers have not accelerated by so much, the reasons are again rooted in a much lower input/output overhead. Unlike a code-generating loader, traditional compilers and linkers need to generate output files, which consumes a lot of time. They also need to consult interface files describing imported libraries. In a dynamic code-generation system, on the other hand, 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 currently staging a come-back, 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. These various kinds of content are supported by independent, dynamically-loadable content editors ("applets") that cooperate in such a way that they appear to the end-user as a single unified application. Load-time code generation allows us to make these applets portable without the performance penalties of interpreted execution, and to run them at full speed without unexpected interruptions.

Acknowledgement

The author's work on load-time code generation continues earlier research begun at Institut für Computersysteme, ETH Zürich, Switzerland. Thanks are due to Niklaus Wirth, for being a competent advisor to the earlier work, and an enlightened mentor ever since.

References

[Cre96] R. Crelier; "Extending Module Interfaces without Invalidating Clients"; *Structured Programming*, 16:1, 49-62; 1996.

[CV65] F. J. Corbato and V. A. Vyssotsky; Introduction and Overview of the Multics System; in *Proceedings of the AFIPS Fall Joint Computer Conference*, 185-196; 1965.

[Fos86] D. G. Foster; "Separate Compilation in a Modula-2 Compiler"; *Software-Practice and Experience*, 16:2, 101-106; 1986.

[Fra93] M. Franz; "The Case for Universal Symbol Files"; *Structured Programming*, 14:3, 136-147; 1993.

[Fra94] M. Franz; *Code-Generation On-the-Fly: A Key to Portable Software* (Doctoral Dissertation, ETH Zürich); Verlag der Fachvereine, Zürich; 1994.

[FK96] M. Franz and T. Kistler; *Slim Binaries*; Technical Report No. 96-24, Department of Information and Computer Science, University of California, Irvine; 1996.

[Gut86] J. Gutknecht; "Separate Compilation in Modula-2: An Approach to Efficient Symbol Files"; *IEEE Software*, 3:6, 29-38; 1986.

[LYJ96] T. Lindholm, F. Yellin, B. Joy, and K. Walrath; *The Java Virtual Machine Specification*; Addison-Wesley; 1996.

[MMS79] J. G. Mitchell, W. Maybury, and R. Sweet; *Mesa Language Manual, Version 5.0*; Report No. CSL-79-3, Xerox Corporation, Palo Alto Research Center, Systems Development Department, Palo Alto, California; 1979.

[Wir88a] N. Wirth; "The Programming Language Oberon"; *Software-Practice and Experience*, 18:7, 671-690; 1988.

[Wir88b] N. Wirth; "Type Extensions"; *ACM Transactions on Programming Languages and Systems*, 10:2, 204-214; 1988.

[Wir95] N. Wirth; "A Plea for Lean Software"; *IEEE Computer*, 28:2, 64-68; 1995.