A Compiler-Supported Unification of Static and Dynamic Loading

Felix Friedrich and Florian Negele
Computer Systems Institute, ETH Zürich, Switzerland
{felix.friedrich,negelef}@inf.ethz.ch

Abstract

In order to provide certain dynamic inference methods such as type tests, garbage collection or method dispatch, metadata for the runtime system of a programming language have to be made available. Such data structures are usually represented using a specific format in object files and are generated during load time. On the way to a particularly simple to understand object file format we found an approach that renders the separation of data and metadata unnecessary. This permits a unification and simplification of static and dynamic loading and makes it possible to concentrate modifications of a system to compiler and runtime. It thus increases the maintainability of a system substantially.

1 Introduction

Nearly all modern programming languages are at runtime supported by a runtime system that provides metadata necessary for the provision of dynamic inference methods, such as method dispatch, type tests, garbage collection, exception handling, module loading and unloading etc. Such data structures are usually represented using a specific format in object files and are generated when an object file is loaded.

Additions and modifications of runtime features of our programming language led to a modification of the relevant runtime data structures. Therefore the object file format became increasingly complex over time. Also the involved tools such as the compiler, loader, linker and decoder had to be adapted accordingly and everything became hard to maintain and understand. We had the idea to exploit the co-design of language, compiler and runtime system to define a new object file format that is particularly simple to understand, easily maintainable and expandable, can be statically linked and dynamically loaded with the same tools and makes it possible to concentrate modifications of the system to as little parts of the code as possible, i.e. to runtime and compiler but not to loader and linker.

Although our ideas are universal and do not depend on a special implementation, we will describe and explicate the new approach with the runtime

support of the programming language Active Oberon, a modular programming language in the tradition of Pascal and Modula.

2 Common Object File Formats

This section discusses the features of some popular object file formats with respect to the metadata contained therein. Although all of them do feature portability across different environments, they incorporate metadata in a proprietary format.

- Portable Executable [2]

  The Portable Executable (PE) file format subdivides the metadata stored therein into two distinct categories. First of all, the headers of the file format describe the physical contents of the file in terms of sections that are loaded into memory by a linker. All data and code that must be loaded can be described using different sections with different settings. On the other hand, there is a so called image data directory which describes metadata that is to be used from within the loaded code itself. This data structure stores various tables with information for exporting and importing symbols, exception handling, debugging and other architecture-specific issues. Each of the 16 tables has its own distinct format and is represented differently in the object file.

  When a Portable Executable file is loaded, the whole contents of the image data directory is directly copied into memory. After that all file offsets of referenced elements within this data structure are replaced by corresponding memory addresses. This way, the data structure is ready to be examined by the runtime system without further transformations.

- Executable and Linkable Format [5]

  The Executable and Linkable Format (ELF) consists of various headers which describe the contents of the rest of the file. This content is partitioned into sections which are used to represent all binary code and data in memory at runtime. Metadata is mostly stored within sections rather than taking over a distinct place in the object file. However, the meaning of the binary content of each section depends on its type and differs for every form of metadata stored therein.

The reason why metadata is represented using a special format that differs from the binary contents of the object file is mainly twofold. On the one hand, the metadata may have to be checked before it is transformed and represented in memory. Secondly, the linker process may be the only one that has the necessary access rights to data structures that have to be updated by the metadata.

However, this design suffers from the following problems. Both cases imply that the linking process itself may be more complex than just loading the binary contents and resolving references therein. In addition to this increased cost of

Feature	Meta-Data
Type test	Type descriptor
Method dispatch	Type descriptor
Dynamic module loading	List of loaded modules
Command execution	List of commands in module
Garbage collection	Pointer offsets in heap and stack frames
Exception handling	Handled code areas and handlers
(Post mortem) debugging	Symbol information for stack frames

Table 1: Features and corresponding metadata for Active Oberon.

loading at run-time, the generated metadata has also to be stored in a specific format at compile-time. This means that the original metadata of the tool-chain may be transformed twice or even more times into a special format, before it can be actually used at run-time. Furthermore, if the designer of the tool-chain or runtime system modifies some characteristics of the metadata, the object file format as well as the linker have to be adapted accordingly. One goal of the design of the new object file format was to overcome all of these problems while still providing the performance of data structures that are directly loaded into memory.

3 Metadata of Active Oberon

In this section we describe the runtime components of a system that supports dynamic module loading. As indicated in the introduction, we use the runtime structures supporting the programming language Active Oberon (cf. [6], [3], [4]) as an example. Active Oberon is a type-safe, object oriented, modular programming language in the tradition of Pascal and Modula. It is garbage collected and features mechanisms for the creation and synchronization of multiple threads using monitors.

Table 1 provides a list of features that are supported by Active Oberon and that require that compiler and runtime system establish a reference to the relevant metadata.

To support the features listed in Table 1 the object files evidently have to comprise the relevant metadata in one form or the other. The binary object file format of Active Oberon adopted prior to this work consisted of sections that provided metadata in a proprietary format that all tools such as compiler and loader had to be able to understand. Table 2 lists the sections of this old object file format and indicates the storage data for each section. Naturally we do not go into details of the storage format here, we only make the remark that sections and data records were tagged with sentinels to find inconsistencies of the format in a reliable way. Moreover, hash values representing the interface of the imported and exported symbols were (and still are) used to detect inconsistencies during loading of object files (admitting so called 'fine grained fingerprinting', cf. [1]).

Section Name	**Purpose**	**Data Stored**
Code	executable code	sequence of bytes
Constants	constant data	sequence of bytes
Commands	Commands	list of: name, arg type, ret type, code offset
Pointers	Addresses of global pointer variables	list of pointer offsets
PtrsInProcs	pointers in procedures	list of: code offset, begin offset, end offset, number pointers, list of pointers.
Imports	imported modules	list of module names
Links	fixup lists	code and data offsets of fixup queues and case tables
Exports	exported symbols	list of: code offset, fingerprint, entry, export type
Use	references to imported modules and system calls	module name, scope, entry, fingerprint
Types	description of types	list of: name, base module, base name, methods (etc.)
Refs	debugging	(long proprietary format)
ExceptionTable	exception handling table	list of: pc from, pc to, pc handler reference

Table 2: Sections of a traditional object file of Active Oberon.

4 The Linking Process: Static vs. Dynamic Linking

In the following, by (dynamic) *loading* we denote the loading of object files together with the subsequent preparation of runtime data structures in a running system. By (static) *linking* we mean the loading of object files together with the subsequent preparation of a (kernel-)image that, once loaded to a fixed address, is intended to be running on bare hardware.

The way modules are integrated into a system or kernel from object files in a system with static and dynamic loading is slightly different. However, any linker or loader must provide at least a mechanism to patch fixups (i.e. resolve symbols) when arranging sections in memory or in a binary boot image. In the conventional setup, metadata are generated from designated parts of the object file stored in a proprietary format.

We first consider the process of dynamically loading a module into a running system. For gaining a shallow understanding, Listing 1 contains the runtime data structures representing loaded modules in the Active Oberon runtime system. It is the job of the loader to

- recursively load all imported modules that are not yet loaded,
- allocate a module data structure,
- load and allocate code and data sections,
- parse metadata from the object file to create and fill in all runtime data structures (such as type descriptors, pointer offset, exception handlers etc.)
- patch all fixups to symbols located in the module and in imported modules.

The Previous Approach: Static Linking using a Simulated Heap

Surprisingly, static linking is conceptually even more complicated than dynamic loading in this context. This is due to the fact that the runtime system relies on the consistency of its static and dynamic components, i.e. the modules in the statically linked kernel and the dynamically loaded modules on top must be represented in the same way. It is thus important that a statically linked kernel reflects a system that behaves as if its modules had been loaded (and allocated) by a loader at runtime. One way to solve this hen and egg problem is to use a *simulated heap*.

The idea is to allocate a pseudo-heap and adopt a simulated module loading to place all modules and required data structures in this heap. Then store the heap as an array of bytes. Therefore a working simulation of the loading and allocating functionalities of the runtime system has to be provided. In essence, this means that a substantial part of the runtime system has to be cloned and slightly modified, for example to protect data structures from being garbage

```
Module= object
var
    next*: Module; (* modules are queued in a global list *)
    name*: Name; (* name of this module *)
    init, published: boolean;
    refcnt*: longint (* #modules importing this module *)
    modules*: pointer to array of Module; (* imported modules *)
    data*, code*, staticTypeDescs*, refs*: Bytes; (* code, data, debugging info *)
    command*: pointer to array of Command; (* commands *)
    ptrAdr*: pointer to array of address; (* pointer offsets in global variables *)
    typeInfo*: pointer to array of TypeDesc; (* type descriptors *)
    procTable*: ProcTable; (* table containing procedure layout information *)
    ptrTable*: PtrTable; (* table containing pointers in procedures *)
    export*: ExportDesc; (* export descriptors *)
    term*: procedure; (* termination procedure *)
    exTable*: ExceptionTable; (* exception handling *)
end Module;
```

Listing 1: Runtime data structures of modules

collected by the runtime system. The so modified loader has to imitate the functionality of the dynamic loader and to patch addresses in the pseudo-heap. The advantage of this approach is that the linker indeed imitates the behavior of the runtime system and thus generates an image 'as if the loader has always been there'.

However this approach has also disadvantages: The loader and all relevant data structures (such as the module displayed in Listing 1) are duplicated. A modification of the kernel implies a lot of work and is error-prone. A reduction of the complexity is hardly possible. Moreover, cross-linking to other platforms is impossible, providing a real show-killer for this approach.

5 The New Approach

As indicated in the previous sections, it was our goal to come up with an approach that provides a unification of loading and linking. Moreover, modifications of the language and runtime system should be reflected only in the compiler and the runtime system but not in all other tools dealing with object files. Recall that the linker and the loader have to provide at least a facility to patch fixups.

Of course it is necessary that metadata are made available in the runtime after load- and boot time. Our trick is that the compiler generates all metadata *in ordinary data sections* and uses the fixup mechanism to ensure that the necessary links are established by loader and linker. This has the following implications:

- The object file consists of code and data sections only. No further section

```
module M;
import Trace;

type
    A = object end A;
    B = object (A) end B;

procedure TypeTest(a: A);
begin
    if a is B then Trace.String("a IS B"); end;
end TypeTest

end M.
```

Listing 2: A sample module M

types have to be introduced.

- For a modification of the runtime structures only the compiler and little parts of the runtime modules have to be adapted.
- Loader and linker do only need to arrange data in memory / kernel image and patch the fixups.
- Loader and linker are nearly identical and can use the same code base for patching fixups.

Optimizations, such as sorting and generation of hash-tables for symbols, can still be performed by the loader. We do not have to sacrifice performance.

An Example

For an illustration of the new approach Listing 2 displays a very small module. To illustrate how meta data are generated and referenced in the generated object file it contains a simple type test. During loading of module M, the runtime data as displayed in Listing 1 have to be generated from the object file by the loader.

A part of the object file generated from module N is displayed in Listing 3 in a textual format. The object file format is described in detail in the appendix of this paper. However, without looking into any details, the reader can gain a rough understanding how the fixup mechanism of the linker will provide the referencing of the module data structure to procedure TypeTest and to the type descriptor of B.

Metadata Registration

The linker does not have any knowledge about the metadata contained in an object file. The registration of the metadata into the runtime system conse-

```
const M.@Module −533068328 8 aligned 4 12 220
    (∗ fixups ∗)
    Modules.Module 1561901731 1 abs 76 0 1 0 32 1 84
    M.@CommandArray −877097149 1 abs 32 0 1 0 32 1 144
    M.@PointerArray 1604245058 1 abs 32 0 1 0 32 1 148
    M.@TypeInfoArray 1732675305 1 abs 32 0 1 0 32 1 152
    M.@ProcTable 1783720974 1 abs 32 0 1 0 32 1 160
     ...

const M.B 287569012 8 aligned 4 3 96
    (∗ fixups ∗)
    M.B 287569012 2 abs 72 0 1 0 32 1 60 abs 80 0 1 0 32 1 76
    M.A 2028155916 1 abs 72 0 1 0 32 1 64
    M.B@Info −796607100 1 abs 0 0 1 0 32 1 68
     ....

code M.TypeTest −2126198741 8 aligned 1 3 45
    (∗ fixups ∗)
    M.B 287569012 1 abs 72 0 1 0 32 1 12
    M.@const0 −380732481 1 abs 0 0 1 0 32 1 30
    Trace.String −696762289 1 rel −4 0 1 0 32 1 35
    (∗ code ∗)
    55985EB8D780B877CF18E74F00000000F048500000009EC0000000A670860000
    00008E9DFFFFFF98CED52C4000

const M.@ProcTable 1783720974 8 aligned 4 4 88
    (∗ fixups ∗)
    Heaps.SystemBlockDesc −88187294 1 abs 72 0 1 0 32 1 4
    M.@ProcTable 1783720974 1 abs 32 0 1 0 32 1 12
    M.TypeTest −2126198741 4 abs 0 0 1 0 32 1 48 abs 45 0 1 0 32 1 52 ...
    M.@Body −509539564 4 abs 0 0 1 0 32 1 68 abs 38 0 1 0 32 1 72 ...
     ...
```

Listing 3: Part of the object file of sample module N

```
module N;

procedure P;
begin ... end P;

begin P
end N.
```

Listing 4: A sample module N

```
.initcode N.$$BODYSTUB // guaranteed to be executed if statically linked
    0: call u32 Test.$$Body:0,0

.bodycode N.$$Body   // called by loader or from initcode
    0: enter 0,0
    1: push u32 N.@Module:21
    2: call u32 Modules.PublishThis:0,4    // try registration of module
    3: brne u32 N.$$Body:7, u8 $RES, 1     // if not successful then escape
    4: call u32 N.P:0,0                    // otherwise execute body (call P)
    5: push uew N.@Module:21
    6: call u32 Modules.SetInitialized:0,4 // .. and mark module initialized
    7: leave 0
    8: return 0
```

Listing 5: Part of the intermediate code of the sample module N

quently has to take place in the code that is executed. Therefore the linker has to make sure that the registering code gets executed.

In the modular programming language Active Oberon, the body of a module provides the initialization code of a module. It has to be executed at module load time and therefore provides the ideal place for metadata registration. Thus, our compiler instruments the code for module registration in the body. This is illustrated with the sample code provided in Listings 4 and 5 in source code and intermediate code, respectively.

6 Evaluation

To be able to judge the benefit of the new approach, we measured the complexity of the implementations very roughly by code size. Tables 3 and 4 comprise the lines of code and number of characters of the source code and the size of the compiled modules for all components necessary to compile, link and load an object file using the old and new approach, respectively. The sizes of the new approach are substantially smaller than those of the old one. In addition, the new object file approach allows the addition of more supported targets without major modifications and any change in the kernel do only imply modifications

Module	Lines Of Code	Characters	Code Size
Linker0	1554	57k	26k
Linker1	887	28k	17k
Linker	95	3k	3k
Loader	891	28k	18k
Object File Writer	2009	71k	37k
Sum	5456	187k	101k

Table 3: Numbers for the old approach

Module	Lines Of Code	Characters	Code Size
Generic Object File Writer	278	8k	6k
Compiler Object File Writer	135	5k	6k
Generic Linker	238	8k	8k
Static Linker	400	16k	10k
Loader	380	12k	6k
Sum	1427	49k	36 k

Table 4: Numbers for the unified approach

of the compiler and runtime system, not the loader and linker.

7 Conclusion

The increasingly complex format of the previously used object files of our operating system, also had dramatic impact on the complexity of the loader and linker. One of the goals of the development of a new object file format was to reduce the complexity of the format and therefore also of the tools that generate and process object files. The search for a simple format fulfilling these goals led to interesting observations.

First of all, there was the idea to treat metadata differently from the way most of currently used object file formats do. We instead proposed to unify binary data as well as metadata using the same representation for both. Not having to distinguish between the two automatically reduced the complexity of the object file, since everything that has to be stored can be represented as plain binary content. The duty of both the linker and the loader therefore boiled down to the two fundamental functions of arranging the binary content in memory and resolving inter-references therein. Both tools could therefore be unified and are essentially the same. Together with the carefully chosen fixup format, the linker and loader gained full cross-linking capabilities. Finally, all phases during compilation, linking and loading that process object files do not depend on the actual content or layout of the metadata stored therein any longer. This naturally decreased the code size of the tool-chain and the runtime system while increasing the maintainability of them substantially.

A Object File Representation

Object files are represented as a set of uniquely identified *sections*. Each section contains the binary data of code or global data that naturally maps to an entity of the programming language like procedures or global variables. Sections may contain information about how this binary data has to be placed in memory by a linker. In addition each section contains a list of *fixups* that refer to other sections. Fixups specify how the linker has to modify the binary data with the unique address of the referenced section once it has been placed. The remainder of this section describes the information stored in section in more detail and shows how it is used during linking.

A.1 Section Types

Each section has a specific *section type* which describes the content and the role of the binary data stored within the section. It can be one of the following types.

- Standard Code Sections

  Standard code sections contain ordinary code and are usually used to model procedures. They are typically called from within other code sections. Standard code sections have no special requirements for their placement in memory other than an optional alignment.

- Initializing Code Sections

  Initializing code sections are special code sections that are placed by linkers at the very beginning of a statically linked image. This guarantees that all initializing code sections are executed automatically before any other code section when the execution control is transferred to the image. This is used to generate all necessary calls to the procedure bodies of Active Oberon modules in a platform-independent way.

- Body Code Sections

  Body code sections are just standard code sections used to identify the bodies of Active Oberon modules. This is used while dynamically loading the module where the runtime system instead of the code itself has to call the procedure body of the module.

- Standard Data Sections

  Standard data sections provide the space and contents of global data. This data is usually modified during the execution of a program by the code within code sections. Data sections may contain predefined data that is initialized accordingly by the linker.

- Constant Data Sections

Constant data sections are data sections that are not supposed to change their contents during the execution of the program. They are used to model global constant data like strings.

A.2 Section Fixups

The sections in an object file refer to each other on various occasions. The binary code in code sections usually needs the relative address of a procedure when calling it. Data sections on the other hand are oftentimes used to store the absolute address of other data or code sections. This interconnection of sections is provided by *section fixups* that are contained within sections.

Each section fixup refers to a single section by name. The name has to be resolved by a linker and is replaced by the corresponding memory address. Section fixups additionally store a list of *patches* which specify how this memory address has to be patched within the binary contents of the section. A patch stores information that allows to specify whether the address has to be patched relative or absolute with respect to the address of the binary data where the patch happens. This place is specified by an offset relatively to the beginning of the binary contents. In addition, the address can be displaced as well as shifted before it is finally written to binary data.

A list of *patch patterns* specifies how the patched address is actually written to the binary data. A pattern consists of an offset relative to the patch offset as well as a signed number of bits that specifies the size of the pattern and its direction. For each pattern in the list, the specified amount of bits are taken from the patched address and written at the specified offset. The direction of the pattern specifies whether the offset increases or decreases during this process, and allows therefore to conform to endianness constraints. Before continuing with the next pattern, the patched address is shifted accordingly by the size of the pattern.

A.3 Linking Phases

The linker processes object files in several phases which are described below.

1. Reading

   All object files are read from disk. Besides the binary format that is used for fast input, there exists also a human readable text format to represent object files as shown in Figure 1 in this appendix.

2. Resolving

   The fixup list of each section is traversed and their names are resolved in order to remove unreferenced sections.

3. Arranging

   All referenced sections are placed in memory adjacent to each other according to their section type and optional alignment constraints. The

binary contents of the section is copied to the resulting unique address. A section may be fixed in which case the object file dictates the memory address where the section has to be placed.

4. Fixup Patching

   The fixup list of each section is traversed again in order to get the addresses of the resolved sections. For each patch of a single pattern the address is patched as specified and written to the memory according to the list of patch patterns.

The actual bit size of the unit of some quantities used during linking is configurable and stored separately for each section. This link mechanism as described above is generic enough to allow to conform to any possible bitmask predefined by any instruction set architecture. Therefore, object files as presented here allow to store any binary code and data for any hardware architecture.

References

[1] R. B. J. Crelier. *Separate compilation and module extension.* Phd thesis, ETH Zürich, 1994.

[2] Microsoft Corporation. *Microsoft Portable Executable and Common Object File Format Specification*, 2010. Revision 8.2.

[3] P. J. Muller. *The Active Object System Design and Multiprocessor Implementation.* PhD thesis, ETH Zrich, 2002.

[4] P. R. C. Reali. *Using Oberon's active objects for language interoperability and compilation.* PhD thesis, ETH Zrich, 2003.

[5] A. Telephone and T. Company. *System V application binary interface: UNIX System V.* UNIX software operation. Prentice-Hall, 1990.

[6] N. Wirth and J. Gutknecht. *Project Oberon : the design of an operating system and compiler.* ACM Press, New York etc., 1992.

```
ObjectFile       = {Section}.
Section          = Type Name Unit Relocatability
                   NumberOfFixups NumberOfBits
                   {Fixup} {Octet} [Sentinel].
Type             = "code" | "initcode" | "bodycode" |
                   "data" | "const".
Name             = identifier Fingerprint.
Fingerprint      = integer.
Unit             = integer.
Relocatability   = "aligned" Alignment | "fixed" Address.
Alignment        = Unit.
Address          = Unit.
NumberOfFixups   = integer.
Fixup            = Name NumberOfPatches {Patch}.
NumberOfPatches  = integer.
Patch            = Mode Displacement ShiftScale
                   NumberOfPatterns {Pattern}
                   NumberOfOffsets {Offset}.
Mode             = "abs" | "rel".
Displacement     = Unit.
ShiftScale       = integer.
NumberOfPatterns = integer.
Pattern          = BitOffset NumberOfBits.
BitOffset        = integer.
NumberOfBits     = integer.
NumberOfOffsets  = integer.
Offset           = Unit.
Octet            = hexadecimal-digit hexadecimal-hexdigit.
Sentinel         = "n".
```

Figure 1: Textual object file format expressed in EBNF