

University of Bath



PHD

An architecture for interpreted dynamic object-oriented languages

Kind, Andreas

Award date:
1998

Awarding institution:
University of Bath

[Link to publication](#)

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 13. May. 2019

An Architecture for Interpreted Dynamic Object-Oriented Languages

submitted by

Andreas Kind

for the degree of Ph.D


of the **University of Bath**

1998

Copyright

Attention is drawn to the fact that copyright of this thesis rests with its author. This copy of the thesis has been supplied on the condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no quotation from the thesis and no information derived from it may be published without the prior written consent of the author.

This thesis may be made available for consultation within the University Library and may be photocopied or lent to other libraries for the purposes of consultation.

Signature of Author  ANDREAS KIND

UMI Number: U106735

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



UMI U106735

Published by ProQuest LLC 2013. Copyright in the Dissertation held by the Author.
Microform Edition © ProQuest LLC.

All rights reserved. This work is protected against
unauthorized copying under Title 17, United States Code.



ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

UNIVERSITY OF WYTH	
35	- 1 OCT 1993
MIS	

Summary

This thesis is concerned with the implementation of object-oriented dynamic programming languages based on bytecode interpretation. A new interpretive implementation architecture is proposed that meets the requirements of code and system portability, execution performance, static and dynamic memory efficiency as well as language interoperability.

The different quality of the architecture compared to other virtual machine approaches is related to the key techniques developed within this work: (i) C embedded virtual machine code, (ii) indexed code threading, (iii) optimal virtual instruction ordering and (iv) quasi-inline method caching. C embedded virtual machine code refers to the representation of bytecodes as constant C arrays that are located in sharable text segments after compilation. Interoperability, application start-up and dynamic memory usage benefit from this representation. Indexed code threading addresses the performance problem with virtual instruction mapping (i.e. loading, decoding and invoking) by using a fast threaded instruction transfer. Unlike with standard code threading, virtual machine code remains compact and executable also with a non-threaded virtual machine emulator. A further performance boost is achieved with optimal virtual instruction ordering. This technique helps to cluster the native code implementing virtual instructions so that native instruction cache performance is increased. Finally, the efficiency problem involved with dynamic method lookup is alleviated with an inline caching scheme that is applicable with constant bytecode vectors. The scheme exploits type locality similar to polymorphic inline caching. However, dynamic memory is saved by avoiding redundant method entries and by being adaptable to generic function invocation which typically comes in waves with hot-spots on particular methods.

A realization of the architecture is presented in form of an implementation of the dynamic object-oriented language EuLisp. The implementation demonstrates the feasibility and effectiveness of the proposed architecture. The average performance increase with indexed code threading is 14% (P5) and 17% (MIPS). The average increase with optimal instruction ordering in the indexed threaded interpreter is 21% (P5) and 15% (MIPS). Sharable read-only data is increased on average by a factor of two and finally, the miss ratio with quasi-inline method caching is measured as 1.06%.

Acknowledgements

Thanks go to my supervisor, Julian Padget, for giving me the great opportunity to start and finish this work. He initiated many aspects in this thesis and provided regular input and help.

I am most grateful for discussions with Russell Bradford, Rob Simmonds, Simon Merrall, Duncan Batey, Luc Moreau, David DeRoure and Guiseppe Attardi.

I would like to thank Horst Friedrich, Ingo Mohr, Malte Forkel and Hans-Otto Leilich for teaching me some basics.

Finally, special thanks to Josephine, Paulina and Mira for giving me a reason.

Contents

I	2
1 Introduction	3
1.1 Why Dynamic Objects?	4
1.2 Virtual Machines	5
1.3 Problems with Interpreted Dynamic Objects	6
1.4 An Architecture for Interpreted Dynamic Object-Oriented Languages	7
1.5 Outline	9
2 Dynamic Object-Oriented Programming	10
2.1 Smalltalk	10
2.2 Lisp	11
2.3 The Common Lisp Object System	13
2.4 Dynamic Objects	14
2.4.1 Dynamic Linking	15
2.4.2 Reflection	15
2.4.3 Automatic Memory Management	16
2.5 Decentralized Artificial Intelligence	17
2.6 Terminology	18
3 Virtual Machines and the Requirements of Dynamic Objects	21
3.1 Virtual Machines	21
3.1.1 Speed	23
3.1.2 Space	23
3.1.3 Versatility	25
3.2 Requirements of Dynamic Objects	27
3.2.1 The Performance Cost of Execution Dependencies	28
3.2.2 The Memory Cost of Execution Dependencies	31
3.2.3 Scripting	31

3.3	Conclusion	32
II		34
4	The Architecture	35
4.1	Embedding Virtual Machine Instructions	35
4.2	Indexed Code Threading	37
4.3	Optimal Instruction Ordering	39
4.4	Quasi-Inline Method Caching	40
4.5	EuLisp	42
4.6	Conservative Garbage Collection	43
4.7	Conclusion	44
5	The Mechanics of Dynamic Objects in youtoo	46
5.1	The youtoo Compiler	46
5.1.1	Dynamic Linking	49
5.1.2	Source Code Interpretation	49
5.1.3	Example 1	50
5.1.4	Example 2	53
5.2	Code Mobility	54
5.3	Demonstrating Performance	55
5.3.1	Indexed Code Threading	56
5.3.2	Optimal Instruction Ordering	57
5.3.3	The SPARC Oddity	59
5.3.4	Quasi-Inline Method Caching	60
5.4	Demonstrating Memory Efficiency	62
5.5	Demonstrating Interoperability	62
5.6	Conclusion	65
6	Related Work	66
6.1	Smalltalk-80	66
6.2	Self	67
6.3	Slim Binaries	68
6.4	Translation into C	68
6.5	Java	69

6.6	Other Techniques	70
6.6.1	Type Inference	70
6.6.2	Method Lookup Optimization	70
6.6.3	Stack Caching	71
6.6.4	Sealing	72
6.6.5	Method Inlining	72
7	Conclusions	73
A	Assembler Code	87
A.1	Virtual Instruction Transfer on P5	87
A.2	Virtual Instruction Transfer on MIPS	88
A.3	Virtual Instruction Transfer on SPARC	89
B	Various Tables	90
C	Measurements	92

Part I

Chapter 1

Introduction

The real world is a highly dynamic and complex system. This suggests using programming languages that can handle dynamism and complexity in a natural and simple way: dynamic object-oriented languages. This thesis is concerned with the implementation of dynamic object-oriented programming languages based on bytecode interpretation.

The ability to handle dynamism and complexity naturally with dynamic object-oriented programming (or for short *dynamic objects*) is based on emphasizing execution dependencies as opposed to emphasizing compilation dependencies in more static languages. The emphasis on compile-time dependencies in languages like C++ is illustrated by the fact that changing a compilation unit generally requires recompilation of all dependent compilation units. Such static dependencies result from the efficient mapping from the object-oriented performance model to the hardware performance model. An implication of this mapping is that class and method definitions may not change after compilation. The emphasis on execution dependencies in languages like CLOS is likewise illustrated with the ability to change, add or remove class and method definitions at run-time. Dynamic and complex applications can benefit from such execution dependencies as recompilation can be avoided.

In a further step toward dynamic dependencies, classes and methods are used to implement the object system itself so that a modification of these defining classes and methods results in a modification of the semantics of the object system. By doing so, the semantics of the language can be customized for individual application domains [KdRB91]. Such reflective capabilities in dynamic object-oriented languages is typically combined with dynamic typing, automatic memory management and run-time linking [Nas92].

In contrast, static object-oriented languages, like C++, Eiffel or Haskell, do not support

run-time control over the object system to that extent. These languages are designed with a clearer separation between compile-time and run-time. Classes, methods and functions are compile-time concepts and do not appear as run-time data objects.

1.1 Why Dynamic Objects?

The importance of dynamic object-oriented languages is based on the fact that the trend towards distribution, symbolic computation and evolutionary software development requires today's applications to deal with a high degree of dynamism and complexity. The increased use of dynamic object technology with languages like Smalltalk, CLOS and (partly) Java in the commercial environment reflects this importance [Gro95, Sha95].

Distribution Infrastructures for distributed computing are now ubiquitous and ready to be used by a new generation of applications based on distributed software components and mobile code. Object-oriented programming has long been acknowledged as advantageous for dealing with complexity inherent in distributed systems [NWM93]. However, popular object-oriented *static* languages with their emphasis on compilation dependencies are not ideally suited for code mobility in heterogeneous computer networks. In contrast, the combination of dynamic linking, automatic memory management, dynamic typing and multi-threading has a natural potential to handle the intrinsic dynamic behaviour of distributed systems, such as in the form of asynchronously arriving active objects [CJK95].

Symbolic Computation The second reason for an increased call on dynamic computation derives from applications which incorporate symbolic computation. Today's standard hardware finally gives acceptable performance to knowledge-based systems. Typically, these systems process symbolic data rather than numeric or other low level data (e.g. bit/character strings). Symbolic data is intrinsically abstract and irregular which can make static typing less precise as well as processor and memory consumption unpredictable. Dynamic typing and automatic memory management are therefore desired features of knowledge representation languages.

Evolutionary Software Development It has been realized that static object-oriented languages cause versioning problems when components, which are used by other components, require modification. These languages normally use static type information for efficient method lookup based on indirection through a statically computed method table. This approach compiles method lookup *into* applications. Changing a class definition in a compilation unit thus forces recompilation of all dependent compilation units. This "fragile base class

problem” limits the scalability of complex software systems and does not exist with method lookup based on execution dependencies in dynamic object-oriented languages. Here, even a dynamic evolution of software is actually feasible for long-running (potentially distributed) applications that cannot be simply stopped, modified and restarted (e.g scheduling systems). Therefore, dynamic object-oriented programming supports the customization and rapid modification of applications according to individual or changing needs [LV97, Cor97, Phi97, DB97].

Distribution, symbolic computation and evolutionary software development share a demand for dynamic flexibility provided with automatic memory management, introspection and dynamic linking—key features of *dynamic* object-oriented programming [Nas92].

Research into integrating and coordinating human and automated problem solvers in large computer and telecommunication networks is driven by the metaphor of group interaction and social organization and is known as decentralized (or distributed) artificial intelligence (DAI) [Gas92]. DAI as a high-level form of distributed computing, is one of the most promising future application areas across the Internet and on company-wide intranets. DAI is linked to the three areas above—distribution, knowledge representation and software evolution. Dynamic object-oriented programming is therefore suited for implementing DAI applications (see [Ham97, Way95, RNSP97] and others).

1.2 Virtual Machines

Compilation into native machine code and direct (or tree) interpretation offer different trade-offs. Both implementation techniques are not optimal with regard to a compound measure including the size, speed and versatility of the corresponding executable program representation [Hoe74, DVC90]. A much better overall value can be achieved with bytecode interpretation. Source code is here transformed into semantically equivalent instructions (bytecodes) of a virtual machine. An interpreter program that emulates the virtual machine executes the virtual machine instructions.

The virtual machine (or bytecode) approach provides an architecture neutral and compact executable program representation that enables code mobility in heterogeneous environments [Gos95]. With regard to the application domain of high-level distributed computing identified in the previous section, this approach is thus a reasonable implementation technique for an dynamic object-oriented language system and sets the general scope for this thesis: dynamic objects with virtual machines.

1.3 Problems with Interpreted Dynamic Objects

Bytecode interpretation and the emphasis on run-time dependencies in dynamic object-oriented languages are not without disadvantages. Speed and space problems create a popular stigma: dynamic object-oriented applications executed on virtual machines are inefficient with regard to execution time and dynamic memory consumption.

Dynamic object-oriented applications show reduced performance compared to more static languages due to the higher overhead and higher frequency of dynamic method lookup. Furthermore, the lack of static type information, higher-order functionals (including continuations) and the capability to extend a program at run-time by dynamic linking, eliminate precise control and data flow prediction which is necessary for standard optimizations and a direct map onto the underlying hardware performance model.

The emphasis on run-time dependencies by means of dynamic linking (with potential run-time evaluation) as well as the use of classes and methods as first-class values require applications to carry around much more code and data than actually necessary [Shr96]. The detection of unused code (e.g. evaluator) and unused data (e.g. metaobjects) is difficult and hinders the delivery of small executables.

Bytecode interpretation is responsible for a further decrease in performance and increase in dynamic memory consumption. Mapping (i.e. loading, decoding and invoking) of virtual machine instructions in the emulator program decreases performance compared to native compilation by at least an order [DVC90].

The separation between execution on the bytecode level and execution on the native machine code level is also disadvantageous for dynamic memory footprints since virtual machine code is treated by the operating system as data rather than sharable text, as with native compilation. Operating systems are therefore not able to share virtual machine code in memory among all the applications executing it concurrently. Such shareability can be of great importance when several applications are started by a user that is using a machine directly and/or by different users that are running applications over a network connection to a machine.

1.4 An Architecture for Interpreted Dynamic Object-Oriented Languages

Efficiency problems with dynamic objects can be addressed by restricting the dynamic character of the source language and introducing means for enhanced static analysability, such as explicit typing, sealing of program parts, reducing reflective capabilities or imposing a closed program assumption with a immutable set of bindings. The ideas put forward in this thesis are different from these approaches as it is attempted to achieve efficiency *without* sacrificing the dynamic character of the source language. The dynamic aspect is regarded as the distinctive feature of dynamic object-oriented programming. Instead of starting at the source code level, the approach here is concerned firstly with the efficiency of the executable program representation and derives from there a novel technique to alleviate the cost of dynamic method lookup.

The contribution of the work presented here lies in the design of a new virtual machine architecture for dynamic object-oriented languages. The approach succeeds in improving efficiency of object-oriented applications by combining four techniques: (i) C embedded virtual machine code, (ii) indexed code threading, (iii) optimal virtual instruction ordering and (iv) quasi-inline method caching.

Embedding Virtual Machine Code The architecture achieves modest dynamic memory consumption with C embedded constant virtual machine code which is compiled into sharable native code (located in text segments) after C compilation. Such sharing results in small memory footprints since code is not duplicated but shared in memory by different processes (i.e. applications) executing it. Read-only code vectors further lead to optimized automatic memory management as they are not heap-allocated and therefore not considered (traced or copied) with automatic memory management.

C embedded virtual machine code simplifies development of software that uses third party software (or which is itself part of another software package) with interoperability on the C language level. Foreign addresses can be used directly within separate compilation units. Neither the virtual machine, nor the run-time support code, need to be extended for interoperability with foreign code. The representation of virtual machine code on the C language level does not compromise the architectural neutrality of the executable program representation. The architecture neutral bytecodes can be extracted from a generated C file or a functional object in order to be sent to other virtual machines in a distributed

heterogeneous environment. The representation of virtual machine code on the C language level therefore combines the advantages of bytecode interpretation and native compilation (or translation into C).

Indexed Code Threading The architecture reduces execution time of applications with optimized transfer between virtual machine instructions by using indexed code threading, a variation of code threading [Bel73]. In contrast to standard code threading, virtual machine code with indexed code threading is compact and portable in the sense that it can be linked to a code threaded or switched version of a C-based interpreter. Average speed ups for this technique are measured as 14% and 17% on P5 and MIPS processors respectively.

Optimal Instruction Ordering Another performance improvement of 21% and 15% can be reported with optimal virtual instruction orderings, again on P5 and MIPS respectively. Clustering the native code of virtual machine instructions that are likely to be executed consecutively, increases the chances that the code is already in the native instruction cache. An approximation of such a clustering can be derived from profiling the dynamic invocation frequency of virtual instructions. The native code of the virtual instruction called most will then be next in memory to the native code of the virtual instruction called second most etc. Particularly, for the small range of virtual instructions that typically dominate applications, such an ordering results in much better native instruction cache performance than the typical ad-hoc ordering. With the help of a tool developed within the context of this thesis, optimal orderings can easily be derived for any range of applications.

Quasi-Inline Method Caching The cost of method lookup with dynamic objects is reduced with a flexible and successful inline caching technique, called quasi-inline method caching. Similar to classical inline method caching [DS84, HCU91], type locality with dynamic method lookup is exploited to achieve high method-cache hit rates. Quasi-inline method caching, however, is particularly suited to read-only virtual machine code. This technique is furthermore designed to adapt to hot-spots common in dynamic object-oriented programs so that cache sizes can be smaller than with other approaches. The average cache miss ratio accomplished with a realization of quasi-inline method caching is around 1.06%.

Although each of the four techniques can be applied without the others in a bytecoded object system, they are not unrelated in the architecture. The combination of the techniques strives to continue the evolution of bytecode interpretation driven by the specific requirements of dynamic object-oriented programming, including efficiency, interoperability and portability.

1.5 Outline

This document is divided into two major parts. Part I introduces dynamic object-oriented programming languages with its distinctive characteristics (Chapter 2) and requirements (Chapter 3). Important issues for implementing these dynamic languages are identified and the approach taken within this thesis is distinguished from others.

Part II presents a new architecture for implementing dynamic object-oriented languages with virtual machines (Chapter 4). The techniques of C embedded virtual machine code, indexed code threading, optimal instruction ordering and quasi-inline method caching are described. Furthermore, this part provides insight into the `youtoo` system, an implementation of the dynamic object-oriented programming language EuLisp (Chapter 5). The implementation shows the feasibility and effectiveness of the proposed architecture. The empirical results collected with its realization are compared with other related work (Chapter 6). Finally, the thesis is concluded and summarized (Chapter 7).

Chapter 2

Dynamic Object-Oriented Programming

Smalltalk, as the most influential object-oriented language, features classes and methods as dynamic language constructs which can be created, inspected, modified and linked during run-time. Later, *static* object-oriented languages, like C++ and Eiffel, put much more stress on static dependencies and deliberately abandoned the dynamic character for better performance and earlier error detection.

This chapter introduces Smalltalk and CLOS, two representative members of the family of dynamic object-oriented languages. Key concepts of dynamic object technology are reviewed, namely dynamic typing, automatic memory management, dynamic linking and reflection. Finally, a terminology is given that is used within the rest of this thesis.

2.1 Smalltalk

Historically, Smalltalk [GR83, Gol95] is (after Simula) the second object-oriented language. The language is based on objects as a uniform representation of data. The structure and behaviour of a set of objects is defined by a class. An object is called an instance of a class if the class defines its structure and behaviour. The structural information in a class is used to create new instances. The behaviour of instances is defined in terms of operations called methods. An object executes a method when it receives and recognizes a message. The execution of a method can involve accessing the state of the receiver object, invoking a primitive or sending a new message.

Super- and subclass relationships among classes define a class hierarchy so that structural and behavioural information can be shared along the hierarchy. A class can extend the structural description of instances defined in its superclass. Likewise, the methods applicable to instances of a class (effective methods) are given by the methods directly defined at the class (direct methods) and the effective methods of the superclass. The access of structural and behavioural descriptions defined in class C_1 from a class C_2 which is a direct or indirect subclass of C_1 , is called inheritance. By sharing structural and behavioural information along the hierarchy, inheritance reduces the need to specify redundant information, simplifies modification and therefore facilitates software reuse [SB86, Sny87, Wei97].

Message sending is different from procedure calling in statically-typed procedural languages. The exact class of the object receiving a message may only be known at run-time because methods are applicable to instances of subclasses of the class defining the method. Thus, the appropriate method for a message being sent to an object has to be determined dynamically¹. The dynamic binding process between messages and methods (late method binding) is specified by a method lookup algorithm.

A Smalltalk message is given by a message selector and a receiver object. Each class in the class hierarchy has a dictionary that maps message selectors to methods. The method lookup for a message send is then as follows:

1. Search for the message selector in the message dictionary of the receiver class. If the selector is found, return the corresponding method; otherwise go to 2.
2. If the receiver class has no superclass go to 3; otherwise set the superclass to be the receiver class and go to 1.
3. The message is not understood by the receiver.

Although message dictionaries are normally implemented as hash tables [Kra83], selector collision and superclass chain traversal result in slow method lookup times.

2.2 Lisp

In its semantics the non-object-oriented part of Smalltalk is very similar to Lisp, one of the first programming languages. Lisp, short for List Processor, was developed by John McCarthy

¹This fact is paraphrased later in the context of multi-methods (Section 2.3) into: The appropriate method for a call site of a generic function has to be determined dynamically.

in the late 1950s [McC59]. The idea was to express computation by symbolic functions similar to the lambda calculus. The List Processor was defined as a Lisp function itself, called the evaluation function. To simplify the evaluation process, functions were represented as lists, the basic Lisp data structure. This duality of data and program, together with a method to extend the evaluation function made Lisp a programmable programming language [Fod91]—a reflective language.

Lisp is no longer a single language. With many different dialects, it has become a family of languages characterized by the following features:

Dynamic Typing Dynamic typing is best explained as the opposite to static typing. With static typing, type correctness is decidable at compile-time. A language is dynamically-typed otherwise. For instance, ML is a statically-typed language because the correct invocation of functions with regard to the type system is decided statically. The type of any value can be determined during run-time of a dynamically-typed program in order to signal dynamic type violation and to provide type predicates. In certain cases, type correctness can be partly decided as well for dynamically-typed languages.

Closures Functions retain the lexical bindings in effect when created. A function can thus be regarded as code plus an environment. The environment can be (partly) shared with other functions that have been created within the same lexical scope.

Continuations The state of computation can be captured in a functional object called a continuation. When the continuation is invoked, the state of computation is re-activated and execution resumes from the point where the continuation was created. New control structures can be easily defined with continuations.

Higher-Order Functionals Functions and continuations have first-class status in the sense that they can be dynamically created, passed as arguments, returned from other functions, assigned to variables or stored in data structures. Functionals are thus treated like any other value.

Automatic Memory Management Unused storage is automatically reclaimed (see also Section 2.4.3).

Dynamic Linking New code can be linked to a running program (see also Section 5.1.1). Although this feature is not always explicitly mentioned in Lisp language definitions,

dynamic linking is typically provided with implementations by a loading or evaluation mechanism.

Macros Lisp source code is written in a Lisp-like syntax so that normal Lisp functions can act as syntax functions (i.e. macros) to pre-process source code. Generally, macros can use the full language.

The most prominent members of the Lisp family are Common Lisp [Ste84, Ste90, AI96] and Scheme [CE91, IEE91]. Common Lisp has successfully superseded a variety of Lisp dialects. For the sake of backward compatibility to the replaced dialects, Common Lisp is known to be “baroque” in its wealth of features. Scheme on the other hand is based on a few orthogonal concepts and minimalist in design. Close to the lambda calculus, Scheme allows for reasoning about its semantics and potential language extensions. Unlike Common Lisp, Scheme has no modules and no object system and is not suited for, nor aimed at commercial software development.

The ability to extend its syntax and its set of control structures gives Lisp the flexibility to host, or even change into, a new programming language [BKK⁺86]. The fact that an object-oriented paradigm can be added to Lisp by means of macros and some defining forms [Bra96, Que96], demonstrates this flexibility and links up to the following section: the Common Lisp Object System (CLOS).

2.3 The Common Lisp Object System

Common Lisp incorporates a variation on the Smalltalk approach to object-oriented programming implemented as the Common Lisp Object System (CLOS) [AI96, BDG⁺88b]. Like Smalltalk, this approach which is rooted in CommonLoops [BKK⁺86] and Flavors [Moo86], uses classes to define the structure, creation and access of objects. However, the notion of message sending is replaced by generic function invocation. With generic functions, all arguments are considered with dynamic method lookup, in contrast to the first argument (i.e. the receiver object) only as with message sending. Dynamic method lookup based on one argument only is called *single-method* dispatch; dynamic method lookup based on more than one argument is referred to as *multi-method* dispatch.

Methods in CLOS are not stored at classes together with other methods defined for instances of the same class, but at generic functions together with other methods that may be selected as the most specific method for a generic function invocation. Similar to Smalltalk,

the binding process between call sites of generic functions and methods is dynamic and specified by a method lookup algorithm.

Generic function invocation is motivated by the fact that the notion of passing a message to a single receiver is in many cases not adequate. For example, with single-methods an operation to write data to various streams (e.g. file, socket connection, string) has to be implemented with explicit discrimination on the type of either, the data or stream argument, in each method. Both objects could however be regarded as receivers of a write message. With multi-methods data *and* stream argument can be considered for method dispatch. Commutative arithmetic is another example which is unsuited to single-method dispatch.

Multi-method dispatch is sometimes described as a generalization of single-method dispatch. This is however misleading, since single-method dispatch provides much better encapsulation of data and code [CL94, DeM93, Cha92]. With generic functions, methods are not local to a specific class so that instance variables always require accessor functions for reading and writing. Direct access to instance variables from methods local to a class combined with restricted public access is only possible with single-method dispatch. This approach therefore provides much better support for encapsulation and information hiding.

The full method lookup with generic functions and multi-methods involves normally the access and invocation of a discriminating function which in turn calls a method lookup function that returns a list of methods applicable (with regard to their domain) to the supplied arguments. The method list is sorted by specificity (see Section 2.6). Hence, the first method is the *most specific*. Its associated method function is finally applied to the arguments. The rest of the method list is saved in case the next method is later requested with `call-next-method` (which is the equivalent to `super` in other languages).

2.4 Dynamic Objects

Smalltalk and CLOS have different approaches to object-oriented programming. However, they share the emphasis on execution dependency and clearly differ in that respect from object-oriented static languages. Compilation of these languages requires much more information about programs and is thus not ideally suited to productivity, but more targeted towards execution efficiency and implementational simplicity [Str93]. This difference is reflected in the three key features of dynamic object technology (or short dynamic objects): dynamic linking, reflection and automatic memory management [Nas92].

2.4.1 Dynamic Linking

The popularity of Lisp and Smalltalk stems from the comfortable and interactive development environments that have traditionally accompanied these languages [Shr96]. Typical features of interactive development environments are profiling, tracing, stepping, inspection and dynamic error handling with the option to resume computation after a modification and, in some cases, graphical user interfaces.

The basic method of interaction is through a read-eval-print loop² that reads and evaluates statements and finally prints the result of the evaluated statement. Definitions (and redefinitions) can be tested immediately without going through a time-consuming edit-compile-link-run cycle.

Such incremental development is related to dynamic linking, used in Lisp, Smalltalk and even C. In each case, the definition of bindings is not necessarily fixed at run-time. New program parts can be linked into a system as they are needed. Tight memory restrictions or dynamic configurability are typical reasons to use dynamic linking.

However, there is a performance tradeoff incurred by dynamic linking. Beside the time it takes to retrieve (from the local file system or off a network) the binding value, a significant overhead has to be accepted by the fact that the application is not closed. Interprocedural and global optimizations generally assume that function bindings do not change after compilation.

2.4.2 Reflection

Computational reflection is the ability of a program to access its structure and state during execution [Smi84]. Reading access—in the sense that a program observes and reasons about its structure and execution state—is referred to as *introspection*. Modification of a program's structure and execution state by a program itself is called *intercession*. Both aspects of computational reflection, introspection and intercession, are based on *reification*, the encoding of program and execution state as data.

Reflection in computational systems is driven by demand for extended flexibility. Perhaps the simplest introspective operator is **type-of** which is typically provided in dynamically typed programming languages. The operator returns a value that represents the type of its argument and therefore reveals already some insight into the representation of data during run-time. A bit more introspective information is necessary to write a generic walker to “walk” over arbitrary data structures including primitive and (possibly user defined) com-

²Also known as top-level.

pound data structures. A print function could use such a generic walker to visualize arbitrary data structures in a nested way. In this case, introspection can help to find out about the length, type, structure and access of data objects.

Intercession can be useful to handle evolving models. Some problem domains are intrinsically dynamic and cannot be correctly represented by a static model in a computational system [KAJ93]. Suppose, we have a model of a heterogeneous network where nodes are represented as instances of different classes. Unpredictably, within the real world new nodes appear, which require to be added as well to the model. With regard to a specific feature which these new nodes incorporate (but no other node had before), it may be desirable to represent the nodes as instances of a new class. A dynamic reorganization of the class hierarchy is however only possible by modifying existing classes, i.e. by intercession.

In both, Smalltalk and CLOS, classes and methods are first-class. However, only CLOS is metacircular in the sense that the object system itself is implemented in terms of objects, classes, and methods (reification). Since classes and methods are first-class values in the language, the structure and behaviour of the object system can be observed (inspection) and modified (intercession). The interface for inspection and intercession is generally known as a metaobject protocol [Coi87, KdRB91]. The essential idea of a metaobject protocol is to enable language users to adapt the semantics of the language to the particular needs of their applications.

Metaobject protocols can be regarded as an extension of the reflective features at the core of the List Processor (see Section 2.2) in the object-oriented context. By doing so, behaviour in the language level can be reused in the application level, resulting in less development cost and—theoretically—less execution time. In practice, reflective capabilities tend to be inefficient due to the fact that default semantics for object creation, slot access and method lookup cannot be “hard-wired” into the system. Like higher-order functionals and dynamic typing, reflection aggravates data and control flow prediction and impedes many optimizations.

2.4.3 Automatic Memory Management

In principle, dynamically allocated memory should be deallocated when no longer in use in order to avoid storage exhaustion. In practice, memory management is tedious and error-prone. Dynamic object-oriented languages critically depend on complete reclamation of unused activation records, closures and explicitly allocated application data because of high (non-tail³)

³A function call is in tail-position if its result determines the result of the function in which the call is located. The enclosing function returns with the result of the tail-call. Thus, the context of the function in which the

function invocation frequency and the typical profile of dynamic problem domains. An automatic kind of memory management is thus not only a standard feature in pure Lisp systems but also a defining characteristic of dynamic object-oriented languages.

Automatic memory management is generally known as dynamic memory allocation combined with automatic reclamation of storage that is no longer accessible by following pointers from program variables [App91, Wil95]. Automatic reclamation is also referred to as garbage collection.

2.5 Decentralized Artificial Intelligence

The Internet is constructed from open services built around a standard communication framework. Due to computer mobility, varying network latency, bandwidth and connectivity there is an increasing demand for off-line computation. The idea is that program modules are sent off as *mobile software agents* to run on remote machines and later return to report to the user [GK94]. In order to fulfil a task without user interaction, agents need to have some degree of mobility, autonomy and determination. Furthermore it can be envisaged that particular problems require cooperation with other agents. Software agents are studied in the field of decentralized artificial intelligence (DAI) which uses the metaphor of group interaction and social organization to integrate and coordinate human and automated problem solvers [Gas92].

Before agent-based systems became *en vogue* recently, related work has been done in the field of object-based concurrent systems [Hew77, AH87, WY88]. Typically, object-based concurrent applications rely significantly on flexible control of computation at run-time. Reflective capabilities can provide such flexibility by means of *metaobjects* that model structural and behavioural aspects of objects. From the modelling aspect, the combination of object-oriented programming and reflection is therefore a natural one [Mae87]. Encapsulation, data abstraction and incremental extension provide a suitable “hook” for computational reflection.

tail-call is located can be discarded/dismissed (i.e. removed from the context stack) just before the tail-call is performed. In the case of a recursive tail-call this has the important effect that tail-recursive functions (i.e. functions with tail-recursive calls) only need a constant amount of context stack. Iterative control structures can be simulated by tail-recursive functions with storage demand equivalent to true iteration. Tail-call optimization is a compulsory optimisation for implementations of some Lisp dialects, including Scheme and EuLisp.

2.6 Terminology

Throughout this thesis the general terminology and paradigm of object-oriented Lisp systems [BKK⁺86, Coi87, BDG⁺88a, KdRB91, BKDP93] is followed. The defining terms are classes, generic functions and methods.

A *class* stores structural and behavioural information about a set of objects which are its *instances*. The class⁴ of any object can be dynamically determined and accessed. Super- and subclass relationships among the classes define a class hierarchy. If not stated explicitly, a single superclass relationship per class (single inheritance) is assumed. A class hierarchy may look like this:

```

<object>
  <character>
  <collection>
    <table>
      <hash-table>
    <sequence>
      <character-sequence>
        <string>
      <vector>
      <list>
        <cons>
        <null>
  <number>
    <float>
    <double>
  <integer>
    <int>
    <bigint>
  <class>
    <simple-class>
    <function-class>
  <method>
    <simple-method>
  <slot>
    <local-slot>
  <function>
    <simple-function>
    <generic-function>
    <simple-generic-function>
  <name>
    <symbol>
    <keyword>

```

Procedures are generally called *functions* even when they are not referentially transparent (i.e. have side-effects). We refer to the source code position of a function call as *function call site*. The dynamic process of calling a function is called *function invocation*.

Simple functions are distinguished from generic functions. A *simple function* is defined by a single function body and is applicable to arguments of any type. Dynamic type checking

⁴There is no distinction between class and type.

signals inappropriate type usage. Types for arguments and return values are not specified. A *generic function* is defined in terms of methods which describe the behaviour of the generic function for different argument types. All arguments are considered for method selection (multi-method dispatch).

Each method is defined with a domain which specifies the applicability of the method to supplied arguments. A method is applicable to arguments if the class of each argument is a subclass of the corresponding domain class. A method with domain D_1 is said to be more specific than a method with domain D_2 if some domain class of D_1 are subclasses of the corresponding domain classes in D_2 and if all remaining classes in D_2 are not subclasses of the remaining corresponding classes in D_1 .

The process of selecting a list of applicable methods—sorted by specificity—for a generic function and supplied arguments is called *method lookup*. Method lookup followed by the application of the most specific method (i.e. the first element in the sorted method list) to the supplied arguments is called *method dispatch*.

The following example defines a generic function `element` to select the i -th element of an ordered collection (i.e. instance of class `<sequence>`). The simple function `foo` calls `element`; in turn `foo` itself is called with a vector, a string and a list object.

```
(defgeneric element (x i))

(defmethod element ((x <string>) (i <integer>))
  (string-ref x i))

(defmethod element ((x <vector>) (i <integer>))
  (vector-ref x i))

(defmethod element ((x <list>) (i <integer>))
  (if (= i 0)
      (car x)
      (element (cdr x) (- i 1))))

(defun foo (x)
  ...
  (element x 1)
  ...)

(foo #(a b c))
(foo "abc")
(foo '(a b c))
```

Since there is no possibility to distinguish in general between simple and generic function call sites, like in `(defun foo (x y) (x y))`, the function invocation protocol handles both cases. Simple functions and methods retain the lexical bindings in effect when created. A single lexical environment for the evaluation of variables, operators and operands is

assumed (i.e. Lisp-1).

Chapter 3

Virtual Machines and the Requirements of Dynamic Objects

As argued in the introduction, a major application domain for dynamic object-oriented programming is distributed computing and in particular the field of decentralized artificial intelligence. It is the affinity to this application domain, requiring code and thread mobility, which makes virtual machine code the most interesting program representation format for dynamic object-oriented programming today [Gos95].

Much attention has been devoted to object-oriented language implementations based on virtual machines in the context of Smalltalk, Lisp and recently Java. However, interpretation and the high-level performance model of dynamic object-oriented programming impose a significant efficiency overhead.

This chapter introduces the concept of virtual machines and identifies key problems, the cost of virtual instruction transfer and suboptimal dynamic memory economy. Furthermore, this chapter concerns the general purpose requirements of dynamic object-oriented languages that have to be met with a successful implementation. The general approach to language implementation taken within this thesis is finally delimited from other approaches.

3.1 Virtual Machines

A programming language is implemented on a hardware platform if a source program can be transformed into a semantically equivalent executable representation and an executor (or

evaluator) for this program representation exists¹. A classification of programming language implementations can be based on the kind of executable program representation [DVC90]:

Native Machine Code (NMC) Source code is transformed into semantically equivalent native machine instructions, a representation that is directly executable on a real machine.

Virtual Machine Code (VMC) Source code is transformed into semantically equivalent instructions of a virtual machine. An interpreter program that emulates the virtual machine executes the virtual machine instructions.

Source Code (SC) Source code is executed by an interpreter program directly without preceding transformation.

The transformation of a program unit from source code into executable code—for a real or virtual machine—is generally referred to as compilation; whereas the stepwise evaluation of code conforming to a semantics is called interpretation. Figure 3-1 illustrates the three principle implementation techniques.

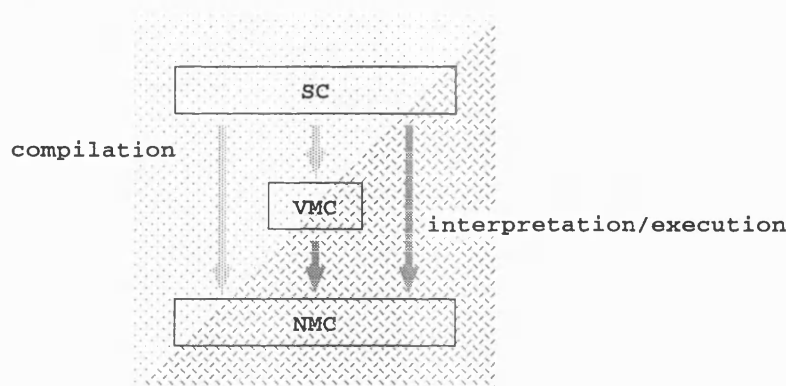


Figure 3-1: *Programming language implementation techniques*

Execution of native machine code and interpretation of source code can be regarded as special cases of interpretation of virtual machine code. In the first case, the virtual machine *is* the native machine and no emulation is required; in the second case, the virtual machine code *is* the source code and no compilation is required. However, virtual machine code is typically an abstract syntax tree or in a bytecode format (i.e. encoded in the the range [0..255]). This

¹The special case of direct execution architectures with hardware support for high-level interpretation is not considered.

work is not concerned with tree interpretation so that within the following, virtual machine instructions are thought of as bytecodes.

Another popular route to language implementation is via source code translation into the C programming language and subsequent compilation with a standard C compiler [Bar89, TLA92, DPS94, Att94, SW95]. Despite some differences, C code translation is here regarded as compilation into native machine instructions which simply happens in two steps (see also Section 6.4).

Also unaddressed in Figure 3-1 is the fact that virtual machine instructions can be further compiled into native machine instructions [DS84, HAKN97]. Again, such a transformation in two steps has advantages that are discussed later (Chapter 6) but here regarded as compilation into native machine code. The three different implementation techniques—referred to as native compilation, bytecode interpretation and direct interpretation respectively—offer different tradeoffs as discussed in the following three subsections.

3.1.1 Speed

Native machine code can be executed directly on a real machine as opposed to virtual machine code which requires mapping (i.e. loading, decoding and invoking) of each executed virtual machine instruction in the emulator program. Consequently, execution times of bytecoded applications are normally about ten times longer than with native machine code [Deu73, Ert95, DS96]. Instruction mapping is necessary with native code too but the processor hardware is able to perform the mapping much faster and in parallel with instruction execution. Direct interpretation imposes an even bigger run-time penalty, as lexical and syntax analysis are performed dynamically and the incremental form of evaluation excludes standard optimizations.

3.1.2 Space

The instruction set of a real machine is designed as an interface between the machine hardware and the software envisaged to run on the machine. With native compilation this predefined interface is in some cases not ideally suited as target code (e.g. to implement full continuations or closures). In contrast, the instruction set of a virtual machine is generally not predefined. The language implementor invents the virtual machine and can therefore design the instruction set specifically for the interpreted language, so that large savings in the space occupied by compiled code can be effected by a suitable designed instruction

set [Deu73, Heh76, Gre84, Pit87, EEF⁺97]. Some standard language operations (or even common operation sequences) are typically compiled into a single virtual machine instruction (e.g. `y=x+1` or `(car (cdr (cdr (cdr x))))`).

Such increase in the semantic content of virtual instructions reduces the frequency of instruction mapping. Since more time is spent in the native code which implements the virtual instructions, smaller code vectors not only lead to memory savings, but as well to execution speed up. The Reduce algebra system is reported to run twice as fast for a particular application by increasing semantic instruction content [Nor93]. However, the impact of bigger instructions on hardware cache performance is difficult to predict. On one hand, bigger virtual instructions can result in a higher hardware instruction cache miss ratio [Ert95]. On the other hand, data cache hits are more likely with smaller code vectors.

The segments of an executable file (e.g. in ELF) fall into two basic categories. The *text* segment contains all read-only memory, typically native code and constant data, whereas the *data* segment is dedicated to read/write data. Modern virtual memory systems support sharing of read-only memory pages with *shared objects*² (also known as shared libraries) in a general way [GLDW87]. Each process that uses a shared object usually generates a private memory copy of its entire data segment, as the data segment is mutable. The text segment, however, need to be loaded into main memory only once. An overriding goal when developing a shared object is to maximize the text segment and minimizing the data segment, thus optimizing the amount of code being shared [Sun93].

While compact in its explicit representation, virtual machine code is suboptimal with regard to sharing of dynamic memory. With the virtual machine approach, the bytecoded application file (image) is normally loaded at start-up. Although image files mainly contain read-only data, namely code vectors, their contents cannot easily be shared in memory by different processes executing the code vectors. Bytecode systems typically use one of the following choices to load an application image:

- reading the image file and dynamically allocating the relevant data structures to build the functional objects,
- memory mapping the image file or
- undumping a complete virtual machine process.

²Shared objects are not related to objects in object-oriented programming.

Reading the image file is clearly the most portable, but as well, a slow solution. A further drawback is that bytecodes cannot be shared in memory by different processes executing it. Undumping process images is difficult to port, problematic in the presence of shared libraries and in general too heavy-weight. Memory mapping as well is difficult to port to non-Unix platforms, but it can enable a quick start-up and sharing of mapped memory pages by different processes executing it. The benefits of virtual memory management can here lead to significant savings in the consumption of dynamic memory (i.e. small memory footprints) [GLDW87, Sun93].

3.1.3 Versatility

Beside performance and dynamic memory consumption, other characteristics have to be considered with the different executable program representations with regard to their general versatility.

Software Development The emulation in the bytecode interpreter can be of importance for interactive development environments. By controlling the state of computation in the virtual machine, support for debugging and inspection can be provided easily.

Portability The custom instruction set of virtual machines greatly simplifies the compilation process. The code generation phase of the bytecode compiler is portable since the peculiarities of different platforms (including the native instruction sets) are absorbed by the virtual machine emulator program. In comparison to native compilation, compiler complexity is further reduced by the fact that the target instruction set is tailored to the source language. However, direct interpretation is still the simplest way to implement a programming language. An interpreter can be directly inferred from a denotational definition of the language semantics [Cli84].

Architectural Neutrality Beside system portability, the portability of the executable program representation is an important feature of direct and bytecode interpretation. Portability of executable code is also known as architectural neutrality.

Infrastructures for distributed computing are now ubiquitous and ready to be used by a new generation of applications based on mobile code (e.g. applets, mobile software agents or multimedia control inside of set-top boxes). Architectural neutrality combined with code compactness of language-derived virtual machine instruction sets has major advantages for such applications. Code can be shipped unchanged to an heterogeneous

collection of machines and executed with identical semantics. Data transfer, however, requires marshalling in order to make the actual data representation transparent between different architectures.

Mobile code owes some of its success to the popularity of the World-Wide Web, and is referred to as applets, in the context of Java [GJS96]. Java supports implicit run-time linking of classes. If an application requires a class, which is not linked already, an exception is generated and handled by an appropriate class loader to retrieve the class from the file system or the network.

Code transfer with proprietary native machine code in heterogeneous environments can only be achieved when code is transformed into an architecture neutral distribution format (e.g. ANDF [BCD⁺91]) before shipping and transformed from the distribution format into the local native machine code after shipping. These transformations are non-trivial and increase shipping time considerably.

An extension to mobile code, required with mobile software agents, is thread migration. Here, not only program code is shipped in a heterogeneous network but also the execution state, in order to resume execution at different locations (i.e. machines).

The emulation of the virtual machine embraces the emulation of devices, e.g. printers and disks. This allows to control the mapping from virtual devices to real devices and helps to guarantee that transferred code does not abuse the local machine or compromise its security. Such security measures are more difficult to implement with transferred native code.

Interoperability The need for single high-level language environments is questioned by a growing public reservoir of mainly C-based libraries. Functionality, like persistence, distribution, concurrency, automatic memory management or graphical user interaction, can be used through application programming interfaces (API) of existing software packages on a wide range of platforms. Such functionality does not need to be included *per se* into high-level languages. In fact, it is often desirable to switch to a multi-lingual paradigm when developing an application. It should therefore be possible that applications are written in different, complementary languages, sharing data and (threads of) control. Such interoperability needs to be based on a simple and flexible foreign function interface which is relatively straightforward with compilation into native machine code where foreign addresses can be directly embedded.

Interoperability in the virtual machine approach, however, means a bytecoded function can be invoked from another language and likewise, an external function is callable from within a bytecode vector. The separation between execution on the bytecode level and execution on the native machine code level establishes a proprietary use of data and control. A foreign address cannot directly be embedded into virtual machine code. Furthermore, the state of computation in the virtual machine is defined in other terms (i.e. *virtual* registers, *virtual* stack, *virtual* program counter etc.) than the state of computation on the hardware level, complicating the transparency of multiple threads of control.

Although a virtual machine emulator program can be extended to embrace foreign code, this is not a practical way to interoperability. In general, it cannot be anticipated which foreign functions will be required by a potential application at the time the virtual machine is installed. In any case, extending the virtual machine emulator program statically is not practical since the virtual machine may be used by different processes and should therefore not be customized to an individual application.

Some systems (e.g. Self [ABC⁺96], Java [Wil97]) use the native run-time linker to extend the virtual machine dynamically with code from shared objects. This technique, however, can be difficult with legacy packages which are not compiled as shared objects.

The versatility of virtual machine code is the key reason for a large number of bytecode interpreted language implementations (e.g. Pascal [NAJ⁺91], Smalltalk [Kra83, DS84], Oaklisp [PL91], Scheme 48 [KR94] and recently, Java [Gos95, LY96]).

3.2 Requirements of Dynamic Objects

A successful language system has to satisfy certain requirements in order to foster the distinctive features of the implemented language and as well to make them practical. The following four requirements are focussed upon within this work:

1. performance,
2. static and dynamic memory efficiency,
3. system and code portability and
4. interoperability.

The previous section showed that by choosing bytecode interpretation as the underlying execution model, portability and static memory efficiency (i.e. compact code) is achieved. However, the remaining requirements cannot be easily satisfied. In fact, the following subsections show that the dynamic object-oriented approach introduces efficiency problems itself.

3.2.1 The Performance Cost of Execution Dependencies

The existence of code that uniformly works for objects of a range of (sub)classes (i.e. inclusion polymorphism [CW85]) in object-oriented languages makes it generally impossible to determine statically which method will be used at a generic function call site. Exact argument classes cannot be determined until run-time because an object x can appear in place of an object y if x is an instance of a subclass of the class of y . The binding of generic function call sites to the most specific applicable methods (i.e. the method lookup) is therefore dynamic.

With dynamic method binding, effective optimizations, notably inlining, dead-code elimination and constant propagation, cannot be performed. Code fragmentation, due to encapsulation in object-oriented languages, results in high function invocation frequency and further aggravates the impact of omitted optimizations [HCU91].

These problems apply to statically-typed and dynamically-typed object-oriented languages equally. Statically-typed languages however, use static type information to compile the method binding process (i.e. virtual function invocation in C++) as a dynamic method table lookup. Each instance carries a pointer to a class-specific method table. Using the offset, that is assigned to each method name statically, the most specific method for a given object can be found in constant time.

The dynamic features typical for dynamic object-oriented languages require a more flexible technique to speed up the method lookup process. Chapter 2 introduced object-oriented languages with classes, methods and generic functions as first-class objects that are used as building blocks of the object system itself. By specializing the metaobjects of an object system, the semantics of the object system (e.g. inheritance) can be customized for an application. Typically, metaobject protocols allow to add, change and remove classes, methods and generic functions at run-time.

A flexible and successful technique to speed up the method lookup in dynamic object-oriented languages is memoization, also known as caching [Mic68, KS86]. This technique is based on the fact that results of side-effect-free functions can be saved and reused to bypass subsequent function applications with identical arguments. Memoization can be applied

to the method lookup function simply by saving the most specific method which has been computed for a generic function and the provided argument types. Such memoization of frequently used methods reduces the number of full method lookups during run-time and therefore can speed up generic function invocation significantly. Subsequent calls to the generic function with arguments of same classes can then reuse the cached method.

The Smalltalk definition [GR83] suggests a vector as a method cache with four consecutive locations for each method entry:

```
initializeMethodCache
  methodCacheSize <- 1024.
  methodCache <- Array new: methodCacheSize

findNewMethodInClass: class
  | hash |
  hash <- (((messageSelector bitAnd: class) bitAnd: 16rFF) bitShift: 2) + 1.
  (((methodCache at: hash) = messageSelector)
   and: [(methodCache at: hash + 1) = class])
  ifTrue: [newMethod <- methodCache at: hash + 2.
           primitiveIndex <- methodCache at: hash + 3]
  ifFalse: [self lookupMethodInClass: class.
            methodCache at: hash put: messageSelector.
            methodCache at: hash + 1 put: class.
            methodCache at: hash + 2 put: newMethod.
            methodCache at: hash + 3 put: primitiveIndex]
```

The `findNewMethodInClass` routine attempts to retrieve an entry for `messageSelector` and `class` in the `methodCache`. If an entry is available and the class value of the entry is identical with `class`, the compiled method is restored and assigned to `newMethod`. Otherwise, `lookupMethodInClass` performs a full method lookup and the cache is updated with result. The hash index is computed with a `bitAnd` operation on the selector and the class object pointers and a second `bitAnd` to map the index into the range of the cache size³.

The CLOS specification [KdRB91] states that `compute-discriminating-function` is used to create the discriminating function of a generic function. When a generic function is invoked, its discriminating function is then used to determine and call the effective method for the provided arguments. Like message sending can be optimized with method caching based on message selectors and receiver classes, the speed of generic function invocation can be increased by method caching based on generic functions and argument classes. Kiczales *et al* [KdRB91] suggest the following method caching scheme for CLOS:

```
(defmethod compute-discriminating-function ((gf standard-generic-function))
  (let ((cache (make-hash-table :test #'equal)))
    #'(lambda (&rest args)
        (let* ((cls (mapcar #'class-of (required-portion-of args))))
```

³For cache sizes of 2^n , the modulo operator can be replaced by the faster bitwise and-operator (i.e. `bitAnd`).

```
(fun (gethash cls cache nil)))
(if fun
  (funcall fun args)
  (let* ((meths (compute-applicable-methods-using-classes gf cls))
        (fun (compute-effective-method-function gf meths)))
    (setf (gethash cls cache) fun)
    (funcall fun args))))))
```

Two differences between the CLOS and Smalltalk approach are obvious: With generic functions, methods can be cached in a distributed fashion at their corresponding generic functions (or as above in the environment of the associated discriminating functions) rather than globally. Cache hit rates are thus expected to be better and the method cache can be updated more easily after, for instance, a method has been removed dynamically. On the other hand, several argument classes generally have to be considered with multi-methods to retrieve a stored method which makes the hash function more expensive.

Improvements can be achieved with variations of the two techniques above [CPL83, FS83]. However, two quite different approaches are method caching with static method tables and inline method caching.

Static method tables are initialized statically so that the dynamic method lookup can be realized in constant time by indexing into the table. Virtual function tables, used in Simula and C++ are static method tables that can be kept within class scopes due to static type information available in these languages (as pointed out earlier in this subsection).

With message sending, static method tables store methods globally. Typically a two-dimensional array is indexed with class and selector codes. Global method tables can become very large and sparsely filled [AR92]. Various techniques [AR92, Dri93, DH95] help to compress static method tables. Unfortunately, a dynamic memory overhead exists if applications have temporal hot-spots. Furthermore, new class or method definitions can cause time consuming re-compression of the method tables. More complications arise with table indexing for multi-methods [AGS94].

Argument types at a generic function call site change rarely. This means, even though functions are defined polymorphic, argument types of function applications within the defined body are constant in about 90% of the cases. This spatial locality of type usage can be observed in Smalltalk [DS84] and Self [HCU91]. Caching of previously used methods directly at a generic function call site in order to exploit this type locality is called inline method caching. This approach is much more appropriate for be applied with dynamic objects.

As stated earlier in Section 3.1.3, interoperability emerges with the trend to a multi-lingual development paradigm. A further argument for co-habitation between higher-level and lower-

level languages is based on performance reasons. The opportunity to have time critical parts of a higher-level language application coded in a lower-level language with tight control over hardware devices, can help to overcome performance drawbacks without switching entirely to a less flexible lower-level language.

3.2.2 The Memory Cost of Execution Dependencies

Traditionally, dynamic object-oriented languages have emphasized incremental software development and rapid prototyping with the underlying idea that programming equals customization and extension of an interactive programming environment [BSS84]. The implication is a blending of compile-time and run-time as well as the loss of a small core language. Both aspects make the delivery of small applications with modest memory requirements difficult. The ability to discard or change the source code definitions of classes and methods by means of a metaobject protocol, as well as the general potential for run-time evaluation, force most applications to carry around much more code than actually needed [Shr96].

Further memory is required to realize reflection with classes, methods and generic functions. In CLOS, a class object stores information about how to allocate and initialize instances; a method object has links to generic functions, to the actual function object defining the method behaviour and to the classes defining its domain; finally, a generic function refers to the associated method objects and to the discriminating function in order to handle the full method lookup.

3.2.3 Scripting

Scripting is a way to control and combine various applications and operating system features by means of a relatively simple, interpreted programming language—a so-called scripting language. Some popular scripting languages are Unix's `sh`, Perl [WS91] and Tcl [Ous94]. Scripting is included in this section because it raises in a practical context some of the requirements that should be met with an implementation of an dynamic object-oriented language.

Typically, scripting commands can be interactively entered into a shell or stored in a file that can be passed to the interpreter of the scripting language. A third alternative provides the interpreter as a library with a defined application programming interface (API). The scripting language can then be used inside an application, either for special purpose computations (for which the scripting language might be more suitable than the implementation

language of the application) or for the purpose of an extension language. Extension languages can raise the power of an application significantly by giving the user a means to extend a software tool. For instance, Lisp is used as extension language with Emacs and AutoCAD.

The utility of a scripting language can be increased by a tight coupling between computation within the scripting language and computation external to the scripting language. Input/output redirection as well as sharing data types are important aspects of such tight coupling. In order to start-up an external application from the operating system level with parameters, a scripting language requires at least the notion of character strings. However more desirable is support to use the API of the external application on the implementation language level. In this case, the scripting language needs to share not only strings with the external application, but also other primitive data types (like numbers, characters) and possibly pointers to handle arbitrary compound data structures. A further step towards tight coupling between the scripting and external language comes with the possibility of controlling the input and output of external application from within the scripting language (e.g. in Unix by redirecting `stdin` and `stdout`).

For many reasons (object-oriented) dynamic languages could be regarded as ideal scripting languages. Dynamic typing, automatic memory management and extensibility is provided with dynamic languages. However, large application sizes, long start-up times and the gap between the programming language and the operating system makes OODLs impractical for scripting purposes. Although Perl, scsh [Shi97] and to some extent Tcl have improved over standard operating system shells, programs written in these systems do not scale well and are not well-suited for production work. If problems with interoperability, start-up and memory usage were overcome, scripting would benefit much from the scalable and well defined features of dynamic object-oriented languages.

3.3 Conclusion

This chapter showed that compilation into native machine code and direct or tree interpretation offer different tradeoffs. Both implementation techniques are not optimal with regard to a compound measure including the size, speed and versatility of the corresponding executable program representation. A much better overall value can be achieved with bytecode interpretation. Source code is here transformed into semantically equivalent instructions (bytecodes) of a virtual machine. An interpreter program that emulates the virtual machine, executes the virtual machine instructions.

The virtual machine approach is at the heart of the implementation architecture developed in this thesis. While fruitful for code/system portability and static memory usage, bytecode interpretation impairs performance, interoperability and dynamic memory usage. In the next chapter an architecture is described which specifically addresses these problematic issues.

Part II

Chapter 4

The Architecture

In this chapter, a novel virtual machine architecture for object-oriented dynamic programming languages is described. The architecture differs from other approaches by using the following techniques developed in the context of this thesis:

- virtual machine code represented as constant C vectors,
- virtual instruction transfer with indexed code threading,
- native code clustering with optimal virtual instruction ordering in the emulator and
- method lookup using quasi-inline method caching.

A further integral part of the architecture is the use of a conservative memory management system [BW88, Bar88].

As explained in Chapter 3, the contributions evolved with the architecture are particularly driven by the problematic issues of performance, dynamic memory consumption and interoperability in the context of bytecode interpretation and dynamic objects. This chapter is dedicated to the description of these key contributions and their theoretical benefits. Chapter 5 follows with a description of the realization of the architecture and empirical results to support the cases made within this chapter.

4.1 Embedding Virtual Machine Instructions

The architecture proposed here differs from the typical virtual machine approach by representing virtual machine code as constant C arrays, i.e. `const long cv[] = { ... }`. Such C embedded virtual machine code has several positive implications:

Stand-Alone Executables Bytecoded modules hosted by `.c` files are compiled with a standard C compiler and linked with the run-time support code (including the virtual machine emulator) to form a stand-alone executable program. The execution of the program does not require a separate bytecode image file.

Separate Compilation Bytecoded sources are compiled separately into object files. Object files can be collected and managed in native libraries side-by-side with compiled foreign code. Such transparency is useful for embedding applications.

Interoperability Foreign addresses can be used directly from within the actual compilation unit (e.g. from a binding vector linked to a closure). Neither the virtual machine, nor the run-time support code, need to be extended for interoperability with foreign code (see Section 3.1.3).

Shareability Virtual machine code is located in sharable text segments of the final executable file since it is defined constant and does not contain any addresses in the raw bytecode vector¹. Code vectors can thus be shared by all processes executing the application, resulting in small memory footprints. Important savings are achieved when modules are compiled into shared objects. In this case, also processes of different applications share virtual machine code in memory (see Section 3.1.3).

Automatic Memory Management A further advantage with C embedded virtual machine code is that bytecode vectors are not heap-allocated and therefore not considered (traced or copied) with garbage collection.

Start-Up A fast application start-up can be crucial for some applications, for instance with scripting (see Section 3.2.3). C embedded code vectors represent a better alternative to typical start-up solutions, like reading a bytecode file, explicitly memory mapping a bytecode file or undumping an entire virtual machine process (see Section 3.1.2), because it is fast, being based on memory mapping, as portable as C and memory efficient on platforms supporting shared objects.

The representation of virtual machine code on the C language level does not compromise on the architectural neutrality of the executable program representation. Although, repre-

¹Virtual machine code that does include absolute addresses can be declared constant in C as well, for instance like `const long cv[] = {0x2e4a561f0, (long)&tab+34, ...};`. However, with standard linking, such code vectors end up in non-sharable read/write data segments.

sented in C, the actual executable program representation format is still virtual machine code.

The performance of bytecoded programs will always remain a concern when compared with native machine code. Virtual machine code requires mapping (i.e. loading, decoding and invoking) of each executed virtual machine instruction in software and is therefore considerable slower than native machine code when executed (see Section 3.1). Two measures are typically taken to accelerate virtual machines: increasing the semantic content of instructions and code threaded virtual instruction transfer.

While effective, increasing the semantic content is not further explored here since much work has been done in this field (see Section 3.1.2)². Instead, (i) a new portable approach to code threading (i.e. *indexed* code threading) and (ii) the possibility of better hardware cache coherency by optimizing the physical ordering of virtual instruction code in memory is investigated in depth. It is pointed out in Section 3 that system portability is an important requirement for language implementations. To establish portability of the virtual machine emulator and in order to take C embedded bytecodes into account, the widely available and optimized C programming language is assumed for its implementation.

4.2 Indexed Code Threading

A direct and reasonable efficient way to implement a virtual machine is to use a switch-statement. The actual instruction in the code vector (*cv*) pointed to by the program counter (*pc*) is used to branch to the instruction code. The following C-like code illustrates this technique:

```
char cv[] = {47, 11, ...};
char *pc = cv;

while (1) {
    switch (*pc) {
        case 0: ... pc++; break;
        case 1: ... pc++; break;
        ...
        case 255: ... pc++; break;
    }
}
```

Although standard C compilers compile dense switch-statements into jump tables, an overhead is involved with a table range test³, the table lookup as well as the jump to the

²It is simply assumed that this issue is considered when applying the architecture.

³A range test can be avoided for some compilers if unsigned char is used.

instruction code and the loop jump (see Figures A-1, A-4 and A-7⁴). This overhead can be reduced by a technique called code threading [Bel73]. Instructions are here no longer encoded as values in the range [0..255], but as addresses of the corresponding instructions. The result is that instructions are “threaded together like beads on a chain” [Kli81]. The interpreter skeleton shows that no loop around a switch-table is necessary since the dereferenced program counter can be used directly to jump to the next instruction:

```
void *cv[] = {&&instr47, &&instr11, ...};
void **pc = cv;

instr1: ... goto *(pc++);
instr2: ... goto *(pc++);
...
instr255: ... goto *(pc++);

goto **pc;
```

Code threading can help to reduce native code cycles necessary for virtual instruction dispatch 3–4 times [Ert95] with an involved actual performance increase of up to 30% compared to the classical switching technique [PK98]. The assembler code generated for the code segments (see Figures A-2, A-5 and A-8) supports these measurements.

However three drawbacks are involved with code threading. First, common four-byte memory addressing leads to four times larger code vectors. Second, code vectors must be declared in the same scope as the instruction labels and hence, not only represented in the implementation language of the virtual machine but also compiled at the same time. Finally, only few (fast and portable) languages implementations support first-class labels on which the performance increase is based⁵. If function addresses were used instead of label addresses, the function invocation overhead (in particular if not tail-call optimized) and the fact that virtual machine registers have to be defined globally would not be acceptable either [Ert95].

In order to resolve some of the drawbacks accompanying code threading, *indexed code threading* is proposed as part of the architecture. Here, instructions are again encoded in the range of [0..255]. However the instruction codes are used to access the instruction labels from a label table (`labels`) which is computed at link time:

```
void *labels[] = {&&instr1, &&instr2, ...};
char cv[] = {47, 11, ...};
char *pc = cv;

instr1: ... goto *labels[*(++pc)];
instr2: ... goto *labels[*(++pc)];
...
```

⁴The GNU C compiler gcc version 2.7.2.1 (P5) and version 2.7.2 (MIPS, SPARC) is used.

⁵Clearly, assembler could be used to realize code code threading in a machine dependent way.

```
instr255: ... goto *labels[*(++pc)];  
goto *labels[*pc];
```

Due to an extra level of indirection, the transfer between virtual machine instructions is slower than with code threading, but still better than with a jump table (see Figures A-3, A-6 and A-9). It is notable that the CISC code for the P5 processor needs three native instructions with both standard and indexed code threading so that performance should be similar.

Beside the performance improvement over switching, the big advantage of indexed threading is that the executable program representation (i.e. code vectors) is suitable with both, a virtual machine emulator based on switching or indexed threading. For instance, for an emulator which is implemented in the C programming language this means, programs can be transformed into bytecode vectors regardless of whether the virtual machine will be based on switching or threading. In fact, by using a compiler flag to alter between switching and indexed threading when compiling the virtual machine, the portability of the emulator is independent of the availability of a C compiler with first-class labels (e.g. gcc [Sta92]).

4.3 Optimal Instruction Ordering

Typically, programs only access a relatively small portion of the available address space at any fraction of time. Memory hierarchies in today's hardware architectures take advantage of such spatial and temporal locality of data and code references in order to bridge the widening gap between processor speed and memory access time.

Clustering native code in the virtual machine emulator which is likely to be executed consecutively, increases the chances that the virtual instruction code is already in the native instruction cache. An approximation of such a clustering can be derived from profiling the invocation frequency of virtual instructions during the execution of an application. The native code of the virtual instruction called most will then be next in memory to the native code of the virtual instruction called second most etc. Particularly, for the 50 virtual instructions that typically dominate applications, such an ordering results in much better native instruction cache performance than the typical ad-hoc ordering.

Such physical ordering of code in memory is applicable with C threaded code because the actual source code organization will be reflected in the machine program and the (literally) "threaded" instruction transfer leads to good locality with native code in execution. The

situation is different with C switched code. The switch branches retain their order in the machine program too, but the jump table range check and the additional loop jump cause the native instruction pointer to cover much more memory than with threaded code (see tables in Appendix A). Optimal instruction ordering with a switched interpreter is therefore not expected to show the same performance increase as with a threaded interpreter.

The practical problem arises now with the fact that the virtual machine cannot be re-compiled for each application but should be provided as a shared object. Consequently, an ordering cannot be *customized* to address a specific execution profile. An acceptable solution is here to average instruction frequencies of representative applications in order to derive a (so called) *optimal* ordering. It turns out that the increase in the native instruction cache hit ratio with optimal ordering is nearly as good as with the custom ordering, since most virtual instructions are in fact application independent, being concerned with the operation of the virtual machine itself (i.e. access of virtual registers and stack values).

A tool has been developed within the context of this thesis to aid deriving optimal virtual instruction orderings. Section 5 introduces this tool and presents figures which show a speed improvements of 21% and 15% for the MIPS and P5 architectures respectively.

As hardware caches continue to get larger there is an argument that at some point the entire virtual machine emulator will fit into the instruction cache. The effect of virtual instruction clustering on instruction cache performance would then indeed become neglectable. However, as long the tradeoff between memory access time and its price exists, new levels in the memory hierarchy will emerge as soon as current level 1 cache technology becomes affordable in larger scale. With this tradeoff very likely to continue in the future, there will continue to be a case for optimal ordering of virtual instruction code as described.

4.4 Quasi-Inline Method Caching

Up to this point, the architecture is not restricted or specifically designed for dynamic objects. And although the architecture is concerned with reducing efficiency overheads and with enabling language interoperability in the context of bytecode interpretation, there is still the high cost of dynamic method lookup that has to be addressed explicitly.

One of the most successful techniques to speed up dynamic method lookup is method inline caching [DS84, HCU91] as described in Section 3.2.1. Unfortunately, this technique requires writable compiled code in order to cache methods at call sites locally. Mutable code, however, stands in direct conflict to the approach of sharable virtual machine code which has

to be defined constant in order to be placed in text segments. Quasi-inline method caching is a technique to make polymorphic inline caching [HCU91] applicable with read-only virtual machine code.

An important obstacle with inline caching is the difficulty of cache flushing when cache entries may become invalid, e.g. after dynamic method removal. Scanning the entire code of an application to invalidate inline addresses is not acceptable. The problem is solved in Smalltalk by using the mapping tables between virtual machine code and native machine code which are involved with the two executable program representation formats [DS84]. Without native code generation, inline method caching is useless in the context of dynamic languages.

In an attempt to simplify the problem of cache flushing and keep virtual machine code in sharable text segments, it is now proposed to hold the method cache at each generic function object (as showed in Section 3.2.1), however, instead of the actual domain, the virtual machine program counter is used to map into the cache. By using the program counter, type locality is exploited in the same way as with classical polymorphic inline caching. Since the cache is not really inlined, the scheme is called *quasi-inline* method caching.

Two more advantages that come with quasi-inline method caching are worth mentioning. Firstly, computing a hash index from a program counter can be realized much faster than computing a hash index from the argument classes that are considered with a generic function invocation⁶. Quasi-inline method caching is therefore suited for single- and multi-method dispatch equally. Secondly, with a linear search hashing policy⁷ on collision, redundancy with classical inline caching can be avoided. Suppose, there are n generic function call sites that are all used with instances of identical classes. Classical inline caching requires n entries, one for each call site. In comparison, in the worst case quasi-inline method caching fills all entries of the generic function cache, i.e for an initial cache size of four there is a maximum of four entries for the 100 call sites.

An implementation of an object-oriented dynamic language augmented with quasi-inline method caching is in no way restricted in its flexibility and is very likely to benefit in performance as cache misses can be reduced to 1.06% (see Section 5). The scheme saves dynamic memory by avoiding redundant method entries and by being adaptable to generic function

⁶Table B.2 shows empirical evidence that multi-argument dispatch is not a rarely used feature with multi-methods. Over 68% of all generic function calls in the OPS5 system implemented in EuLisp [OP93] discriminate on more than one argument.

⁷For tables with a default size of four entries, more sophisticated collision handling is not necessary. Tables are flushed when over 90% filled.

invocation which typically comes in waves with hot-spots on particular methods. A fixed size hashing scheme associated with quasi-inline caching can here adapt to such temporal and spatial locality as dynamic method lookup has to face.

4.5 EuLisp

Assuming a “sufficiently smart compiler” [SG93] the transformation of object-oriented dynamic source code into an efficient executable representation is in many cases actually possible—despite the high-level performance model. However, for complexity reasons the development and maintenance of such a compiler is difficult.

The goal of recent standardization efforts with object-based Lisp dialects was therefore to define a clean, commercial-quality Lisp dialect which would not be bound to former Lisp tradition simply out of backward compatibility reasons but which would foster the original key idea of the List Processor in co-habitation with a dynamic object system [PCC⁺86].

Three of these languages are EuLisp [PNB93, PE92], ISLisp [Int97] and Dylan [Sha96]. A feature common to these languages is a clear separation between the core language and language extensions (e.g. the development environment) as well as a clear separation between compile-time and run-time of a program. The rest of this section is dedicated to EuLisp, the language of choice throughout this thesis.

The distinguishing features of EuLisp are (i) the integration of the classical Lisp type system into a class hierarchy, (ii) the complementary abstraction facilities provided by the class and the module mechanism and (iii) support for concurrent execution (multi-threading). The object system is accompanied in level-1 of the language definition by a metaobject protocol (MOP) to enable reflective programming similar to CLOS (see Chapter 2.3). The EuLisp MOP however, provides a better balance between the conflicting demands of efficiency, simplicity and extensibility [BKDP93].

The EuLisp language definition breaks with some Lisp traditions which were not beneficial for efficient language implementations. The introduction of strict module interfaces leads to more opportunities for compile-time optimizations. And the separation between a small core language and supporting libraries particularly helps to overcome the problem of extracting a program from an interactive development environment. Applications are no longer required to carry around a lot more code than actually needed. However, if necessary, runtime evaluation can be supplied with a corresponding module library so that an efficiency penalty has to be accepted only for functionality actually used.

Although EuLisp (or a language with similar characteristics) is from this point on regarded as integral part of the architecture, this work does not investigate the relationship between efficiency and the EuLisp language design. The architecture described here is applicable in principle to any object-oriented dynamic language and its success is not bound to the EuLisp language design.

4.6 Conservative Garbage Collection

Common to all garbage collection techniques is the need to recognize pointers within allocated memory during run-time of a program. Traditionally, compilers cooperate with automatic memory management systems and provide information about the layout and/or location of pointers. However, such cooperative data representations complicate foreign-function interfaces and thus interoperability, since foreign pointers that do not follow the tagging/boxing scheme cannot be handled by the garbage collector.

Conservative pointer finding techniques [BW88, Bar88] emerged with the desire to add garbage collection to the C programming language. Since C is statically typed and has therefore no need for run-time type information, automatic deallocation introduces here the problem of identifying pointers in a *conservative* environment. Garbage collection based on conservative pointer finding treats aligned bit patterns within a certain range as pointers. With this technique, integer values may in some cases mistaken as a pointer. However, empirical evidence shows that potential memory leaks, which may result from integer values accidentally classified as pointers, are very rare [Boe93]. In any case, such an ambiguous pointer may not be relocated by the collector because a relocation would change its value so that conservative pointer finding can only be applied with a *non-relocating* garbage collection algorithm.

A non-relocating garbage collection scheme has major advantages when pointers are passed to a program part written in other language. The external program part is free to store the pointer for later reuse, even when the garbage collector has gained control in between. Object pointers thus don't need to be protected against deallocation and relocation if passed outside the language implementation territory.

By adding a conservative collection scheme to the virtual machine architecture, the important aspect of interoperability is addressed, internal and external pointers can be mixed without precaution. The fact that garbage collection can be easily added to a system by replacing the default allocation interface (i.e. mainly `malloc()`) is responsible for the pop-

ularity of one particular conservative memory management system written by Boehm and Weiser [BW88, Boe93]. Its popularity led to robustness and availability on many platforms (even in the presence of pre-emptive multi-threading). The issue of multiple threads of control in conjunction with interoperability is addressed in Section 5.5.

4.7 Conclusion

System portability, architectural neutrality of executable code, language interoperability, performance and memory efficiency are important requirements to implementations of object-oriented dynamic languages partly derived from the typical application domains and partly linked to the deficiencies of these languages (see Chapter 3). Figure 4-1 illustrates the relationship between the key characteristics of the architecture proposed in this chapter and these requirements.

It was argued earlier, that with an emphasis on run-time dependencies, object-oriented dynamic languages have a natural potential to handle dynamism in distributed systems. To reinforce this key advantage the architecture is based on a virtual machine approach which provides code compactness and architectural neutrality, crucial features for code mobility in distributed systems. As a side effect, the virtual machine approach results in portability of code generation since different hardware platforms do not need to be considered.

Virtual machine code is slow compared to native machine code. This penalty can be alleviated with code threading and optimal instruction ordering. The high cost of dynamic method dispatch is reduced with quasi-inline method caching.

By using a conservative garbage collection scheme and virtual machine code as static C data, applications becomes easily interoperable with other programming languages. Furthermore, constant C embedded code vectors can be located in sharable text segments leading to modest memory footprints.

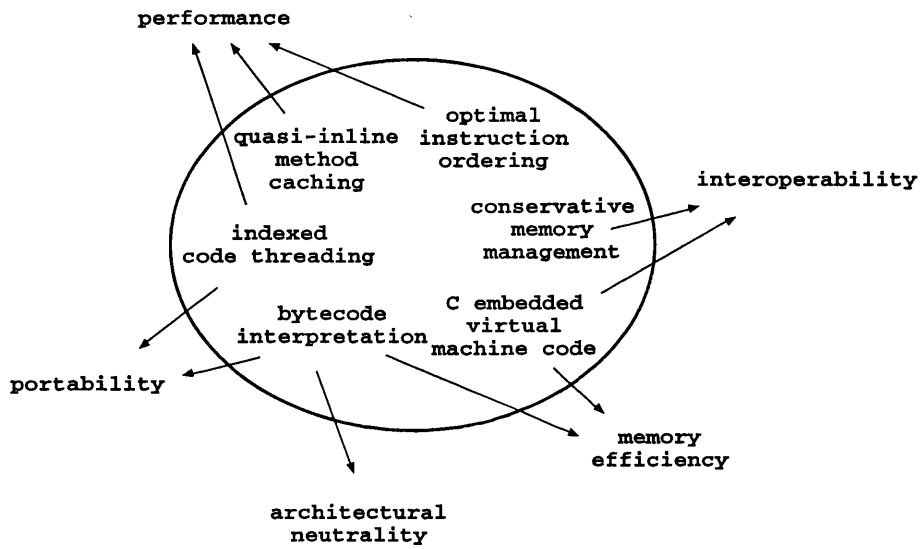


Figure 4-1: *Design and requirements*

Chapter 5

The Mechanics of Dynamic Objects in `youtoo`

The architecture proposed in Chapter 4 is applied in `youtoo`¹, an implementation of the dynamic object-oriented programming language EuLisp. EuLisp is a single-valued Lisp dialect with an integrated object system, a defined metaobject protocol, a module system and a simple light-weight process mechanism.

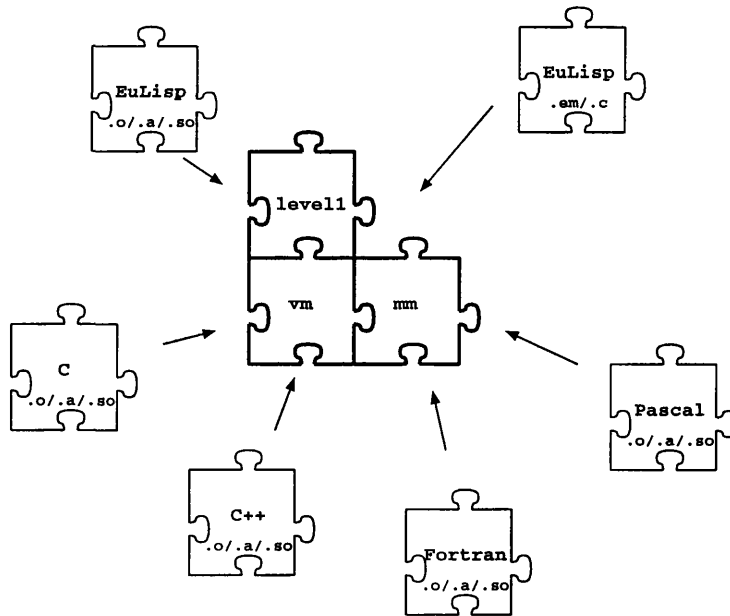
This chapter presents `youtoo` and demonstrates the feasibility and effectiveness of the proposed ideas, namely C embedded virtual machine code, indexed code threading, optimal virtual instruction ordering and quasi-inline method caching.

5.1 The `youtoo` Compiler

In `youtoo`, the technique of embedding virtual machine instructions in C leads to a correspondence between EuLisp modules and C files (`.c/.h`). Compiled modules can be collected in a library (`.a/.so`) or immediately linked with the virtual machine (`vm`), the conservative memory management package² (`mm`) and the EuLisp standard language library (`level11`) into an executable file (see Figure 5-1). If necessary, additional foreign code (C, C++, Pascal, Fortran) can be linked in form of libraries or object files. Dynamic linking (described in detail in Section 5.1.1) on the EuLisp-level allows the addition of new modules at run-time.

¹The name derives from the fact that `youtoo` is the second EuLisp reference implementation. The system is publicly available from <ftp://ftp.maths.bath.ac.uk/pub/eulisp/youtoo>.

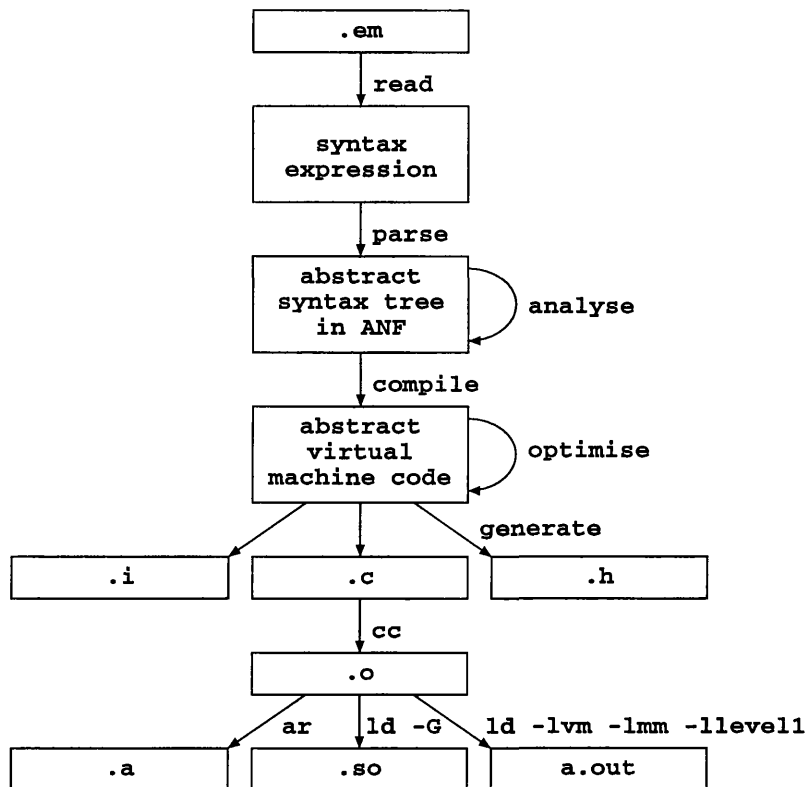
²A conservative mark-and-sweep garbage collector implemented by Boehm and Weiser [BW88] is used.

Figure 5-1: *Compiling an application*

The `youtoo` compiler (see Figure 5-2) is a EuLisp program which takes a module name, reads the source code of the module and generates an abstract syntax tree (AST) in A-normal form (ANF). ANF is an intermediate representation that captures the essence of continuation-passing style (CPS) including the reductions normally followed after standard CPS transformation [FSDF93]. With the exception of tail-call optimization, `youtoo` does not exploit the optimization opportunities that emerge after translation into ANF. As noted earlier, the work presented here is not concerned with typical compile-time optimizations.

After some analysis on the AST, the compiler generates abstract virtual machine code for the functions defined in the module and finally generates a C file (`.c`) and two interface files (`.i` and `.h`).

The C file effectively defines virtual machine code as constant C vectors local to the unit of C compilation (i.e. `static const long cv[] = {...}`) and a module initialization function to initialize the module on the C-level. The initialization includes the final initialization of statically defined Lisp literals. Initialization on the Lisp-level, i.e. execution of the top-level forms of each module starts after all modules are initialized on the C-level. A module binding vector is defined that holds defined module binding values and literals. The generated interface files map binding names of defined bindings in the module to offsets in the module binding vector. The Lisp-level interface file (`.i`) is used for separate compilation and dynamic linking of modules; the C level interface file (`.h`) is not only used for separate compilation

Figure 5-2: *The youtoo compiler*

but also with foreign in-calls.

Foreign function declarations in the source module result in a C stub function in the `.c` file to handle argument and result conversion between Lisp and C as specified in the foreign function declaration. A pointer to the C stub function is stored in the module binding vector and is used by a specific virtual machine instruction to invoke the foreign function. Converters are provided for basic data types and for handling addresses. If the requested C function is not linked to an application by default, it can be passed to the compiler as a parameter. See Table B.5 for a list of available foreign-function converters.

Some static Lisp objects can be represented as static C values and thus be truly statically allocated. This is the case for primitive values like characters, integers, doubles, strings, the empty list as well as for composed values like vectors and lists that exclusively contain values which again can be represented as static C data. If an element of a static (on the Lisp-level) vector or list cannot be represented as static C data, at least the empty data structure is statically allocated (on the C-level) and filled in during C-level initialization of the module. Symbol and keyword literals cannot be defined constant as they need to be interned, i.e. included into or accessed from the symbol/keyword table depending on the

module topology in an application.

5.1.1 Dynamic Linking

EuLisp is syntactically extendible with syntactic transformation functions (macros) which may be applied during compile-time. Typically, the user can add new application-relevant macros to the standard set of macros. Macros are defined in full EuLisp and potentially use defined functions and other defined macros. EuLisp's strict separation between compile-time and run-time enforces the restriction that no module can appear in the transitive closure of its own compilation environment [DPS94]. This requirement induces the differentiation between syntax and lexical modules and frees a compiler to resolve compilation/execution dependencies within a compilation unit, in this case, a module. If a syntax module `m1` defines macros which are used in another module `m2` then `m1` has to be compiled prior to `m2`. In this way a compiler can avoid using either multiple compilation cycles or having an interpreter as a constituent. Instead a compiler can be incrementally extended by dynamically linking the required previously compiled syntax modules.

Dynamic loading of code is implemented in `youtoo` by scanning the C file to find the relevant offsets to the start of the array definitions and reading the contents of the arrays. This process is assisted by layout information hidden as C comments generated by the compiler. Macro expansion with syntax modules is based on such Lisp-level dynamic linking.

5.1.2 Source Code Interpretation

The `youtoo` system does not provide a direct interpreter (as described in Section 3.1) for EuLisp. Instead, there is a read-compile-execute-print-loop in place of the read-eval-print-loop normally provided with direct interpretation. Each expression to be interpreted is compiled into virtual machine code (using the same transformation as in the compiler) and then immediately executed on the virtual machine emulator (see Figure 5-3). After `youtoo` is bootstrapped, i.e. compiled into a stand-alone application by itself, the virtual machine emulator is linked to it anyway so that the execution of the compiled code in the interpreter is straightforward. In fact, interpreter and compiler are the same program called `youtoo`. If a file name is presented to `youtoo`, like in

```
youtoo test.em -l level1
```

the corresponding module will be compiled and linked with the (shared) run-time libraries and perhaps with other imported compiled modules into an executable file `test`. Invoking

youtoo without any parameter (or parameter *-i*)³ starts up the pseudo read-eval-print loop and the following interaction can be envisaged:

```
EuLisp System 'youtoo 0.94'

[user]: (+ 1 2)
- 3
[user]: (defun fact (x) (if (< x 2) 1 (* x (fact (- x 1)))))
- #<simple-function: fact>
[user]: (fact 100)
- 933262154439441526816992388562667004907159682643816214685929638952175999
93229915608941463976156518286253697920827223758251185210916864000000000000
000000000000
[user]:
```

Evaluation of expressions and creation of new global bindings is performed in the lexical and syntax environments of the module named in the prompt. By default this is the artificial *user* module which can be regarded as an alias for the EuLisp standard language extended with functionality to control the compiler. The interpreter provides more functionality, not described further here, common to interactive programming environments.

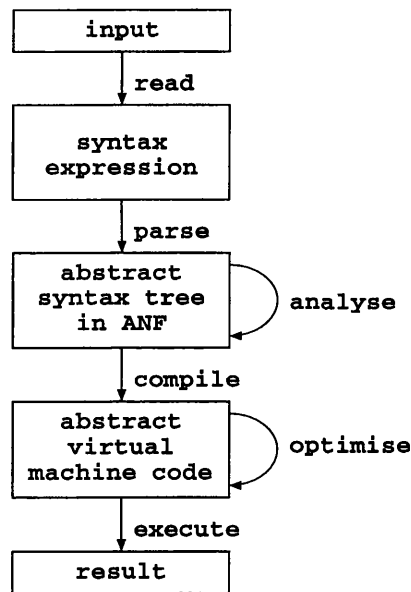


Figure 5-3: *The youtoo interpreter*

5.1.3 Example 1

The following example illustrates the way code vectors are actually embedded in C. The EuLisp module *test* defines the well-known factorial function in EuLisp:

³Information about compiler flags can be found in Figure B.4.

```
(defmodule test
  (import (level1)
    export (fact))

  (defun fact (x)
    (if (< x 2)
      1
      (* (fact (- x 1)) x)))

) ; end of module
```

The language `level1` bindings are imported and the defined lexical binding `fact` is exported. The bytecode compiler in `youtoo` translates the module into the following C code in file `fact.c`⁴:

```
#include <eulisp.h>
#include "test.h"

/* Module bindings with size 3 */
void *test_[3];

/* Initialize module test */
void eul_initialize_module_test()
{
  eul_initialize_module_level1();
  {
    /* BYTEVECTOR fact arity: 1 size: 28 index: 2 nbindings: 1 */
    static const long fact_bv[] = {I(aa,1b,84,1a),I(1b,44,04,83),I(36,0e,1c,
2c),I(1b,24,00,3c),I(01,1b,1f,04),I(16,22,02,45),I(02,00,00,00)};
    static long fact_code[] = {INT(28),NIL,(long)fact_bv};
    static long fact_bnds[] = {INT(1),NIL,B(test_,2)};
    static long fact[] = {INT(6),NIL,NIL,INT(1),NIL,NIL,(long)fact_code,(lon
g)fact_bnds};

    /* BYTEVECTOR initialize-test arity: 0 size: 20 index: 0 nbindings: 5 */
    static const long init_test_bv[] = {I(87,25,01,24),I(03,3e,07,24),I(02,3
c,00,21),I(01,23,04,2a),I(86,ac,00,00)};
    static long init_test_code[] = {INT(20),NIL,(long)init_test_bv};
    static long init_test_bnds[] = {INT(5),NIL,B(test_,0),B(test_,1),B(level
1_,0),B(level1_,1),B(test_,2)};
    static long init_test[] = {INT(6),NIL,NIL,INT(0),NIL,NIL,(long)init_test
_code,(long)init_test_bnds};

    /* Initialization of lambda: fact */
    eul_set_string_class(fact_code);
    eul_set_vector_class(fact_bnds);
    eul_set_lambda_class(fact);
    eul_set_lambda_name(fact,"fact");
    /* Initialization of lambda: initialize-test */
    eul_set_string_class(init_test_code);
    eul_set_vector_class(init_test_bnds);
    eul_set_lambda_class(init_test);
    eul_set_lambda_name(init_test,"initialize-test");

    /* Initialize module binding vector */
    test_[0] = init_test;
    test_[1] = NIL;
    test_[2] = fact;
  }
}
```

⁴The code is slightly simplified to enhance readability.

```
}  
} /* eof */
```

The following C macros are provided with `eulisp.h`:

```
#define NIL 0  
#define B(m,i) (long)&m[i]  
  
#ifdef LITTLE_ENDIAN  
#define I(x1,x2,x3,x4) 0x##x4##x3##x2##x1  
#else  
#define I(x1,x2,x3,x4) 0x##x1##x2##x3##x4  
#endif
```

The `I()` macro reverses the four bytecodes constructing a long on little endian machines so that bytevectors can be safely cast to `(char *)` when interpreted. The `INT()` macro converts C integer representation into tagged Lisp integer representation cast to long. The resulting run-time representation of the Lisp closure `fact` is a Lisp object with six slots.

```
Instance #<simple-function: fact> of class <simple-function>  
  name = fact  
  arity = 1  
  setter = ()  
  environment = ()  
  code = "\x0aa\x01b\x084\x01a\x01b\x044\x004\x083\x036\x00e\x01c\x02c..."  
  bindings = #(<C: 0x1007B800>)
```

The `code` and `binding` slots are initialized statically with the Lisp string `foo_code` (itself referring to the constant bytevector `foo_bv`) and the Lisp vector `foo_bnds` that holds bindings used from within the bytevector. In the example, the binding vector refers to the factorial function in the same module. The remaining part of the closure is initialized with `eul_set-` statements. The closure is finally stored in the module vector `test_` at index 2⁵.

By avoiding inlined addresses in bytevectors directly, the virtual machine code has almost no unused padding space for alignment and most importantly, is located in sharable read-only text segments, i.e. `.rodata` in ELF (see also Section 5.5).

The interface file `fact.h` defines the exported binding offset on the C level.

```
#ifndef EUL_TEST_H  
#define EUL_TEST_H  
  
#include <level1.h>  
  
extern void *test_[];  
extern void eul_initialize_module_test();  
  
/* Local module bindings */
```

⁵At index 0 is the module initialization function (i.e. top-level forms) stored; at index 1 is `NIL` if the module initialization function has not yet been called.

```
#define eul_test_fact_binding_ 2
#endif /* eof */
```

The interface file `fact.i` contains the exported binding offset with additional information about local literals and function binding vectors which is used with separate compilation and dynamic loading.

```
(definterface test
  (import (level1)
    bindings (
      (fact 2 test fact ())
    )
    local-literals (
    )
    lambda-bindings (
      ...
    )
  )
)
```

5.1.4 Example 2

Calling a foreign C function is similar to calling the factorial function in the previous example. The main difference is that a stub-function is generated to deal with argument and result conversion as specified by the `defextern` declaration in the source code. This stub function can then be referred from a binding vector in the same way `fact` is referred from `fact_bnds` in the previous example. A special virtual instruction is necessary to call the stub function.

```
(defextern atoi (<string>) <int>)
(defun foo (x) (atoi x))
```

is compiled into the following bytevector:

```
/* Foreign stub functions */
static LispRef ff_stub_atoi (ARG(LispRef *,sreg_value_sp))
ARGDECL(LispRef *,sreg_value_sp)
{
  LispRef G003, res;

  EUL_EXTERNAL_POPVAL1(G003);
  EUL_FF_RES_CONVERT0(res,atoi(EUL_FF_ARG_CONVERT3(G003)));
  return res;
}
...
/* Initialize module test */
void eul_initialize_module_test()
{
  ...
  /* BYTEVECTOR foo arity: 1 size: 8 index: 2 nbindings: 1 */
  static const long foo1_bv[] = {I(aa,41,00,45),I(01,00,00,00)};
  static long foo1_code[] = {INT(8),NIL,(long)foo1_bv};
  static long foo1_bnds[] = {INT(1),NIL,B(test_,3)};
}
```

```
static long foo1[] = {INT(EUL_LAMBDA_SIZE),NIL,NIL,INT(1),NIL,NIL,(long)
foo1_code,(long)foo1_bnds};
    ...
    test_[3] = ff_stub_atoi;
}
```

The foreign-function interface explained so far defines the means to describe a C function's arguments and return type to EuLisp. However, it is also desirable to call a EuLisp function from C and then to do mutually recursive calls between EuLisp and C. An in-call is realized as a call to the interpret function of the virtual machine emulator with the EuLisp function to call as argument. The following example illustrates how the EuLisp function `fact` in module `test` can be invoked from C.

```
#include <eulisp.h>

EUL_IMPORT(test)
EUL_DEFINTERN(fact,1,test)

main()
{
    int res;

    EUL_INITIALIZE();
    res = eul_int_as_c_int(fact(c_int_as_eul_int(10)));
    printf("fact(10)=%d\n", res);
}
```

The include file `eulisp.h` provides macros to access bindings of EuLisp modules as well as routines to convert between EuLisp and C data. The foreign function interface is similar to that described for Ilog Talk [DPS94] and is extended for other programming languages (e.g. Fortran and Pascal) under the Solaris operating systems where function calling follow same conventions. Further interoperability with C++ is achieved with a C function wrapper and the `extern "C"` declaration.

5.2 Code Mobility

Since the architecture realized with `youtoo` is designed to support dynamic object-oriented languages with first-class functions and reflective capabilities, it is consistent to provide read *and* write access to the `code` slot of closures at run-time. The bytecode vector of a closure object in `youtoo` is naturally represented as a Lisp string (`fact_code` in Section 5.1.3) that can be freely accessed and for instance shipped unchanged to a virtual machine process that runs on different machine. Its execution will have the same semantics regardless of the underlying hardware. This shows that functions, although represented as C data structures, do not lose their architectural neutrality.

Serialization of functional objects (i.e. simple/generic functions, continuations, threads), for instance for migration between processes, can only be provided if the module name and its respective offset can be determined from the absolute C address used in the binding vectors of the closures on the bottom of the functional object (e.g. `fact_bnds` in Section 5.1.3). These addresses generally point into the vector holding the bindings of a module (i.e. `test_` in Section 5.1.3). With serialization, the binding vector at the closure is unlinked from the current address space and annotated with the relevant information to retrieve the binding in a different address space. The algorithm to determine the module name and the binding vector offset from a binding reference is described in Figure 5-4.

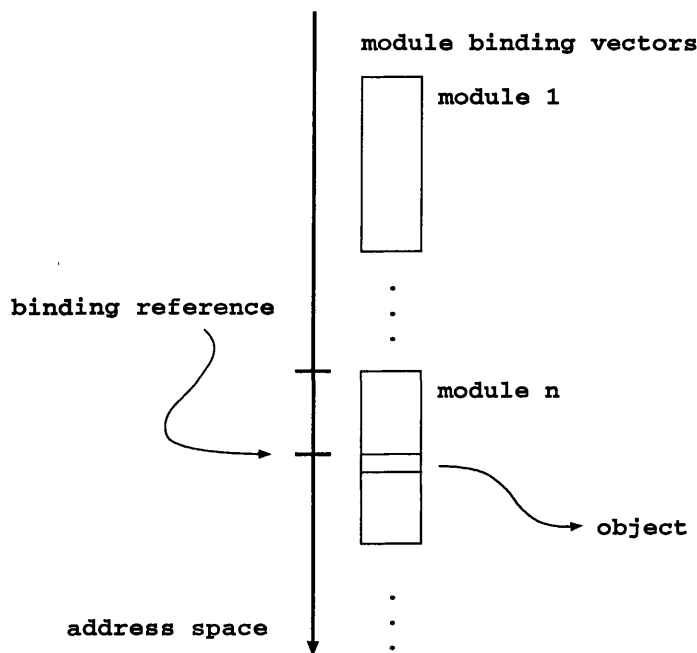


Figure 5-4: *Resolving binding references for code vector serialization. Step 1: module binding vectors of all linked modules are ordered with regard to their location in memory (fragmentation does not matter). Step 2: find the module binding vector into which the binding reference is pointing (e.g. with a linear search). Step 3: return the module name and subtract the start address of the module binding vector from the binding reference to obtain the offset.*

5.3 Demonstrating Performance

It is not claimed that `youtoo` is a high performance system. This is mainly due to a lack of local compile-time optimizations that can be applied even in the presence of dynamism.

However, `youtoo` shows reasonable overall performance (see Table 5.1) which is mainly due to the combination of the indexed variation of code threading, optimal instruction ordering and quasi-inline method caching. Individual figures for these techniques are presented later in this section.

Program	clisp 3.28		scheme 48 0.36		youtoo 0.93	
arith0	67.50s	5.80	22.88s	1.96	11.63s	1
mem	15.61s	1.07	8.35s	0.57	14.52s	1
nfib	38.49s	2.47	220.74s	14.18	15.56s	1
rec	64.66s	2.29	49.14s	1.74	28.20s	1
tak	16.35s	1.05	34.44s	2.22	15.45s	1
takl	9.85s	0.71	51.69s	3.77	13.71s	1
vec	26.51s	2.89	19.08s	2.08	9.16s	1

Table 5.1: *Comparison with other bytecode systems*

It is very difficult to perform a fair comparison of language implementations, so that Table 5.1 is not discussed in great detail. Worth noting is `youtoo`'s performance drop with a memory management stressing benchmark. The cost of conservative garbage collection [Zor93] can here be observed.

The actual timing is provided by the Unix `time` function uniformly for all language implementations and benchmark programs. The benchmark programs are briefly described in Table B.3. Execution times are recorded by timing benchmark programs twice, with and without calling the entry function. The difference in time reflects the raw execution time of the benchmark and eliminates implementation differences in program invocation that are not discussed here (e.g. initialization, program loading, program preprocessing or even compilation). For better accuracy, the average over ten execution times is taken into account.

Section 4 listed indexed code threading and optimal instruction ordering as key techniques that are included in the implementation architecture. The following subsections show the impact of these techniques on the performance of `youtoo` individually.

5.3.1 Indexed Code Threading

The impact of indexed instruction transfer which was identified in Section 4.2 as a practical variation of code threading is reported for `youtoo` in Tables C.1, C.3 and C.3 in the Appendix and summarized in Table 5.2.

The figures indicate that on average about 15% execution time can be saved (in some cases even up to 29%). However, execution times of virtual machines cannot be improved

	P5		MIPS		SPARC	
	switched	threaded	switched	threaded	switched	threaded
tak	1	0.86	1	0.78	1	0.98
arith0	1	0.85	1	0.88	1	0.99
arith1	1	0.92	1	0.95	1	0.98
rec	1	0.87	1	0.77	1	0.97
takl	1	0.77	1	0.79	1	1.26
meth	1	0.82	1	0.73	1	1.00
vec	1	0.71	1	0.81	1	0.99
hanoi	1	0.96	1	0.80	1	0.89
nfib	1	0.83	1	0.78	1	0.99
mem	1	0.99	1	1.00	1	1.01
		0.86		0.83	1	1.01

Table 5.2: *Threaded Instruction Dispatch (relative)*

with indirect threading on the SPARC architecture (see also Section 5.3.3).

5.3.2 Optimal Instruction Ordering

Three different instruction orderings for the suite of 10 programs (listed in Table B.3) are applied in the virtual machine emulator program. The first ordering is generated by a random generator⁶. The second ordering is derived from profiling information about the dynamic instruction frequency of the program being measured and thus called custom ordering. With custom ordering, the instruction executed most in the source text of the emulator program is next to the instruction being executed second most etc. Since it is not practical to modify and recompile the emulator for each program to execute, a third ordering is used, referred to as optimal. The optimal ordering reflects the custom orders of a set of programs and thus can be used in the interpreter. Figures 5-5 and 5-6 illustrate the different instruction orderings for the `meth` benchmark. The latter figure clearly shows that the instruction frequencies of `meth` deviate slightly from the average instruction frequencies. Nevertheless, the optimal ordering is much closer to the ideal custom ordering than the initial random ordering.

Tables C.1, C.3 and C.3 in the Appendix and Table 5.3 in this section show the absolute and relative effect of installing the random, custom and optimal instruction orders. The benefit is of the order of 20% on P5 and MIPS. The SPARC architecture however shows again no speed up for the different orderings. The optimal ordering that resulted from the programs in Table B.3 are showed in Table C.4.

⁶The specific random order may have an impact on the execution time of one particular benchmark program. However, we have checked that the average execution time is independent of the actual random order.

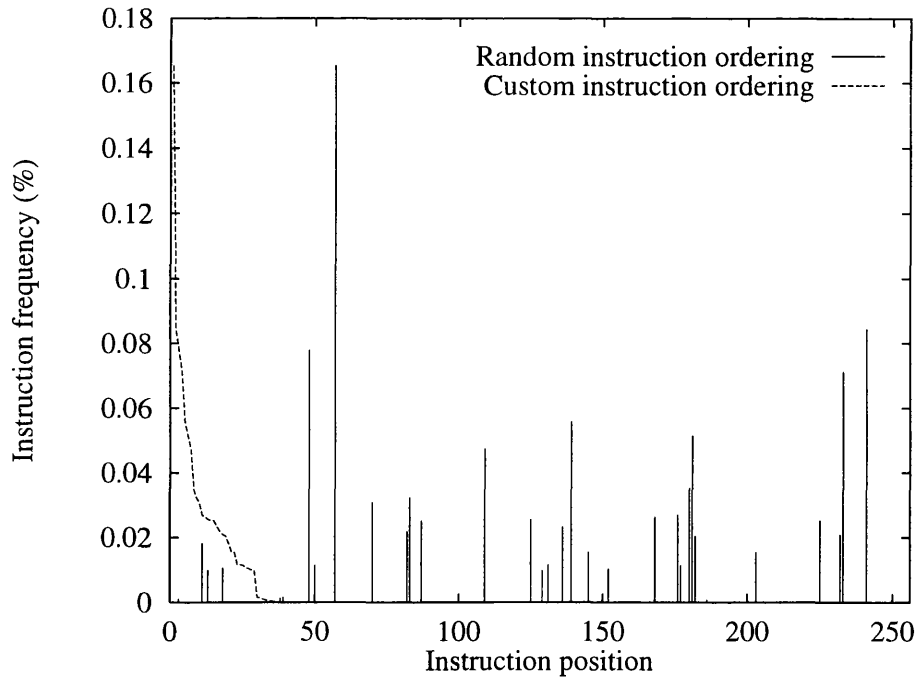


Figure 5-5: *Random instruction ordering*

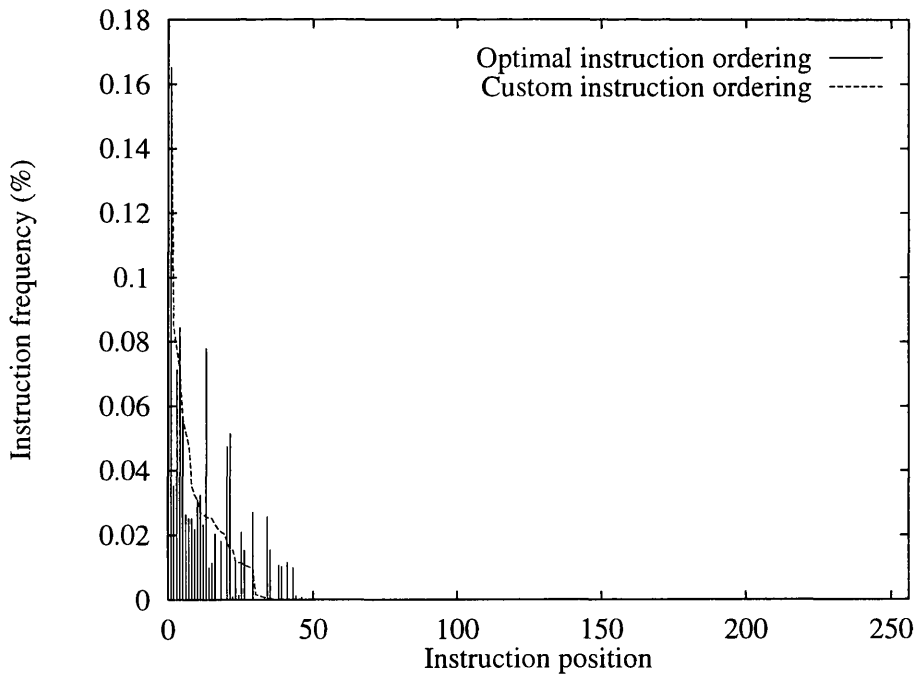


Figure 5-6: *Optimal instruction ordering*

	P5			MIPS			SPARC		
	rand	custom	optimal	rand	custom	optimal	rand	custom	optimal
arith0	1	0.72	0.72	1	0.60	0.70	1	1.00	0.97
arith1	1	0.86	0.89	1	0.83	0.87	1	0.87	0.86
hanoi	1	0.78	0.81	1	0.74	0.63	1	0.98	0.95
mem	1	0.96	0.96	1	0.95	0.96	1	1.08	0.97
meth	1	0.76	0.77	1	0.80	0.81	1	0.89	1.10
nfib	1	0.72	0.66	1	0.97	0.98	1	0.84	0.84
rec	1	0.94	0.97	1	0.97	0.98	1	1.00	0.99
tak	1	0.71	0.74	1	0.93	0.93	1	0.98	0.99
takl	1	0.77	0.77	1	0.93	0.94	1	0.99	1.27
vec	1	0.66	0.61	1	0.80	0.71	1	0.98	0.98
		0.79	0.79		0.85	0.85		0.96	0.99

Table 5.3: *Instruction ordering (relative)*

5.3.3 The SPARC Oddity

Neither indexed code threading nor optimal instruction ordering has significant (positive or negative) influence on the performance of `youtoo` on the SPARC architecture. This subsection attempts to explain this oddity.

As the instruction cache size of the used SPARC machine is twice as large as the instruction caches with the P5 and MIPS architectures (see Table B.1), the SPARC cache performance profile is likely to differ from the MIPS and P5 architectures. In order to bring more light into the relationship between native code size and instruction cache performance Figure 5-7 illustrates the cumulative native code (text segment) size of the emulation function according to the optimal ordering in Table C.4.

Figure 5-7 shows that the 19 most frequently invoked virtual machine instructions can be within the MIPS instruction cache (16K). For the P5 it can even be the 30 most frequently used instructions. The better results of optimal instruction ordering on P5 compared to MIPS can perhaps therefore be attributed to the compactness of the CISC instruction set. However, the first 29 most frequently invoked virtual instructions fit with their native code implementation as well into the native instruction cache of the SPARC processor (32K). Unfortunately, a simple explanation for the SPARC oddity can therefore not be derived from Figure 5-7.

However, the hardware information in Table B.1 bears some uncertainty in the SPARC case. Shared memory machines usually have large level-2 caches, between 1 to 8 MB per processor [Sim97]. In case the SPARC specification which was available during the bench-

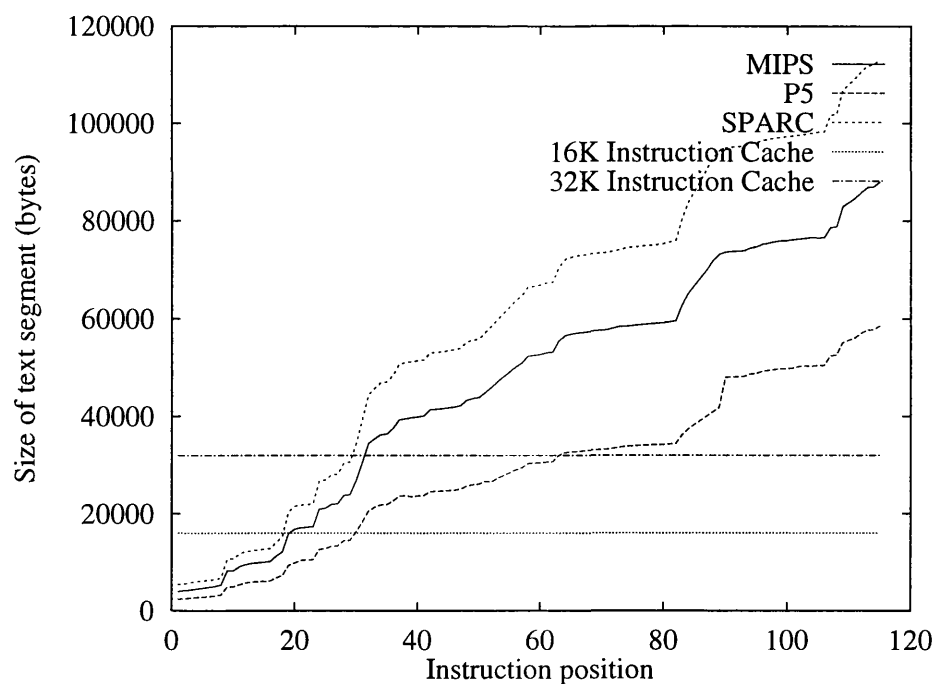


Figure 5-7: *Accumulative instruction size*

marking is in this respect not accurate, instruction ordering can have only little impact on the instruction cache performance because the entire virtual machine fits into the level-2 instruction cache.

A further source of uncertainty is related to the fact that hardware caches are shared by all the processes in execution. The benchmarking was deliberately performed at times of low machine load to reduce the influence of other concurrently running applications. But still at these times the used SPARC machine was busy with other applications much more than the P5 and MIPS machines. The SPARC oddity can have therefore here another reason.

For the sake of brevity, a more sophisticated discussion on instruction caching is not included here.

5.3.4 Quasi-Inline Method Caching

In order to understand the effect of quasi-inline method caching on the dynamic method lookup in `youtoo`, several measures are displayed in Figure 5-8. The data in the figure is collected by profiling the OPS5 system (implemented in EuLisp [OP93]) when resolving a non-trivial rule set.

Figure 5-8 shows normal high method cache miss ratio during initialization (of the EuLisp level-1 modules). However, when the OPS5 specific code is entered (after generic function

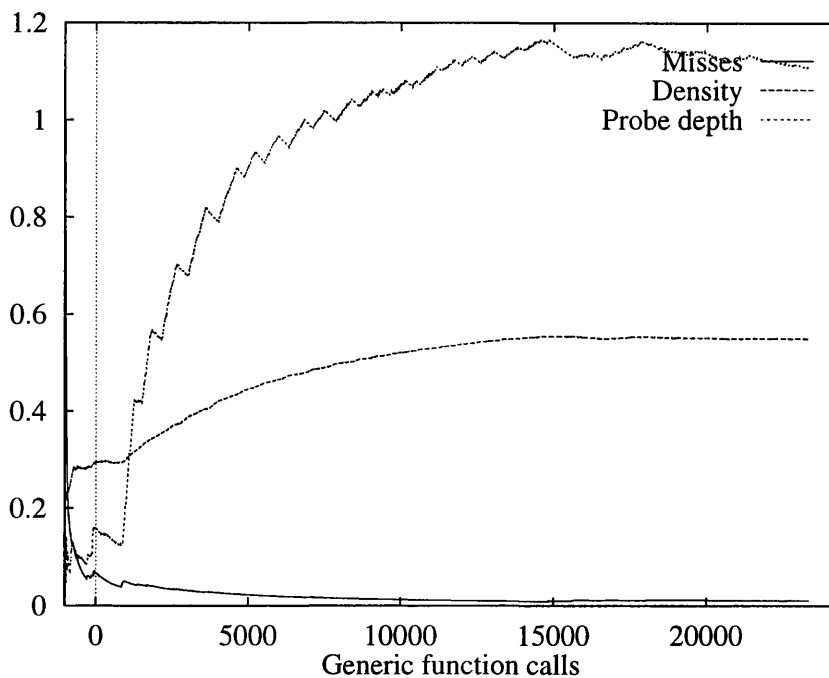


Figure 5-8: *Quasi Inline Method Caching*

call 0), miss ratio is already below 10% and from then on exponentially dropping down to 1.06%. Application dependent code is regularly called now so that the method caches fill up and the probe depth (i.e. the average number of steps necessary to reach a valid entry in the cache) increases up to 1.11. Thus, most table entries are either in the initial or the very next cache entry. Average cache density around 0.5 indicate that caches are only half filled in general. The sparse filling is important to keep the probe depth small. And in turn, a small probe depth is important to minimize the overhead of linear search in the cache.

OPS5 is chosen to demonstrate the success of quasi-inline method caching because of its abrupt change in behaviour after the RETE network is constructed for the input rule set. This point is reached circa after 15000 generic function calls. The dynamic change can be observed as a fluctuation in the probe depth. However, the miss ratio is hardly affected so that the dynamic method lookup continues to benefit from the caching scheme. Similar cache hit ratios have been measured with other applications as well.

Measuring the actual performance increase with method caching in a hybrid object-oriented language (i.e. with both, simple and generic functions) is difficult. A three fold speed up could be observed with some pure object-oriented benchmarks when the caching scheme changed from hashing on the actual argument classes (à la CLOS) to hashing on the program counter.

5.4 Demonstrating Memory Efficiency

It is claimed in Chapter 4 that C embedded virtual machine code reduces static and dynamic memory usage.

Measurements with `youtoo` presented in Table C.5 show that the average ratio of the size of read-only data to read/write data changes from 1.32 to 0.29 when the `const` allocation qualifier is omitted with the code vector definitions in the EuLisp standard modules. In absolute terms, the total size of the compiled EuLisp modules is 361296 bytes, of which read-only data is 104880 and read/write data is 79248, with `const` declaration and 41568+142496 bytes, respectively, without. Sharable read-only data is therefore increased by a factor of two.

Thanks to shared objects, also the static memory usage is very good for applications of `youtoo`. An executable file—regardless of whether the hello-world program or the entire `youtoo` system compiled by itself—is generally less than 10K in size.

5.5 Demonstrating Interoperability

Based on `youtoo`'s foreign function interface it was possible to enrich EuLisp with three basic functionalities provided by publicly available libraries:

- distribution by linking `mpich`⁷ an implementation of the Message Passing Interface (MPI),
- a graphical user interface with a binding to Tcl/Tk and
- pre-emptive multi-threading with a link to POSIX kindred thread libraries.

All three of these libraries are being used in the development of a multi-agent system modelling the Spanish Fishmarket [RNSP97]. The rest of this section gives some details about the latter interface because it uncovered at least all of the problems that also occurred with the other libraries.

Multi-threading is a powerful and structured way to concurrent execution supported by operating systems, libraries and high-level programming languages. But so far threads cannot be transparently shared between multiple languages in one application.

The distribution of `youtoo` includes ports to three external thread libraries (Solaris, MIT and PPCR) [KP98]. Thread transparency is achieved by running EuLisp functions on foreign

⁷See <ftp://info.mcs.anl.gov/pub/mpi>.

threads. First, an out-call from EuLisp to C creates a foreign thread with an initial C function that then performs an in-call to invoke the EuLisp function when the foreign thread is being started.

In detail, the out-call is made to a C function that takes an instance `thr` of the class `<thread>` representing foreign threads in Lisp and arguments for the initial Lisp closure. The Lisp closure is stored with thread initialization in the `function` slot of the instance (see Figure 5-9). The Lisp and C thread objects are linked with each other via the slot `thread-handle` and thread specific data (TSD). Finally, the C thread calls the interpreter entry function with the closure and arguments.

EuLisp's `thread-value` is implemented by a foreign C wrapper function that checks if the initial closure has returned already (i.e. foreign thread has terminated). In this case the slot `return-value` is no longer unbound, but set to the corresponding return value. If the initial closure has not yet returned (i.e. foreign thread is still running), C's function to join a thread will be called. After joining, the return value of the initial closure is accessible at the Lisp thread object. Figure 5-9 shows how the Lisp thread instance is linked with the corresponding C thread structure. A mutex is necessary to assure a thread-safe implementation of `thread-value`.

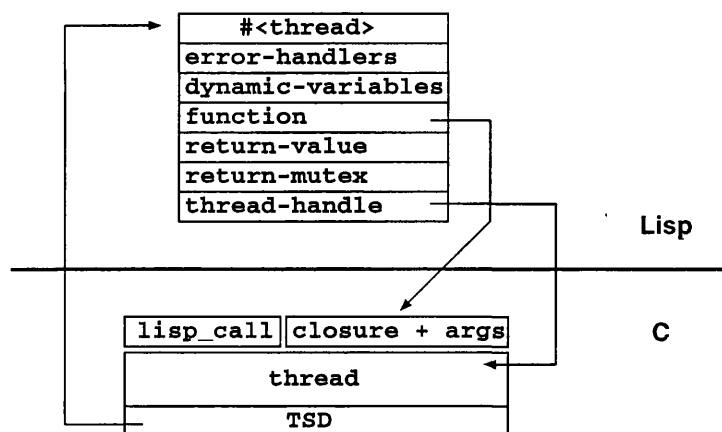


Figure 5-9: *Thread representation*

Yielding control in favour of another thread by means of `thread-reschedule` can be directly mapped onto the corresponding foreign thread functions using the `thread-handle` slot. Inverse pointers from C threads to EuLisp thread objects are kept as thread specific data. EuLisp's `current-thread` returns such a backward pointer from the currently running C thread.

So far, it has been explained how EuLisp threads are based on foreign threads. We still need to show how threads in EuLisp and C can be synchronized in a transparent way. EuLisp locks cannot be implemented with some of the foreign thread libraries (e.g. Solaris), because a mutex can only be unlocked by the thread that last locked the mutex (the mutex owner), but EuLisp locks are defined to be accessible by any part of the program, e.g. to permit producer/consumer style synchronization. Creating a binary semaphore in EuLisp results in the creation of a foreign counting semaphore with count initialized to 1. EuLisp uses a boxed C pointer to handle the semaphore. The class `<lock>` can be easily subclassed to implement counting semaphores. The EuLisp functions `lock` and `unlock` are straight out-calls to foreign semaphore functions `wait` and `signal`.

The comparison in Table 5.4, based on a four-processor SPARC 4d architecture (SS100E, each 50MHz) running Solaris 2.5.1, shows the performance benefit of using foreign thread libraries compared to `youtoo`'s built-in non-pre-emptive multi-threading with one-shot continuations that operate on the stack architecture of the virtual machine.

foreign threads	real	user	sys
padd	0m11.595s	0m9.303s	0m7.940s
dphil	0m19.033s	0m16.512s	0m24.081s
built-in threads	real	user	sys
padd	5m33.657s	1m48.164s	0m34.949s
dphil	2m24.026s	1m50.893s	0m30.571s

Table 5.4: *Performance comparison*

The timing is deliberately performed on a multiprocessor machine, which explains that the total of the values printed for user and system time exceeds real time when using the foreign thread library. However, the sum of user and system time reveals that applications can also benefit on single-processor machines by running 2 to 8 times faster using foreign thread libraries instead of built-in threading.

Safe memory (de)allocation in a pre-emptive multi-threaded run-time environment is assured by using a conservative memory management system that is designed to work with the thread libraries in question [BW88, WDH89]. Furthermore it is worth mentioning that it is easy for the user to switch between the co-operative, built-in threads and the foreign threads when writing a program. By importing the `fthread` module and linking the corresponding library, the standard classes `<thread>` and `<lock>` are simply redefined so that programs can run unchanged for different thread implementations.

5.6 Conclusion

The most important aspect of this chapter is that it demonstrates the applicability and effectiveness of the interpreter architecture proposed in Chapter 4. The realization of the architecture in form of the `youtoo` system shows increase in performance, memory efficiency and ease of interoperability.

Many details which are further addressed in `youtoo` are not mentioned here in order to keep the focus on the aspects related to the key elements of the architecture.

Chapter 6

Related Work

This thesis is concerned with the implementation of object-oriented dynamic programming languages based on bytecode interpretation. A new interpretive implementation architecture is proposed that meets the requirements of code and system portability, performance, static and dynamic memory efficiency as well as language interoperability.

Much of the knowledge about bytecode interpretation that has accumulated over the years can be found in Debaere *et al* [DVC90]. And experiences in the object-oriented context are collected in [Kra83]. In this chapter it is tried to compare existing language systems and their individual techniques with the approach taken in the proposed architecture.

6.1 Smalltalk-80

Smalltalk technology today is in many aspects still defined by the Deutsch and Schiffman Smalltalk-80 system [DS84]. This implementation uses two executable program representations, virtual machine code and native machine code. A compiler, as part of the interactive development environment, dynamically generates on demand native code from virtual machine code. The Smalltalk-80 system demonstrates with this run-time translation that it is possible to implement a complete dynamic object-oriented programming environment (including a window system) with modest hardware resources.

The Smalltalk-80 system, although powerful for rapid prototyping, has major disadvantages for the delivery of stand-alone applications. A Smalltalk application is typically bound to the development environment. This results from the dynamic character of the Smalltalk language (see also Chapter 3) but is also made worse by using run-time code generation. With

dynamic code translation between different representations a compiler and a lot more classes than actually necessary have to be linked to applications. In fact, a Smalltalk-80 application is an incrementally modified version of the development environment [Gol84]. Explicit configuration of stand-alone applications [Sri88] and various stripping techniques address this delivery problem with a lot of (user) effort. In contrast, applications with `youtoo` allow to deploy small stand-alone executables thanks to the technique of compiling EuLisp modules into C embedded virtual machine code and archiving compiled modules in native libraries.

Beside dynamic compilation on demand, inline method caching was first developed with Smalltalk-80. The relation between inline method caching and quasi-inline method caching as proposed in the implementation architecture is discussed in the following section.

6.2 Self

The Self system [CUL89] is probably the most advanced experimental language implementation in the context of dynamic object-oriented programming. Self is similar to the Smalltalk-80 system but differs in its use of prototypes instead of classes and its use of messages instead of variables to access state.

At the heart of the system is an optimizing native compiler that is designed to reduce polymorphism and enable method inlining with techniques like, customization [CU89], message splitting [CU90], type inference [APS93] and polymorphic inline method caching [HCU91]. The latter technique has been described already in Chapter 4. Customization is a technique to reduce polymorphism in Self by duplicating methods for more specific arguments. Similarly to method inlining, the body can be optimized with regard to the specific argument types [CU89, DCG95]. Splitting is similar to customization but aimed to reducing polymorphism in a single method by duplicating and re-arranging control paths [CU90]. Type inference is discussed later in this chapter.

The success of these techniques is very good when applied together and in an iterative manner. In addition, profiling information is used to determine profitable areas of optimization and to compile code with a bias toward common type cases (type feedback [HU94]). The techniques developed with Self emphasize the environment character of the Self system. Like with Smalltalk-80, the delivery of small executables is however difficult since compile-time and run-time are interleaved. Type inference and customization aggravate the delivery problem since the entire source code has to be present in order to achieve the reported performance increase. Such a total compilation furthermore precludes dynamic linking [HU94].

The critique on the Self approach from the viewpoint of this thesis is that memory inefficiency is simply unacceptable for “stand-alone” Self applications. Agesen et al [AU94] claim to have developed an automated application extractor for Self based on type inference. It is however unclear how well type inference, as well as customization, scale with larger applications. Without heuristics customization is likely to result in code explosion and type inference typically requires very long compilation times. It seems therefore that mapping source modules with restrictive import/export declarations onto C compilation units which can be subsequently archived in shared libraries, as described in this work, is better suited to produce stand-alone applications (or deliverable APIs) with modest memory requirements. The speed of the Self system which comes close to C performance is however undisputed.

6.3 Slim Binaries

Chapter 3 introduced source code and bytecode as architecture neutral program representations. Another format is used with Oberon [FK97]. The abstract syntax tree of an Oberon source module is encoded with a compression scheme based on LZW into a *slim binary*. Slim binaries cannot be interpreted directly but have to be compiled at load-time. The advantage over bytecode is that the control-flow structure of the source module is captured in the representation. The resulting native code generated on-the-fly can therefore be much better optimized than native code generated from bytecode.

The tree-based encoding technique applied with slim binaries requires a loading facility to generate the final executable application from encoded modules. The loader is basically a fast native compiler with all its architecture dependence and development cost. The approach proposed in this thesis is different in this aspect. After generation of C embedded virtual machine code, the presented architecture joins the standard route of software development and delivery of the underlying operating system (i.e. standard C compiler, native linker, virtual memory management).

6.4 Translation into C

Chapter 3 mentioned translation into the C programming language as another popular route to achieve a high-level language implementation. There are two variations with C translation:

1. High-level code is compiled into C code that simulates a virtual machine.

2. High-level code is compiled into C code that uses C control structures.

Simulating a virtual machine involves generating C *code* and is therefore different from C embedded code vectors as the C language is here only used as a vehicle for data representation. The representation shares however the performance drawbacks of virtual machines.

The latter approach to C translation is applied in many systems [Bar89, TLA92, DPS94, Att94, SW95, Que96]. The approach supplies system portability and performance with standard C compiler technology. However, some high-level language features (e.g. `call/cc`, tail-call optimization) are difficult to implement. A major drawback for C translation is that the executable program representation is native code, an architecture dependent format. The approach is therefore not well suited for the prospective application domain of high-level distributed computing (see Chapter 3).

6.5 Java

Java is in many ways close to be a truly *dynamic* object-oriented language. Dynamic linking, automatic memory management and dynamic method lookup that allows run-time class linking, place Java close to dynamic object-oriented programming. However, Java lacks general reflective capabilities. The class `Class` cannot be subclassed and the language does not include a metaobject protocol, like CLOS for instance.

Java's interesting features from the perspective of this thesis are generic run-time class loading (the user can redefine a class loader) and bytecode verification based on Java's monomorphic instruction set [LY96]. Beside the critique on unshared virtual machine code (discussed in Chapter 4) which is involved with class loading, it has to be noted that the dynamic configurability is much more limited compared to other language implementations, including `youtoo`. Java application can only be extended during run-time with *new* classes. However, methods and classes cannot be modified or removed dynamically.

Application start-up with Java is performed more or less in the traditional way by reading bytecode files. Interoperability, application start-up and dynamic memory usage benefit therefore much more from the C embedded virtual machine code representation.

6.6 Other Techniques

6.6.1 Type Inference

Generally, type inference is used to derive enough type information to ensure safe run-time execution of a program. In order effectively to optimize object-oriented languages, multiple levels in generic function call trees need to be considered. Normally a (generic) function application call other (generic) functions, which might call again other (generic) functions and so on until a primitive call is reached. Only few type inference approaches actually deliver type information for multiple levels of polymorphism. Two of these are generic type schemes [KF93] and labeled type variables [PC94]; the latter approach shows better results, as the analysis effort is focussed on promising program parts only.

Some static analysis techniques (e.g. type inference) can help to diminish this penalty but often reduce as well the dynamic character of the language by employing compile-time dependencies.

Much work is devoted to reduce run-time type checks in Lisp systems [JW95, WC94, Hen92, Shi91a, SH87]. These approaches are typically focussed on global techniques to optimize list processing. In an object-oriented context, list processing is less dominant because objects can be used where before only lists were available. With the typical type inflation in object-oriented programming run-time check/tag removal is superseded by the more general problem of a fast dynamic method lookup.

Some flow analysis techniques are similar to type inference in the attempt to trace polymorphism over multiple call tree levels. Both, 1CFA [Shi91b] and flow directed inlining [JW96] disambiguate different function clones for different function call sites of the same function. Again, these techniques are targeted to optimizations, like method inlining, and work against the emphasis of few compile-time dependencies proclaimed in this thesis.

6.6.2 Method Lookup Optimization

With polymorphic inline method caching [HCU91] the monomorphic caching scheme used with Smalltalk-80 [DS84] is augmented to store a type case statement (in native code) in place of the full lookup call. The case statement reflects the previously computed methods which have been used at this call site and defaults to the full method lookup. It is argued in Chapter 4 already that quasi-inline method caching has advantages over the classical caching techniques as implemented in Smalltalk-80 and Self. Memory efficiency, cache flushing and

the fact that compiled code should be sharable (i.e. read-only) cause severe problems with the classical inline schemes.

Quasi-inline caching, as described in Chapter 4, uses a hashing scheme which is based on the (virtual) program counter. By doing so, type locality can be exploited although the cache is not really inlined but located at the generic function. The advantage of this scheme is that cache flushing and sharable virtual machine code is much simpler to realise. Furthermore, with a linear search hashing policy on collision, redundancy with classical inline caching can be avoided (see Section 4.4). While the space overhead with polymorphic inline caching increases linear with compiled code (about 2% [HCU91]), the space overhead with the quasi-inline technique increases only linear with the number of generic functions.

The cache miss ratio with polymorphic inline caching is reported by Hölzle *et al* [HCU91] as being between 1% and 11% for a suite of five benchmarks. The miss ratio with quasi-inline method caching is measured as 1.06% as well for several applications.

Driesen *et al* [DH95] present several dispatch techniques for statically- and dynamically typed languages in a common framework and discuss their cost on pipelined processors. The results are useful for native code generation and address multiple-inheritance. Dynamic method and class creation as well as memory consumption are however unaddressed.

An interesting approach to optimized method dispatch is proposed by Queinnec [Que95]. Like quasi-inline method caching, this technique is concerned to preserve the dynamic capabilities of dynamic object-oriented languages, e.g. run-time class/method definition/removal. In contrast to many other approaches, the technique performs method dispatch based on decision trees and uses an optimized subclass predicate. Although compact and fast, scalability—particularly with multi-method dispatch—appears to be a problem since the scheme does not adapt to the typical hot-spots that appear with generic function invocation in object-oriented programming.

6.6.3 Stack Caching

An generic technique that delivers notable improvements for bytecode interpreters is stack caching [Ert95]. *Dynamic* stack caching requires multiple copies of the interpreter to keep track of the state of the stack. With *static* stack caching the compiler keeps track of the stack state. These techniques speed-up instruction dispatch by adding significant complexity to the interpreter or compiler and do not promise to be equally successful on different processors¹.

¹The second issue partly applies as well to the proposed architecture.

6.6.4 Sealing

Sealing (or freezing) is used with CMU Common Lisp [Me92] and Dylan [Sha96] to control the dynamism of functions and classes. Not all parts of the object system need to (or must) make use of the potential dynamism and opportunities for incremental development. The idea is that the performance of these parts should not be compromised for flexibility they don't embody.

If a class is sealed it may not be subclassed further. And similarly, after sealing a generic function, no additional methods may be added. Sealing of dynamically created classes and generic functions is generally not supported since this declaration is aimed at compile-time optimizations. Sealing enables a better starting point for static analysis of separately compiled parts of a dynamic object-oriented program. In some cases however it may be difficult to anticipate which classes could be subject to future reuse.

6.6.5 Method Inlining

Inlining is an effective technique to improve execution efficiency by replacing a function call with the body of the called function. In this way, time for handling arguments and allocating control frames can be saved. For very small functions the function call overhead can even exceed the execution time spend in the function body. Generally, inlining enables ensuing optimizations. The inlined body can be optimized with respect to the context in the host function as well as the host function can be optimized with respect to the inlined code. Function inlining is particularly effective for programming languages with a high function invocation frequency.

Because of (i) high expense and frequency of generic function invocation arising with inheritance and encapsulation, (ii) difficulties with static method binding and (iii) the effect on the success of ensuing optimizations, inlining of methods can be regarded as the key optimization in object-oriented programming. Because of this, the following techniques are in one or another way aimed towards method inlining.

Chapter 7

Conclusions

This work strives for a language implementation architecture that addresses the requirements of (bytecode) interpreted object-oriented dynamic programming (or short *dynamic objects*). To achieve efficiency and interoperability without restricting the distinctive flexibility of dynamic objects, several new implementation techniques are developed and tested, including *embedded virtual machine code*, *indexed code threading*, *optimal instruction ordering* and *quasi-inline method caching*.

C embedded virtual machine code refers to the representation of bytecodes as constant C arrays that are located in sharable text segments after compilation. Interoperability, application start-up and dynamic memory usage benefit from this representation. Indexed code threading addresses the performance problem with virtual instruction mapping (i.e. loading, decoding and invoking) by using fast threaded instruction transfer. Unlike with standard code threading, virtual machine code remains compact and can also be executed by a non-threaded virtual machine emulator. A further performance boost is achieved with optimal virtual instruction ordering. This technique helps to cluster the native code implementing virtual instructions so that native instruction cache performance is increased. Finally, the efficiency problems involved with dynamic method lookup are alleviated with an inline caching scheme that is applicable with constant bytecode vectors. The scheme exploits type locality similar to polymorphic inline caching. However, dynamic memory is saved by avoiding redundant method entries and by being adaptable to generic function invocation which typically comes in waves with hot-spots on particular methods.

With indexed code threading and optimal instruction ordering, *youtoo*, an implementation of the proposed architecture shows an average performance increase of about 20% on the

P5 and SPARC architectures. Embedded virtual machine code increases sharable read-only data by a factor of two. Virtual machine code can be shared in memory by different client applications executing it concurrently on the same machine, so that at most one copy of a module's virtual machine code exists in memory. Quasi-inline method caching, finally, results with `youtoo` in a method lookup miss ratio of 1.06%.

The implementation architecture is realized in `youtoo` with single inheritance and multi method dispatch. Nonetheless, the techniques are of general applicability. Any object-oriented dynamic language, regardless if single/multiple dispatch/inheritance, can benefit from quasi-inline method caching (e.g. Smalltalk and CLOS). And embedded virtual machine code, indexed code threading and optimal virtual instruction ordering can help to enhance the performance of any bytecoded language implementation.

With much focus on Java and its virtual machine approach recently, it would be interesting to see the impact of applying some of the techniques to an implementation of Java. Although Java owes its success to code mobility within the World-Wide Web [GJS96], more and more applications use Java as a general purpose programming language without applets (and related classes). In this context, the architecture should be applicable also to Java virtual machines.

Unaddressed in the proposed implementation architecture is the problem of method cache access with pre-emptive multi-threading [KJ93]. In general, it has to be ensured that one thread does not modify a method cache while another thread is reading it. The default thread implementation in `youtoo` is realized on the level of the virtual machine so that atomic read/write access to cached methods can be assumed. With foreign threading this assumption does not hold any longer. A mechanism for locking method tables is required.

The combination of dynamic object-oriented programming and bytecode interpretation is exciting because of the involved tradeoffs between efficiency and flexibility. In contrast to other approaches, this work tried to balance these tradeoffs so that the distinguishing flexibility of object-oriented dynamic programming is not compromised. In summary, it is believed that the architecture that is developed within this thesis can be regarded as a consistent continuation of the evolution of bytecode interpretation driven by the specific requirements of dynamic object-oriented programming, including efficiency, interoperability and portability.

Bibliography

- [ABC⁺96] O. Agesen, L. Bak, C. Chambers, B.-W. Chang, U. Hölzle, J. Maloney, R. B. Smith, and M. Wolczko. *The SELF 4.0 Programmer's Reference Manual*, 1996.
- [AGS94] E. Amiel, O. Gruber, and E. Simon. Optimizing multi-method dispatch using compressed dispatch tables. In *Proceedings of the Conference on Object-Oriented Systems, Languages, and Applications*, volume 29(10) of *ACM SIGPLAN NOTICES*, pages 244–258, 1994.
- [AH87] G. Agha and C. Hewitt. Actors: A conceptual foundation for concurrent object-oriented programming. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 49–74. MIT Press, 1987.
- [AI96] American National Standards Institute and Information Technology Industry Council. *American National Standard for Information Technology: programming language — Common LISP*. American National Standards Institute, 1996.
- [App91] A. W. Appel. Garbage collection. In Peter Lee, editor, *Topics in Advanced Language Implementation*, pages 89–100. MIT Press, Cambridge, Massachusetts, 1991.
- [APS93] O. Agesen, J. Palsberg, and M. I. Schwartzbach. Type Inference SELF: Analysis of Objects with Dynamic and Multiple Inheritance. In O. Nierstrasz, editor, *European Conference on Object-Oriented Programming (ECOOP)*, LNCS 707, pages 247–267. Springer, July 1993.
- [AR92] P. André and J.-C. Royer. Optimizing method search with lookup caches and incremental coloring. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 110–126, October 1992. Also available as SIGPLAN Notices, 27(10), 1992.

- [Att94] G. Attardi. The embeddable Common Lisp. In *Proceedings of the Lisp Users and Vendor Conference*, Berkeley, California, August 1994.
- [AU94] O. Agesen and D. Ungar. Sifting out the gold. In *Proceedings of the Conference on Object-Oriented Systems, Languages, and Applications*, pages 355–370. ACM, 1994.
- [Bar88] J. F. Bartlett. Compacting garbage collection with ambiguous roots. Technical Report 88/2, Digital Equipment Corporation Western Research Laboratory, Palo Alto, February 1988.
- [Bar89] J. F. Bartlett. Scheme- \rightarrow C a Portable Scheme to C Compiler. Technical Report 89/1, DEC Western Research Laboratory, 1989.
- [BCD⁺91] M. E. Benitez, P. Chan, J. Davidson, A. Holler, S. Meloy, and V. Santhanam. ANDF: Finally an UNCOL after 30 years. Technical Report CS-91-05, University of Virginia, March 1991.
- [BDG⁺88a] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kicsales, and D. A. Moon. *Common LISP object system specification X3J13 Document 88-002R*. SIGPLAN Notices, 23(9). 1988.
- [BDG⁺88b] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon. Common LISP object system specification X3J13 document 88-002R. *ACM SIGPLAN Notices*, 23, 1988. Special Issue, September 1988.
- [Bel73] J. R. Bell. Threaded code. *Communications of the ACM*, 16(6):370–372, 1973.
- [BKDP93] H. Bretthauer, J. Kopp, H. Davis, and K. Playford. Balancing the EuLisp metaobject protocol. *Lisp and Symbolic Computation*, 6(1/2):119–138, August 1993.
- [BKK⁺86] D. G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel. CommonLoops, merging Common Lisp and object-oriented programming. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 17–29, October 1986. Also available as SIGPLAN Notices 21(11), November 1986.

- [Boe93] H.-J. Boehm. Space efficient conservative garbage collection. In *Proceedings of SIGPLAN'93 Conference on Programming Languages Design and Implementation*, volume 28(6) of *ACM SIGPLAN Notices*, pages 197–206, 1993.
- [Bra96] R. J. Bradford. An implementation of Telos in Common Lisp. *Object Oriented Systems*, 3:31–49, 1996.
- [BSS84] D. R. Barstow, H. E. Shrobe, and E. Sandewell. *Interactive Programming Environments*. McGraw-Hill, New York, 1984.
- [BW88] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software—Practice and Experience*, 18(9):807–820, 1988.
- [CE91] W. Clinger and J. Rees (Editors). Revised⁴ report on the algorithmic language scheme. Available from `ftp://nexus.yorku.ca/pub/scheme`, November 1991.
- [Cha92] C. Chambers. Object-oriented multi-methods in Cecil. In *ECOOP '92, European Conference on Object-Oriented Programming*, volume 615 of *Lecture Notes in Computer Science*, pages 33–56. Springer-Verlag, 1992.
- [CJK95] H. Cejtin, S. Jagannathan, and R. Kelsey. Higher-order distributed objects. *ACM Transactions on Programming Languages and Systems*, 17(5):704–739, 1995.
- [CL94] C. Chambers and G. T. Leavens. Typechecking and modules for multi-methods. In *Object-Oriented Programming Systems, Languages, and Applications*, pages 1–15, October 1994. Also available as SIGPLAN Notices 29(10).
- [Cli84] W. Clinger. The Scheme 311 compiler: An exercise in denotational semantics. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 356–364, August 1984.
- [Coi87] P. Cointe. Metaclasses are first class: the objvlisp model. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 22, pages 156–167. ACM Press, December 1987.
- [Cor97] D. D. Corkill. Countdown to success: Dynamic objects, GBB, and RADARSAT 1. *Communications of the ACM*, 40(5):48–58, May 1997.

- [CPL83] T. J. Conroy and E. Pelegri-Llopart. An assessment of method-lookup caches for Smalltalk-80 implementations. In *in [Kra83]*, pages 239–247. 1983.
- [CU89] Craig Chambers and David Ungar. Customization: optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, volume 24, pages 146–160, Portland, OR, June 1989.
- [CU90] C. Chambers and D. Ungar. Iterative type analysis and extended message splitting: optimizing dynamically-typed object-oriented programs. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 150–164, June 1990.
- [CUL89] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of SELF a dynamically-typed object-oriented language based on prototypes. *ACM SIGPLAN Notices*, 24(10):49–70, October 1989.
- [CW85] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.
- [DB97] B. Davies and V. Bryan Davies. Patching onto the Web: Common Lisp hypermedia for the Intranet. *Communications of the ACM*, 40(5):66–69, May 1997.
- [DCG95] J. Dean, C. Chambers, and D. Grove. Selective specialization for object-oriented languages. *ACM SIGPLAN Notices*, 30(6):93–102, 1995.
- [DeM93] L. G. DeMichiel. CLOS and C++. In A. Paepcke, editor, *Object-Oriented Programming: the CLOS Perspective*, chapter 7, pages 157–180. MIT Press, Cambridge, Mass., 1993.
- [Deu73] L. P. Deutsch. A LISP machine with very compact programs. In Nils J. Nilsson, editor, *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, Standford, CA, August 1973.
- [DH95] K. Driesen and U. Hölzle. Minimizing row displacement dispatch tables. In *Proceedings of the ACM OOPSLA '95 Conference*, pages 141–155, 1995.

- [DPS94] H. E. Davis, P. Parquier, and N. Séniak. Sweet Harmony: The TALK/C++ Connection. In *Proceedings of the Conference on Lisp and Functional Programming*. ACM Press, New York, 1994.
- [Dri93] Karel Driesen. Selector table indexing & sparse arrays. In *Proceedings of the ACM OOPSLA '93 Conference*, 1993.
- [DS84] P. Deutsch and A. M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 297–302, January 1984.
- [DS96] R. V. Dragan and L. Seltzer. Java speed trials. *PC Magazine*, 15(10), October 1996.
- [DVC90] E. H. Debaere and J. M. Van Campenhout. *Interpretation and Instruction Path Coprocessing*. The MIT Press, 1990.
- [EEF⁺97] J. Ernst, W. Evans, Ch. W. Fraser, S. Lucco, and T. A. Proebsting. Code compression. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 358–365, June 15–18, 1997.
- [Ert95] M. A. Ertl. Stack caching for interpreters. In *SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 315–327, 1995.
- [FK97] M. Franz and Th. Kistler. Slim binaries. *Communications of the ACM*, 40(12):87–94, December 1997.
- [Fod91] J. Foderaro. LISP: Introduction. *Communications of the ACM*, 34(9):27–27, September 1991.
- [FS83] J. R. Falcone and J. R. Stinger. The Smalltalk-80 implementation at Hewlett-Packard. In Glenn Krasner, editor, *Smalltalk-80: Bits of History, Words of Advice*, pages 79–112. Addison-Wesley, 1983.
- [FSDF93] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. *ACM SIGPLAN Notices*, 28(6):237–247, 1993.
- [Gas92] L. Gasser. *An Overview of DAI*, pages 9–30. Kluwer Academic Publishers, 1992.
- [GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

- [GK94] M. R. Genesereth and S. P. Ketchpel. Software agents. *Communications of the ACM*, 37(7):49–53, July 1994.
- [GLDW87] R. A. Gingell, M. Lee, X. T. Dang, and M. S. Weeks. Shared libraries in SunOS. In USENIX Association, editor, *Proceedings of the Summer 1987 USENIX Conference: June 8–12, 1987, Phoenix, Arizona, USA*, pages 131–145, Berkeley, CA, USA, 1987. USENIX.
- [Gol84] A. Goldberg. The influence of an object-oriented language on the programming environment. In D. R. Barstow, H. E. Shrobe, and E. Sandewall, editors, *Interactive Programming Environments*, pages 141–174. McGraw-Hill, 1984.
- [Gol95] A. Goldberg. Why Smalltalk? *Communications of the ACM*, 38(10):105–107, October 1995.
- [Gos95] J. Gosling. Java intermediate bytecodes. *ACM SIGPLAN Notices*, 30(3):111–118, March 1995.
- [GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.
- [Gre84] R. Greenblatt. The LISP machine. In D. R. Barstow, H. E. Shrobe, and E. Sandewall, editors, *Interactive Programming Environments*. McGraw-Hill, 1984.
- [Gro95] Gartner Group. Research note on LISP, July 1995.
- [HAKN97] J. Hummel, A. Azevedo, D. Kolson, and A. Nicolau. Annotating the java bytecodes in support of optimization. 1997.
- [Ham97] S. Hamilton. New products: Agent-based distributed computing in Java: Voyager agents. *Computer*, 30(5):97–98, May 1997.
- [HCU91] U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented programming languages with polymorphic inline caches. In *ECOOP '91 Conference Proceedings*. Springer Verlag LNCS 512, 1991.
- [Heh76] E. C. R. Hehner. Computer design to minimize memory requirements. *Computer*, 9(8):65–70, August 1976.
- [Hen92] Fritz Henglein. Global tagging optimization by type inference. In *Conference on Lisp and Functional Programming*, pages 205–215. ACM, 1992.

- [Hew77] C. Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, (8):323–364, 1977.
- [Hoe74] L. W. Hoewel. ‘Ideal’ directly executable languages: An analytical argument for emulation. *IEEE Transactions on Computers*, 23(8):759–767, 1974.
- [HU94] U. Hölzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *SIGPLAN ’94, Orlando*, pages 326–336. ACM, 1994.
- [IEE91] IEEE Computer Society, New York. *IEEE Standard for the Scheme Programming Language*, IEEE standard 1178-1990 edition, 1991.
- [Int97] International Standards Organization. *Information Technology—Programming languages, their environments and system software interfaces—Programming language ISLISP*. International Standards Organization, 1997.
- [JW95] S. Jagannathan and A. Wright. Effective flow analysis for avoiding run-time checks. In *Proceedings of the Second International Static Analysis Symposium*, volume 983 of *LNCS*, pages 207–224. Springer-Verlag, 1995.
- [JW96] S. Jagannathan and A. K. Wright. Flow-directed inlining. In *Proceedings of the ACM SIGPLAN ’96 Conference on Programming Language Design and Implementation*, pages 193–205, May 1996.
- [KAJ93] G. Kiczales, J. M. Ashley, and L. H. Rodriguez Jr. Metaobject protocols: Why we want them and what else. In A. Paepcke, editor, *Object-Oriented Programming: the CLOS Perspective*, chapter 14, pages 101–118. MIT Press, Cambridge, Mass., 1993.
- [KdRB91] G. Kiczales, J. des Rivières, and D. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, Massachusetts, 1991.
- [KF93] A. Kind and H. Friedrich. A practical approach to type inference for EuLisp. *Lisp and Symbolic Computation*, 6(1/2):159–176, 1993.
- [KJ93] G. Kiczales and L. H. Rodriguez Jr. Efficient method dispatch in PCL. In A. Paepcke, editor, *Object-Oriented Programming: the CLOS Perspective*, chapter 14, pages 335–348. MIT Press, Cambridge, Mass., 1993.

- [Kli81] P. Klint. Interpretation techniques. *Software—Practice and Experience*, 11(9):963–973, September 1981.
- [KP98] A. Kind and J. Padget. Multi-lingual threading. In *Euromicro Workshop on Parallel and Distributed Processing, Madrid*, pages 431–437. IEEE Computer Society, 1998.
- [KR94] R. Kelsey and J. Rees. A tractable Scheme implementation. *Lisp and Symbolic Computation*, 7(2):315–335, 1994.
- [Kra83] G. Krasner, editor. *Smalltalk-80: Bits of History, Words of Advice*. Addison-Wesley, Reading, 1983.
- [KS86] R. M. Keller and M. R. Sleep. Applicative caching. *ACM Transactions on Programming Languages and Systems*, 8(1):88–108, 1986.
- [LV97] R. Laddaga and J. Veitch. Dynamic object technology. *Communications of the ACM*, 40(5):36–38, May 1997.
- [LY96] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.
- [Mae87] P. Maes. Concepts and experiments in computational reflection. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 147–155. ACM Press, December 1987.
- [McC59] J. McCarthy. Programs in LISP. Report A. I. MEMO 12, M.I.T., RLE and MIT Computation Center, Cambridge, Massachusetts, May 1959.
- [Me92] R. A. MacLachlan (editor). *CMU Common Lisp User’s Manual*. Department of Computer Science, Carnegie-Mellon University, July 92.
- [Mic68] D. Michie. Memo functions and machine learning. *Nature*, 218:19–22, Apr 1968.
- [Moo86] D. A. Moon. Object-oriented programming with Flavors. *ACM SIGPLAN Notices*, 21(11):1–8, November 1986. OOPSLA ’86 Conference Proceedings, Norman Meyrowitz (editor), September 1986, Portland, Oregon.
- [NAJ⁺91] K. V. Nori, U. Ammann, K. Jensen, H. H. Nageli, and C. Jacobi. Pascal-P implemenation notes. In D. W. Barron, editor, *Pascal—The Language and its Implementation*, pages 125–170. Wiley & Sons, Ltd., 1991.

- [Nas92] I. Nassi. *in: Dylan—An Object-Oriented Dynamic Language*. Apple Computer, 1992.
- [Nor93] A. C. Norman. Compact delivery support for REDUCE. *Lecture Notes in Computer Science*, 722:331, 1993.
- [NWM93] J. R. Nicol, C. Th. Wilkes, and F. A. Manola. Object orientation in heterogeneous distributed computing systems. *Computer*, 26(6):57–67, June 1993.
- [OP93] M. H. Odeh and J. A. Padget. Object-oriented execution of OPS5 production systems. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 178–190. ACM, 1993.
- [Ous94] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison Wesley, Reading Massachusetts, 4 edition, 1994.
- [PC94] J. Plevyak and A. A. Chien. Precise concrete type inference for object-oriented languages. *ACM SIGPLAN Notices*, 29(10):324–340, 1994.
- [PCC⁺86] J. Padget, J. Chailloux, Th. Christaller, R. deMantaras, J. Dalton, M. Devin, J. F. Fitch, T. Krummnack, E. Neidl, E. Papon, S. Pope, Ch. Queinnec, L. Steels, and H. Stoyan. Desiderata for the standardization of LISP. In *Proceedings of the ACM Conference on LISP and Functional Programming*, pages 54–66, August 1986.
- [PE92] J. Padget and G. Nuyens (Eds.). *The EuLisp Definition*. Version 0.99; available from <ftp://ftp.maths.bath.ac.uk>, 1992.
- [Phi97] R. E. Phillips. Dynamic objects for engineering automation. *Communications of the ACM*, 40(5):59–65, May 1997.
- [Pit87] T. Pittman. Two-level hybrid interpreter/native code execution for combined space-time program efficiency. In *Proceedings SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, pages 150–152. ACM, June 1987. Also available as SIGPLAN Notices 22(7) July 1987.
- [PK98] J. Padget and A. Kind. A tunable architecture for delivering interpreted programs. In S.A. Cerri and C. Queinnec, editors, *Actes de JFLA98—Journées Francophones des Langages Applicatifs*, number 17 in Collection Didactique, pages 117–139. INRIA, 1998.

- [PL91] B. A. Pearlmutter and K. J. Lang. The implementation of Oaklisp. In P. Lee, editor, *Topics in Advanced Language Implementation*. The MIT Press, Cambridge, 1991.
- [PNB93] J. Padget, G. Nuyens, and H. Bretthauer. An overview of EuLisp. *Lisp and Symbolic Computation*, 6(1/2):9–98, August 1993.
- [Que95] Ch. Queinnec. Fast and compact dispatching for dynamic object-oriented languages. Technical Report 10, École Polytechnique, Project ICSLA, 1995.
- [Que96] Ch. Queinnec. *Lisp In Small Pieces*. Cambridge University Press, Cambridge, UK, 1996.
- [RNSP97] J. A. Rodríguez, P. Noriega, C. Sierra, and J. A. Padget. FM96.5 A Java-based Electronic Auction House. In *Second International Conference on The Practical Application of Intelligent Agents and Multi-Agent Technology: PAAM'97*, 1997.
- [SB86] M. Stefik and D. G. Bobrow. Object oriented programming: Themes and variations. *The AI Magazine*, 6(4):40–62, 1986.
- [SG93] G. L. Steele, Jr. and R. P. Gabriel. The evolution of lisp. In *History of Programming Languages Conference*, volume 28, pages 231–270, March 1993.
- [SH87] P. Steenkiste and J. Hennessy. Tags and type checking in Lisp: Hardware and software approaches. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 1987.
- [Sha95] Y.-P. Shan. Introduction: Smalltalk on the rise. *Communications of the ACM*, 38(10):102–104, October 1995.
- [Sha96] A. L. M. Shalit. *Dylan Reference Manual*. Addison-Wesley, 1996.
- [Shi91a] O. Shivers. Data-flow analysis and type recovery in Scheme. In P. Lee, editor, *Topics in Advanced Language Implementation*, chapter 3, pages 47–87. The MIT Press, 1991.
- [Shi91b] O. G. Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda*. PhD thesis, Carnige-Mellon Univeristy, May 1991.

- [Shi97] O. Shivers. Automatic management of operating-system resources. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, pages 274–279, 9–11 June 1997.
- [Shr96] H. Shrobe. Why the AI community still needs Lisp. *IEEE Expert Online*, February 1996.
- [Sim97] R. Simmonds. Personal communication. 1997.
- [Smi84] B. C. Smith. Reflection and semantics in Lisp. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 23–35. ACM, January 1984.
- [Sny87] A. Snyder. Inheritance and the development of encapsulated software components. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*. MIT Press, 1987.
- [Sri88] S. Sridhar. Configuring stand-alone Smalltalk-80 applications. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 95–104, November 1988. Also published as ACM SIGPLAN Notices, 23(11).
- [Sta92] R. M. Stallman. *Using and Porting GNU CC*. Free Software Foundation, Inc., December 1992.
- [Ste84] G. L. Steele, Jr. *Common Lisp: The Language*. Digital Press, first edition, 1984.
- [Ste90] G. L. Steele Jr. *Common Lisp: The Language*. Digital Press and Prentice-Hall, second edition, 1990.
- [Str93] B. Stroustrup. A history of C++: 1979-1991. In *ACM SIGPLAN HOPL-II. 2nd ACM SIGPLAN History of Programming Languages Conference (Preprints)*, volume 28, pages 271–297. ACM Press, March 1993.
- [Sun93] SunSoft. *SunOS 5.3 Linker and Libraries Manual*, 1993.
- [SW95] M. Serrano and P. Weis. Bigloo: a portable and optimizing compiler for strict functional languages. *Lecture Notes in Computer Science*, 983:366, 1995.

- [TLA92] D. Tarditi, P. Lee, and A. Acharya. No assembly required: compiling standard ML to C. *ACM Letters on Programming Languages and Systems*, 1(2):161–177, June 1992.
- [Way95] P. Wayner. *Agents Unleashed: A Public Domain Look at Agent Technology*. Academic Press: London, 1995.
- [WC94] A. K. Wright and R. Cartwright. A practical soft type system for Scheme. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pages 250–262, 1994.
- [WDH89] M. Weiser, A. Demers, and C. Hauser. The portable common runtime approach to interoperability. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 114–122, December 1989. Also published in *ACM Operating Systems Review* 23(5).
- [Wei97] K. Weihe. Reuse of algorithms: Still a challenge to object-oriented programming. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 34–48. ACM Press, October 1997.
- [Wil95] P. R. Wilson. Garbage collection. *Computing Surveys*, 1995.
- [Wil97] A. Wilson. The Java Native Method Interface and Windows. *Dr. Dobb's Journal of Software Tools*, 22(8):46–50, August 1997.
- [WS91] L. Wall and R. L. Schwartz. *Programming Perl*. O'Reilly Associates, Inc., 1991.
- [WY88] T. Watanabe and A. Yonezawa. Reflection in an Object-Oriented Concurrent Language. In *Proceedings of the OOPSLA '88 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 306–315, November 1988. Published as ACM SIGPLAN Notices, volume 23, number 11.
- [Zor93] B. G. Zorn. The measured cost of conservative garbage collection. *Software—Practice and Experience*, 23(7):733–756, July 1993.

Appendix A

Assembler Code

A.1 Virtual Instruction Transfer on P5

```
.L4:      movzbl (%eax),%edx      # get instruction
          cmpl $255,%edx    # range check
          ja .L4           # default jump
          jmp *.L262(,%edx,4) # jump table lookup and jump to next
                               # instruction
.L262:   .long .L260       # jump table
          ...
.L260:   incl %eax        # increment pc
          jmp .L4         # loop jump
```

Figure A-1: *Virtual instruction transfer with switch on P5 (gcc -O2)*

```
.L18:   movl (%edx),%eax    # get label address
          addl $4,%edx     # increment pc
          jmp *%eax       # jump to next instruction
```

Figure A-2: *Virtual instruction transfer with code threading on P5 (gcc -O2)*

```
.L18:   incl %edx        # increment pc
          movzbl (%edx),%eax # get label offset
          jmp *-1024(%ebp,%eax,4) # jump to next instruction
```

Figure A-3: *Virtual instruction transfer with indexed code threading on P5 (gcc -O2)*

A.2 Virtual Instruction Transfer on MIPS

```

$L4:      lbu $3,0($4)           # get instruction
          #nop
          sltu $2,$3,256       # range check
$L265:    beq $2,$0,$L265      # default jump
          sll $2,$3,2          # multiply by 4
          lw $2,$L262($2)      # jump table lookup
          #nop
          j $2                 # jump to next instruction
$L262:    .gpword $L260        # jump table
          ...
$L260:    j $L4                # loop jump
          addu $4,$4,1         # increment pc

```

Figure A-4: *Virtual instruction transfer with switch on MIPS (gcc -O2)*

```

$L75:     lw $2,0($3)          # get label address
          #nop
          j $2                 # jump to next instruction
          addu $3,$3,4         # increment pc

```

Figure A-5: *Virtual instruction transfer with code threading on MIPS (gcc -O2)*

```

$L8:      addu $6,$6,1         # increment pc
          lbu $2,0($6)        # put pc in $2
          #nop
          sll $2,$2,2         # multiply by 4
          addu $2,$sp,$2      # add label table offset
          lw $2,8($2)         # get instruction address
          #nop
          j $2                 # jump to next instruction

```

Figure A-6: *Virtual instruction transfer with indexed code threading on MIPS (gcc -O2)*

A.3 Virtual Instruction Transfer on SPARC

```
.LL4:      ldub [%o0],%g2      # get instruction
           cmp %g2,255      # range check
.LL265:    bgu .LL265       # default jump
           nop
           sll %g2,2,%g2    # multiply by 4
           ld [%g2+%g3],%g2 # jump table lookup
           jmp %g2          # jump to next instruction
           nop
.LL262:    .word .LL260     # jump table
           ...
.LL260:    b .LL4          # loop jump
           add %o0,1,%o0   # increment pc
```

Figure A-7: *Virtual instruction transfer with switch on SPARC (gcc -O2)*

```
.LL147:   ld [%g3],%g2     # get label address
           jmp %g2        # jump to next instruction
           add %g3,4,%g3  # increment pc
```

Figure A-8: *Virtual instruction transfer with code threading on SPARC (gcc -O2)*

```
.LL9:     add %i0,1,%i0    # increment pc
           ldub [%i0],%o0  # put pc in %o0
           sll %o0,2,%o0   # multiply by 4
           add %fp,%o0,%o0 # add label table offset
           ld [%o0-1040],%o0 # get instruction address
           jmp %o0        # jump to next instruction
           nop
```

Figure A-9: *Virtual instruction transfer with indexed code threading on SPARC (gcc -O2)*

Appendix B

Various Tables

	Type	Processor	RAM	Data Cache	Instr Cache
P5	PC	150MHz P150	32M	16K+512K	16K
MIPS	SGI Indy	100Mhz MIPS R4600	64M	16K	16K
SPARC	SUN SS100E	4x50MHz SPARC 4d	192M	16K	32K

Table B.1: *Architectures*

arity	dynamic ratio
1	0.315350
2	0.536946
3	0.105638
> 3	0.042066

Table B.2: *Generic function arity ratio in OPS5 implemented in EuLisp*

Program	Stress
arith0	integer arithmetic
arith1	float arithmetic
hanoi	slot access
mem	memory management
meth	method invocation
nfib	recursion, integer arithmetic
rec	recursion
tak	recursion, integer arithmetic
takl	list processing, recursion
vec	vector access

Table B.3: *Benchmark programs*

flag	description
-help	show usage
-load_path <dir>	add <dir> to load path
-c	create C linkable module file only
-ar	create C linkable library file
-l <lib>	specify C linkable library
-L <dir>	extent C linkable library load path
-fff <file>	specify C foreign function file
-ffl <lib>	specify C foreign function library
-o <file>	destination file
-script	script mode
-no_gc	garbage collection library not linked
-cc <compiler>	used C compiler
-ld <linker>	used C linker
-ar_cmd <cmd>	used C ar command
-ranlib_cmd <cmd>	used C ranlib command
-cflags <flag>	additional C compiler flag
-static	no shared libraries used
-g	C debug info
-i	force interpretation mode

Table B.4: *youtoo flags (extract)*

Lisp	C
<character>	char
<int>	int
<double>	double
<string>	char *
boolean	int
ptr	void *
<int*>	int *
<double*>	double *
<string*>	char **

Table B.5: *Foreign function converters*

Appendix C

Measurements

P5 gcc	threaded			switched		
	random	custom	optimal	random	custom	optimal
arith0	9.61s	6.96s	6.96s	8.31s	8.43s	8.22s
arith1	7.90s	6.78s	7.05s	8.30s	7.89s	7.66s
hanoi	6.24s	4.88s	5.08s	6.17s	5.85s	5.31s
mem	6.18s	5.95s	5.94s	6.02s	5.97s	5.99s
meth	10.19s	7.78s	7.89s	9.87s	11.83s	9.58s
nfib	14.29s	10.22s	9.38s	11.44s	11.70s	11.29s
rec	19.04s	17.98s	18.40s	22.83s	21.68s	21.11s
tak	13.14s	9.38s	9.70s	11.35s	12.07s	11.30s
takl	10.30s	7.94s	7.94s	9.15s	9.78s	10.34s
vec	8.94s	5.93s	5.46s	6.76s	6.73s	7.72s

Table C.1: *Instruction ordering and threaded dispatch on P5*

MIPS gcc	threaded			switched		
	random	custom	optimal	random	custom	optimal
arith0	16.58s	9.95s	11.63s	14.59s	13.11s	13.22s
arith1	16.23s	13.50s	14.19s	14.91s	14.72s	14.97s
hanoi	12.97s	9.62s	8.16s	9.25s	9.15s	10.15s
mem	15.16s	14.40s	14.52s	14.50s	14.20s	14.45s
meth	16.57s	13.29s	13.45s	18.40s	16.91s	18.49s
nfib	15.86s	15.43s	15.56s	20.91s	20.05s	19.96s
rec	28.84s	28.00s	28.20s	37.54s	36.27s	36.75s
tak	16.59s	15.50s	15.45s	20.26s	19.86s	19.93s
takl	14.66s	13.57s	13.71s	17.67s	16.90s	17.41s
vec	12.83s	10.23s	9.16s	11.45s	11.29s	11.34s

Table C.2: *Instruction ordering and threaded dispatch on MIPS*

SPARC gcc	threaded			switched		
	random	custom	optimal	random	custom	optimal
arith0	14.13s	14.14s	13.77s	14.00s	17.70s	13.93s
arith1	15.67s	13.63s	13.43s	14.33s	16.69s	13.50s
hanoi	11.53s	11.30s	10.93s	11.70s	11.86s	11.10s
mem	13.20s	14.20s	12.80s	12.06s	13.23s	13.10s
meth	18.98s	16.86s	20.93s	19.00s	22.07s	16.57s
nfib	25.39s	21.33s	21.40s	29.73s	28.17s	21.30s
rec	40.53s	40.46s	40.03s	40.49s	50.17s	40.20s
tak	22.93s	22.53s	22.76s	45.43s	33.06s	25.29s
takl	20.57s	20.36s	26.17s	20.47s	25.83s	26.26s
vec	13.10s	12.90s	12.79s	12.89s	15.67s	12.56s

Table C.3: *Instruction ordering and threaded dispatch on SPARC*

code	position	invocation	name
27	1	0.12647739	STACK_REFO
31	2	0.11775139	STACK_REF
28	3	0.09931751	STACK_REF1
68	4	0.07262022	BRANCH_NIL_POS
29	5	0.05743256	STACK_REF2
36	6	0.05520266	BINDING_REF
69	7	0.02998005	RETURN
60	8	0.02931382	CALL_OPERATOR
26	9	0.02794180	FPI_LT
54	10	0.02538586	BRANCH_POS
171	11	0.02368542	CHECK_ARGUMENTS2
61	12	0.02332673	TAIL_CALL_OPERATOR
71	13	0.01895022	DISPLAY_REF
130	14	0.01887169	STATIC_REFO
18	15	0.01838423	NULLP
34	16	0.01729920	NOBBLE
67	17	0.01707753	CHECK_ARGUMENTS
17	18	0.01700065	THE_CDR
21	19	0.01443220	FPI_DIFFERENCE
2	20	0.01355292	PRIMITIVE_REF
72	21	0.01322283	SET_DISPLAY_REF
42	22	0.01243134	POP1
131	23	0.01236767	STATIC_REF1
20	24	0.01143196	FPI_SUM
35	25	0.01110556	STATIC_REF
170	26	0.01089714	CHECK_ARGUMENTS1
132	27	0.01080677	STATIC_REF2
44	28	0.00994132	FPI_DEC
134	29	0.00851682	STATIC_REF_NIL
25	30	0.00837974	FPI_EQUAL
23	31	0.00753782	FPI_QUOTIENT
22	32	0.00753782	FPI_PRODUCT
45	33	0.00718992	FPI_ZEROP
16	34	0.00650257	THE_CAR
138	35	0.00589914	STATIC_FPI_BYTE_REF
3	36	0.00572763	SET_PRIMITIVE_REF
43	37	0.00375411	FPI_INC
70	38	0.00367226	ALLOC
59	39	0.00356534	MAKE_LAMBDA
38	40	0.00247272	STATIC_FPI_REF
80	41	0.00201856	EQ
107	42	0.00138637	INIQ
6	43	0.00109182	PRIMITIVE_SIZE
135	44	0.00104827	STATIC_REF_T
4	45	0.00083096	PRIMITIVE_CLASS_OF

Table C.4: *Optimal virtual instruction ordering with programs in Table B.3*

MIPS/ELF module file	size	with const			without const		
		.rodata	.data	ratio	.rodata	data	ratio
boot.o	10544	2656	2848	0.93	816	4704	0.17
boot1.o	15280	2864	3040	0.94	2080	3808	0.55
callback.o	3984	1248	944	1.32	736	1440	0.51
character.o	7760	2096	976	2.15	1424	1648	0.86
collect.o	6512	2288	1088	2.10	624	2752	0.23
compare.o	4096	1056	1040	1.02	272	1824	0.15
condition.o	5712	1552	1408	1.10	640	2320	0.28
convert.o	800	144	144	1.00	96	192	0.50
convert1.o	8608	3168	1952	1.62	960	4176	0.23
copy.o	3872	1168	912	1.28	384	1696	0.23
dynamic.o	3184	832	816	1.02	352	1296	0.27
event.o	1376	288	272	1.06	160	400	0.40
float.o	3152	960	560	1.71	336	1184	0.28
format.o	3888	1360	800	1.70	368	1792	0.21
fpi.o	5328	1488	1136	1.31	576	2048	0.28
handler.o	15136	4560	1936	2.36	2544	3936	0.65
integer.o	2160	496	464	1.07	256	720	0.36
let-cc.o	1440	336	320	1.05	128	528	0.24
levell.o	1728	384	304	1.26	96	592	0.16
list.o	17424	4928	4656	1.06	1232	8336	0.15
lock.o	4048	1088	864	1.26	432	1520	0.28
mop-access.o	8672	3136	1952	1.61	1104	3984	0.28
mop-alloc.o	9728	3584	2112	1.70	1232	4464	0.28
mop-class.o	22160	4272	5888	0.73	2640	7520	0.35
mop-defcl.o	10112	2656	2720	0.98	912	4448	0.21
mop-gf.o	7408	2224	1808	1.23	832	3200	0.26
mop-init.o	5104	2880	576	5.00	400	3056	0.13
mop-inspect.o	3984	960	1056	0.91	384	1632	0.24
mop-key.o	1408	352	288	1.22	176	448	0.39
mop-meth.o	7312	2688	1616	1.66	928	3376	0.27
mop-prim.o	1888	336	480	0.70	240	576	0.42
number.o	7760	2128	1824	1.17	592	3360	0.18
read.o	10352	3600	2448	1.47	912	5136	0.18
socket.o	8912	2304	2192	1.05	864	3632	0.24
stream.o	12416	4128	3120	1.32	960	6288	0.15
stream1.o	13616	4032	320	12.60	3872	496	7.81
stream2.o	19968	5648	4736	1.19	2240	8144	0.28
stream3.o	8640	3088	2112	1.46	752	4432	0.17
string.o	14912	4336	2672	1.62	2048	4944	0.41
symbol.o	4512	1184	1088	1.09	368	1904	0.19
table.o	11920	3824	2704	1.41	1216	5296	0.23
table1.o	10640	2976	2528	1.18	1152	4352	0.26
telos.o	976	208	160	1.30	96	272	0.35
thread.o	15824	4128	4016	1.03	1792	6352	0.28
vector.o	17040	5248	4352	1.21	1344	8272	0.16
total	361296	104880	79248	1.32	41568	142496	0.29

Table C.5: Impact of constant virtual machine code on EuLisp level-1 modules on MIPS