OPTIMIZATION OF OBJECT-ORIENTED AND CONCURRENT PROGRAMS

BY

JOHN BRADLEY PLEVYAK

B.A., University of California, Berkeley, 1988

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1996

Urbana, Illinois

Abstract

High level programming language features have long been seen as improving programmer efficiency at some cost in program efficiency. When features such as object-orientation and fine-grained concurrency, which greatly simplify expression of complex programs, are used parsimoniously, their effectiveness is mitigated. It is my thesis that these features can be implemented efficiently through interprocedural analysis and transformation. By specializing their implementation to the contexts in which they are used, the program's efficiency is not adversely affected by the flexibility of the language. The specific contributions herein are: 1) an adaptive flow analysis for practical precise analysis of object-oriented programs, 2) a cloning algorithm for building specialized versions of general abstractions, 3) a set of optimizations for removing object-oriented and fine-grained concurrency overhead, and 4) a hybrid sequential-parallel execution model which adapts to the availability of data. The effectiveness of this framework has been empirically validated on standard benchmarks. It is publicly available as part of the Illinois Concert system (http://www-csag.cs.uiuc.edu).

To Cindy,

Acknowledgments

I would like to thank everyone at the University of Illinois at Urbana-Champaign who made this possible. I would especially like to thank Vijay Karamcheti, my officemate, colleague and friend, for his help, support, knowledge and patience; Xingbin Zhang for putting up with my tirades about the "right way" to build a compiler; Julian Dolby with whom I shared many a thought and cup of coffee; Scott Pakin and the rest of the Concurrent Systems Architecture group for many interesting discussions, some technical; and my advisor, Andrew Chien, for information, help, time and giving me the opportunity to pursue so many fascinating directions. I would like to thank my committee for appreciating this work, and Uday Reddy for conversations and his patience with my lack of proclivity for rigor.

Life in Chambana would have been unbearable if it were not for my friends there. Especially, Keith, keeper of the box, George, towny, Von, Bridget, Chris, Steve, Mahesh, Tara, Ulrike, Jeff, Igor and Ellen, Bob and the rest of the Illinois Sailing Club. I also want to mention The Inn, The Blind Pig, Thats Rentertainment, the several Korean restaurants and trips to the BVI as havens and escapes from a dull Midwest corn and cow town.

I would like to thank my dear friends in California for their support while I have been away pursuing this work. Especially, Joe, Patty, Chris and Anita for putting me up when I came to visit; Kian and Aaron for coming to visit me in this wasteland; John, Craig, Greg, Deedee, Lily, Mark, Tracy, Dan, Gina, for many a past and future day of fun; Kelly, Simon and Justin, for reminding me of what is important; Kevin, Kari, Terry, Linda, the Ashlands crew, and all the rest.

I would like to thank my family, for their support, understanding and patience: Dave and Cameron, Candy and Skip, and my dear mother and father. Words cannot express.

And most of all I would like to thank Cindy, my true love.

Preface

> The reasonable man adapts himself to the world; the unreasonable one persists in trying to adapt the world to himself. Therefore all progress depends on the unreasonable man.
>
> *George Bernard Shaw*

This thesis had both a conception and a birth. The conception occurred when I discovered Smalltalk in 1981. A thought was planted that expressiveness could be joined with simplicity and clarity. However, being unreasonable, I desired efficiency and concurrency as well. The birth occurred in 1992, when I began work on what would later be the Concert project, whose purpose was to combine the simplicity of Smalltalk with the pervasive concurrency of Actors, and yet automagically be as efficient as FORTRAN on distributed memory MIMD machines. At the time, the best pure object-oriented systems were several times slower than C, and shared-memory vector and SIMD machines reigned supreme. Today, concurrent object-oriented programming has been popularized and networks of workstations (NOWs) are the rage. Today, the sequential efficiency of the Concert system matches C, exceeds that of C++, and the parallel efficiency approaches FORTRAN with message passing.

This thesis is organized into four parts. The first part is background (Chapters 2 and 3). If you are already familiar with object-oriented and concurrent object-oriented concepts, you may still wish to read the summaries (Sections 2.5 and 3.7) which define some useful terms. The second part describes the compilation framework in general (Chapter 4) and the Illinois Concert system in particular. The third part comprises the body of this thesis. For readers interested in particular analyses or transformations, I have tried to make Chapters 5 through 8 relatively independent by including some background information and references back to the pertinent earlier parts of this thesis. Finally, part four (Chapter 9) discusses adaptive sequential-parallel execution.

Table of Contents

Chapter

List of Figures

List of Tables

Chapter 1

Introduction

> The fact is, that civilization requires slaves. The Greeks were quite right here. Unless there are slaves to do the ugly, horrible, uninteresting work, culture and contemplation become almost impossible. Human slavery is wrong, insecure, demoralizing. On mechanical slavery, on the slavery of the machine, the future of the world depends.
>
> *Oscar Wilde. The Soul of Man Under Socialism, 1895*

The key to constructing large software systems is to write and reason through abstractions, leaving the details of the implementation to the compiler. This follows from what I call the programmer's uncertainty principle. This principle holds that when the limits of human understanding are reached something must go, either the big picture or the small picture. Object-orientation and fine-grain concurrency are tools for reasoning about the big picture. This thesis is about automating the details, the small picture.

Object-oriented programming (OOP) is the byword of software engineering; promising to increase productivity through abstraction and software reuse. Concurrent object-oriented programming (COOP) applies those tools to parallelism and distribution, with applications from supercomputing to web browsers. It is an axiom of this thesis that object-oriented programming and fine-grained concurrency are "The Right Thing" [75]. Instead of belaboring the point, I defer to the references provided at the end of this thesis, the reader's intuition and the wisdom of future generations.

OOP and COOP produce programs with structure quite different from standard procedural codes, but with the same high demands for efficiency. Traditional local optimization techniques are poorly suited to the dynamic nature of OOP and COOP codes. The abstractions which free

programmers from implementation details, hide information from compilers, and can result in poor performance.

It is my thesis that:

> Object-oriented programming and concurrent object-oriented programming can be made efficient through interprocedural analysis and transformation.

That is, programs written in natural OOP and COOP style can be made as fast as the equivalent programs written in conventional languages and styles (i.e. procedural C).

The core of this work is the recognition that as programs pass from abstract descriptions of potential behavior to concrete events more information becomes available about what *might* happen until the potential collapses into a singularity of what *did* happen. The key to building an efficient computation is to take advantage of this information as soon as it becomes available when transformation is cheaper. For example, knowing the type, size and location of data and the set of operations to perform enables the compiler to statically schedule the computer's resources for maximum efficiency. If some information is not known, for example, the size of the data, some scheduling must be done dynamically which is generally more costly; and the less information available, the less efficient the computation.

This emphasis on information leads to an optimization framework based on four elements:

analysis The discovery of information by examination of the program text. As in real life, much interesting information is predicated: if **A** is true then **B** is true.

specialization The optimization of a program section for particular conditions (e.g. **A** is true); may involve replicating the section for several different conditions.

speculation The testing of a property (e.g. is **A** true?); followed by specialization.

adaptation Modification of behavior based on accumulated information.

Analysis is capable of determining information early in the program's life cycle when it can be applied with maximum impact; however, this information is often predicated or incomplete. For example, analysis might indicated that at a particular program point a piece of data might have one of three values (**a**,**b**, and **c**). Early in the program's execution the data might take on one of the values **a** later taking on the others. Moreover, the analysis might have additional

information for the case *if* the piece of data has the value **a**. How does analysis determine this information, and what good is it? The answer to the first question is found in Chapter 5, which contains a discussion of adaptive flow analysis. Essentially, this analysis symbolically executes the program, then examines the information obtained, adapts itself to answer questions about things it did not resolve, and reanalyzes. The information obtained by analysis is used throughout compilation and execution.

One use for the information produced by analysis is to specialize portions of the program for different situations. In describing the desired program behavior, object-oriented programmers use general abstractions, such as sets of objects, again and again in a variety of circumstances. However, the very flexibility which makes these abstractions useful under many conditions can make them inefficient in particular situations. By automatically building specialized versions of the general abstraction for those cases which are performance critical, both flexible expression and efficient implementation can be achieved.

In programs the operations performed often depend on the values of the input data. Therefore, it is not possible to know everything about the execution of the program from the program text. However, analysis can often delineate the range of possible behavior, enabling to building of specialized code for different possibilities. These specialized versions are based on speculations which must be verified before they are actually executed, typically by testing at run time. These tests can be costly, so careful management of speculative information is important for overall efficiency.

Finally, run time information can be used to adapt the implementation of the program for higher efficiency. For example, a portion of the program which often deals with data stored on distant parts of the machine should overlap remote access with other operations. On the other hand, parts of the program which use local data should proceed sequentially through their operations, eliminating the overhead of juggling outstanding remote requests for data. This situation is analogous to prescribing a specialized training regimen to an athlete with special needs. In this case, the running program adapts itself to the location of data.

These four elements, analysis, specialization, speculation and adaptation, are useful, in turn, as the program passes from abstract description to concrete events. Analysis involves no run time cost, but often provides only predicated information. Specialization can improve efficiency, but may involve an increase in the size of the program as it replicates code for the

different predicated conditions. Speculation involves potentially costly run time tests, but it can determine information unavailable at compile time. Finally, the program can adapt to behavioral trends in behavior by collecting run time information and modifying its activities accordingly.

These elements are the central concepts in this thesis, and they appear both as the central concept of their own chapters: analysis (Chapter 5), specialization (Chapter 6), speculation (Chapter 8) and adaptation (Chapter 9), as well as in combination and in supporting roles in these and other chapters. The contents of these chapters are summarized in Section 1.1 below. The contributions represented by this work in relation to the state of art are described in Section 1.4, and the overall organization of this thesis is discussed in Section 1.5

1.1 Thesis Summary

This thesis describes a optimization framework containing novel techniques for context-sensitive analysis, specialization of abstractions, speculative optimization and adaptive execution. It begins by describing object-orientated and fine-grained concurrent programming models, introducing terms and providing a basis for understanding the difficulty of producing efficient implementations for such programs. Then it describes the execution model, through which the program is mapped to the hardware. Next it describes the compilation framework which performs this mapping. This framework consists of a new context-sensitive analysis which is both precise and practical, a new cloning algorithm for constructing specialized versions of abstractions, and a collection of individual optimizations for object-oriented and fine-grained concurrent programs. Finally, a new hybrid sequential-parallel execution enables the program to adapt to the location of data at run time.

Programming Model

The programming model is a simple pure object-oriented programming model [76] with objects (data), methods (operations), and classes which link them. It makes no distinctions are made between primitive types (e.g. integers) and user defined types (e.g. Set), and requires no type declarations, prototypes, or canonical hints (e.g. `inline`). To this object-oriented model, fine-

grained concurrency is added in a manner consistent with the principles of simplicity and abstraction. The concurrency model has three key features:

- a shared name space,
- dynamic thread creation, and
- object level access control.

A shared namespace allows programmers to separate data layout and functional correctness. Dynamic thread creation allows programmers to express the natural concurrency of the application, leaving the system to map it to the underlying machine, and object-level access control provides a basic mutual exclusion mechanism which can be used to construct larger atomic operations and synchronization structures.

Execution Model

The execution model is based on a set of conventional single-threaded processing elements with a memory hierarchy where some memory is less costly to access (i.e. "local"). In general, only objects local to a processing element are operated on directly. The system synthesizes the global namespace of the programming model by detecting and mapping operations on "remote" objects to communication between processing elements. Concurrency in the programming model is achieved by multithreading the processing elements in software and parallel execution across nodes. Thus, each processing element can be viewed as a sequential machine augmented with runtime primitives for naming, locking, location, and concurrency control. This model supports existing massively parallel processors [173, 51] and networks of workstations [12].

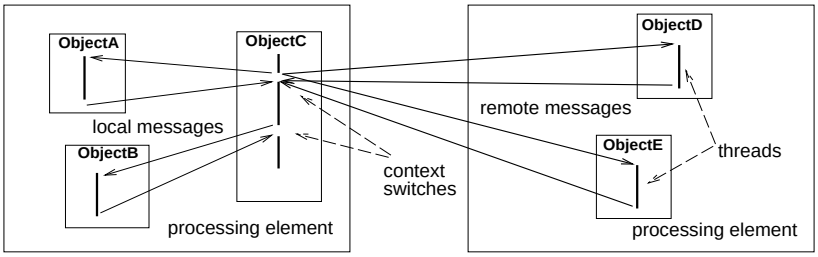


Figure 1.1: Execution Model Example

Each object has a global name, a set of locks to implement access control, and a queue for ready and suspended threads which store their state in *contexts*, heap-allocated activation records. When a message is sent to an object, a *future* is created representing the return value and a thread is started to compute the future value. When the return value is required by the initial thread, the future is *touched* and the thread suspends until the value is present. Thus, a logical thread may split then rejoin or seem to migrate from processor to processor as in Figure 1.1.

Compilation Framework

The compilation framework starts with a new interprocedural context-sensitive flow analysis which breaks through abstraction barriers. Classes and functions are then replicated (cloned) for the contexts in which they are used. An iterative cloning algorithm rebuilds the cloned call graph using a modified dispatch mechanism. Using the information fixed by cloning, classes and functions are specialized removing much of the overhead of object-orientation: instance (member) and local variables are unboxed, methods (virtual functions) are statically bound and inlining is performed speculatively based on the class of objects or function pointers for OOP and location or lock availability for COOP. Other optimizations include conversion of member and global variables to locals, lifting and merging of access regions, removal of redundant lock and locality check operations, and redundant array operation removal.

Analysis

The flow analysis is a new context sensitive interprocedural analysis which adapts to the structure of the program to efficiently derive information at a cost proportional to the precision of the information obtained. Flow analysis of object-oriented programs is complicated by the interaction of data values and control flow information through dynamic dispatch and imperative update of instance variables. To address these problems, this analysis combines simultaneous data and control flow analysis with iterative adaptation to the structure of the program. A simple, less control and data flow sensitive analysis is used to determine where more precise analysis is needed. *Contours* are used to summarize stack frames and groups of objects. Adaptation is achieved through the *contour representation* which describes the summarization mapping. It is extended locally to provide more precision and then the program is reanalyzed.

Cloning

Cloning builds specialized versions of classes and methods for optimization purposes. It begins with the results of flow analysis, including the call graph and the set of contours. These contours are partitioned into prototypical clones based on optimization criteria. Object contours are partitioned into concrete types (implementation classes), and method contours are partitioned into method clones. Next, an iterative algorithm is applied which repartitions the contours until the call graph is *realizable*; until the objects can be created of correct concrete types, and the correct clones can be invoked for each invocation site. The standard dynamic dispatch mechanism which selects the desired method based on the selector and class of the target is modified to be context sensitive using an invocation site identifier.

Optimization of Object-oriented Programs

Several sources of inefficiency are evident for object-oriented programs including: abstraction boundaries and polymorphism, small method size, high invocation density, data dependent invocations, data access overhead and potential aliasing. Invocation density is addressed through invocation optimizations: static binding, speculation and inlining. Data access overhead is addressed by unboxing and the elimination of pointer-based accesses through conversion of instance variables to Static Single Assignment (SSA) form. Array alias analysis and a suite of standard low level optimizations are also performed with the resulting implementations as efficient as conventional C implementations.

Optimization of Concurrent Object-oriented Programs

Concurrent object-oriented programs also suffer from inefficiency, in particular, resulting from a lack of information about the location and locking status of objects. Several optimizations address these problems. Lock operations are optimized by taking advantage information provided by the the call graph about when the access rights required by a method have already been acquired when the method is called. Also, analysis can recognize *stateless methods* which do not required access at all. Speculative inlining inserts tests around inlined code. These tests introduce *access regions* in which access to an object has been granted. These regions are transformed to amortize the cost of speculation and to increase potential for conventional op-

timizations. Memory hierarchy traffic is optimized by using flow analysis information and that provided by access regions to cache data at higher levels of the memory hierarchy. Likewise, distributed global variables are optimized by using the call graph to detect *temporally constant* globals and by caching their value. Finally, synchronization of threads is is optimized by careful placement of touches.

Hybrid Execution

The program is implemented using hybrid sequential-parallel execution which enables the program to adapt at run time to the concurrency structure of the program and the location and availability of data. Hybrid execution provides separately optimized sequential and parallel versions of methods. When possible, methods are executed sequentially using FIFO (First In First Out) order on a stack with low overhead. The parallel versions store their local data in persistent heap-based contexts and are specialized for generating parallel work and for hiding the latency of long running and/or non-local operations. One of three sequential calling convention of increasing flexibility is selected automatically for each method based on interprocedural analysis, enabling the use of sophisticated synchronization structures at no cost to other parts of the program.

1.2 The Concert System

The framework and algorithms described in this thesis were developed as part of the Concert project and implemented as part of the Concert system (Section 4.1). The Concert system is a complete programming system for developing high performance concurrent object-oriented programs for execution on large scale parallel machines. In addition to the compiler, which embodies the optimizations in this thesis, there is a runtime specialized for each target platform, an emulator for quick turnaround debugging, a debugger and a standard library. The results reported in this thesis are for implementations produced by the Concert system.

1.3 Results

This thesis demonstrates a number of different results for the various analyses and optimizations. However, this is a real and complete system. Each part of the framework depends on the other parts. For example, the analysis (Chapter 5) alone does not improve the program. Cloning (Section 6) depends on flow analysis, but again does little *in itself* to improve efficiency. The optimizations in Chapters 7 and 8 which do improve efficiency fundamentally depend on flow analysis and cloning. Therefore, the overall performance results are presented in later chapters. Chapter 7 demonstrates that a pure dynamically-typed object-oriented language can obtain the same efficiency as C for a number of standard benchmarks. Moreover, it shows that the same language can be *more* efficient than C++ as compiled with a standard C++ compiler (G++). Likewise, Chapter 8 demonstrates that, for the Livermore Loops, essentially all the overhead of a shared namespace and object-based protection scheme can be eliminated so that the COOP programs are as efficient as C when the data is available. Finally, Chapter 9 shows that fine-grained concurrency can be implemented efficiently with adaptive sequential-parallel execution. Such *hybrid* execution can approach optimal efficiency given by the total work and communication overhead implied by the data layout.

1.4 Contributions

The general contribution of this thesis is an optimization framework for object-oriented and fine-grained concurrent languages. Individual contributions are: 1) an adaptive flow analysis for practical, precise analysis of object-oriented programs, 2) a cloning algorithm for building specialized versions of general abstractions, 3) a set of optimizations for removing object-oriented and fine-grained concurrency overhead, and 4) a hybrid sequential-parallel execution model which adapts to the availability of data.

1. A new adaptive flow analysis [140, 141] which, is a significant extension over previous work. When initially published, it was demonstrated to be more efficient than previous analyses [135], and it was the only analysis for object-oriented programs capable of handling arbitrarily complex type structures. It remains the only practical and demonstrated analysis capable of analyzing both polymorphic methods and polymorphic classes in the presence of imperative update (i.e. real object-oriented programs as opposed to functional or functional object-oriented

programs). The most powerful comparable analysis [2] is limited to polymorphic methods and has not been used for context sensitive optimization (in [3] context sensitive information was summarized before optimization).

2. A new cloning algorithm [142] which represents the first application of whole program cloning to object-oriented programs. It solves several unique problems, including: specialization of classes including data layout, modification of the dispatch mechanism for context sensitivity, and the discover of a realizable call graph. The closest comparable work by Cooper and Hall [85, 84, 48, 86, 49], is directed to handling specialization of FORTRAN based on values of parameters, and is limited to forward data flow problems. In contrast, the new cloning algorithm handles a more general class of data flow problems and includes data structure specialization.

3. OOP and COOP specific optimizations which include several new or substantially new. The optimizations of locks, access regions and touches (Sections 8.1, 8.3 and 8.5) are new. The application of flow analysis and cloning information to the problem of decreasing memory hierarchy traffic is similar to conventional alias analysis, but the use of access region information is new. However, the substantial contribution of this work is the demonstration over a suite of standard programs that the set of optimizations described are *sufficient* to enable a pure dynamically-typed language to match the efficiency of C. Previous systems [28, 30, 95] were several times slower than C.

4. A new hybrid sequential-parallel execution model which differs from its predecessors in that it provides two separately optimized versions of code; one optimized for sequential efficiency and one for parallel efficiency. It also provides a hierarchy of calling conventions of increasing flexibility and cost. These are selected automatically by the system based on the requirements of the code. As a result, hybrid execution is capable of matching the speed of C when the required data is available and yet still provides support for continuation passing (Section 9.2.3) and latency hiding where required.

Details of these contributions appear in Chapters 5; 6; 7 and 8; and 9 respectively.

1.5 Organization

This thesis is organized into four parts. The first part is background material on the OOP and COOP programming and execution models (Chapters 2 and 3). The second part describes the

compilation framework in general (Chapter 4) and the Illinois Concert system in particular. The third part comprises the body. It describes adaptive flow analysis (Chapter 5), cloning (Chapter 6), and optimization of OOP and COOP (Chapters 7 and 8 respectively). Finally, part four (Chapter 9) discusses adaptive sequential-parallel execution.

Chapter 2

Programming Model

The best way to do research is to make a radical assumption and then assume it's true. For me, I use the assumption that object-oriented programming is the way to go.

Bill Joy

This chapter describes objected-oriented and fine-grained concurrent programming concepts, terminology and languages. It is not intended to be a comprehensive discussion of these topics, but rather to define clearly the terms used in this thesis. Thus, the focus is on features *as they affect optimization and ultimately performance*. Three concurrent object-oriented languages are discussed. Two, Concurrent Aggregates (CA) and Illinois Concert C++ (ICC++) are supported by the Concert system. The last is the simple pedagogical language used in examples throughout this thesis.

2.1 Object-Orientation

Object-oriented programming is characterized by information hiding and reuse of specifications.[1] Object-oriented programming languages contain special features for these purposes, including abstract data types for encapsulation, polymorphism for specification over abstractions, and inheritance for successive refinement of specifications. These language features in turn are supported by implementation techniques which are discussed in Section 3.

[1]For want of a better term, a *specification* is a part of a program which may be incomplete by itself (e.g. an abstract class, C++ template or generic function).

2.1.1 Abstract Data Types

The objective of abstract data types is to encapsulate information about the implementation of an object. They describe an interface consisting of a set of operations. These operations are abstract in the sense that their implementations are opaque; they are described by the information they require and produce, but not by the steps they perform. For example, compare the two C++ [68] stack classes in Figure 2.1. The class on the left represents an abstract data type. It describes only the abstract operations `push()` and `pop()` (the parentheses indicate that these are functions). On the other hand, the class on the right describes a concrete implementation of a stack as an array and count of elements.

```
class Abstract_Stack {
  void  push(int x);
  int   pop();
};
```

```
class Concrete_Stack {
  int   data[100];
  int   ndata;
  void  push(int x) { data[ndata++] = x; }
  int   pop() { return data[--ndata]; }
};
```

Figure 2.1: Abstract Class (left) and Concrete Implementation (right)

This goal of information hiding or encapsulation, has two important ramifications. First, the interface should by as simple as possible. In particular, extraneous information such as optimization annotations and implementation directives should be avoided. Notions such as the location of data, function calls, referencing and dereferencing should not be part of the abstraction. Second, abstract data types can be implemented by any data structure supporting the operations. Thus, the more general the abstraction, the more options are available for implementation. In Chapter 6, we will see how the compiler can leverage this flexibility to build high performance implementations.

2.1.2 Polymorphism

Polymorphism is the ability of a specification to operate on any abstraction which provides the required abstract behavior. This enables specifications to be reused and factored into shared and unshared parts.[2] Sets of abstract operations are called *signatures* [131, 16]. For example,

[2]The appropriate specialized parts are selected by dynamic dispatch (Section 2.1.4).

the abstract class on the left in Figure 2.1 defines a signature containing the `push()` and `pop()` operations.

The concept of polymorphism applies to variables and, by extension to classes and functions containing them. Polymorphic variables can contain any object supporting the operations performed on the variable. The set of these operations forms a signature, and the variable can contain any object which *conforms* to that signature. Classes which contain polymorphic instance variables define polymorphic objects, and methods which take polymorphic arguments are polymorphic functions. We will see in later chapters that these two types of polymorphic specifications require special techniques if they are to be analyzed and optimized by the compiler.

2.1.2.1 Polymorphic Variables

Polymorphic variables are locations which can store any object conforming to some signature. Two polymorphic variable declarations appear in Figure 2.2, one on the left side for Lisp [107] and one on the right for C++. The Lisp variable `a` can be assigned objects of any type (facilities exist in Lisp for arbitrarily limiting the range of objects assigned). The C++ variable `b` can be assigned a pointer to any object which is of class `B` or a subclass of (derived from) `B`[3].

```
(defvar a nil)
```

```
B* b = NULL;
```

Figure 2.2: Polymorphic Variables in Lisp (left) and C++ (right)

2.1.2.2 Polymorphic Classes

A common use of polymorphic variables is container classes, which can hold objects of more than one class. For example, the two polymorphic stacks in Figure 2.3 can be used to contain various kinds of objects. In the case of the Lisp (CLOS) [107] class on the left, the stack may contain objects of any class, even different classes simultaneously. The only operations the stack performs on its contents (assigning and moving) are supported by all Lisp objects, hence the signature of the contents conforms to any object. The C++ class (right), defines a stack which

[3]In the future, we will use *subclass* to mean non-strict subclass; that is "a subclass of `B`" may include `B`.

can contain objects of any subclass of A. Defining a stack in C++ which can contain any type of object requires templates [165] and is discussed in Section 2.1.5.

```
(defclass Stack () ...)
(defun push ((s Stack) e) ...)
(defun pop ((s Stack)) ...)
```

```
class Stack {
  void push(A* e);
  A* pop();
};
```

Figure 2.3: Polymorphic Containers in Lisp (left) and C++ (right)

2.1.2.3 Polymorphic Functions

Polymorphic functions operate on arguments conforming to some signatures. Thus, we can have the polymorphic functions `dbl()` in Figure 2.4 which can operate on any object supporting the `+` operation. The Lisp code (top left) defines a function over all types supported by the `+` function. The Smalltalk [76] code (bottom left) defines a *method* (in C++ parlance, a *virtual member function*) applicable to any class supporting the `+` method. In both Lisp and Smalltalk polymorphic functions can be used as any other function. They can be passed as values `#'dbl` (Lisp) and `#dbl` (Smalltalk), and stored in variables. The ability to treat functions as *first class* citizens is a significant source of expressive power which complicates analysis.

```
// polymorphic function (Lisp)
(defun dbl (a)
  (+ a a))

// polymorphic method (Smalltalk)
dbl
  ^ self + self.
```

```
// subclass polymorphic (C++)
Addable& dbl(Addable& a)
  { return a + a; }

// signature polymorphism (C++)
signature S { S operator+(S); };
S& dbl(S& a) { return a + a; }
```

Figure 2.4: Polymorphic Functions: Lisp, Smalltalk (left) and C++ (right)

C++ has several mechanisms supporting polymorphism. Figure 2.4 (right) provides some examples. The primary mechanism in C++ for polymorphism is subclassing. The function `dbl()` (top right) is applicable to objects of any subclass of `Addable`. This mechanism cannot be used with primitive types (`int`, `float`, etc.) which are not part of the class hierarchy. A mechanism [16] has been proposed which extends the C++ language to enable the definition of `dbl()` directly in terms of signatures (bottom). This has the added benefit of separating the typing and inheritance [23] (Section 2.1.3). Likewise, C++ templates [165] describe a set of

monomorphic functions, an element of which is *instantiated* for each use. These are covered in Section 2.1.5.

2.1.3 Inheritance

Inheritance is a language construct for building hierarchies. Inheritance can be applied both to interfaces (signatures) and implementation (behavior). Applied to interfaces, inheritance can be used to create a new signature with all the operations of an old signature and some additional operations.[4] Similarly, a new implementation can be built by *inheriting* the behavior of a more general one, and then adding to or redefining part of the behavior. For example, consider the up down counter on the left in Figure 2.5. Suppose we wish to create a modified specification which records the highest point reached. We simply subclass the class `UpDown` to be `UpDownCount`, and redefine the behavior of the `up()` method. The new `up()` method calls the old `up()` method and updates `max`.

```
class UpDown {
  int val;
  void up();
  void down();
};
```

```
class UpDownCount : UpDown {
  int max;
  void up() {
    UpDown::up();
    if (max<val) max=val;
  }
};
```

Figure 2.5: Superclass (base class) (left) and Subclass (derived class) (right)

2.1.4 Dynamic Dispatch

Dynamic dispatch (virtual function call) is the selection of the function to be executed based on the run time class of the target object and the selector or generic function name. This allows function calls to exhibit different behavior based on the class of an objects and supports programming with polymorphism. Each method (virtual function) is an implementation of the generic function specific to objects of a particular class.

For an example of dynamic dispatch, recall the two classes defined in Figure 2.5 which both define the `up()` method. If `up()` is invoked on a polymorphic variable `a` which might contain an object either of the two classes `UpDown or UpDownCount` (left side of Figure 2.6), the appropriate

[4]Viewing signatures as types, the new signature is a subtype of the old.

version will be selected at runtime. The effect will be as if a sequence of conditionals selected the appropriate version of `up()` (right side). Transformations for minimizing dynamic dispatch overhead are discussed in Chapters 6 and 7.

```
void func(UpDown * a) {
  a->up();
}
```

```
void func(UpDown * a) {
  if (a->class == UpDown)
    a->UpDown::up();
  else if (a->class == UpDownCount)
    a->UpDownCount::up();
}
```

Figure 2.6: Dynamic Dispatch Unfolded

2.1.5 Parametric Polymorphism

Parametric polymorphism is a limited form of polymorphism which allows classes and functions to be parameterized by the types of objects they support. Like inheritance, parametric polymorphism can be used for both interfaces and implementations. For interfaces, parametric polymorphism, like type declarations, constrain variables, allowing the programmer to more easily reason about the correctness of code. For implementations, parametric polymorphism builds a specialized version of a specification for a particular situations as indicated by the values of the parameters.

Templates [166] in C++ and generics in Ada [103] use parametric polymorphism simultaneously for both interfaces and implementations. For example, in Figure 2.7 the template (left) describes a set of the `dbl()` functions applicable to any type supporting the `+` operator. However, in C++, no general `dbl()` function exists, hence it cannot be passed as an argument or assigned to a variable. Instead, a specific function is generated from the specification for each use. Particular *monomorphic* instances are derived by applying a general specification to a set of parameters (in this case derived from the calling environment) at compile time.

Data types can also be specified using parametric polymorphism. In Figure 2.7 the class `Stack` is parameterized over the element type. When the stack is created, it is explicitly instantiated with the contents of type `E` (bottom right of Figure 2.7). In C++, instantiation typically involves the creation of specialized copies of templated code. This replication can expand the size of the executable program tremendously (i.e. cause code bloat) [166]. While the stack specification is polymorphic, each instance is as if it were declared with a particular type and

```
// template polymorphism (C++)
template <class A>
A dbl(A a) {
  return a + a;
}
...
  int i = dbl(1);
```

```
// template polymorphism (C++)
template <class E>
class Stack {
  void push(E e);
  E pop();
};
...
  Stack<int> s;
```

Figure 2.7: Function (left) and Class (right) Templates

therefore nominally monomorphic. These instances are still subject to subclass polymorphism and the associated inefficiencies (Section 2.1.6 below). Techniques for automatically discovering and optimizing parametric polymorphism and managing code size are discussed in Chapter 6.

2.1.6 Implementation Issues

Abstractions appear in the programming model as classes and functions. If these are implemented directly, each class and function would have a single unique implementation and each function (e.g. empty constructors) would have to be called. This can lead to very inefficient code. For example, take the (generic) class defined in Figure 2.8 and the two uses (one with an `int` and one with a `double`). One implementation would be to create a single version of the class with an indirection to a separately stored and tagged `a` field. Another implementation might build special classes for each uses of `A` and specialize all the code on objects of these classes.

```
class A {
  a;                  // instance variable
  A(aa) : a(aa) {};   // constructor initializing a to aa
  func();             // method
} x(1),y(1.0);        // two instances
```

Figure 2.8: Generic Class

Functions abstract the operations of the executing program, and direct implementations which cross these these abstraction boundaries can be inefficient. While the function call itself may require several instructions including dynamic dispatch, the greatest cost results from the loss of potential optimization. For example, Figure 2.9 uses the generic class `A` defined

in Figure 2.8. The `while` loop on the left contains seven function calls (three read and one write accessor[5] call for `A::a`, `doit()`, `A::func()` and `done()`). If we have the definitions for `A`, `doit()` and `func()` we can remove six of those calls producing the code on right. However, object-orientation complicates such *inlining* of methods.

```
...
  A x(intarg),y(floatarg);
  while (!done(x,y))
    y.a = doit(x.func,y.a);
```

```
doit(x,y) { x+y }
A::func() { a*2 }
...
  let tx = intarg,ty = floatarg;
  while (!done(tx,ty))
    ty += tx*2;
  y.a = ty;
```

Figure 2.9: Potential Optimization Example

In general, inlining of methods requires information about the actual (as opposed to declared) type of objects. Recall Figure 2.5 which defines two classes `UpDown` and `UpDownCount`. Now consider the code on the left in Figure 2.10.[6] The function `func()` can be called on `UpDown` or `UpDownCount` objects. This prevents the code for `up()` from being inlined directly. The type declarations of C++ do not solve this problem. In the C++ code on the right `func()` is similarly called on objects of these types. In general, inlining such calls requires program analysis (Section 5) and transformation (Section 6).

```
func(x) {
  ...
  x.up;
}
```

```
void func(UpDown &x) {
  ...
  x.up();
}
...
  func(*(new UpDown));
  func(*(new UpDownCount));
```

Figure 2.10: Inlining of Methods (virtual functions)

2.2 Concurrency

This thesis is concerned with *concurrent* programming languages (and their execution on parallel hardware) not parallel programming languages. Parallelism is two operations happening

[5]An accessor is a method which simply returns or sets an instance variable.

[6]This example is written in the language described in Section 2.3.3.

at the same time, a situation reflected in the physical world by the simultaneous action of two different pieces of hardware. Concurrency is a looser term, indicating that active periods of the two operations *may* overlap. Concurrency allows, but does not require parallelism, and is more natural in the abstract world of programming where it is often desirable to abstract away the mapping of conceptual operations to physical hardware. For example, concurrency includes coroutines where a single locus of control bounces between two tasks so that both are "active" but only one of which is physically executing at any given time. The key aspect of concurrency is that that it is non-binding, so a perfectly valid execution of two concurrent operations is sequential; execute one, wait for it to complete, then execute the other.

Under this definition, a compiler can easily produce efficient sequential code for any level of concurrency in the program specification (by simply ignoring it). In fact, because two concurrent tasks can be executed in any order, the more concurrency in the specification, the greater implementation freedom for the compiler. For example, in Figure 2.11, the code to the left specifies a traditional sequence of four operations. There is precisely one way to execute these calls correctly. The code to the right specifies four concurrent operations. There are twenty-four (four factorial) correct orders of execution for these calls.

```
{                                    conc {
  operation1();                        operation1();
  operation2();                        operation2();
  operation3();                        operation3();
  operation4();                        operation4();
}                                    }
```

Figure 2.11: Concurrency Example

Fine-grained concurrent programs are simpler (in a information theoretic sense) since they do not specify unnecessary sequencing. In a more practical sense, modern superscalar and pipelined microprocessors are highly parallel (Section 3.1.1). Optimizing compilers for such processors must work hard to remove excess sequentiality from specifications in order to produce efficient code for sequential languages. Thus, at some level, all programs executing on modern hardware are fine-grained concurrent.

2.2.1 Concurrent Statements

Concurrency can either be *tree-structured* or *irregular*. Tree-structured concurrency is so named because the task graph (where the nodes are tasks and the edges are dependencies between tasks), is a tree. In Figure 2.12, a single task (top) has created four subtasks (bottom) through concurrent calls (possibly message sends to concurrent objects). These subtasks must synchronize with their parent upon completion. This notification of termination makes tree-structured concurrency easier to reason about, analyze and optimize than irregular concurrency where the task can form a general graph. For this reason, many of the analyses and transformations discussed in this thesis assume tree-structured concurrency.

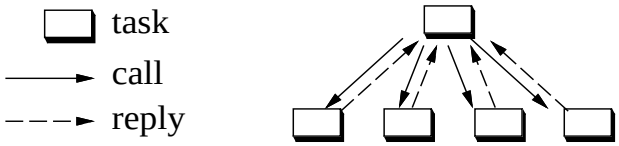


Figure 2.12: Tree-Structured Concurrency

While tree-structured concurrent child tasks generally complete before their parents, this is not always the case. A subtask which has no outstanding subtasks may delegate the responsibility of synchronizing with the parent to its last subtask. This allows the formation of synchronization structures such as forwarding and barriers as in Figure 2.13. The subtask forwarded to assumes the responsibility to synchronize with the parent, much like a tail recursive call [45]. Similarly, all the tasks waiting on the barrier use it to synchronize with their parent.

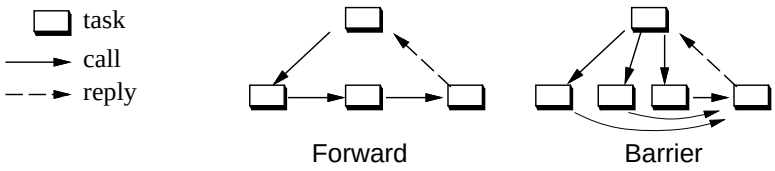


Figure 2.13: Other Synchronization Structures

There are two main mechanisms for generating tree-structured concurrency: concurrent blocks and concurrent loops. Within concurrent blocks statements are partially ordered to preserve a consistent view of local data.

2.2.1.1 Concurrent Blocks

As in Figure 2.11, a set of statements can be declared concurrent. The block completes only when all the enclosing statements have completed. Again, concurrency is not mandatory: if the statements are simple (arithmetic operations for example) they may be executed in some sequential order. Note that even without concurrency, this flexibility can provide extra freedom for instruction scheduling (see Section 3.1.1).

2.2.1.2 Concurrent Loops

Concurrent loops declare that (subject to the conditions expressed in Section 2.2.1.3) the iterations of the loop are concurrent. As a convenience to the programmer, ICC++ (Section 2.3.2) implicitly declares any concurrent loop body to be concurrent. Thus, the loop on the left of Figure 2.14 is a sequential loop, each iteration of which is a set of concurrent statements. In this case, only one `operation1()` will ever be active at any time. On the other hand, all the statements in all the iterations of the loop on the right are concurrent. Thus, `MAX` invocations of `operation1()` might be active simultaneously.

```
for (i=0;i<MAX;i++) conc {
  operation1();
  operation2();
  operation3();
}
```

```
conc for (i=0;i<MAX;i++) {
  operation1();
  operation2();
  operation3();
}
```

Figure 2.14: Sequential Loop of Concurrent Blocks (left) and Concurrent Loop (right)

2.2.1.3 Local Consistency

Concurrent objects ensure the consistency of their internal state by controlling concurrent access to that state. In order to help ensure the consistency of function local state, operations against it are not allowed to interfere with each other. The result of any such computation must be that which would be produced by execution of the dependent chain in normal sequential execution order. This preserves sequential semantics for function local operations by inducing a partial order over statements.

For example, the three statements on the left in Figure 2.15 are dependent since `L2` depends on `L1` (`L2` reads the value of `i` which `L1` writes) and `L3` depends on `L2` (it reads the previously

```
conc {                          conc {
  i = h;       // L1               func(i); // L4
  i = i + j;   // L2               i++;     // L5
  func(i);     // L3               func(i); // L6
}                               }
```

Figure 2.15: Local Consistency Examples

written value of `i`). Thus, the argument of `func()` will be the result of the sequential execution of these three statements. However, these statements need not be actually executed sequentially. On the right of Figure 2.15 only the statements `L5` and `L6` are dependent, in that order. In particular, statements `L4` and `L5` are dependent on the original assignment of `i`, but are otherwise independent. Such *write after read* dependencies are often called "false dependencies" for this reason.

2.2.2 Concurrent Objects

Concurrent objects are abstract data types which provide a consistent interface in a concurrent environment. They control concurrent access to their instance variables, provide for distributed concurrency and make concurrency guarantees which enable the programmer to reason about progress.

2.2.2.1 Concurrency Control

In order to provide a consistent interface, concurrent objects must control which messages are processed concurrently. For instance, a concurrent abstraction representing a collection cannot compute the number of elements it contains at the same time that it is adding or deleting elements. Many control mechanisms are possible. The simplest is to allow an object to process only one message a time [34, 43]. More complex schemes enable messages to be processed based on the state they access [81] or through set inclusion [73, 128].

The Concurrent Aggregates (Section 2.3.1) language subscribes to the model that a single message can be processed by a given object at one time. This was found to be restrictive both for the programmer and the optimizer (see Section 8.1). ICC++ (Section 2.3.2) takes a more permissive view, stating simply that intermediate object states cannot be seen. Essentially, this requires that result of a set of operations conform to some serialization. For example, in Figure 2.16, the calls to `f3()` and `f2()` can go on concurrently since they both read `x`. However,

```
class A {
  x = 1;
  y = 1;
  f1() { x = x + y; }
  f2() { y = y + x; }
  f3() { func(x); }
} a;
```

```
...
  conc {
    a.f1();
    a.f2();
    a.f3();
  }
```

Figure 2.16: Concurrency Control Examples

neither `f1()` and `f2()` nor `f1()` and `f3()` can go on concurrently since `f1()` writes `x` while both the calls to `f2()` and `f3()` read `x`. This allows the programmer to control nondeterminism without requiring the program to be overspecified. The final values of `a.x` and `a.y` are nondeterministic, but the sum of `x` and `y` must be 5.[7].

2.2.2.2 Distributed Consistency

Object-oriented programs are composed of a number of interacting objects. In order to reason about the composite behavior of a group of objects it must be possible to compose a set of transactions (messages) on a group of objects into a single transaction. The simplest mechanism for distributed consistency is to provide a single object whose abstract type represents the composite behavior. For example, Figure 2.17 shows a concurrent hash table abstraction which is implemented with a set of bucket objects. Since interactions with the buckets are moderated by the hash table object, consistence can be maintained over the abstraction.

2.2.2.3 Concurrency Guarantees

Some programs require concurrent objects to be able to overlap the processing of certain messages in order to prevent deadlock. These *concurrency guarantees* can be as simple requiring that an object be able to send a message to itself. Given an object consistency model in which only a single message can be executed at one time, an object sending a message to itself would result in deadlock (e.g. as in CC++ [34]). Stronger concurrency guarantees increase the expressiveness of the language as more programs which do not deadlock are possible. However,

[7]More relaxed consistency models are possible, for example, one in which a consistent set of "old" values can be read would allow the example to result in `x+y = 4` instead of `5` (see Section 10.2)

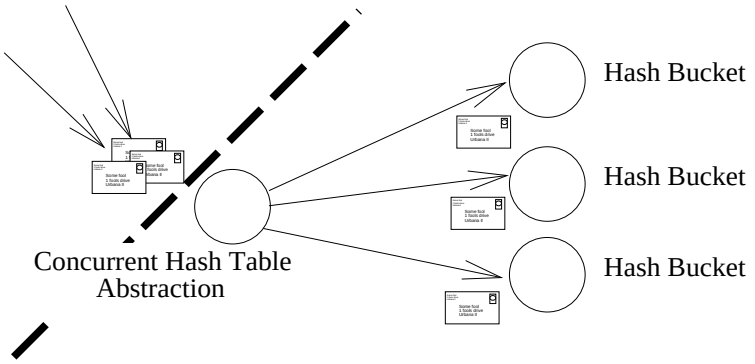


Figure 2.17: Example: Distributed Concurrent Abstraction

weaker guarantees grant the scheduler more flexibility, so concurrency guarantees must be balanced against implementation cost. For example, guarantees such as *strong* and *weak fairness* [6] can be expensive to implement since they require sophisticated scheduling.

Concurrent Aggregates provides for guaranteed concurrency between different objects, and the ability to invoke methods on the current object, `self` (`*this` in C++). Moreover, CA guarantees that any message which is sent will eventually be processed.[8] This has proven very expensive for applications with recursive inner loops, even when global analysis is used to carefully place scheduling operations. As a result, a compiler option allows this the fairness guarantee to be disabled. ICC++ makes no such guarantee. Concurrency guarantees are discussed further in Sections 3.2.5 and 3.2.1.2.

2.2.2.4 Actors

Since they can operate asynchronously, concurrent objects are sometimes called *actors* [5]. Likewise, the invocation of a method on a concurrent object is sometimes called a *message* because, like a piece of mail, they need not be responded to immediately, and more than one can be outstanding at one time. For example, messages sent to an actor (Figure 2.18 on left) build up until the actor can process them. Likewise, an actor (right) can send a number of messages off at the same time.

[8]Version 3.0 of the Concert compiler does not break iterative infinite loops for local scheduling, though it does break recursion.

Figure 2.18: Examples: In Box (left) and Outstanding Requests (right)

2.2.3 Naming, Location and Distribution

The fine-grained concurrent model does not include the location of objects or their distribution on the target platform within the language semantics. That is, the location of objects does not influence the meaning of a program. Programmer level names of objects do not include their location, which is managed transparently. A discussion of the execution model which underlies the implementation of the transparent global shared name space on distributed memory hardware appears in Chapter 3. Implementation and optimization of this model are discussed in Chapters 8 and 9.

2.2.4 Implementation Issues

Fine-grained concurrent programs can be very inefficient if implemented naively. These inefficiencies can result from the flexibility of the programming model. In particular, protecting the consistency of objects, providing location independence and simply managing the high degree of concurrency provided by programs written to this model can be very expensive.

2.2.4.1 Concurrency Control Boundaries

All objects mediate access to their internal state through an abstraction boundary. Likewise, concurrent objects mediate concurrent access to their state through concurrency control boundaries. These boundaries are more expensive to pass through than abstraction boundaries since the accessing task may have to be delayed. For example, in Figure 2.19, the code on the left defines three objects, two of which concurrently send a message to the third. As we can see

on the right, one of the messages (nondeterministically) has to be delayed. The interaction of scheduling and concurrency control boundaries is discussed in Sections 8 and 9.

```
class A {
  msg();
  send(a) { a.msg(); }
} x,y,z;

conc {
  y.send(x);
  z.send(x);
}
```

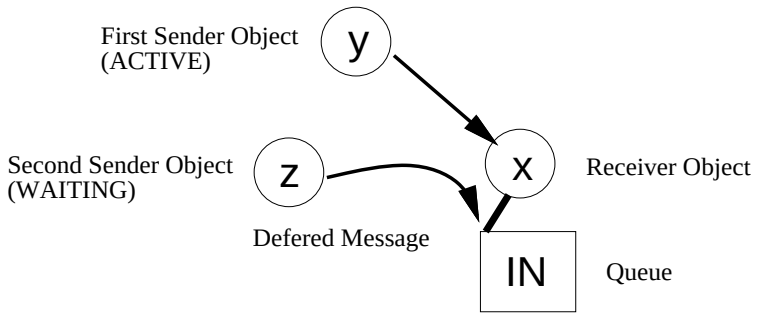


Figure 2.19: Concurrency Control Boundaries Example

2.2.4.2 Locality Boundaries

The fine-grained concurrent programming model does not include the notion of location of objects (Section 2.2.3). However, modern parallel and distributed computers are Non-Uniform Memory Access (NUMA); the cost to access "nearer" data is less than the cost to access data further away. As we will see in Chapter 3, the cost of access to data can vary tremendously. In order to provide good performance, most accesses must be to near data. This induces regions of locality, groups of objects which may access each other at lower cost. While the programmer need not manage these regions explicitly, portions of the program which are to be executed in parallel must necessarily span them, requiring the compiler to manage them. In Chapter 3 discusses the impact of crossing these *locality boundaries*.

2.2.4.3 Excess Concurrency

The fine-grained concurrent model encourages maximal expression of concurrency. Conceptually, each interaction between concurrent activities requires a *synchronization*. This includes message sends, which must determine if the target object is capable of receiving the message, and scheduling, when a delayed message becomes active. An example of a set of concurrent objects x, y and z synchronizing appears in Figure 2.20. Because y is processing a message from z when x sends its message, x's message is delayed. When the amount of concurrency exceeds that required, unnecessary synchronization can lead to inefficiency. The interaction of excess concurrency and scheduling is discussed in Chapter 3.

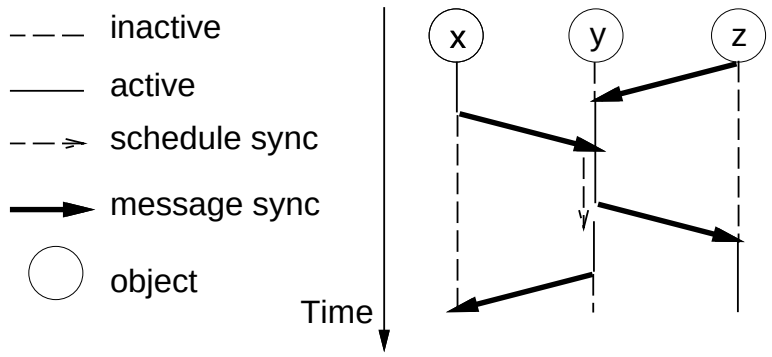


Figure 2.20: Synchronization between Concurrent Objects

2.3 Languages

This section discusses languages which embody the concurrent object-oriented programming model. The two languages with supported by the Concert system (Section 4.1) are Concurrent Aggregates (CA) [41, 43, 42, 40] and Illinois Concert C++ (ICC++) [38, 81, 35]. While superficially very different (CA has a Lisp-like syntax while ICC++ is derived from C++), the compilation techniques required to obtain efficiency for each language are almost identical. The language which will be used in the examples throughout this thesis (which is a blend of the two) reflects this. However, since the particularities of the two languages supported by Concert influenced the implementation of the analyses and transformations as well as the test suite used in their evaluation, they are discussed here.

2.3.1 Concurrent Aggregates (CA)

Concurrent Aggregates (CA) is a simple, untyped, pure object-oriented language [43]. It is closely related to Concurrent Smalltalk (CST) [55, 54, 100] which itself is closely related to Smalltalk [76]. The syntax of CA is reminiscent of Lisp [107]; parentheses group each expression and declaration. It has single inheritance, no type declarations, and first class selectors and messages (essentially a reification of a message send as a vector of arguments). It is pure in the sense that everything is an object, including integers, floating point numbers and globals. However, unlike Smalltalk control flow is built in; it has conditionals, `if`, and `while` loops. Furthermore, all statements, expressions and arguments are assumed to be evaluated concurrently

unless surrounded with an explicit `sequential` block. Figure 2.21 contains the naive Fibonacci program. Note that the recursive calls are concurrent.

```
(method integer fib ()
  (reply (if (< n 2) 1)
    (+ (fib (- n 1)) (fib (- n 2)))))

(method osystem initial_message ()
  (reply (fib 10)))
```

Figure 2.21: Fibonacci in Concurrent Aggregates

Concurrent Aggregates includes homogeneous collections of objects which collaborate to form an unserialized parallel abstraction. These are called *aggregates* (hence the name). Figure 2.22 contains an example of the use of aggregates in CA. The `aggregate` top level form defines an aggregate class with `nr_reps` representatives (elements). The `value` field in each element is initialized to zero by the `forall` (concurrent `for`) loop. Intra-aggregate addressing is through the `sibling` method. A `count` message sent to the aggregate is vectored to an arbitrary element where it (sequentially) updates the local `value` and `replys`. The `reply` is used for synchronization; the sender of `count` knows the accumulation operation has been completed when it receives the reply. The `sum` method begins the summation on the first element. Each element accumulates to the total `sum` and the last element returns the total count over all the elements to the original caller of `sum`.

2.3.2 Illinois Concert C++ (ICC++)

ICC++ [38, 81, 35] is a fine-grained concurrent dialect of C++, possessing the characteristics described in this chapter. It has concurrent blocks and loops (Sections 2.2.1.1 and 2.2.1.2), collections and object-based concurrency control (Section 2.2.2.1). It differs from CA and the example language in that type declarations are required for all variables, functions and classes.[9] However, this information is simply discarded (after semantic checks in the front end) by the compiler which calculates more precise information in the analysis phase (Section 5).

[9]As of Version 1.0. We hope to change this in the future.

```
(aggregate counters value           ;; aggregate (class) definition
  (parameters nr_reps)              ;; initialization parameters
  (initial nr_reps                  ;; initialization code, nr_reps elements
    (forall index from 0 below groupsize
      (set_value (sibling group index) 0))))

(handler counters count (val)       ;; handler (method) definition
  (sequential (set_value self (+ val (value self)))
              (reply val)))

(handler counters sum ()            ;; pass message on to first sibling
  (forward (sum_internal (sibling group 0) 0)))

(handler counters sum_internal (sum)
  (let ((newsum (+ sum (value self)))
        (nextindex (+ myindex 1)))
    (if (< nextindex groupsize)
      (forward (sum_internal (sibling group nextindex) newsum))
      (reply newsum))))
```

Figure 2.22: Counter Collection (Aggregate) in CA

One of the goals of ICC++ was to be as compatible with C++ as possible. Nevertheless, ICC++ departs from C++ where such departures are necessary to preserve consistency in a concurrent environment. In particular, like Java [168], ICC++ does not allow pointer arithmetic, pointers into objects, to fundamental types, or interconversion between pointers and arrays. ICC++ also requires accessor functions to be used to access all member variables. These functions are automatically defined. Furthermore, operations such as `++` and `+=` which are normally considered to be atomic are implemented as a single transaction. Figure 2.23 shows the atomic ICC++ increment, a C/C++ style pointer based atomic increment (illegal in ICC++ as it requires a pointer into an object, breaking the object abstraction). Since C++ is a sequential language, the final example (C++ style non-atomic increment) is equivalent in C++ but not in ICC++ where a concurrent operation setting the value of `count` between the read and the write might be lost.

```
// ICC++ atomic increment
self[].count++;

// C/C++ style pointer based atomic increment
int * i = &(*this)[ARBITRARY].count;
(*i)++;

// C++ style non-atomic increment
Counter &element = (*this)[ARBITRARY];
int i = element.count;
element.count = i + 1;
```

Figure 2.23: Atomic Operations

2.3.3 Example Language

Sections 2.1.2, 2.1.3 and 2.1.5 point out several ways in which C++ fails to support the principles of object-oriented programming. Moreover, C++ includes a number of features which are either overly complex (overload resolution), dangerous (casts) or simply poorly considered ("access control", templates). To wit, the examples in this thesis will be in a variation of C++/ICC++ with the following changes:

- Methods need not be defined in the class definitions. This is a violation of encapsulation for which C++ substitutes "access control".
- No C++ style "access control" (i.e. `private`, `protected`, and `public`).
- No type declarations, the `let` pseudo type can be used to introduce new bindings [164]. The weak typing system of C++ is neither safe (because of casts) nor powerful. C++ substitutes templates.
- No templates. Instead, polymorphic functions are specified by omitting type declarations.
- No pointers, references, or inline objects. Instead we follow the Smalltalk/Scheme model where all objects are by reference and any copies must be made explicitly.
- All instance variables are accessed through accessor functions. For a write of instance variable `a` the accessor is `operator=a()`.

- Methods (member functions) without arguments do not require parentheses (i.e. use `a.f` instead of. `a.f()`). These parentheses violate encapsulation since they differentiate system and user defined accessor functions, revealing the implementation.

- Compound statements can contain a final (unterminated) expression which is the value of the statement (i.e. `{ f(); 1 }` has 1 as its value).

- The return value of a function without a return is the value (if any) of its compound statement.

- Tuples, comma separated lists of values which are essentially anonymous records, can be used to return multiple values from a function (e.g. `(x,y) = func()`, ICC++ [81]).

The resulting language has the look of C++ and the clarity and simplicity of Smalltalk or Scheme. Moreover, as we will see in succeeding chapters, it can be compiled to execute with the efficiency of C (Chapters 7 and 8)). That is not to say that the techniques that will be discussed are not applicable to C++, only that many of C++'s features are not necessary to obtain that efficiency.

2.4 Related Work

The subjects of this chapter are represented in the literature by three traditions. First, is object-oriented programming alone. Then, there is the concurrent object-oriented tradition, including Actors. Finally, the subgroup of C++ based concurrent object-oriented languages is so large and diverse that it deserves separate coverage.

2.4.1 Object-Oriented Programming

Object-oriented programming was initially popularized in the United States by Smalltalk [76] and Flavors [133]. More recently, C++ [165] has become very popular for general purpose programming in industry. Object-oriented languages are very diverse, but they generally fall along a continuum between *static* and *dynamic*. Generally speaking, the more static the language, the more properties of its programs are determined at compile time. For example, Ada [103] has strong typing which constrains the types of objects at compile time, while in Smalltalk all

data is dynamically typed (the type safety of operations is checked at run time). Similarly, C++ [165] does not provide for automatic garbage collection, while Modula-3 [89], Sather [164] and Eiffel [130] provide it as an option and it is an integral part of Lisp and Smalltalk. SELF [176] is perhaps the most dynamic object-oriented language, allowing dynamic modification of the delegation style method lookup path at runtime. Generally speaking, the more dynamic the language the more powerful and the more difficult to compile for efficient execution.

2.4.2 Concurrent Object-Oriented Programming

Concurrent object-oriented programming extends the object-oriented paradigm for concurrent, parallel and/or distributed computing. The Actors [92, 46, 5, 4] model, is based on a simple but powerful semantics. A number of different systems using this model have been created. For example, languages based on Actors include: the dialects of ABCL [183, 170, 186], HAL [101, 115], ACORE [126] and Rosette [174]. These systems are based on asynchronous messages. Other languages have added stronger typing systems, for example, Cantor [19] and POOL-T [10, 9, 11]. Several forms of concurrent Smalltalk have been created, including Concurrent Smalltalk [184] and the language CST [98] which CA resembles. Sather, an Eiffel-like language has a more traditional parallel extension called pSather [134]. More recently, the language Ocore [116] has been developed by the Real World Computing Partnership.

2.4.3 Parallel C++

The other large body of COOP research has centered around parallel extensions to C++. This work can be roughly divided into two groups: data (object) parallel and task parallel. Languages in the data parallel group include pC++ [122] and C** [119]. In these languages, the operations specified in a single thread of computation may be executed on disjoint data without interaction. These operations are expressed over aggregations of data objects [41, 149]. This differs markedly from more general parallel languages which allow the user to specify more than one logically concurrent thread of control as well as interactions between threads.

There are many task parallel extensions to C++ which are divided into those based on fork-join and semaphore concurrency, object-based concurrency and extensible systems. Systems based on fork-join and semaphore concurrency including ESKit [158, 157], Presto [17], COOL [33], CC++ [34] and CHARM++ [108] provide facilities for programmers to construct objects

containing threads and objects which protect their state. However, these systems do not provide language level object consistency nor do they provide mechanisms for building of multi-object abstractions.

ESKit C++ [158], Mentat [80], CHARM++ [108], and Compositional C++ [34] are all medium-grained explicitly task parallel languages where the user controls grain size. None of these systems has developed a global optimization framework, probably owing to a desire to leverage existing C++ compiler technology. On the other hand, the subject of this thesis is automatic optimization of fine-grained concurrency through global analysis and transformation.

2.5 Summary

Object-oriented programming is the process of describing *abstractions*. Through *polymorphism* any abstraction can be used which supports the required set of operations (a *signature*). Through *inheritance* one abstraction can extend the definition of another. Fine-grained concurrency enables the programmer to specify which operations *may* be executed in parallel. Fine-grained concurrent object-oriented programs consist of a set of concurrent objects which interact by sending *messages* to each other. These objects *encapsulate* their state in a concurrent environment through *object-based concurrency control*. Together, object-oriented programming and fine-grained concurrency ease the task of writing and understanding programs by enabling abstractions to be built with well defined behavior in a concurrent environment, enabling local reasoning about the behavior and meaning of programs. However, programming systems which implement these abstractions directly can be very inefficient.

Chapter 3

Execution Model

> *The villainy you teach me I will execute, and it shall go hard, but I will better the instruction.*
>
> William Shakespeare. The Merchant of Venice

An execution model abstracts the execution of a program. It is the medium of compilation, by which the high level programming language is mapped to the low level hardware, and a way for the compiler writer to reason about the efficiency of that mapping. The execution model presented in this chapter consists of a model of the hardware (Section 3.1) of the target platform (i.e. CPU, network), a model software implementation (Section 3.2) and a runtime interface (Section 3.5) which connects the two. From these we infer a cost model which motivates the optimizations in later chapters. As we will see, those optimizations sometimes break through these simple models when necessary for efficiency.

3.1 Hardware

The hardware model describes the target platforms and is used by the compiler to model the cost of operations. The goals of the model are portability and scalability; to provide accurate cost estimates for a number of different systems of different size. It has two parts: the sequential microprocessor, whose characteristics are of interest in the evaluation of optimization for OOP (Chapter 7), and the parallel machine model for the evaluation of COOP specific optimizations (Chapter 8). Since the design of large scale parallel machines has not stabilized, we assume the least common denominator: a collection of commodity computers connected by

a communication medium. This induces a simple two level locality model (i.e. *local* vs. *remote*). The impact of particular hardware features is considered in [111, 110].

3.1.1 Microprocessor

The vast majority of computers today, including large scale parallel machines, are constructed from commodity microprocessors. Such microprocessors consist of a number of functional units for arithmetic, memory and control flow operations and a memory hierarchy. The memory hierarchy may contain registers, reservation stations (intermediate values in the pipeline), register windows, hardware thread contexts, level one, two and three caches and finally local memory. Figure 3.1 contains a block diagram of an example microprocessor. Assuming that it is work conserving, an execution is *efficient* if it can keep the functional units busy. Two factors fundamentally limit efficiency: effective use of the memory hierarchy and control flow ambiguities. Higher levels in the memory hierarchy can deliver more data per time unit, hence, given the ability of the functional units to sink large amounts of data, efficiency depends on how effectively the program uses those higher levels. Primarily this means that a program should use the very highest level (registers) for most operations. As a secondary consideration, the program should exhibit *temporal locality* of memory access [138].

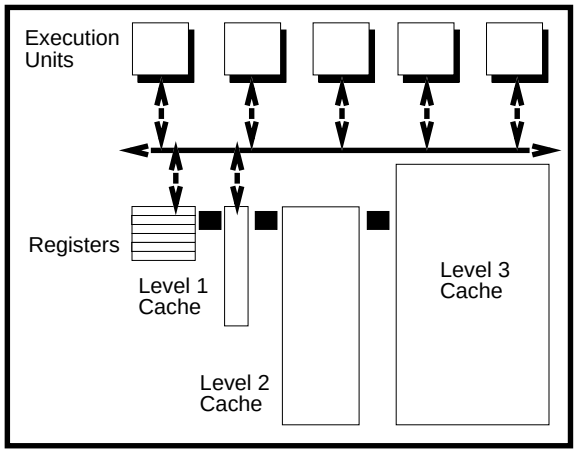


Figure 3.1: Microprocessor Block Diagram

Control flow ambiguities occur when the processor cannot predict the target of a branch instruction. This forces the processor to speculate on the target with incorrect speculations resulting in waste of execution resources. Take for example the pipeline of the Intel Pentium Pro

in Figure 3.2. This processor executes up to 40 instructions at once in a pipeline 12 clock cycles deep. A mispredicted branch typically incurs a 15-cycle latency [82]. For a processor which can dispatch three instructions per cycle to five functional units, this penalty is substantial. Dynamic dispatch is a prime culprit (see Section 3.1.3.1 below) contributing to control flow ambiguity and inefficient use of processor resources.

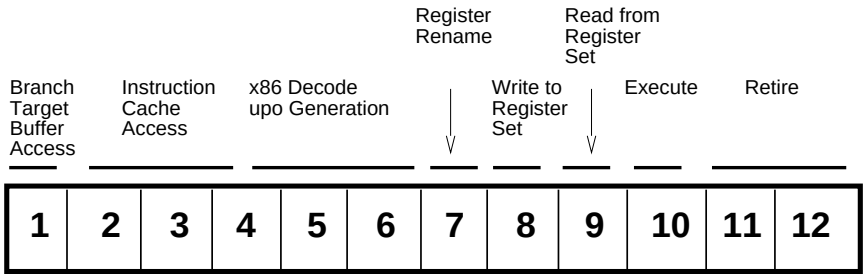


Figure 3.2: Microprocessor Pipeline

3.1.2 Distributed Memory Multicomputer

Multiple microprocessors are composed to form a parallel computer. Each of these processing elements has some local memory (possibly overlapping with other processors) which it can access more quickly than *remote* memory (which is generally local to other elements). In this model it is not important whether or not the hardware supports a single shared address space or cache coherence for remote memory. These access mechanisms are issues for the software implementation. What is important is that memory access cost is *non-uniform*, This model conforms to single processor or SMP (symetric multi-processing) nodes spanned by an interconnection network. Figure 3.3 diagrams such a multi-computer with two dimensional interconnect.

In the simplest case, mapping the program onto the hardware involves mapping objects to local memory and methods to threads (Section 3.2.1) on processing element associated with that memory. This mapping is expressed in Figure 3.4. Each thread logically operates within (in order) an object, a processing element, a local memory space. When a thread sends a message, the processing element on which it is located may switch to a new thread. Since switches involves flushing the higher levels of the memory hierarchy in order to make room for data for the new thread, processor efficiency dictates that they should be minimized. However,

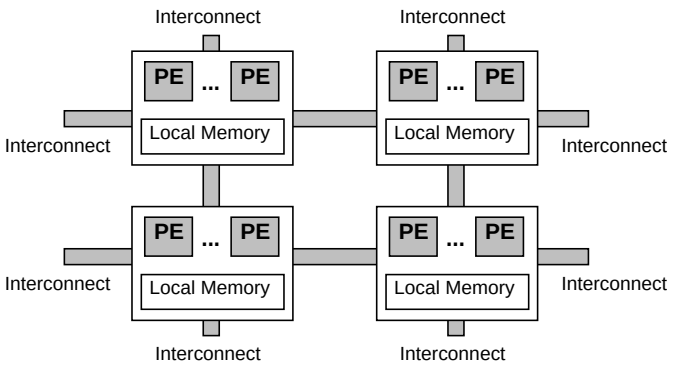


Figure 3.3: Hardware Model

active threads, running in parallel, generate work (additional threads) which is used to keep the nodes in a parallel machine busy. So, some number of thread switches are generally required for parallel efficiency.

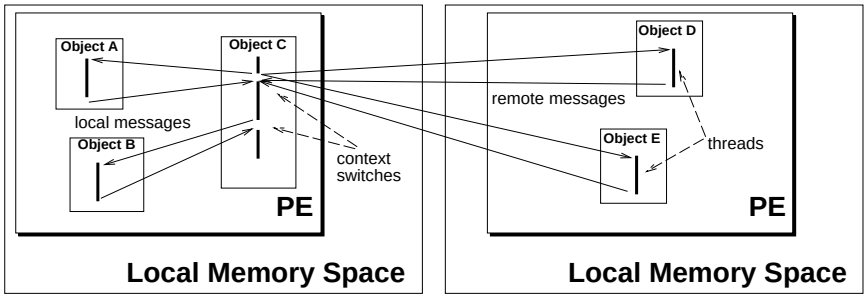


Figure 3.4: Software to Hardware Mapping

3.1.3 Implementation Issues

The hardware model indicates several potential sources of inefficiency from object-orientation and concurrency. Dynamic dispatch (Section 2.1.4) induces control flow ambiguities which (among other things) inhibit functional unit utilization. Object centrism (Section 2.1.1) encourages persistent memory-based computation over the use of temporary register-based data. Finally, distributing the computation across a distributed memory machine implies communication between local address spaces which is more expensive than are local memory operations.

3.1.3.1 Control Flow Ambiguities

Object-oriented programming encourages the use of abstract interfaces, inheritance and polymorphic methods. The implementation of these features at run time by dynamic dispatch leads to control flow ambiguities because the code to be executed depends on the type of an object or the value of a selector variable. Eliminating dynamic dispatch requires analyzing the program to determine under what circumstances control will flow in which direction. The code can then be transformed such that specialized versions are used when these conditions are static (i.e. known at compile time). The dynamic dispatch can then be replaced with a direct call, eliminating the ambiguity. Chapters 5, 6 and 7 are concerned with such analysis and transformation.

3.1.3.2 Memory Hierarchy Traffic

Efficient use of the memory hierarchy requires most data to be allocated in registers. Object-oriented programming encourages the use of dynamically allocated (indefinite extent) objects which are accessed indirectly (by pointers). As a result, the instance variables of an object are potentially aliased memory locations. In order to allocate instance variables to registers and eliminate the memory traffic associated with accessing them, the compiler must show that during their allocation to registers, the variables cannot be read or written through some other pointer. Chapter 7 considers this optimization in detail. Likewise, threads should also use registers for local data. However, these registers must be flushed to memory at context switches. In order to minimize this memory traffic, the compiler groups and minimize context switch points (see Chapter 8).

3.1.3.3 Communication

The cost of communication consists of two factors: overhead and latency. The overhead of communication can be reduced by specializing the communication mechanisms using compile time information. For example, directly executing a method from the communication buffer incurs less overhead than using a general purpose interface (Section 9.2.4). Latency is a function of the underlying communication hardware and the thread scheduling algorithm. By immediately scheduling messages which arrive at a node, latency can be reduced (Section 9.3). The remaining latency can be hidden by performing multiple long latency operations (message

sends) concurrently, context switching and then restarting when all the results have arrived (Figure 8.17).

3.2 Software

The soft droppes of rain perce the hard marble

John Lily

The execution of fine-grained concurrent object-oriented program can be view as the interaction of software constructs which implement program level constructs. There are three main program level constructs: *threads* (Section 3.2.1), *objects* (Section 3.2.2), and *messages* (Section 3.3). Threads are associated with a *contexts* (Section 3.2.1.1) which hold the temporary data use by the threads. Threads synchronize using *futures* (Section 3.2.1.3) which promise a value to be delivered later and *continuations* (Section 3.2.1.4) which deliver the value. Objects are composed of *slots* (Section 3.2.3) which can contain polymorphic variables or tagged data locations. Objects protect this state from race conditions with *locks* (Section 3.2.5). They interact by asynchronous method invocation (*message* passing, Section 3.3).

3.2.1 Threads

Concurrency is implemented at run time as a collection of fine-grained (short lived) threads. The cost of creating, blocking and resuming these threads sets a lower bound on the number of instructions which they must contain for the program to be considered efficient. For example, if the average thread executes one thousand instructions, but creating a thread requires two thousand instructions, only one-third of all instructions will be doing "useful" work. Thus, fine-grained threads are not implemented at the operating system level which would require an expensive change of hardware protection domain for scheduling. Instead, all the operations on threads – creation, suspension and resumption – are implemented at the user level. Efficient implementation of fine-grained threads is discussed in detail in Chapter 9.

3.2.1.1 Contexts

A context is a non-LIFO (Last In First Out) store for thread local state. Contexts can be thought of as heap allocated stack frames. In sequential computation, a method must complete

before its caller can continue. In fine-grained concurrent computation, the caller may continue, and invoke additional methods. The storage for the temporary data for the second method cannot be allocated on a simple stack since the first method has not yet completed. In general, the concurrent model induces a forest of contexts as opposed to a stack of frames (see Figure 3.5).

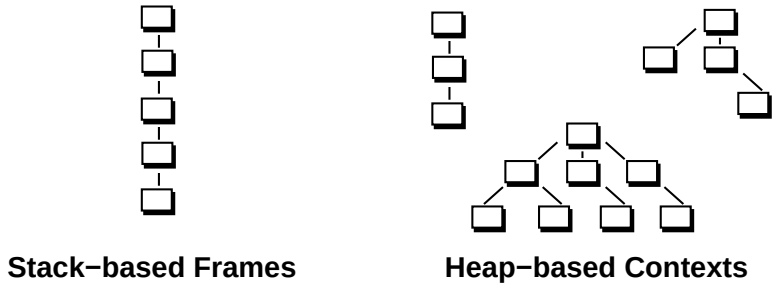


Figure 3.5: Stacks of Frames vs. Trees of Contexts

A context is similar to a stack frame with additional fields peculiar to object-orientation and concurrency. Figure 3.6 diagrams the model implementation of a context. **Method** is a reference to a method descriptor which is used during scheduling to determine the locking requirements and code of the method and, along with the **Program Counter**, by the garbage collector to determine the types of unboxed temporary variables. **Object** refers to the object on which the method was invoked, when the method exists, any locks acquired on this object are released. **Program Counter** is the location within the method where the thread last suspended. **Continuation** is the continuation for this method and, much like a return address, forms the linkage with the calling method. Finally, **Arguments** and **Temporaries** contain tagged or unboxed data used by the thread (see Section 3.4.3).

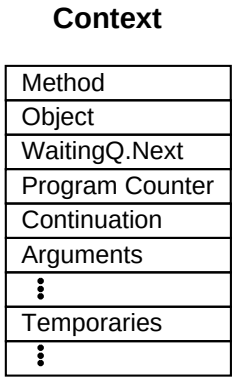


Figure 3.6: Context Layout

This model implementation is only a starting point for optimization. Cloning (Chapter 6) and inlining (Chapter 7) break the connection between methods and the code bodies to which contexts are associated. Speculative inlining (Chapter 7) and access regions (Chapter 8) break the connection between objects and contexts by allowing the thread associated with a context to operate on many objects, acquiring and releasing locks as required. For a variety of reasons, including portability, interfacing with external sequential code, efficiency and resource reuse, a stack-based mechanism can be preferable. Chapter 9 describes a hybrid stack-heap scheme which preserves the benefits of both systems by breaking the connection between contexts and threads, only creating contexts for threads that require scheduling.

3.2.1.2 Scheduling

When threads are scheduled effects the execution in several ways. First, it determines the dynamic task structure and the amount of parallelism. A "bushy" task tree provides additional units of computation for load balancing and latency hiding, but scheduling and synchronizing these tasks incurs overhead. Second, *when* a thread is scheduled effects the dependent threads waiting for its result. If the result is not returned quickly, the dependent threads will be flushed from the higher levels of the memory hierarchy. They will then have to be reloaded at some cost. Finally, when one thread is created by another on the same processor, the second thread can be scheduled immediately, enabling the threads to communicate through the highest levels of the memory hierarchy (i.e. registers). This motivates an eager scheduling model default, and the hybrid execution model presented in Chapter 9.

3.2.1.3 Futures

New threads are logically created for each concurrent invocations (Section 3.3). These threads synchronize with their parent thread using *futures* [87, 121]. Futures are essentially promises made by a task to provide a result, possibly at some later time. In a fine-grained concurrent model they are implicit, unlike MultiLisp [87] and Mul-T [117] where their insertion is the responsibility of the programmer. They are automatically inserted so as to enforce user specified concurrency constraints (sequential blocks) and local data flow (Section 2.2.1.3). For example, in Figure 3.7 the calls to `func1()` and `func2()` promise the results of their calculations as `a` and `b` respectively. These calls can be implemented as concurrent threads whose results may

be computed at any time. The parent thread continues to the call `func3()` which requires the results `a` and `b`. When these results are ready `func3()` can begin executing. Finally, when `func3()` completes, `func4()` can begin executing.

```
{
  conc {
    a = func1();
    b = func2();
    c = func3(a,b);
  }
  func4();
}
```

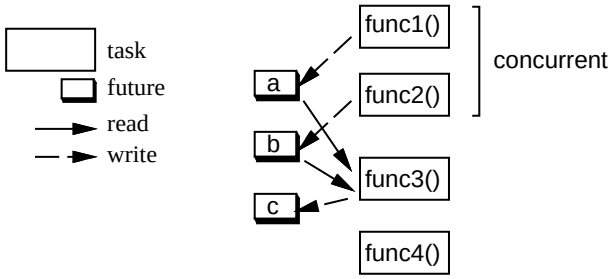


Figure 3.7: Futures Examples

3.2.1.4 Continuations

When a future is originally created it is *empty*, and the right to determine the value of the future is called a *continuation*. The continuation is essentially a small closure which is applied to provide the value of the future. Continuations are *first-class* citizens; they can be passed to another method or thread and it can be stored in memory. For example, in Concurrent Aggregates, the continuation can be accessed through the pseudo-variable `requester` or passed on to another invocation using the `forward` construct (see Section 2.3.1), while in ICC++ it is accessible directly as an object called `reply` available in every method. When a thread wishes to test a future to see if its value has arrived yet, it `touches` the future. If the value has not arrived, the thread must suspend.

```
conc {
  a = func1();
  ...
  b = func2(a);
}
```

```
(aFuture,aContinuation) = MAKE_FUTURE(a);
MAKE_THREAD(func1(),aContinuation);
...
if (!TOUCH(aFuture)) SUSPEND;
(bFuture,bContinuation) = MAKE_FUTURE(b);
MAKE_THREAD(func2(a),bContinuation);
```

Figure 3.8: Continuations and Touches

Consider the code on the left in Figure 3.8. The method `func2()` depends on the value of `a`. The pseudo-code on the right describes the steps taken by the abstraction to implement this dependence. First, `a` is made into an empty future and a continuation is created to return a value to that future. Second, a thread is created to execute `func1()` and passed the continuation

for a. Concurrent with the execution of func1() the future a is touched. If the value a is not present, the thread suspends. In any case, when a is available its value is passed to the method func2(). Of course, since building futures and threads and suspending is expensive, the most general forms of these operations are rarely performed. Optimizing these operations is the topic of Chapters 8 and 9.

```
conc {                      func2(a) {                func2(a,continuation) {
  ...                         return func3(a);          func3(a,continuation);
  b = func2(a);             }                         }
}                           func3(a) {                func3(a,continuation) {
                              return a+a;               continuation(a+a);
                            }                         }
```

Figure 3.9: Use of Continuations

Continuations can be used like normal methods which complete immediately (clearly we cannot use a future to wait until the first future received the value) and returns no value. Figure 3.9 shows a method invocation on the left. In the center is the definition of the method as written by the programmer. On the right is the same definition with the "hidden" continuation variables made explicit. Notice how the right to determine the value of b is forwarded from func2() to func3(). The continuation is eventually used by func3() to return the final result to b.

3.2.1.5 Counting Futures and Continuations

Counting futures are futures which represent a number of outstanding values. Along with counting continuations, counting futures enable a thread to synchronize once with an arbitrary (determined at runtime) number of threads (much like the *future sets* of MultiLisp [121]). On the left in Figure 3.10 a concurrent loop invokes func1() on a number of objects. The loop cannot complete until all of the invocations have completed. On the right, a future cFuture is created with a initial count of zero. Each time an invocation is made, the number of outstanding results is incremented. The TOUCH() operation, when applied to a counting future, checks that *all* the results have returned, suspending the thread if this is not the case. The implementation implications of counting continuation are considered in more detail in Chapter 9.

```
conc for (i=0;i<n;i++)
  o[i]->func1();
```

```
cFuture = ZERO_COUNT;
LOOP
  aContinuation = cFuture.INC();
  MAKE_THREAD(o[i]->func1(),cContinuation);
if (!TOUCH(cFuture)) SUSPEND;
```

Figure 3.10: Counting Futures and Continuations

3.2.2 Objects

Objects are represented by a region of memory laid out with fields for instance variables, synchronization and scheduling (Section 3.2.2.1. Each instance variable field is called a *slot* (Section 3.2.3). These slots may be tagged with the type of the contents, enabling them to represent polymorphic instance variables (Section 3.2.4). Race conditions resulting from concurrent access to these slots are prevented by locks (Section 3.2.5). When a method cannot be scheduled immediately, for example if it cannot acquire the locks it requires, it is delayed in a queue of threads associated with the object.

3.2.2.1 Object Layout

Concurrent objects are, in general, more heavyweight than their sequential counterparts. In addition to the class descriptor (virtual function table pointer [166]) required for object-oriented dispatch, they must maintain lock information and a queue of outstanding (blocked) messages. Also, the compiler must be able to generate code to access instance variables and find object pointers for garbage collection purposes. Figure 3.11 shows the layout of an object. Both instance variables and array elements are optional, and can be included in any object. Since the programming model does not permit the interconversion of pointers and arrays, array are like other objects and can contain instance variables. The instance variables and array elements may or may not be tagged slots (Figure 3.11 below).

3.2.3 Slots

A slot is a data location which is *tagged* so that the type of the contents can be determined at run time. Slots can contain immediate values (integers, floating point numbers, system constants), global names, local object pointers, continuations, and futures (Section 3.2.1.3). Figure 3.12 gives a example how a slot might be declared in C. The slot tags are used for dynamic dispatch

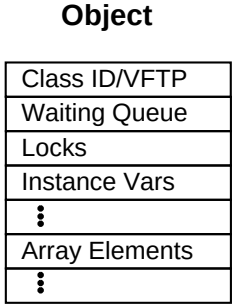


Figure 3.11: Object Layout

and by the garbage collector (Section 3.4.3) to determine if a slot contains a pointer. For example, Concurrent Aggregates is a pure object-oriented language in which a variable can store an integer one moment and a reference to an object the next. Tag manipulations can be expensive and slots can occupy more space than *unboxed* (untagged) values. Section 7.4 discusses eliminating these inefficiencies through unboxing.

```
struct Slot {
  enum {INT, FLOAT, FUTURE, GLOBAL_PTR, LOCAL_PTR} tag;
  union {
    int i;
    float f;
    Future fut;
    Continuation cont;
    GlobalPtr gptr;
    LocalPtr lptr;
  } value;
};
```

Figure 3.12: Slot Abstraction

3.2.4 Tags

Slots are used to store the value of polymorphic variables and the return values of concurrent invocations which require futures. Slots contain tags which indicate the type of data which they contain. These tags differentiate fundamental types (`int`, `float`, etc.), global and local names, collections, selectors, continuations, and futures, both empty and full. When analysis can show that a slot can contain only one type of data at any one time, the slot can be converted into

unboxed data. The tags are removed and space is allocated only for the contents, allowing the removal of tag manipulation operations and the recovery of the tag space.

3.2.5 Locks

Object-based access control is the underlying mechanism used to ensure consistency as required by the programming model. While the consistency model is described in terms of visible state changes, the implementation is based on atomicity. That is, the internal state changes of a transaction (method) are hidden by prohibiting other transactions from happening concurrently. While in some cases it is possible to ensure this by analyzing the control flow of the program, in general, an object must be locked and conflicting messages delayed. Concurrent methods have the general form of the code on the left in Figure 3.13.

```
A::foo() {
  LOCK(self);
  ... operate on self ...
  UNLOCK(self);
}
```

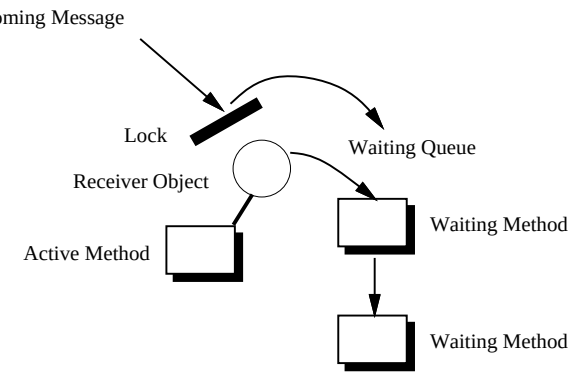


Figure 3.13: Object-based Access Control

When a method begins executing a thread is created which locks the target object to prevent interference from other threads. Locks can be checked, taken and released, requiring access and manipulation of the lock fields stored in the object. In the case of a check, the new thread may be required to suspend pending availability of the lock. Such messages arriving at a busy object are delayed in the waiting queue until the currently active thread frees the lock.

The region of code over which a thread has access to the object is called the *access region*. There are two main ways to remove lock operations. First, one locking operation subsumes another when the second occurs only when the first lock operation has succeeded. Such lock operations are unnecessary and can be removed. Second, consecutive acquisitions and releases of the same lock can be grouped, and the intermediate release-take pairs removed. Optimizations

concerning lock subsumption detection and manipulations of access regions in order to minimize the number of lock and scheduling operations are discussed in Chapter 8.

3.3 Messages

An invocation (message send) in the fine-grained concurrent object-oriented model consists of a number of steps (illustrated in Figure 3.14). These steps are abstract; using the techniques described in this thesis, many of them can be optimized away or performed in a different order. First (1), the global name is translated, and the target address space determined. Then (2), the message is constructed from the arguments and the continuation is built. Next (3), the message is transfered to the target address space. Some time later (4), the message is scheduled. Finally (5), the message is dynamically dispatched, and the appropriate code begins executing.

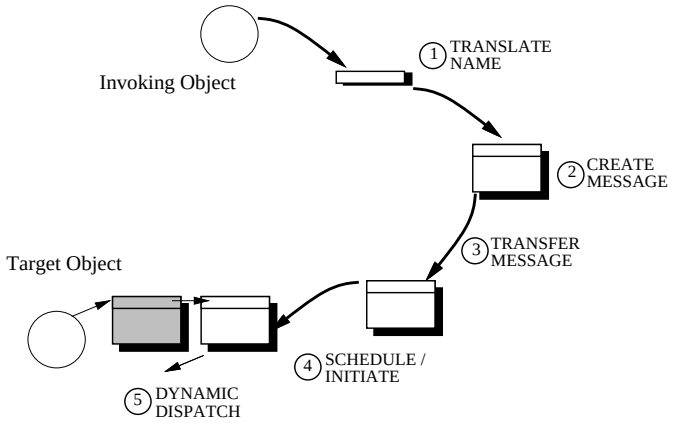


Figure 3.14: Invocation Sequence

3.3.1 Global Shared Name Space

A shared name space means that there is a single name space and that the names are always valid, nomatter where the data is located. Many concurrent object-oriented systems distinguish the hardware local and global namespaces (e.g. Split-C, CC++, Charm++, Mentat). In such systems only global names can be passed between and used to refer to objects nodes on different nodes of a distributed memory machine (Section 3.1). A shared name space greatly simplifies programming, by enabling the expression of the algorithm to be separated from the location

of data. Global names can implemented as a node number and a local memory address of the current or initial location of the object or an index into a distributed hash table [109].

```
...                                  if (NODE(a) == THIS_NODE) {
a.foo();                               o = LOCAL_POINTER(a);
...                                    INVOKE(o,"foo")
                                     } else
                                       SEND_MESSAGE(NODE(a),a,"foo");
```

Figure 3.15: Global Name Translation

For distributed memory machines both with and without hardware support for a shared name space, it is desirable to manage the location of threads with respect to the data they are operating on in order to ensure locality of access. This involves a translation from global names to local names, which are valid within a single node and typically represented by an address in the local memory. The translation mechanisms are provided by the runtime system (Section 3.5). For example, in Figure 3.15, the code on the left shows program level invocation. On the right, the global name a is first checked to see if it corresponds to a local object, and either a local invocation is made or a remote message send. Chapter 8 discusses automatic management of locality and the optimization of translation operations.

3.3.2 Scheduling

Scheduling is the process of selecting which tasks run when. When a message arrives at the object the default behavior is execute it immediately, acquiring any required locks. If the locks cannot be acquired, the message is delayed. The scheduler maintains a queue per object in which is stored waiting messages in the order in which they arrived. When locks are release on the object, waiting messages attempt to acquire their locks and execute.

The scheduler must support the model of fairness required by the programming model. In this thesis, we assume a restricted version of weak fairness. Messages sent to objects will be handled eventually assuming that methods holding locks on the object eventually terminate, threads executing on the processing element eventually block, and the result of the computation depends on the result of the message. These conditions can be violated by the program deadlocking, looping infinitely, or not synchronizing on the termination of an operation. The compiler must not to induce these conditions so long as the programmer relies only on the con-

currency guarantees provided by the programming model. This affects the locking optimizations in Chapter 8.

3.3.2.1 Dispatch Mechanism

The code executed as a result of a method invocation depends on the (dynamic) type of the target object(s) (Section 2.1.4). A *dispatch mechanism* selects the code to be executed using a set of *dispatch criteria* derived from the calling environment which can include the signature, declared and actual type of one or more arguments, and the name of the method. Several dispatch mechanisms implementations are possible, including run time searching of the method dictionary, class-based tables [166], hash tables [178], inline cashes [61] and polymorphic inline caches [96]. The caching mechanisms attempt to reduce the amortized cost of dispatch by providing a fast path for related temporally proximate dispatches.

Perhaps the most common implementation (C++ [166]) is a table of methods indexed by the order in which methods are declared in a class. This method has the disadvantage that methods to be dispatched on must be defined in some shared superclass, a restriction C++ imposes in the type system. This restriction is isomorphic to the object layout problem (Section 3.4.1). Faster mechanisms can also be used when the values of some of the criteria are known at compile time, and the fastest mechanism is to avoid the call entirely and inline the code. Chapters 5, 6 and 7 are concerned with these optimization.

3.4 Implementation Issues

The facilities provided by the execution model are sufficient, but their generality makes them expensive. Variables implemented as slots require additional space and time to manipulate the tags. For non-polymorphic variables, the tags and operations can be elided. Likewise, thread creation, scheduling and synchronization operations should be avoided whenever possible. Three other issues are memory map conformance (Section 3.4.1), load balance and data distribution (Section 3.4.2) and garbage collection (Section 3.4.3).

3.4.1 Specialization and Memory Map Conformance

When an object of a particular class is created, it can be used in a number of different ways. If it is an array, it may be created to hold some arbitrary number of elements or a compile time constant number. Its instance variables may be used to hold single types of data or those of several types. Access to its internal state may be entirely mediated by some surrounding object or the object may be capable of receiving messages directly. These difference represent optimization opportunities for which the object may be specialized. Sometimes it is possible to statically determine these properties over an entire class of objects, in which case the class and all code manipulating objects of that class can be specialized. Inheritance, however, introduces additional complexities.

Memory map conformance is a property of the layout of objects within classes such that code compiled to manipulate objects of the superclass type can be used on objects of the subclass type. For example, consider a class `A` which defines a single instance variable `a` and a class `B` which inherits from `A` and defines an additional instance variable `b` the compiler can ensure that the location of `a` within objects of type `A` and `B` will be the same. This is illustrated in Figure 3.16. This property enables code to be shared between the superclass (`A`) and the subclass (`B`), but it inhibits some optimizations.

```
class A {
  a;
};
class B : A {
  b;
};
```

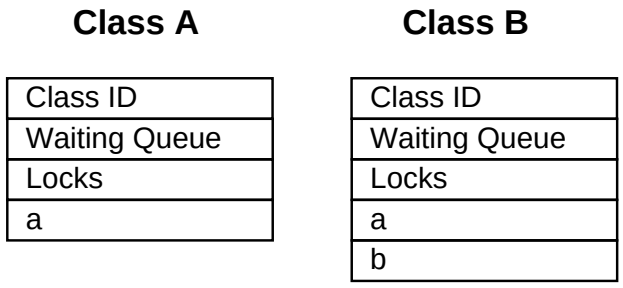


Figure 3.16: Class Memory Map Conformance

Memory map conformance can also be an issue for objects in a single class. For example, if a particular instance variables of an object is known to be a compile time constant, the space for that instance variable need not be allocated. However, this can lead to difficulties since the code generated to manipulate the object may be shared with other objects which require the instance variable. Consider the two objects in Figure 3.17. The memory map of `x` could be altered to remove the instance variable `a` since this variable is a compile time constant. However,

the code which manipulates these objects must access b at different locations. Performing such optimizations, both those which preserve memory map conformance and those which do not, is one of the the subjects of Chapter 6.

```
class A { a; b; };
foo (o) { return o.b+o.b; }
conc {
  A x; x.a = 1; x.b = ..;
  A y; y.a = g; y.b = ..;
  .. foo(x) .. foo(y) ..
}
```

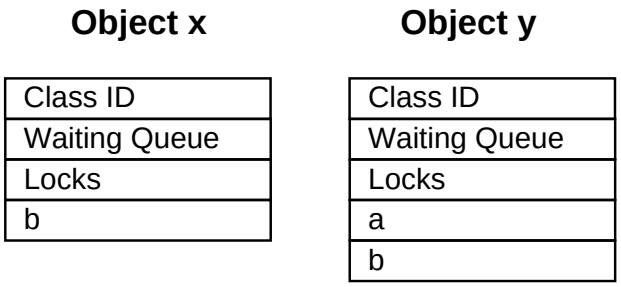


Figure 3.17: Object Memory Maps Which do not Conform

3.4.2 Load Balance and Data Distribution

The efficiency of a parallel program depends on its *processor efficiency* (the efficiency of execution on a processor) and its *parallel efficiency* (the number of processors that the program can keep busy consistently). This thesis is concerned primarily with processor efficiency under the assumption that the work and data are distributed. There are many ways to distribute work and data and balance the work load across the machine. Generally, these techniques involve data shipping (caching, object migration etc.), function shipping (remote procedure calls, distributed loops) or both.

To abstract the problem of processor efficiency, we assume a simple model were methods execute local to the object on which they are invoked and objects are distributed across the machine. Load balance is assumed to be maintained by some combination of static placement, and data and function shipping mediated by the runtime system. The location where a thread should be created (that of the target object) is mediated by runtime system through an interface (Section 3.5) used by the compiler generated code.

3.4.3 Memory Allocation and Garbage Collection

Object-oriented languages encourage dynamic creation and disposal of objects at run time instead of static allocation of them at compile time. It follows that the efficiency of these operations can greatly effect the overall performance of the application. As we will see in Chapter 7,

the efficiency of at least one simple benchmark doubles when the memory management mechanism is improved. The nature of the object memory map effects memory allocation and garbage collection. For example, if an object does not have an array portion or if the array portion is known to be of a particular size a custom bin-based memory allocator can be used. Similarly, in order for the garbage collector to find all the reachable objects in the system it must be able to determine which data represent object pointers. Therefor, if objects and contexts (see Section 3.2.5) are tagged by type descriptors, individual tags on the instance variables and temporary variables can be eliminated in favor of information stored in the type descriptor.

3.5 Runtime

The compiled code interacts with runtime system through an abstract interface which hides many of the details of the underlying hardware. It is the runtime system which defines the translation and message transfer operations. Table 3.1 summarizes the runtime interface. Again, many of these operations may be optimized away. The first set (INVOKE,REPLY) is for invoking methods (sending messages) and replying to continuations. The next set (LOCK_OBJECT,LOCAL_POINTER) exposes the object consistency locks and global to local name translation mechanisms. The next set (OBJ,CONTEXT_SLOT) allows translation to and from unboxed values and operations on heap based contexts. Continuation manipulation operations (MAKE_CONTINUATION,TOUCH) make up the next set. Then we have object creation (NEW_OBJECT) and intra-collection (COL_TO_REP) addressing, and finally operations on globals. These operations will be considered in more detail in the chapters which discuss their optimization.

3.6 Related Work

At the hardware level, there have been a number of systems which were constructed especially to run object-oriented languages, including the Xerox Dorado [139] and Berkeley SOAR [150]. The J-Machine [56] and the MDP [57] were designed to run concurrent object-oriented programs and use the COSMOS [99] operating system as a runtime environment on top of the hardware. Likewise, ABCL has been implemented on the hybrid data flow machine EM-4 [183]. However, recently economy of scale and advances in compiler technology have favored commodity micro-

Operation	Variation and Related Operations
INVOKE	INVOKE_IMMED,INVOKE_LOCAL
REPLY	REPLY_WITH_MSG
LOCK_OBJECT	UNLOCK_OBJECT,LOCKED?
LOCAL_POINTER	
CONTEXT_SLOT	INST_VAR,CONTINUATION_SLOT,ARGUMENT, THE_MSG,THE_SUPER
OBJ	INT,FLOAT,GLOBAL_NAME
MAKE_CONTINUATION	MAKE_COUNT_CONTINUATION
TOUCH	MTOUCH,TOUCH_BEGIN,SWITCH,TOUCH_END,MTOUCH_BEGIN
NEW_OBJECT	NEW_LOCAL_OBJECT,NEW_LOCAL_OBJECT_SIZE, NEW_ARRAY,NEW_LOCAL_ARRAY,NEW_UNBOXED_ARRAY, NEW_LOCAL_UNBOXED_ARRAY,NEW_AGGREGATE, NEW_LOCAL_AGGREGATE,NEW_UNIQUE,NULL_OBJECT
COL_TO_REP	COL_TO_PHYSICAL_REP
READ_GLOBAL	READ_GLOBAL_SEQUENTIAL_UNBOXED_INT, READ_GLOBAL_SEQUENTIAL_UNBOXED_OBJ,READ_GLOBAL_SEQUENTIAL
DISTRIBUTE_GLOBAL	

Table 3.1: Runtime Operations

processors which proven cost effective. For example, ABCL implemented on the AP1000 and the Concert system on the Thinking Machines CM5 [173] and Cray T3D [52] have all proven effective, often thanks to efficient runtime systems [179, 112, 113].

There is a host of material on software models for sequential object-oriented languages, most notably Smalltalk [76, 67, 61], and SELF [29, 28, 30]. These models differ from that discussed in this chapter with respect to support for distribution and concurrency. There are also many concurrent object-oriented systems. Of particular interest are Concurrent Smalltalk [100], ABCL [186], ABCL/R2 [127] and ABCL/f [171]. Many of the concepts in this chapter are found in these systems, but the specific mechanisms, especially for synchronization, differ.

3.7 Summary

The execution model is the compilation target and cost model for the compiler. It is composed of a set of execution abstractions of the hardware, software and runtime system. The hardware

model is based on commodity microprocessors spanned by an interconnection network. This model indicates several potential sources of inefficiency, including control flow ambiguities resulting from dynamic dispatch, memory hierarchy traffic and communication overhead. The software model describes model implementations for *threads*, *objects*, and *messages*.. Threads store their state in a *context* which are non-LIFO stack frames. Threads synchronize with other threads using *futures* which promise a value to be provided later by a *continuation*. Objects store their instance variables in *slots* which may be tagged with the type of the contents. The software model describes the physical layout of contexts and objects and the implications for memory allocation and garbage collection. It also describes the method invocation sequence and dynamic dispatch mechanism. The hardware and the software interact through a *runtime interface*.

Chapter 4

The Compilation Framework

> Computers are useless. They can only give you answers.
>
> *Pablo Picasso*

This chapter describes the compilation framework and the Concert System [37] in which it is implemented. The Concert System is a complete development system for concurrent object-oriented programs consisting of a compiler, runtime system and various support tools and libraries. The Concert vision, expressed in Section 4.1, is to combine high-level expression of programs and efficient implementation. The Concert compiler (Section 4.2) achieves this through aggressive analysis and transformation, and collaboration with a efficient runtime system (Section 4.3). Working together, the compiler and runtime can efficiently map both Concurrent Aggregates (Section 2.3.1) and ICC++ (Section 2.3.2) to a variety of target machines (Section 4.4).

4.1 Concert

The Illinois Concert System [36] is a project of the Concurrent Systems Architecture Group. Its goal is to produce portable, high performance implementations of concurrent object-oriented languages on parallel machines. The philosophy of the project is that programs should be written at a high level, exposing all the fine-grained concurrency and that the language implementation should tune the execution grain size to that supported efficiently by the target platform. The Concert system consists of a compiler, a runtime specialized to the underlying

hardware, an emulator for quick turnaround debugging, a debugger and a standard library. The Concert system has been in development since January, 1993.

4.1.1 Objective

The overriding objective of the Concert project is to show that concurrent object-oriented efficiently can be executed efficiently on stock hardware. We believe that empirical evaluation must be the ultimate arbiter and that necessity is the most effective way to establishes priorities. We established two constraints for our system. First, it should provide high performance. Since the techniques required to improve performance change dramatically as we approach that of conventional languages on uniprocessors, high performance is of both practical and academic value. The second constraint is that the system should be general and portable, applicable to a range of concurrent object-oriented languages and target parallel machines. Portability is important from both a practical as well as an academic standpoint. As a practical matter, parallel machines have diverse characteristics and short lifetimes; while we were one of the early users, the CM5 went out of production shortly after our port was complete. From an academic standpoint, portability demonstrates generality of approach.

4.1.2 Philosophy

The philosophy of Concert is that programmers should be concerned with natural expression of their programs and the system should be concerned with producing an efficient implementation. From our point of view, natural expression involves high level abstractions and *explicit* concurrency. We believe that the compiler should map these dynamic high level abstractions to static implementations at the earliest point possible (e.g. partially evaluating expressions with respect to known parameters). Explicitly concurrent programming languages are strictly more expressive than sequential languages since they can model non-deterministic algorithms. We believe the programming should express the natural concurrency in the algorithm and that it is the responsibility of the system to tune the realized concurrency to that which can be efficiently supported by the target platform.

Our approach to tuning the execution grain size is to specialize the computation relative to the properties of the program as they become know. Some properties are known statically, in the program text, at compile time. Others require the program to be transformed. For example,

replicating code executed under different conditions can be beneficial because it enables creation of unique versions with fixed known properties. Still other properties must be tested at runtime. We can compile different version of code specialized for these dynamic properties, and the version executed is selected at runtime. In any case, taking advantage of these properties as early as possible is desirable, since implementation cost increases as we move closer to the hardware. This induces a hierarchy of mechanisms pictured in Figure 4.1. By fixing properties as early as possible, we separate the dynamism of the algorithm from that of the language used for expressing the algorithm.

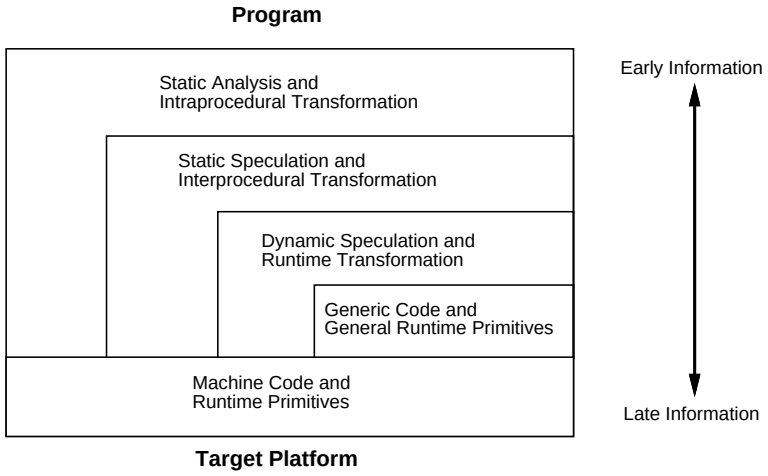


Figure 4.1: Concert Philosophy: A Hierarchy of Mechanisms

In order to provide the system maximal flexibility the runtime system provides a hierarchy of primitives of varying degrees of flexibility. For example, the compiler can select a primitive which statically binds the code to be executed for an invocation (if the code is known at compile time) or one which dynamically binds the code at run time. In turn the compiler provides information obtained by static analysis to the runtime system for use in runtime transformations like caching and load balancing. Thus, the Concert philosophy is that the compiler and the runtime should work together, each leveraging the strengths of the other.

4.1.3 Parts of the System

The Concert System [65, 63] is compose of a compiler, runtime system, emulator, symbolic debugger and a set of libraries. The compiler and runtime are discussed in Sections 4.2 and

4.3 respectively. The emulator [167] works within the Lisp environment by interpreting the *core language* intermediate representation (Section 4.2.2.2). The ParaSight debugger [62, 64] provides source level debugging of concurrent object-oriented programs on workstations. The standard library provides reusable code for two and three dimensional grids, barriers, distributed counters, combining trees, data parallel computations and other standard data structures and algorithms.

4.1.4 Timetable

The Concert System has undergone a staged development, described in Table 4.1. The project originally derived from the thesis work of my advisor, Andrew Chien [39]. That system consisted of a translator and runtime for uniprocessor workstations. Simple programs executed on that system ran approximately six orders of magnitude (one million times) slower than unoptimized C. In the first exploratory stage of work we added inheritance and dynamic dispatch to the Concurrent Aggregates language, developed a runtime interface for the CM5 and prototyped an interprocedural flow analysis. This stage is described in the table by the row labeled **Pre-Concert**. Based on this experience began a new design, Concert, including a new compiler framework and runtime interface.

Version	Date	Language Changes	Major Compiler Changes	Other Major Changes	Speed vs. C
Pre-Concert	6/92-12/92	CA+OO	proto-Flow Analysis	proto-CM5 runtime	1/10e6
Concert 1.0	6-93	+set!	CFG+SSA-based, Flow Analysis	CM5 runtime	1/10e3
Concert 1.1	10-93		Primitive Inlining	Emulator, Debugger	1/100
Concert 2.0	3-94	+functions	Speculative Inlining	Parallel GC	1/10
Concert 3.0	5-95	+annotations	PDG-based, Access Regions Hybrid Execution	T3D runtime User Distributions	1
Concert 4.0	5-96	ICC++	Object Inlining		1

Table 4.1: Development of the Concert System

The first version of Concert (1.0) was an internal release capable of executing the test suite developed for the pre-Concert system. The internal representation was a Control Flow Graph (CFG) in Static Single Assignment (SSA) [53] form. This version was considerably faster than the pre-Concert system owning to a more efficient runtime system. The first external release

(1.1) improved on this by inlining primitive operations (e.g. integer add). Version 2.0 added speculative inlining (Section 7.5), runtime locality and lock tests conditioning inlined code. This brought performance to within a factor of 10 of C for many codes. The last factor of 10 required more radical changes, including shifting to a new internal form, the Program Dependence Graph (PDG) [72]. It is the work required for this last factor which occupies Chapters 8 and 9.

4.2 Compiler

The Concert compiler employs aggressive static analysis and transformations exploiting information from the high level COOP semantics and the low level load, operate, store semantics of the programming model. For example, subsumption of object-level access control operations (Section 3.2.5) requires interprocedural analysis and transformation at the level of the call graph. Likewise, object-level access regions effectively guarantee lack of aliasing which can be used for instruction scheduling and register allocation at the lowest level. To exploit these levels, the Concert compiler provides a compilation framework with eight representations of the program from source code to target code so that optimizations can be applied at the most convenient level. This framework is composed of five phases: parsing and semantic processing, program graph construction, analysis, transformation and code generation. Excluding the two parsers, the compiler is approximately thirty-five thousand lines of Common Lisp/CLOS.

4.2.1 Overview

The Concert compiler is structured as a pipeline of phases with several feedback loops. Figure 4.2 gives this structure along with the intermediate representations used in each phase. The parsing and semantic processing phase consists of reading the program in the source language into an Abstract Syntax Tree (AST) and computing various attributes [7]. Its product is the program translated into the core language intermediate form. This form is then during the program graph construction phase, translated into a Program Dependence Graph (PDG) [72] in Static Single Assignment (SSA) [53] form by way of a Control Flow Graph (CFG). The program graph intermediate form is used by the analysis and transformation phases. Finally, during code generation, the control flow graph is rebuilt, the program is translated into Register Transfer Language (RTL) and target code is generated.

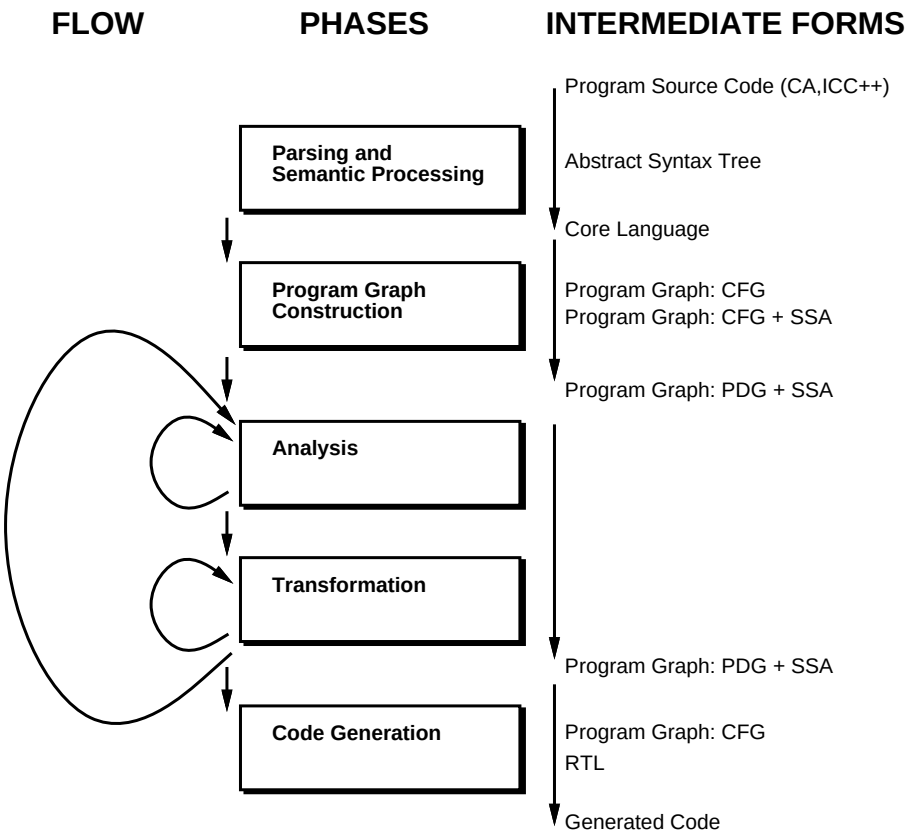


Figure 4.2: Concert Overview: Phases and Intermediate Representations

4.2.2 Retargetable Front End

One of the goals of Concert is portability of the system, the ability to compile a wide range of concurrent object-oriented languages. In order to meet this goal, we designed a retargetable front end. Since we wanted to preserve high level information, instead of immediate translation to RTL (e.g. like GCC [161]), the front end target is a simple "core" language. The language specific front ends (for CA and ICC++) pass a set of methods to the program graph construction phase. These methods are built from statements in the core languages over symbols (globally unique identifiers).

4.2.2.1 Symbols

Symbols are passed as a simple list or flat table of descriptions consisting of a unique identifier and a number of optional fields. There are two types of symbols: variables and classes. The

identifier is used both within symbol descriptions and within the core code to represent the symbol. It must be unique; program language level scopes must be flattened. Internally the identifier is represented as a reference to a symbol object. There is one optional field applicable to both variable and class symbols:

FIELD	TYPE	DESCRIPTION
name	string	The printable name of the symbol (e.g. variable name) used by the debugger etc.

several applicable only to variables:

FIELD	TYPE	DESCRIPTION
value	string, integer or floating point number	Indicates that this variable is a constant and gives the value.
type	class symbol identifier	The type used for dispatching purposes. For the `self` argument, this is class in which the method is defined. For other values, this is the type which the variable should be considered to be of for dispatching purposes (i.e. for `super` in Smalltalk, this is the superclass).
self	boolean	Indicates that this is a "syntactic self send". In both CA and ICC++, invocations which are made directly on `self` or `this` are considered to be part of the same transaction (Section 2.2.2.1).
global	boolean	Indicates that this is a global variable. An initialization body will be executed before the program proper.
counter	boolean	Indicates that this variable should use a counting continuation (Section 3.2.1.5).

and others only to classes:

FIELD	TYPE	DESCRIPTION
instance-variables	list of symbol identifiers	The instance variables for this class. Any inherited instance variables should be included here.
super-class	symbol identifier	The superclass used by dynamic dispatch for method lookup.

4.2.2.2 Core Language

A program in the core language is a set of methods in the form of an invocation template and statement. The invocation template has the form of a **send** statement where the selector is a string constant and the parameters are the arguments. There are five statements in the core language with two variations and one tag. These are presented in Table 4.2.

STATEMENT	ARGUMENTS	DESCRIPTION
send	list of symbols	An invocation.
move	symbol, symbol	Assignment of primitive types (including pointers and references).
if	symbol, statement, statement	Conditional.
while/ conc-while	symbol, statement	A while loop. **conc-while** indicates that the iterations can execute concurrently.
seq/conc	list of statements	A block of statements. **conc** indicates that they can execute concurrently.
future	list of symbols	Tag which indicates that the actual argument is a continuation for the future values of the symbols.

Table 4.2: Core Language Statements

Figure 4.3 is an example of the translation the polymorphic function `dbl()`. The class hierarchy is rooted at `rootclass` (all classes have rootclass as a superclass). There are four symbols including the argument whose **type** is used for dispatch, the selector `dbl` and a temporary value `temp`. The method defines the invocation template including the hidden continuation (Section 3.2.1.3) argument. In the body, the two **send** statements are nominally concurrent, though they will have to execute sequentially because the second requires the result of the first.

```
// polymorphic function to double numbers
dbl(x) {x+x}

// core language translation
(symbol class rootclass)
(symbol variable x :type rootclass)
(symbol variable dbl :value "dbl")
(symbol variable temp)
(method (dbl x c)
 (conc
   (send + x x (future temp))
   (send reply c temp)))
```

Figure 4.3: Core Language Example

Thus, the core language is a bare bones concurrent object-oriented language similar to Smalltalk but without even that language's syntactic conveniences. While it is quite powerful, it has restrictions. Control flow is restricted to **if** and **while** to simplify program graph

construction. Functions are restricted to the top level, so local data can not be accessed by more than one function. This enables later phases to easily preserve the consistence of local data. Both CA and ICC++ are translated into the core language in their respective parsing and semantic processing phases.

4.2.2.3 Parsing and Semantic Processing: CA

The CA front end is build using the PArser GENerator for Common Lisp by Ken Traub, part of the Dataflow Compiler Substrate [175] for the MIT Id 88.0 compiler. The parser generates an abstract syntax tree on which inherited and synthetic attributes are defined by Lisp functions and computed on demand. Since CA is quite close to the core language, this front end is concerned mainly with flattening scopes, constructing accessor functions and initialization code for objects and globals. As a result, it is quite small, requiring approximately two thousand lines of Lisp code.

4.2.2.4 Parsing and Semantic Processing: ICC++

The ICC++ front end was written by Julian Dolby with the help of Hao-Hua Chu and is derived from the CPPP parser from Brown University. It is built using a modified version of the Zebu parser by Joachim H. Laubsch combined with several recursive descent prepasses in the lexer. The parser generates an abstract syntax tree on which substantial computation must be done for type checking, insertion of coercion operations and overload resolution. ICC++ is not as similar to the core language as CA (Section 2.3.1). In particular, ICC++ allows more general control flow (`break`, `continue`, `return` and `goto`) which is translated into conditionals and while loops [69]. Nevertheless, the code related to core language translation is a small portion (twenty percent or three thousand lines) of the ICC++ front end which requires some fourteen thousand lines of Lisp code.

4.2.3 Program Graph Construction

The program graph intermediate form is a variation on the Program Dependence Graph (PDG) in Static Single Assignment form (SSA). Construction of the program graph generally follows standard algorithms, except for building CFG Data Dependences and Static Single Use (SSU) form (both described below). A program graph node is created for each core language statement,

presenting two views: one with with fields for sets of *rvals* (values read by this statement) and *lvals* (values written), and one with only *args* (asynchronous message arguments) including an explicit continuation. Local and traditional sequential optimizations are implemented in terms of rvals and lvals while interprocedural and concurrency minded optimizations operate on the continuation passing view. These nodes are embedded in four graphs. The Control Flow Graph (CFG) is constructed directly from the core language. Forward and backward dominator graphs are computed [124]. The Control Dependence Graph (CDG) is built and the program is then translated into SSU form using a variant of [53]. Note that since the core language contains only **while** loops, the CDG is a tree.

4.2.3.1 CFG Data Dependencies

We record CFG Data Dependencies in the program graph for user specified ordering of non-local operations. They are computed based on the semantics of concurrent and sequential blocks and loops. These, in turn, depend on the read-after-write relationships between local variables. For example, two **send** statements might otherwise be concurrent save that one is required to precede a write to a variable which is read in a statement required to precede the second (see Figure 4.4). When the program is converted into Static Single Use form, the dependencies for **move** statements are dropped since they are implicit in the def-use relationships.

```
...
  // The two sends must be executed in sequence
  (conc (seq (send meth1 a (future x))
             (move 1 b))
        (seq (move b c)
             (send meth2 d (future y))))
```

Figure 4.4: Data Dependencies Example

4.2.3.2 Static Single Use Form

Static Single Use (SSU) form is a variant on Static Single Assignment (SSA) form [53, 160]. SSA form inserts ϕ-Nodes, essentially assignments with multiple right hand sides where control flow merges. For example, after a conditional a ϕ-Node renames variables assigned in either branch. This ensures that that each variable will appear on the left hand side of only one

assignment. SSU form adds ψ-Nodes which, analogous to ϕ-Nodes, rename variables which are read along different control flow paths. These ψ-nodes appear before conditionals. Consider the code in Figure 4.5. In the code on the left, a is used under two different conditions (when *condition* is true and when it is false) and it is assigned twice. SSA form renames variables for each use and assignment. In addition to simplifying the construction of the flow graph, this renaming prevents interference between transfer functions during flow analysis (Chapter 5).

```
if condition                          (a1,a2) = ψ(a0);
  a = a.meth1;                        if condition
else                                    a3 = a1.meth1;
  a = a.meth2;                      else
                                        a4 + a2.meth2;
                                    a5 = φ(a3,a4);
```

Figure 4.5: Code before (left) and after (right) SSU Conversion

4.2.3.3 The Standard Prologue

Concurrent Aggregates is a very simple, pure object-oriented language. All program data are objects, and all operations are message sends.[1] Before any user code is compiled, the compilation environment is augmented by compiling a standard prologue (Appendix D). This prologue contains the builtin classes and functions accessible to the user. For example, the `integer` and `float` classes and their operations are defined there. Likewise, arrays, strings, globals, the constants `true` and `false`, and even the `nil` object are all defined in the standard prologue and their definitions can be overridden by any user program. For example, bounds checking on arrays is implemented simply by overriding the `at` and `put_at` operations on the `array` class. At the lowest level, these operations are defined in turns of *primitives* whose properties are known to the compiler [76]. Given this flexibility, the performance numbers in Table 4.1 for the prototype version of Concert are not surprising. Naive implementation of integer addition as a fully concurrent dynamically dispatched message send would result in extremely inefficient programs.

[1]Primitive control flow, because of its ubiquitousness, is not handled through messages. However, nothing in the system prevents the programmer from using Smalltalk style `True` and `False` classes, and `ifTrue`, `ifFalse` selectors to do so.

4.2.4 Analysis

The goal of the analysis phase is to determine flow sensitive information by constructing an interprocedural data and control flow graph. The nodes of this graph are program variables created under particular conditions and the arcs describe the flow of data. Because the values of data (classes or implementation types) affect the flow of control for object-oriented programs, the analysis determines control and data flow simultaneously. Some of the types of information analyzed for this phase are: implementation types of data, the call graph, sharing (aliasing) patterns, constants, stateless methods and lock subsumption (Section 8.1). The algorithm, described in detail in Chapter 5, is iterative, constructing successively refined approximations.

4.2.5 Transformation

The transformation phase consists of three types of transformations, *intraprocedural* which operate within a procedure, *interprocedural* which operate between procedures and *whole program*. Figure 4.6 shows these transformations and the (approximate) order in which they are applied. The intraprocedural transformations are applied periodically to simplify the program during interprocedural transformation. Cloning is discussed in (Chapter 6). General optimizations for object-oriented programs are covered in Chapter 7 and those specific to concurrent object-oriented programs in Chapter 8.

4.2.6 Code Generation

Code generation is the last phase in which the graph-based intermediate form is converted into an executable form. First, the control flow graph is reconstituted from the partial order of execution specified by the CDG and Data Dependences. Next, SSA assignments are moved from the conditionals (where they are attached for convenience of transformation) into the CFG. Touches are then inserted to enforce data dependences. The hybrid execution model (Chapter 9) requires separately optimized sequential and parallel versions of methods to be created. This is accomplished by translating the program nodes in the CFG into RTL. Registers are allocated over the RTL which is finally written out as a set of macros. These macros are converted by the runtime interface preprocessor into C++ which we use as a very slow but portable assembler.

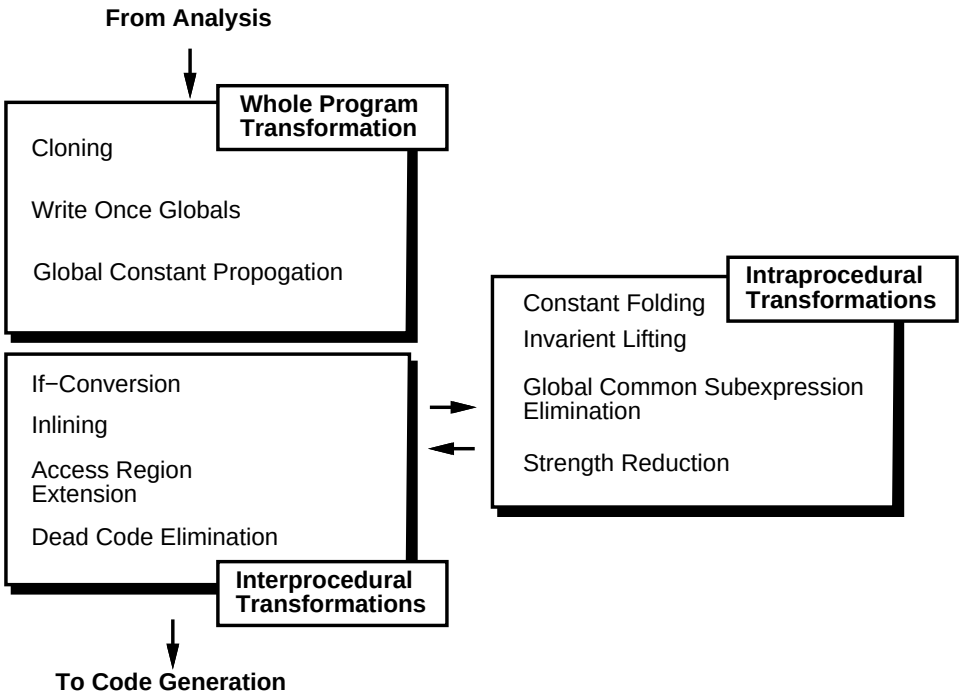


Figure 4.6: Order of Transformations

4.3 Runtime

The runtime manages processor and memory resources at runtime. It provides functions for communication, thread management, load balancing, scheduling, and memory management including global garbage collection. The interface (described in Section 3.5) is given as a set of `m4` and `cpp` macros over a linkable library. The runtime is written in C++ and assembly language, and consists of approximately twenty thousand lines of code.

4.4 Target Machines

Three different target platforms are available for executing Concert programs and used in the evaluation of the compiler. The first is SPARC [159] based uniprocessor workstations from Sun Microsystems Computer Company which is used to evaluate sequential efficiency. The second is the SPARC based distributed memory Connection Machine 5 from Thinking Machines Corporation. This machine shares the same instruction set architecture with the workstation, and includes direct, processor to processor, messaging. The last platform is Cray Research's

T3D which provides hardware support for a shared address space. These last two platforms are used to evaluate parallel overhead and performance.

4.4.1 SPARC Workstation

The SPARC workstation which we will use for evaluate is the SPARCstation 20 with SuperCache. This machine contains a 75Mhz SuperSPARC-II, 64 Megabytes of RAM, a one megabyte second level cache, 50Mhz MBus and 25Mhz SBus and has a SPECint_92 rating of 125.8 and a SPECfp_92 rating of 121.2. The SuperSPARC-II is capable of issuing three instructions per clock cycle. It has a 20-Kbyte instruction level 1 cache, a 16-Kbyte level 1 data cache, and a 64 entry TLB with hardware page-table walking. It is fully SPARC Version 8 compliant. The Mbus is capable of 400 Megabytes/second peak and 125 Megabytes/second sustained bandwidth. The workstation is system is running Solaris 2.4 and the tests were conducted in single-user mode.

4.4.2 Thinking Machines Corporation Connection Machine 5

The Connection Machine 5 (CM-5) machine used is a 512 node SPARC based machine at the National Center for Supercomputing Applications (NCSA). It is a distributed memory parallel machine with a memory mapped network interface residing on the main memory bus (Mbus). Five word packets are injected into the network by storing the data and destination node into designated addresses. Likewise, packets are received by polling a designated address. The processors are single issue units operating at 32 Mhz with an external cache. The network is a Fat Tree [71] capable of up to 20 Megabytes per second with cross-section bandwidth of 5 Megabytes per second per node. While the CM5 contains vector units capable of 140 Megaflops per second, Concert does not make use of these units.

4.4.3 Cray Research T3D

The T3D machine used is a 512 processor (256 node) DEC 21064 based machine at the Pittsburgh Supercomputing Center. It is a distributed memory parallel machine with hardware support for a global address space. Individual processors can read or write the address space of other processors directly. The runtime system builds an efficient messaging layer upon hardware atomic swap, write and prefetch queues. The processors are dual issue superscalar units operating at 150Mhz with an on-chip direct mapped 8-Kbyte data cache. The network is a

three dimensional torus with peak and sustainable processor transfer rates of 160 Megabytes per second and 73 Megabytes per second respectively. Cross section bandwidth depends on the exact topology, but each channel is capable of 300 Megabytes per second.

4.5 Related Work

Instead of surveying the breadth of compiler research, this section concentrates on those systems which either tackle similar problems in terms of programming and execution model, or which employ related optimization strategies. In the area of pure dynamically-typed languages, the various versions of the SELF system are notable for their approach, which consists of capturing and preserving dynamic information. Earlier versions of the system described by Chambers [30], guessed the types of data objects and inserting run time checks to verify the guesses. Inlining was used to increase the dynamic range of these checks, and transformations preserved the information. This approach proved brittle [95]. In later versions of the SELF system, Hölzle used profiling information to produced more robust performance [97]. The Cecil [32] system tackles the same problem by examining the class hierarchy to find methods which are not overridden [59], and using profiling information to specialize multiple-dispatch object-oriented programs [58].

In the realm of concurrent object-oriented programming, there have been a number of systems targeted specifically to fine-grained concurrency. The CST (Concurrent Smalltalk) compiler [100] is for a language largely similar to CA, but it did not perform global restructing. Similarly, [171] concentrates largely on high performance runtime facilities. The HAL system [102] originally provided only simple translation. Recent versions [115] have provided type inference, specialization of Actors constructs and an efficient runtime system. However, direct comparison is difficult because of differences between programming models. Earlier systems concentrated on efficient runtime systems and mappings to the hardware [152, 183, 171], or on language issues [184].

4.6 Summary

The Concert philosophy is that programmers should be concerned with natural expression of their programs and the programming system should produce an efficient implementation. The

goal of the Concert project is to produce portable, high performance implementations of concurrent object-oriented languages on parallel machines. The compiler embodies a compilation framework with five phases: parsing and semantic processing, program graph construction, analysis, transformation and code generation. During these phases, the program is translated into several intermediate forms, including and Abstract Syntax Tree (AST), a simple core language, a Control Flow Graph (CFG), a Program Dependence Graph (PDG) and Register Transfer Lanuage (RTL). In addition, the program undergoes a translation to Static Single Assignment form (SSA). The analysis and transformation phases include feedback loops, and operate on the PDG in SSA. A runtime system provides an abstraction of the hardware, and is used as the target for the compiler. Tests conducted using the Concert system are conducted on SPARC workstations, the Thinking Machines CM5 and the Cray T3D.

Chapter 5

Adaptive Flow Analysis

> You can't prove anything about a program written in C or FORTRAN. It's really just Peek and Poke with some syntactic sugar.
>
> *Bill Joy. The Unix Haters Handbook*

Control and data flow information forms the basis of program transformation. Without this information, it would be impossible to know whether or not a given transformation preserved the meaning of the program. Object-oriented programs are particularly difficult to analyze because the meaning of a statement depends on *context*, in particular, the classes of the objects involved. This chapter presents a context sensitive interprocedural analysis which adapts to the structure of the program to efficiently derive flow information at a cost proportional to the precision of the information obtained. Moreover, the results are applicable to such optimizations as static binding, inlining and unboxing.

This chapter is organized as follows. Section 5.1 describes basic flow analysis and defines notation. Section 5.2 introduces adaptive analysis. The contour abstraction and basic algorithm are presented in Section 5.3 and extended to the adaptive algorithm in Section 5.4. Recursion and termination are discussed in Section 5.5. Sections 5.6 and 5.7 discuss the implementation and empirical results. Finally, we cover related work in Section 5.8, and summarize in Section 5.9.

5.1 Background

Object-orientation increases both the importance of flow information as well as the difficulty of acquiring it. Object-orientation introduces an additional level of indirection at an invocation site: the code executed depends not only on the selector (generic function name), but on the class of the target[1] object as well. Thus, not only do data values depend on the flow of control, but the flow of control depends upon the values of data: the class of objects and the values of selector (function) variables. To optimize the program we need both good data flow information and good control flow information. Likewise, to analyze the program, flow analysis must simultaneously derive control and data flow information [154].

5.1.1 Constraint-based Analysis

Context sensitive flow analysis is a simultaneous control and data flow analysis which combines elements of abstract interpretation [50] and data flow analysis [74]. The flow graph is constructed by abstract interpretation and the approximations of data values are updated by propagation along the edges of the graph. Such techniques have been called *constraint-based* [136] because the flow graph resembles a constraint network, where the edges are constraints and the nodes are variables. For example, a flow analysis to determine the set of classes whose instances a variable might take on generates flow edges when an object of class `C` is created indicating that the result must be in the set containing at least `C`, {`C`}. Using $[\![v]\!]$ to denote the set of classes for variable v and N_v and N_C to denote the flow graph nodes for v and `C` respectively, the constraints and corresponding flow edges for object creation and assignment statements are:

program text	constraint	flow edge
$x = new\ C$	$[\![x]\!] \supseteq \{C\}$	$N_C \longrightarrow N_x$
$x = y$	$[\![x]\!] \supseteq [\![y]\!]$	$N_y \longrightarrow N_x$

When an invocation site is encountered during construction of the flow graph, the data flow values at the invocation site are used to conservatively approximate the method (function) which may be invoked based on the dispatch semantics of the language. For example, in a single-dispatch object-oriented language, the methods are determined by the data flow values

[1]The object to which the message was sent; in C++ the `o` in `o->method()`.

of the selector and target object arguments. The flow graph computes an approximation to these reaching problems.[2] For example, given a invocation site with reaching selectors S and a target object with reaching classes C, flow edges are constructed for all methods in the cross product $S \times C$ (allowing for inheritance).

Since abstract interpretation of the invocations (construction of the flow graph) uses the data flow values (the current solution) to determine the method invoked [135], the data flow values are updated concurrently with flow graph construction. The meet function for the data flow values is union and the data flow transfer functions are modeled by constraints on the values derived by abstract interpretation. For example, the constraint on the set of class names *Class* for the flow graph node o representing the target object with incoming data flow arcs from nodes o_i at an invocation site with possible methods $M \subseteq S \times C$ would be:

$$Class(o) = \{c \mid (x, c) \in M\} \cap \bigcup_i Class(o_i)$$

Likewise, constants, primitive functions and tests for equivalence with singleton objects (like NIL) produce constraints which affect the data flow values at nodes.

The context sensitivity of flow analysis follows from the *contour* abstraction [156, 105]. In the theory of flow analysis, the language to be analyzed is first given an exact semantics which is essentially an interpreter. The contours in such a semantics represent the call stack and determine variable bindings. For practical analysis, cost and precision are balanced by using abstract contours which represent some set of exact contours. A contour abstraction can therefore vary from coarse (one contour per method) to fine (one contour per call frame). Since flow graph nodes are created for each local variable for each contour, a separate (context sensitive) solution is obtained for each calling context represented by a contour. Thus, contours determine both the complexity (cost) of the analysis as well as the precision.

For example, 0th-order Control-Flow Analysis (0CFA) uses one contour per method while 1CFA uses one for each call site [155]. With each additional level of caller context sensitivity (Section 5.3.6), the precision of the information obtained increases, but the cost increases as well. In general, if we take a to be the average number of invocation sites for each method and S to be the number of statements in the program, the cost of NCFA is $O(Sa^N)$ (i.e. exponential

[2]This approximation is only safe when the analysis is complete, requiring the analysis to track changes to the approximation.

in N). Thus, a fixed contour abstraction is cost effective only for very small levels of caller sensitivity [106].

5.1.2 Program Representation

The program is represented in Static Single Use (SSU) form (see Section 4.2.3.2). This form creates a unique variable for each assignment under each set of local control flow conditions. Without these additional variables, constraints could not be safely applied to limit the values propagated through a node (variable) since the constraint might apply only to the variable under one set of control flow conditions. For example, on the left side of Figure 5.1 the variable `a` variously holds instances of class `A` and `integer`, to which are applied `geta` and `+` respectively. If the transfer function required that the type of `a` contain only those classes to which both `geta` and `+` can be applied, analysis would incorrectly report that no type could be found for `a`. SSA conversion prevents this problem by creating new variables for each assignment of `a`.

```
a = new A;
a.geta;
a = 1;
a = a + a;
```

```
if ...
  ... a + 1 ...
else
  a.geta;
... a ...
```

Figure 5.1: Analysis on Static Single Use Form

A more common problem is presented by the use (reading) of a variable under different conditions. In order to prevent these conflicts SSU form creates new variables for each use along different control flow paths. For example, in Figure 5.1 on the right, the two methods (`geta` and `+`) are applied to a variable in the two branches of a conditional. As with the two assigned values, `a` cannot be safely restricted to classes supporting both `+` and `geta`. Instead, the `a` in the left and right branches are restricted separately, and the `a` following the conditional has the meet (union) value of the two restricted `a`'s values. The result is that the final value may be of a class which supports either `+` or `geta`.

5.1.3 Contours

The key to the precision of context sensitive flow analysis is the *contour* [156] abstraction, the mapping of contours to the invocation environments of a method. Many different mappings are possible, and discovery of efficient contour abstractions specific to individual programs is

the subject of this chapter. Clearly, only those contours which represent "usefully distinct" environments need be created. Assume that the goal of flow analysis is to determine the classes of objects contained in variables and the code executed at invocation sites (see Section 5.5.3 for discussion of other flow problems). Environments are composed of the values of the arguments at the time of the invocation. The classes and methods associated with arguments therefore represent a useful basis of distinction. For example, the Cartesian Product Algorithm [2] differentiates contours based on the cross-product of the classes of the arguments. Likewise, Jagannathan and Weeks [105] differentiate contours based on the abstract values of the arguments.

The Cartesian Product Algorithm and Jagannathan and Weeks' solutions determine the contour abstraction immediately. That is, in a single pass of analysis, at the point where a contour is required, the abstraction is selected and fixed. This has two problems. First, the class of an argument does not capture all of its useful distinctions. The class may be polymorphic (Section 2.1.2) so that different objects of that class may have instance variables containing objects of different classes. For example, a instance of the `List` class may be a list of integer, a list of floating point numbers, or a list of lists. In order to capture these distinctions the complexity of the domain of values must be increased; instead of just classes, objects might be represented by their class and the classes of their instance variables. This could be extended some number of levels, and even to recursive and conditional types [8], incurring the concomitant cost.

The second problem is that object-oriented languages are imperative; objects are bits of potentially aliased albeit encapsulated state. This means that the alias structure for the entire store (all data) might affect the classes of objects accessed within a method. Thus, the problem of determining the code executed at invocation sites in object-oriented programs is equivalent to the alias analysis problem [118, 44, 60, 182, 147]. A typical method for finding safe approximations of the may-alias problem is to summarize groups of objects based on their creation point. This technique is applicable to flow analysis of object-oriented programs. For example, [135] differentiates objects by the statement at which they were created. However, this is simply a fixed 1-level abstraction which can be logically extended to N levels by differentiating objects based on the callers of the method containing the creation statement increases cost exponentially.

Our solution is to use a flexible abstraction which can be extended to different levels for different parts of the program being analyzed. This enables efficient analysis because, while it does not decrease the exponent, it does break up the cost into components. For example, the cost for N-level caller differentiation where each i is a regions of local extension falls from $O(Sa^N)$ to $\sum_i S_i a_i^{N_i}$. Each S_i corresponds to a set of polymorphic methods, and each N_i to the depth required to analyze S_i. The advantage of adaptation is in the precision with which invocation sites are mapped to the methods which might be invoked. When the number of such methods is close to 1, a_i is close to 1, bringing the cost of analysis close to $O(S)$ (linear) and enabling the invocation sites to be statically bound in the implementation (Section 7.3.1).

5.2 Adaptive Analysis: Overview

> It's quite possible to have a mixed abstraction, where the features of the program determine whether a precise, expensive abstraction will be used for a given contour, or an approximate, cheap one will be used instead. In fact, we could employ a kind of "iterative deepening" strategy, where the results of a very cheap, very approximate analysis would determine the abstractions used at different points in a second pass, providing precise abstractions only at the places they are required.
>
> *Olin Shivers. Thesis — Extensions*

Adaptive analysis proceeds stepwise by analysis and extension of the contour abstraction. The high level driver is given in Figure 5.2. The reason that these two steps are performed separately is that the structure of the flow graph is determined both by the contour abstraction and the data flow value which, in turn, are determined by the structure of the flow graph! Thus, changing the contour abstraction during an analysis step would either not increase precision (if the old conservative data flow values where preserved), or require invalidating and recomputing affected flow graph values and structure. By delaying changing the contour abstraction until the iteration has finished, we can use all the information produced to guide the changes.

5.3 The Analysis Step

Adaptive flow analysis consists of iteratively applying two steps: analysis and incremental precision extension. The analysis step constructs the flow graph while maintaining the updated data flow values at the nodes. SSU form, which induces an explicit local data flow graph,

```
void analysis_driver() {
  do {
    analysis_step();
    extension_step();
  } until ( extension successful )
}
```

Figure 5.2: Adaptive Analysis Driver

simplifies this construction over other analyses [105, 162]. In order to further simplify the algorithm, we assume that instance variables are accessed via accessor methods and that no variables are captured from surrounding scopes (Section 2.3.3).

5.3.1 Definitions

The definition of the flow graph appears in Figure 5.3. Each constant, expression and definition in the program is associated with a *Label*. Local variables (which are assigned only once) are associated with the expression which assigns them. Instance variables, which can be assigned multiple times, are associated with the label of their definition. These labels are used to uniquely identify the flow graph $Node$ representing the corresponding variable and to represent selectors, and classes in data flow values.

$$
\begin{array}{rclrcl}
n \in Node & = & Label \times Contour & N \in Nodes & = & \mathcal{P}(Node) \\
e \in Edge & = & Node \times Node & E \in Edges & = & \mathcal{P}(Edge) \\
c \in Contour & = & \mathcal{N} & C \in Contours & = & \mathcal{P}(Contour) \\
v \in Value & = & \mathcal{P}(Node) & V \in Values & = & Node \rightarrow Value \\
r \in Restrict & = & Value_1 \times ... \times Value_n & R \in Restricts & = & Contour \rightarrow Restrict \\
i \in Invoke & = & \mathcal{P}(Contour) & I \in Invokes & = & Node \rightarrow Invoke
\end{array}
$$

Figure 5.3: The Flow Graph

Contours are unique identifiers representing abstract calling environments; we use the natural numbers $\mathcal{N}$ where 0 is the top level environment. The $Value$ of a node is the set of nodes representing the values (constants, selectors, or object contours) which reach that node. Each

contour *Restricts* the values its parameters can take on. These restrictions represent qualities of the set of abstract calling environments which the contour represents (see Section 5.3.5). The *Invokes* function represents the abstract call graph by mapping invocation nodes to invoked contours.

The algorithm uses several functions to move around on the flow graph and extract information. *Flow* and *Back* move along the edges of the flow graph, taking a node and returning the set of nodes in the forward and backward data flow directions respectively. *Selectors* takes a node and returns the the set of selectors (labels of generic function names) or primitive functions of its value. That is, it finds the value of the node with respect to the *reaching selectors* problem. Likewise, *Class* takes a node and returns the set of labels for class names or primitive classes, and *Object* returns the set of contours for the constructor methods of objects referenced by the node. Constants are defined in all contours, so they are defined (arbitrarily) by the contour for the top level environment (0).

$$
\begin{aligned}
Flow(n) &= \{m \mid (n,m) \in E\} \\
Back(n) &= \{m \mid (m,n) \in E\} \\
Selectors(n) &= \{l \mid v' \in V(n) \wedge v' = (l,c) \wedge l \in \{\text{primitive function}, \text{selector}\}\} \\
Class(n) &= \{l \mid v' \in V(n) \wedge v' = (l,c) \wedge l \in \{\text{primitive class}, \text{class name}\}\} \\
Object(n) &= \{c \mid v' \in V(n) \wedge v' = (l,c) \wedge l \in \{\text{primitive class}, \text{class name}\}\} \\
Name(v) &= \{(l,0) \mid v' \in v \wedge v' = (l,c)\}
\end{aligned}
$$

Figure 5.4: Functions on the Flow Graph

5.3.2 Analysis Step Driver

Each analysis step begins by placing an interprocedural call graph edge (*Edge*) for the program entry point onto a worklist of *Edge*s. As each *Edge* is processed, new *Edge*s are found to be reachable and placed on the worklist. When the worklist is empty, the analysis step has finished. Figure 5.5 contains the pseudo code for the analysis step driver.

```
void analysis_step() {
  while ( worklist is not empty) {
     extract edge from worklist
    if ( edge has no contour) {
       find contour for edge
       create local flow graph
    }
     attach edge caller to callee
  }
}
```

Figure 5.5: Analysis Step Driver

For each call edge, if the edge does not have a contour from a previous analysis iteration, a contour is selected and the flow graph representing the local data flow inside the called method with respect to this contour is constructed. The local flow graph is attached to the flow graph of the calling method at the parameters and return values. Then this local flow graph is attached to the global flow graph at global variables. Global variables are not unique with respect to method contours and are represented by a single flow graph node. If the contour is not new, only the connections for the parameter and return value need be created since the existing local flow graph will be used to summarize both invocations.

5.3.3 Local Flow Graph Construction

The local flow graph consists of nodes representing the local, instance, and global variables and arcs representing data flow between them. The nodes of this graph are defined in Table 5.1. The node for a local variable is determined by its label and the method contour. The node(s) for an instance variable are determined by the *Object* contours of `self` (the target object of the accessor containing the instance variable). Global variables are unique, and determined by the contour representing the top level environment.

local	(l, c)	c is the method contour
instance	(l, o)	$o \in Object((\texttt{self}, c))$
global	$(l, 0)$	0 is the top level environment

Table 5.1: Local Flow Graph Nodes

The flow graph edges are those induced by data flow including the SSU assignments and reads and writes of instance and global variables. Data flow to or from instance variable is construed to be flow to or from all of the nodes representing that instance variable.

Constraints are imposed on the values of local variables based on how the variables are used. There are three types of local constraints: those for constants, those induced by the use of the variable in a dispatch position, and those induced by primitive functions. Constants are constrained to take on the appropriate abstract value: for example, integers, floating point numbers and strings are all immutable, so we say that they are created a priori in the top level environment and their *Class* value must include `integer`, `float` and `string` respectively. Objects in the dispatch position are constrained to be of a class supporting the selector(s) to which is applied, as in Section 5.1.1. Primitive functions can impose arbitrary constraints on the classes of their parameters. Figure 5.2 provides some examples of local constraints.

`1`	$Class((1,0)) \subseteq \{\texttt{integer}\}$
`1.0`	$Class((1.0,0)) \subseteq \{\texttt{float}\}$
`o.f`	$Class((o,c)) \subseteq$ classes supporting `f`
integer_add(a,b)	$Class((a,c)) \subseteq \{\texttt{integer}\}$ $Class((b,c)) \subseteq \{\texttt{integer}\}$

Table 5.2: Local Constraint Examples

Class is a derived quantity, so the constraints are reflected on the $Value$ of nodes. For example, the value of node $(1,0)$ must include $(\texttt{integer},0)$. Likewise, $V(\texttt{o},c)$ is constrained not to include any (n,c') such that n is a class name in the set of those supporting f, independent of the value of c'.

5.3.4 Global Flow Graph

The global flow graph is the collection of local flow graphs interconnected during the analysis step (Section 5.3.2). For each call edge, the set of possibly invoked methods is determined. Then contours are selected to abstract the environments of the invoked methods. These contours impose constraints on the values of arguments in dispatch positions (Section 5.3.5). Next, flow edges are constructed between the nodes in the calling method representing the formal

parameters and those in the callee method representing the arguments. Finally, any new call edges are added to the worklist.

The called methods are drawn from the applicable methods with the selectors which reaching the invocation. Methods are applicable when an object of the appropriate class can reach each argument. That is, for arguments a_i and parameters p_i, the intersection of $Class(a_i)$ and $\{l \mid (l,c) \in (R(p_i)\}$ is non-empty. Likewise, the selectors reaching the selector argument ($Selectors(a_0)$) must include the name of the selector of the method. Section 5.4.2 discusses selection of contours in more detail.

When the values of arguments change, new methods and contours become reachable, requiring edges to be added to the worklist. Since the data flow graph is maintained up to date, new edges may be created anywhere in the global flow graph in response to the addition of a single local constraint. For example, the addition of an `integer` constant constraint can causes the value of all reachable variables to contain (`integer`, 0) which, in turn, can cause a large number of new edges to be created for each invocation site where those variables appear in the dispatch position. Thus, each time the value of a node changes, all affected invocations must be examined and any new edges added to the worklist.[3]

5.3.5 Restrictions

Constraints are imposed on parameter nodes correspond to the feasible call edges under the dispatch semantics (Section 5.1) and the contour abstraction (*Restricts*). Thus, values of the parameters are subject to the restriction: $V(p_i) \subseteq R(p_i)$. For example, Figure 5.6 on the left shows a polymorphic method which is applied to two arguments which could be either integers or floating point numbers. On the right are several different sets of contour restrictions. The first provides a single contour covering all cases, the second is more specific, and the last provides for all four possible classes. Thus, restrictions enables the use of separate contours for different combinations of values. Since any given variable can only hold one value at one time, separate analysis for different combinations is safe so long as each element of the cross product of values is represented by some contour (Section 5.4.2). This is achieved in the alternative contour representation of [105] by single-value based analysis of curried functions.

[3]Our implementation attaches a list of such invocation sites to each node.

```
f(i,j) { ... }
...
if ... a = 1; else a = 1.0;
if ... b = 2; else b = 2.0;

c = f(a,b);
```

({i,f},{i,f})
({i},{i,f}) ({f},{i,f})
({i},{i}) ({i},{f}) ({f},{i}) ({f},{f})

where i = (**integer**, 0)
and f = (**float**, 0)

Figure 5.6: Restrictions

5.3.6 Imprecision and Polymorphism

To simplify the exposition of the algorithm, we differentiate *method imprecision* from *object imprecision*. Imprecisions are flow graph nodes whose values are not singleton sets. Method imprecisions are those of nodes defined by the surrounding method's contour (local variables). Object imprecisions refer to nodes defined by object contours (instance variables). Imprecisions can result from a number of sources including incomplete input, flow insensitivity, and (for mutable locations) temporal insensitivity. This analysis focuses on the second sort which often results from the use of polymorphic methods or objects. The *level of polymorphism* is the depth of the polymorphic method call path or polymorphic reference path (see Figures 5.7 and 5.8). 0CFA handles no polymorphism, 1CFA [154] handles one level, and this algorithm adapts to handle different levels of polymorphism in different areas of the program.

```
power(x,y) {
  if (y>0)
    x*power(x,y-1);
  else
    one(x);
}

power(1,2);
power(1.0,2);
```

Figure 5.7: Method Polymorphism

```
class tuple {
  l;
  r;
  left() { l }
};

let a = tuple(1,2),
    b = tuple(1.0,2);
  a.left;
  b.left;
```

Figure 5.8: Polymorphic Objects

5.4 Adaptation

Adaptive flow analysis uses the results of the previous iteration (starting with 0CFA) to extend the contour abstraction for the next iteration. After each iteration, the contour abstraction

is extended by *splitting* the set of invocations associated with a contour (see Figure 5.3) to differentiate uses of the method or class it represents. A new analysis iteration starts by clearing the values V and the edges E which make up the flow graph. However, the abstract call graph I which captures the local levels of context sensitivity is preserved. In this way, the analysis adapts to the structure of the program across iterations.

5.4.1 Splitting

Splitting divides contours, increasing the number of flow graph nodes and potentially eliminating imprecisions from the analysis results. Splitting polymorphic methods (method splitting) divides the invocations associated with a method contour over a number of smaller of more specific contours. Splitting polymorphic objects (object splitting) divides the invocations associated with the creation of objects of a particular class over a number of contours representing subsets of the instances which are used in different ways.

The simplest form of splitting relies on argument values, selecting a contour the values of whose formal parameters most closely match those of the invocation arguments. Invocations are processed in order so the arguments have approximations of their final values when the contours for each invocation are selected. To minimize the number of analysis iterations, this partial information is used to "eagerly split" method contours; i.e. select more precise contours to represent the abstract calling environments of new edges. Similarly, we can eagerly split contours representing objects. However, since the selection of object contours occurs at the point where the objects are created, before the instance variables are used, it generally is less effective. Eager splitting occurs as part of contour selection.

5.4.2 Selecting Contours

When an invocation is encountered, the set of applicable methods is determined and the contours are selected. There are two goals for contour selection. First, the contour should not be overly general. That is, the contour should not be used to abstract invocations for which different information is available, since a conservative approximation of the information of all invocations will be used for analysis of the contour. Secondly, contours should be shared whenever possible. Clearly we could satisfy the first goal by selecting a new contour for every invocation,

but then the analysis would never terminate. Sharing contours effectively is the key to efficient analysis.

For a given target method, the conditions of dispatch induce constraints which are applied to the values of the arguments to determine the values which will flow into the parameters for this invocation (Section 5.3.4). While a contour could be created for each element of the cross product of entering values ($w = \prod_i v_i$), this would be expensive and, in general, prevent termination (see Section 5.5). Instead we select contours based on information from the last iteration and then eagerly split contours based on the *Label* component (*Selectors* or *Class*) of the argument values, leaving splitting based on the contour component (*Object*) of the values to be done non-eagerly. For example, Figure 5.9 shows three invocations (left) with different argument values (right). The argument values of the first two invocations differ only in the contour component, so they will share the same contour while the value of first argument of the last invocation has a different label (the class `B`) and will not share the same contour.

`func(new A,new A);`	$\{(\mathtt{A}, c_1)\}\{(\mathtt{A}, c_2)\}$
`func(new A,new A);`	$\{(\mathtt{A}, c_3)\}\{(\mathtt{A}, c_4)\}$
`func(new B,new A);`	$\{(\mathtt{B}, c_5)\}\{(\mathtt{A}, c_6)\}$

Figure 5.9: Selecting Contours

The contours for an invocation from node n are selected in three steps given in Figure 5.10. First, from the cached contours $I(n)$ we select those whose restriction cross product $\prod_i r_i$ intersects w, favoring those which intersect the smallest number of elements, and remove those elements from w. For any remaining elements of w we select from all contours associated with the method those whose restrictions intersect w. Finally, we form subsets out of any remaining elements by applying $Name$ to each parameter value ($\prod_i Name(v_i)$) and create contours for each identical result with the singleton $Name$s as restrictions. Intuitively these contours are insensitive to particular contours reaching their parameters, but are (eagerly) differentiated with respect to the names of the methods or classes reaching those parameters.

As a special case, a unique contour is always created for accessor methods for each *Object* value of the node representing the target object. This simplifies the exposition of the algorithm by eliminating the boundary case where the accessor method contours require method splitting to isolate accessor operations to particular object contours. This enables a cleaner separation between method and object contour splitting.

```
select_contours() {
  for an invocation node n with arguments a_i
  w = ∏_i v_i where v_i = V(a_i)
  for each c ∈ I(n) such that w ∩ ∏ R(c) is not empty
    select c
    w = w − ∏ R(c)
  while w is not empty
    select c, a new context with restrictions NAME(w_0)
    w = w − {x | NAME(x) == NAME(w_0)}
  where
    NAME(x) = ∏_i Name(x_i) }
```

Figure 5.10: Selecting Contours Pseudo Code

5.4.3 Method Contour Splitting

Splitting method contours enables separate information to be obtained for different uses of the method. The idea is to examine the data values after an iteration, find situations where the caller argument values of a call edge are more precise than the values of the corresponding callee parameters, and build new contours for the cases. For example, if the value of one of arguments for a particular invocation is a strict subset of all other corresponding arguments' values, a new contour is created for that invoke. In the new contour, the corresponding formal parameter will have the (more precise) subset value.

Since the domain of values is recursive, splitting for every difference produces a nonterminating analysis. Moreover, not all differences in argument values are meaningful. For example, object contours for a particular class definition may distinguish subsets of the class's instances which are important to only a fraction of methods. So, instead of splitting for every difference in values, we start from a specific imprecision which we wish to eliminate (e.g. where a type check, boxing operation or dynamic dispatch would be required) and look for the imprecisions which caused it. This goal-driven analysis also has the advantage that resource use scales with the precision demanded.

Using the functions described in Figure 5.4 we traverse the flow graph. Starting from the point of imprecision we look back for a set of confluences[4] of values; in data flow terms, meets $a \wedge b$ where $a, b \neq \emptyset$ and $a \neq b$. Given a node n with imprecise Val (one of $Selectors$, $Object$ or $Class$) we find the least set of confluences $Conf(n, Val)$ which obey:

$$\begin{aligned} Conf(n, Val) &\supseteq \begin{cases} \{n'\} & \text{if } \exists n' \in Back(n) \wedge Val(n) \neq Val(n') \\ \emptyset & \text{otherwise} \end{cases} \\ Conf(n, Val) &\supseteq Conf(n', Val) \text{ where } n' \in Back(n) \end{aligned}$$

The contours of parameter nodes in $Conf(n, Val)$ represent the first order contribution to the imprecision, and splitting them is the first way in which the imprecision may be eliminated. Imprecision can also arise from interprocedural control flow ambiguity due to secondary imprecisions in other arguments. For example, since the result value is the meet (union) of the results of all the possible methods invoked, if the set of selectors reaching an invocation is imprecise, the result can be imprecise. Similarly, the parameters of the invoked methods might be imprecise as a result of the extraneous flow edges. Lastly, imprecisions in object contours can lead to imprecise results of instance variable accesses. We extend $Conf(n, Val)$ to $Conf'(n, Val)$ to handles these cases, where i is an invocation:

$$\begin{aligned} Conf'(n, Val) = Conf(n, Val) \cup \\ \{(n'' \mid n' \in Conf(n, Val) \vee |Val(n)| > 1) \wedge \\ (n' \text{ is an argument or return variable of } i \ \wedge \\ (n'' \in Conf'(DispatchArgument(i), Class) \vee \\ n'' \in Conf'(SelfArgument(i), Object) \vee \\ n'' \in Conf'(SelectorArgument(i), Selectors)))\} \end{aligned}$$

The three occurrences of *Conf'* on the right hand side account for the effects of imprecise dispatch arguments, imprecise object contours, and imprecise reaching selectors respectively. The newly split contours are distinguished in the next analysis step through the changes to

[4]This is not to be confused with the Church-Rosser property, though both draw on the common definition; from the Oxford English Dictionary (second edition) **confluence**: *A flowing together; the junction and union of two or more streams or moving fluids.*

abstract call graph I or additional restrictions R. For confluences ($Conf(n, Val)$ is non-empty) (l, c), we create new contours C' with identical restrictions $\forall c' \in C'.R(c') = R(c)$, but with I mapping callers with identical values to separate contours (i.e. $\bigcup V(v' \in Back((l, c')) = V((l, c'))$). For an imprecision ($|Val(n)| > 1$) at argument node n at position i, we create new contours with identical invokes ($\forall l \in Label.I(l, c') = I(l, c)$) and modify the restrictions $r = R(c')$ to differentiate the elements of $Val(n)$ (e.g. $|V(r_i)| = 1$). Different contours will then be selected in the next iteration.

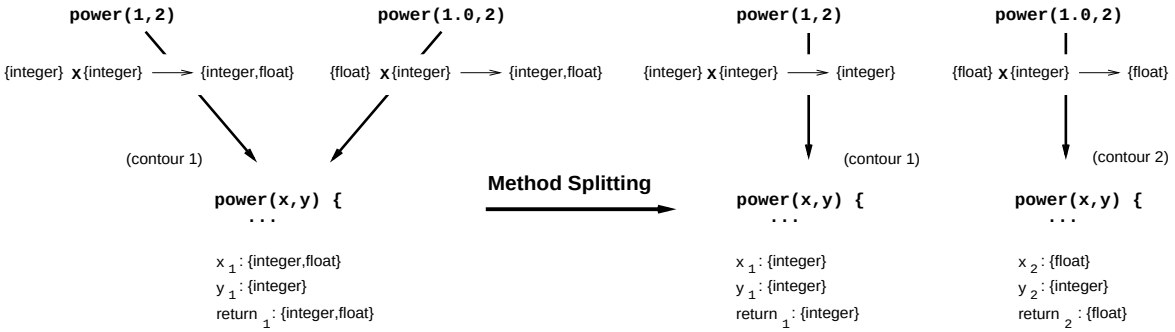


Figure 5.11: Method Splitting for Integers and Floats

Figure 5.11 illustrates method splitting involving the `power` method from Figure 5.7. Note that this particular situation would have been taken care of by eager splitting, however, rather than complicate the example, we will simply imagine that it was not. At the left, the actual arguments for the formal parameters `x` and `y` coming from `power(1,2)` and `power(1.0,2)` have different *Class*es, so there is a confluence. The imprecision manifests itself in a confluence at the first argument {`integer`,`float`}, when it is clear that the value for the first call is `integer` and for the second `float`. Splitting introduces two sets of nodes x_1,y_1 and x_2,y_2, eliminating the confluence.

5.4.4 Object Contour Splitting

Object splitting partitions contours based on the usage of the objects they represent. It is more complex than method splitting because the point of confluence (the instance variable) is separated from the point at which the contour was created in the flow graph. In fact, since object contours flow through the graph, splitting the object contour alone is not enough; we must ensure new contours remain distinct as they flow through the flow graph. Note that

this requires splitting intermediate methods, like the identity method, which make no use of the instance variables of the argument. However, these extra contour are coalesced by cloning (Chapter 6) and do not affect optimization.

Figure 5.12: Object Splitting for Imprecision at `l`

Figure 5.12 is an example of object splitting based on the program example in Figure 5.8. On the left, the two creation points, `tuple(1,2)` and `tuple(1.0,2)` produce the same contour. As a result, the value of the instance variable `l` is {`integer`,`float`}. Splitting the object contour discriminates the two cases, producing precise results for both cases.

Again, we start with an imprecision we wish to eliminate. Object splitting involves four operations.

1. Identifying the assignments to the instance variable which give rise to the imprecision.
2. Identifying the paths in the flow graph which the instance variable's contour took from its creation point to the assignments.
3. Ensuring that these paths are distinct.
4. Dividing the object contour into a set of contours.

The first step is to identify the conflicting assignments to the same instance variable with the same contour. Next we find the flow paths from the creation of the contour c which defines the instance variable node n (e.g. $n = (l, c)$) to the assignments. These paths must be disjoint to propagate the distinct contours we introduce by object splitting, or we will fail remove the imprecision. Instead, all variables which after the paths have conjoined will hold both contours, and, for instance, the read accessors will return the union of values of the corresponding instance variable for both contours. We ensure disjointness by using method and object splitting along the flow paths where necessary. When the paths are disjoint, the conflicting values are assigned into different contours by splitting the original object contour.

```
class A {
  a;
  operator=a (value)  a = value;
}

let x = A()_c,              ;; I((c,0)) = {1})
    y = A()_d;              ;; I((d,0)) = {1})
  x.a = 1_e;                ;; I((e,0)) = {2})
  y.a = 1.0_f;              ;; I((f,0)) = {3})
```

Figure 5.13: Object Splitting Example

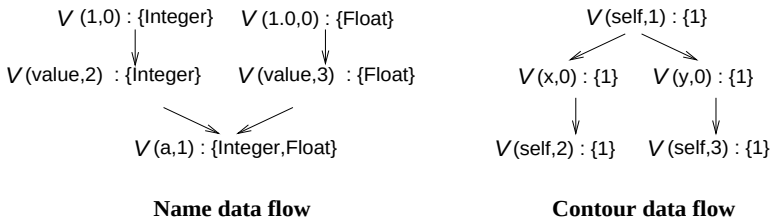
To illustrate the algorithm, we will use the example in Figure 5.13. In this example, two instances of class `A` are created. The write accessor method `operator=a()` (Section 2.3.3) is then used to set the `a` instance variable of each to a different type of number.

Identifying the Assignments First, the nodes which are assigned (have a flow edge to) the imprecise instance variable are found. These are grouped so that all the nodes in each group have identical values with respect to the type of the imprecision, indicated by the parameterizing function Val (again, one of $Selectors$, $Object$ or $Class$). We define the function $AssignSets(n, Val)$ which takes a node n, an imprecise instance variable, and return a set s of sets of nodes s_i, each of which represents a different use of the instance variable.

$$AssignSets(n, Val) \quad = \quad s \text{ where } \bigcup_i s_i = Back(n) \wedge \forall s_i \in s, n' \in s_i, n'' \in s_i . Val(n') = Val(n'')$$

The nodes in $Back(n)$ are the right hand sides of assignments to the imprecise instance variable. Figure 5.14 shows the flow graph for our example, and the assignment sets derived.

Identifying the Paths Next, we compute the flow paths which the instance variable's contour took from its creation point to the assignments. For each element a of $AssignSets(n, Val)$ we find the `self` nodes $Self(a)$ of the accessor methods which contain the assignments a. The $Object$ value of these nodes are the contours which defines the instance variable node n (i.e. for $o \in Object(s \in Self(a))$, $n = (l, c)$). Then we compute the paths taken by the contours from their creation point (the node (`new`...$, c$)) to $Self(a)$. These paths must to be distinct in order



$$\mathbf{AssignSets}((\mathtt{a}, 1), \mathbf{Name}) \quad = \quad \{\{(\mathtt{value}, 2), (\mathtt{value}, 3)\}\}$$

Figure 5.14: Data Flow and Assignment Sets Example

to eliminate the imprecision. Intuitively, if the contours' paths merge they will be applied to the same methods with the same values.

$$\begin{aligned} a &\in AssignSets(n, Val) \\ Self(a) &= \{(\mathtt{self}, c) \mid (l, c) \in a\} \\ Path(a) &= Closure(Back, Self(a)) \end{aligned}$$

We compute the paths $Path(a)$ for each assignment a by taking the closure of the function $Back$ over the set containing the `self` nodes for the methods containing the assignments. The `self` nodes are found with the function $Self(a)$ by extracting the method contour from a.

The paths $Path(a)$ are those which would be taken by a new contour created to eliminate the portion of the imprecision $Val(a)$. For example, in Figure 5.15 contour **1** travels to `(value,2)` through `(self,2)` and `(x,0)`. Since this path must be distinct from the other paths the appearance of a node on more than one of these paths represents a secondary imprecision. For each node we need to know the subset paths in which it is contained.

$$\begin{aligned} AllPaths(n, Val) &= \{p \mid p = Path(a) \wedge a = AssignSet(n, Val)\} \\ NodePaths(n', n, Val) &= \{ps \mid n' \in ps \wedge ps \in AllPaths(n, Val)\} \end{aligned}$$

We define the function $AllPaths(n, Val)$ to be all the paths for all the assignment sets. Further, we define $NodePaths(n, Val)$ to be the subset of all the paths which a particular node n' is on.

Ensuring Discrimination Using the paths determined above, we now apply the confluence finding algorithm recursively to determine the confluences of the potential contours represented by these paths. However, the paths are defined by the assignments and join at the creation point whereas the other values are distinct when created but join at merges in the flow graph. Thus the path can be thought of as flowing backward in the data flow graph. This requires modification of the $Conf$ function:

$$FlowOrBack(Val) = \begin{cases} Flow & \text{if } Val = Path \\ Back & \text{otherwise} \end{cases}$$

$$Conf(n, Val) = \begin{cases} \{n\} & \text{if } \exists n' \in (FlowOrBack(Val))(n) \ \wedge Val(n) \neq Val(n') \\ \emptyset & \text{otherwise} \end{cases}$$

The new $Conf$ uses the $FlowOrBack(Val)$ function which is either $Back$ as before or $Flow$ when Val refers to the paths. $AssignSet$ requires an analogous change, and the rest of the algorithm is identical.

$$\begin{aligned} Path(\{(\texttt{value}, 2)\}) &= \{(\texttt{self}, 2), (\texttt{x}, 0), (c, 0)\} \\ Path(\{(\texttt{value}, 3)\}) &= \{(\texttt{self}, 3), (\texttt{y}, 0), (d, 0)\} \end{aligned}$$

$$Splittable(0, (\texttt{a}, 1), Class) = \{\{(c, 0)\}, \{(d, 0)\}\}$$

Figure 5.15: Paths and Splittability Example

Resolving the Imprecision The last step is the actual splitting of the object contours. When two or more paths do not share any nodes, the contour is split and a new contour created for each path or set of paths not sharing nodes. Figure 5.15 provides an example of a contour which is determined to be splittable. The new contours will cause the accessor methods and the node representing the instance variable at the point of the imprecision to split, removing the imprecision. The function $Splittable(c, n, Val)$ determines the subsets of creation points for contour c which can be profitably split for the imprecision at node n of type Val:

$$
\begin{aligned}
Splittable(c, n, Val) &= \{t \mid t \subset s \wedge \bigcup NodePaths(n' \in s, n, Val) \neq AllPaths(n, Val) \wedge \\
&\qquad \forall n' \in t, n'' \in t, NodePaths(n', n, Val) \cap NodePaths(n'', n, Val) \neq \emptyset\} \\
\text{where } s &= \{n \mid n \in p \wedge p \in AllPaths(n, Val) \wedge Back(n) = \{(\texttt{self}, c)\}\}
\end{aligned}
$$

Using s the nodes which represent the creation points for the object contour c, we determine the subsets of creation points whose paths are not disjoint. Since the creation points are the end of the paths, non-disjointness implies equality and that the union of these subsets will be s. Further, since creation points correspond to invocations on the class (object creation) function, $Splittable(c, n, Val)$ computes the sets the invocations for the new contours. Thus, if

$$
Splittable(c, n, Val) = AllPaths(n, Val)
$$

the object contour cannot be profitably split. When the object contour is split, the newly created contours are substituted for the original in the restrictions for argument nodes along the corresponding paths. Thus the new contours will follow the distinct paths in the next iteration, and their instance variables will be assigned a subset of the values of the original, removing the imprecision.

5.5 Remaining Issues

In this section we discuss recursion, termination and complexity issues and the applicability of this analysis to other data flow problems. Since the contour representation used by adaptive flow analysis is not static but recursively defined, recursion in the program being analyzed requires special handling.

5.5.1 Recursion

Since the definition of contours is recursive, ensuring termination requires limiting the number of contours produced by recursive program structures. There are three types of these structures:

- Recursive methods

- Recursive data structures
- Method-data recursion

The first two types are normal recursive methods and recursive types. The third type represents the case where a recursive method creates objects on which it is later invoked. This is the case for such common programming idioms as insertion into a linked list. While contours are represented by unique identifiers, their uniqueness is determined by their callers I and their restrictions R. The first two types of recursive structures induce other contours by invocation I while the third induces them through restrictions R.

After each iteration and before splitting, we identify the strongly connected components (SCCs) in the graph whose nodes are the contours and whose edges are:

- a contour c and the contours it invokes $\{c' \mid c' \in I((l, c))\}$
- a contour c and the contours it restricts $\{c' \mid (l, c) \in R(c')_i\}$

The SCCs in this graph contain the contours which have a part in defining each other. To prevent non-termination we do not allow invocations or restrictions between contours in the same SCC to cause splitting. Furthermore, invocations into recursive cycles can also lead to non-termination as recursive cycles are successively "peeled". These invocations are also prohibited from splitting beyond a constant level (in our implementation, two levels). Allowing invocations on the cycle to split to a constant level enables analysis of recursive structures with a period less than or equal to the constant since contours can form cycles up to that length.

5.5.2 Termination and Complexity

Termination is ensured by limiting the number of contours produced by recursion. However, since $Node$s and $Value$s are defined by labels (program points) and contours, which in turn can be distinguished by their values at each argument, the theoretical complexity is exponential. Nevertheless, in practice, we have found the complexity to be related to both the size and levels of polymorphism of the analyzed program. Furthermore, we have found that the level of polymorphism in programs increases relatively slowly with program size, and the complexity of analysis along with it (see Section 5.6 for an empirical evaluation).

5.5.3 Other Data Flow Problems

Adaptive flow analysis can be applied to other data flow problems where the precision of deep flow sensitivity is desired. There are two classes of such problems, those that flow forward through only local variables and parameters, and those that flow through instance variables. Since each additional problem adds another dimension to the splitting criteria, additional levels of recursive unfolding (Section 5.5.1) are required to prevent the solution of earlier problems from interfering with any unfolding of recursive methods required for additional problems. To prevent such interference additional problems should be solved in order (that is, all iterations required for one problem should be completed before any splitting is done for any later problem).

For those problems which that flow only through local variables and parameters, only method splitting need be considered. If the domain is small, for example the distinction between *local* and *remote* objects, the data flow problem can be simply added as a new type of imprecision (i.e. bound to Val in Section 5.4.3). If the domain is larger, for example constants, splitting should be limited to those imprecisions likely to be of importance for optimization. Since adaptation is goal driven, integrating optimization criteria is simply a matter of choosing the initial imprecisions (Section 5.4.3).

The second class of problems are those for which flow sensitivity is required for values flowing through instance variables. The resulting information can be used to specialize memory layout of generic classes. For example, analysis of a `list` class which shows that, for a subset of creation points, no destructive updates (i.e. `set-cdr!`) are performed can be used to "CDR code"(allocate the contents in consecutive memory locations). Since all imprecisions at instance variables are resolved or become imprecisions in the paths of potential object contours, no special mechanism is required to add additional problems. However, large domains may require use of optimization criteria.

5.6 Implementation

The full implementation of this analysis involved engineering decisions which are of sufficient interest to warrant discussion here. As construed, this analysis makes some simplifying assumptions about the structure of the program (e.g. the use of accessor methods). Also, certain language features are not specifically covered: arrays, closures and first class continuations. To

make impact of these assumptions more tangible, the main data structures of our implementation are presented, and related to the assumptions about program structure. Finally, support of arrays, first class continuations, and closures is discussed.

5.6.1 Data Structures

The four main data structures describe the interprocedural data and control flow graphs. These are flow graph nodes, interprocedural call edges, method contours and object contours. The two types of contours are separated in the implementation because they require different handling for splitting, recursion, etc. To simplify matters, each method contour is associated with a single object contour (that is, we automatically split the method contour based on the object contours of the target object), see Section 5.4.4.

Flow Graph Node

id The unique identifier of the node, composed of a program variable (label) and method contour and/or object contour.

flow Set of nodes in the forward direction in the flow graph.

back Set of nodes in the backward direction in the flow graph.

upper-type Constraints limiting the type of the variable (e.g. the use of the variable as the target of a message send – Section 5.3.3).

lower-type The current estimate of the variable's type. This is used to determine the applicable methods (Section 5.3.4). (derived from **object-contours** below).

object-contours The current estimate of where objects referenced by the the variable could have been created.

selectors The reaching selectors (generic function names).

continued-value The representative return value nodes for continuations.

path-sets The paths this variable is on which might carry a needed object contour (see Section 5.4.4).

arg-of-message The invocation nodes which might generate new edges for changes in variable.

The **id** is a tuple containing fields for program variables, and both method contour and object contours since some variables are determined by only object contour (instance variables)

or neither (global variables). New outgoing edges resulting from changes in the data flow values of arguments in the dynamic dispatch positions are triggered by checking the checking the **arg-of-message** field when a nodes value changes.

Interprocedural Call Edge

invoking-statement The statement from which the edge originated.

source-contour The method contour from which the edge originated.

object-contour The object contour of the target object which determines **contour**.

contour The contour on which the edge is incident.

parameters The nodes representing the invocation parameters.

The unique identifier of an edge is a tuple containing **invoking-statement**, **source-contour**, **object-contour**, **selector**. That is, it contains the flow sensitive invocation site and the inputs to the dispatch function. This identifier is used to recover the edge in successive analysis iterations from a hash table; since the recovered edge contains the `contour` this table represents the $I \in Invokes$ from Section 5.3.1 which is preserved from iteration to iteration.

Method Contour

method The method for which this contour represents a set of activations.

restrictions The values which can pass in the formal parameters of this contour.

edges The edges representing invocations on this contour.

invokes-edges The edges representing invocations from this contour.

creation-points The nodes representing the nodes at which objects are created within this contour.

recursive-set The set of recursive cycles containing this contour.

Method contours have an identity, which is dictated both by its **method** and **restrictions** as well as its position in the cached interprocedural call graph as dictated by **edges**. The **invokes-edges**, **creation-points** and **recursive-set** fields are used to cache information used in handling recursion (see Section 5.5.1).

Object Contour

creation-points The nodes representing the context sensitive statements where objects described by this contour are created.

method-contours The method contours determined by this object contour.

containing-contours The object contours in whose instance variables objects with this contour are stored.

recursive-set The set of recursive cycles containing this contour.

Object contours are uniquely determined by the **creation-points** they represent. The **method-contours** field is used to quickly determine which nodes may be effected by a split of the creation set during the clearing of the values between iterations. It is also used along with **containing-contours** and **recursive-sets** to handle recursion (see Section 5.5.1).

5.6.2 Accessors

The analysis as described requires all instance variables to be accessed through accessor methods. The reasons for this are two fold. First, they allow the functions which walk the data flow graph to map from a node back to program statements. Instance variables are not associated with a method contour, hence, without an interposing local variable it would be difficult to find the method(s) responsible for a data flow arc between two instance variables. Second, if instance variable accesses were allowed into arbitrary methods, the object contour determining the instance variable node might come from a parameter, a return value or another instance variable. This would require additional rules both to generate the flow arcs and in the $Conf'$ function (Section 5.4.3) to determine the possible cause of an imprecision. Requiring the use of accessors is somewhat conservative. In fact, the implementation supports direct access to instance variables within all methods for single dispatch languages so long as an intermediate local variable is used for assignments between instance variables.

5.6.3 Arrays

Analysis of the contents of arrays is handled analogously to instance variables. Since the analysis is temporally insensitive for instance variables (all reads and writes are to a single node representing the instance variable independent of where they take place in program) and

since a node summarizes all instances represented by an object contour, having a single node summarize all accesses to all elements of an array is a safe, conservative technique. That is, the contents of arrays can be analyzed homogeneously as a single instance variable, using a special *Label* to represent array contents. However, the two dimensions of precision (temporal and element-wise) are amenable to additional techniques. An example of temporal sensitivity is covered in Section 5.6.4.

Element-wise, separate variables can be used to represent subsets of elements. For example, the first class messages of CA are essentially vectors of arguments which can be manipulated as arrays and then "sent" as messages. These message values are analyzed by using a separate node to represent each argument at a known offset and a node representing all other elements. Accesses to using constant indices are applied to the appropriate element while accesses with unknown indices are applied to all including the node for other elements.

5.6.4 First Class Continuations

First class continuations (Section 3.2.1.4) [41, 35] are essentially objects which are used to return values to a future. Since they are ubiquitous (used for every return value) in our COOP programming model, the implementation uses a special temporally sensitive mechanism instead of the standard object contour splitting mechanism. The values returned by continuations are represented by a secondary nodes attached to the primary node representing the continuation proper. That is, a set of secondary flow graphs are created, running parallel to the flow of the continuation but in the *opposite* direction. The values in the secondary graph are those[5] to which the continuation is applied, and the standard update mechanism carries them back to the invocation site where they flow into the return value of the invocation expression.

5.6.5 Closures

Closures are methods which scope mutable variables. The current implementation does not handle closures directly. The primary reason is that COOP languages cannot allow mutable local variables to be scoped by other methods. This would violate encapsulation of local state and allowing race conditions unconstrained by the concurrency control mechanisms. However,

[5]ICC++ supports multiple return values.

closures are easily modeled as objects. The method contour of the surrounding scope represents the object contour for the scoped variables. Thus, closures could be handled by extending the notion of contours to include a map from the *Label*s of those variables to *Contour*s as in [105] and by using data splitting to extend the precision of captured variables.

5.7 Performance and Results

In this section we present an empirical study on a selection of programs.

5.7.1 Test Suite

The test programs are concurrent object-oriented codes written by a variety of authors of differing levels of experience with object-oriented programming. They range in size from kernels to small applications. They all make use of polymorphism for code reuse and abstraction.

Program	ion	network	circuit	pic	mandel	tsp	richards	mmult	poly	test
User Lines	1934	1799	1247	759	642	500	378	139	49	39
Total Lines	2384	2249	1697	1209	1092	950	828	589	499	489

The first three programs simulate the flow of ions across a biological membrane (**ion**), a queueing network (**network**) and an analog circuit (**circuit**). **pic** performs a particle-in-cell calculation, and **mandel** computes the Mandelbrot set using a dynamic algorithm. The **tsp** program solves the traveling salesman problem. **richards** is an operating system simulator used to benchmark the SELF system [30, 97]. The last three programs are kernels representing uses of polymorphic libraries. **mmult** multiplies integer and floating point matrices, **poly** evaluates integer and floating point polynomials and **test** is a synthetic code which uses multi-level polymorphic data structure. All the programs were compiled with the standard CA prologue of 450 lines of code (Appendix D).

5.7.2 Analysis

We implemented three different analysis algorithms: 0CFA with one flow graph node per program variable, OPS [135] with contours distinguished by their immediate caller (i.e. one level of caller-based splitting for methods and objects), and the adaptive flow analysis describe in this chapter (AFA). We compared these algorithms based on precisions, time complexity and space complexity.

Algorithm	*Progs Typed*	*Progs Failed*	*Type Checks*	*Runtime (secs)*
AFA	9	0	0	199
OPS	3	6	99	150
0CFA	0	9	718	34

5.7.3 Precision

We use two criteria for precision: typing (assignment of types such that run time type checks are not required) and elimination of dynamic dispatch. In this section we cover the former, leaving the latter for Section 6. The table above shows that 0CFA was unable to type even simple programs. OPS fared little better, typing only three of nine programs. However, AFA was able to type all the programs. The times are for our implementation in CMU Common Lisp/PCL on a Sparc10/31.

5.7.4 Time Complexity

Figure 5.16 shows the time taken by the three algorithms which were implemented in the same framework using identical data structures. Note that the speed of AFA compares favorably to that of OPS in two of the three cases where the were both able to type the program. This is because AFA focuses its effort only on areas of the program where it is required. However, when AFA produces better information, it requires more time.

Program	*Lines*	OPS *Typed?*	*Time Sec.*	AFA/ OPS
ion	1934	NO	714	1.2
circuit	1247	NO	290	2.1
pic	759	NO	363	2.5
tsp	500	NO	56	1.4
mmult	139	NO	78	3.5
test	39	NO	15	5.1
network	1799	YES	234	.65
mandel	642	YES	25	.42
poly	41	YES	18	2.2

Figure 5.16: Efficiency of Type Inference Algorithms

5.7.5 Space Complexity

We compare the space complexity by examining the number of contours used per method (the number of nodes used by each algorithm are reported in the Appendix B). Figure 5.17 show the number of contours required AFA and OPS as a multiple of the methods in the program. AFA requires 1.5 and 2.5 per method while OPS requires 2.5 – 4. While additional contours can result in greater precision, AFA's goal directed splitting reduces the number required for a given level of precision.

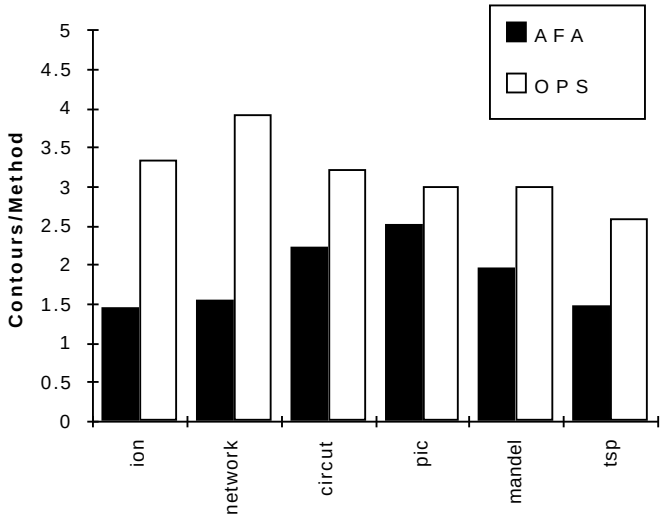


Figure 5.17: Contours per Method

5.8 Related Work

Control and data flow information is vital to optimizing compilers of high level languages. It is useful for constant, copy and lambda propagation [154], static binding, inlining and speculative inlining [30, 97], type recovery [156], safety analysis [137], customization [30], specialization [58] and cloning [83, 142] and other interprocedural optimizations [47].

The use of non-standard abstract semantic interpretation for flow analysis in Scheme by Olin Shivers [156] provides a good basis for this and other work on practical type inference. In particular, the ideas of a call context cache to approximate interprocedural data flow and the reflow semantics to enable incremental improvements in the solution foreshadow this work. Recently, Stefanescu and Zhou [162] as well as Jagannathan and Weeks [105] have provided

simplified frameworks for flow analysis. However, these frameworks are theoretical in nature, with no provisions for managing cost, and not suitable for a practical implementation.

Iterative type analysis and message splitting using run time testing are conceptually similar techniques developed in the SELF compiler [26, 27, 28]. However, iterative type analysis does not type an entire program, only small regions. Essentially, these techniques simply preserve the information obtained by runtime checks inserted into the code. Later work by Hölzle [97] on the SELF-93 compiler uses the results of polymorphic inline caches (i.e. profiling) to determine likely run time types, inserting type tests to ensure that the expected actually occurs.

Type inference in object-oriented languages in particular has been studied for many years [169, 78]. Constraint-based type inference is described by Palsberg, Schwartzbach and Oxhøj in [136, 135]. Their approach was limited to a single level of discrimination and motivated our efforts to develop an extendible approach. Agesen [1, 3] extended the basic one level approach to handle the features of SELF [176]. His technique is limited to eager splitting, and is incapable of handling polymorphic data structures which are destructively updated as a result of his single pass approach.

The soft typing system of Cartwright and Fagan [24] extends a Hindley-Milner style type inference to support union and recursive types as well as insert type checks. To this Aiken, Wimmers, and Lakshman [8] add conditional and intersection types enabling the incorporation of flow sensitive information. However, these systems are for languages which are purely functional where the question of assignment does not arise and extensions to imperative languages are not fully developed. Lastly, our algorithm shares some features of the closure analysis and binding time analysis phases used in self-applicative partial evaluators [148], again for purely functional languages.

5.9 Summary

Flow analysis of object-oriented programs is complicated by the interaction of data values and control flow information through dynamic dispatch and imperative update of instance variables. This chapter presents a flow analysis technique which combines simultaneous data and control flow analysis with iterative adaptation to the structure of the program. Essentially, a simple, less flow and data sensitive analysis is used to determine where more precise analysis is needed.

The *contour representation*, a summarization of stack frames or groups of objects, is extended locally, to provide more precision and the program is reanalyzed. Using a number of COOP programs, this adaptive analysis is shown to be both effective and efficient.

Chapter 6

Cloning

> Property was thus appalled,
> That the self was not the same;
> Single nature's double name
> Neither two nor one was called.
>
> *Shakespeare, The Phoenix and the Turtle, 1601*

Cloning is the process of building specialized versions of classes and methods from generic specifications in order to improve efficiency. These versions are specialized with respect to the manner in which they are used by the programmer and/or implemented (their context). In particular, polymorphic classes and methods are specialized into a set of monomorphic classes and methods, whose storage maps, dispatch tables, and call bindings are optimized for the corresponding classes.

This chapter presents a cloning algorithm which attempts to maximizes optimization opportunities while minimizing code replication. The number of clones is minimized by creating only those dictated by a given set of optimization criteria. Example criteria are minimization of dynamic dispatch and maximization of unboxing within performance critical portions of the code. These clones are shared across the program to limit overall code expansion. The algorithm is both efficient and effective. In our study it produces modest code size increases in the range of -20% to +70% while statically binding approximately 99% of all invocations and, through inlining, eliminating 45% to 99% of invocations overall.

The structure of this chapter is as follows: Section 6.1 introduces a matrix multiply example which will be used in this and later chapters. Section 6.2 describes how the contours produced by context sensitive flow analysis (Chapter 5) are used to guide cloning. Section 6.3 presents

a modified dispatch mechanism required to make it possible for the call graph containing the specialized clones to be realized at run time. Selection of clones is covered in Section 6.4. Optimization of the clones, including data layout and call bindings is discussed in Section 6.5. Finally, a study of performance of the algorithm and its effectiveness for optimization appears in Section 6.6.

6.1 Example: Matrix Multiply

In order to illustrate the cumulative effects of cloning and later OOP and COOP specific optimizations, Figure 6.1 contains an example which is threaded through this thesis. The multiplication of encapsulated two dimensional matrix objects was selected because it is simple and illustrates many sources of inefficiency. While not a conventional object-oriented example, this code illustrates both polymorphism and several levels of encapsulation. Moreover, this program could be written cleanly in FORTRAN (or assembly language for that matter) with good performance. However, our goal is to have both high level abstraction and low level performance. This thesis demonstrates how to build an implementation with the performance and loop structure of the efficient procedural algorithm, while retaining the abstraction expression of object-oriented programming.

The code in Figure 6.1 declares the two-dimensional polymorphic array class `Array2D` and the `innerproduct` and matrix multiply `mm` methods. The `Array2D` class encapsulates a contiguously allocated two dimensional array object. The method `at()` accesses an element of that array by the standard technique of linearizing the indices [66]. These methods are then used to multiply two arrays containing integers `ai` and `bi` into an integer array result `ri`, and two arrays containing floats `af` and `bf` into a float array result `rf`.

6.2 Clones and Contours

The result of flow analysis (Chapter 5) is context sensitive information where a context is given in terms of call paths for methods and creation points for objects. More precisely a context c is a method m, invoked from a statement s in context c' on an object created at statement s' in

```
class Array2D : Array {
  rows;
  cols;
  at(i,j);
  at_put(i,j,value);
  innerproduct(a,b,i,j);
  mm(a,b);
};

Array2D::at(i, j) {
  return self[(cols * i) + j];
}

Array2D::at_put(i, j, value) {
  self[(cols * i) + j] = value;
}
```

```
Array2D::innerproduct(a,b,i,j) {
  let result = a.at(i,0) + b.at(0,j);
  for (let k=1;i<a.cols;k++)
    result += a.at(i,k) + b.at(k,j);
  at_put(i,j,result);
}

Array2D::mm(a,b) {
  for (let i=0;i<b.rows;i++)
    for (let j=0;j<a.cols;j++)
      innerproduct(a,b,i,j)
}

main() {
 Array2D aI,bI;  // L1
 Array2D aF,bF;  // L2
 ri.mm(aI,bI);
 rf.mm(aF,bF);
}
```

Figure 6.1: Matrix Multiplication Example

context c''.[1] For each context the analysis provides the classes which each variable might point to, and likewise for the object creation points, the classes of instance variables of objects created there. Furthermore, the analysis provides an interprocedural call graph over the contexts. The cloning phase uses this information to decide which contexts should be instantiated as unique methods and which sets of objects should be instantiated as unique classes. Essentially, the analysis phase treats the user's methods and classes as a set of uninstantiated templates (see Section 6.7) and determines how they might be instantiated automatically by cloning for both efficient and compact code.

For the matrix multiply example, the analysis determines that the objects `aI,bI` are of class `Array2D` and contain integers (call them `Array2Dint`), and `aF,bF` contain floats (`Array2Dfloat`). Furthermore, it shows that there are two versions of `mm()`, one called on `aI,bI` which operates entirely on `Array2Dint` and another called on `aF,bF` which operates on `Array2Dfloat`. Within these two versions of `mm()` the corresponding versions of `innerproduct()` were called and within

[1]The recursion in the definition is headed off by having `main` invoked from a distinguished context (Section 5.3.1).

them the corresponding versions of `at()` and `at_put`. Thus, the `at()` within `innerproduct()` on `Array2Dint` is known to return an integer.

This information is sufficient to build specialized optimized versions of classes and methods. For example, in Figure 6.2 the generic class `Array2D` is used to hold 100 integers. The method `foreach` is used to invoke the method for a given selector on each element of the array. By cloning a special versions of `Array2D` and `foreach`, the code on the right can be generated. The integer elements of the array are unboxed (the class identification tag removed). The loops have been merged, and the double operation converted to a shift. Instead of requiring hundreds of dynamically dispatched method invocations, multiplications, and indirections, the operation to double every element require only one statically bound function call.

```
foreach(a) {
  for (let i=0;i<a.rows;i++)
    for (let j=0;j<a.cols;j++)
      a.at_put(i,j,f(a.at(i,j)));
}
dbl(i) { i+i }
...
Array2D a(10,10);
foreach(a,dbl);
```

```
foreach_dbl(a) {
  for (let i=0;i<100;i++)
    a[i] = a[i]<<1;
}
...
Array2Dint a;
foreach_dbl(a);
```

Figure 6.2: Cloning Optimization Example

While all the information required to make these transformations is supplied by the analysis, the analysis results cannot be used directly for cloning. First, the natural candidates for replication, contours [156], are too numerous.[2] Second, contours can be distinguished during analysis by elements of the calling context which are not covered by the standard dispatch mechanism (Section 2.1.4). Thus, cloning requires a modification to the dynamic dispatch mechanism, and the power of this mechanism must be balanced with any additional cost. A good cloning algorithm enables efficiency to be balanced with code size.

[2]This is a result both of manner in which the analysis discovers the program structure and of the relative complexity of the data and control flow information required by analysis as compared to that needed for specific optimizations. That is, what the analysis discovers in one part of the program, may not be relevant locally, but enable optimization of another part of the program.

6.2.1 Overview of the Algorithm

The cloning algorithm proceeds by applying optimization criteria (e.g. maximize static binding (Section 7.3.1)) to the contours provided by analysis. These criteria induce partitions over the contours based on the information available for optimization. These partitions are candidate clones. These partitions are then iteratively refined (broken into smaller partitions) subject to the requirement that the call graph of the program represented by the partitions (clones) is *realizable*. That is, the modified dispatch mechanism is capable of selecting the appropriate clone for each invocation site. Section 6.3 presents an efficient modified dispatch mechanism which requires at most one additional instruction and Section 6.4 covers constructing the initial partitions and their refinement. First we introduce the data structures used by the algorithm.

6.2.2 Information from Analysis

The information produced by analysis and described in Figure 5.3 is imported by the cloning algorithm as a set of functions. Figure 6.3 provides a terse description of these functions which are used by the cloning algorithm. Contours are the basic element of context sensitivity. Each of the **method_contours** represent some abstract set of method activations summarized by the analysis. Similarly each element of the set of **class_contours** summarizes some abstract set of objects analyzed as a unit. Each contour is associated with its particular **method** or **class**. Each method contains a set of statements representing invocations (**invsites**) and the creation of new objects (**creation_points**). Finally, each method contains a set of **variables** and each class contains a set of **instance_variables**).

method_contours the set of method contours produced by analysis

class_contours the set of class contours produced by analysis

method(m) the method associated with method contour **m**

class(c) the class associated with class contour **c**

invsites(m) the invocation statements in method **m**

variables(m) the variables in method **m**

creation_points(m) the object creation statements in method **m**

instance_varaibles(c) the instance variables of class **c**

Figure 6.3: Primary Cloning Information

Each contour corresponds to a unique set of context sensitive information. For example, each contour determines the types of variables and binding of invocation sites. Figure 6.4 describes the functions used by the cloning algorithm to access this information. A creation statement and a method contour uniquely determines the class contour of objects created at that statement in that context (**created_contour**). Likewise, an invocation site and a method contour determine the call **binding**, whether or not the site can be statically bound. For instance and local variables, the class or method contour, repectively determine the **boxing**, whether or not the variable can be unboxed (Section 3.2.3). Finally, the interprocedural call graph (**call_graph_edges**) is represented as a set of edges for each invocation statement. Each edge represents a invoke on a method contour, the **callee**. Each edge is further associated with a particlar **selector** (generic function name) and class contour of the target object (**object**),

created_contour(s,m) the class contour of objects created at statement **s** in the context of method contour **m**

binding(s,m) the set of methods which might be invoked at statement **s** in method contour **m**

boxing(v,m) the data layout (e.g. raw integer, raw floating point number, boxed integer or pointer) required for variable **v** in method contour **m**

call_graph_edges(s) the set of call edges for statement **s**

callee(e) the callee method contour for edge **e**

selector(e) the selector (generic function name) whose availability at the call site, in part, induced the edge **e**

object(e) the class contour whose availability at the invocation site, in part, induced the edge **e**

Figure 6.4: Cloning Optimization Information

The information in Figure 6.4 represents only a small subset of that which can be determined by our adaptive context sensitive analysis. In general, the optimization criteria can depend on any analyzed property. Also, analysis may distinguish method contours by arbitrary aspects of the calling environment including: the contours from which they were invoked [135], the types of all the arguments [1] and other criteria [105]. As a result, a call graph on contours cannot, in general, be realized by the standard dispatch mechanism.

6.3 Modified Dynamic Dispatch Mechanism

Cloning modifies the call graph by replicating subgraphs which are then called by only a subset of the previous callers. The information within the cloned subgraphs is then specialized for the subset. If an invocation site within a specialized subgraph can only invoke a single target method, that site can be statically bound, connected directly to the appropriate clone. However, if the invocation site requires a dynamic dispatch, the standard single-dispatch mechanism (i.e. the one used by C++ or Smalltalk) is, in general, insufficient to distinguish the correct callee clone. The problem is that this dispatch mechanism determines the method to be executed based on the selector (generic function name) and runtime class of the target object $< selector, class >$, and these are identical for all clones of a given method. Figure 6.5 illustrates one limitation.

```
class Stream;
class StringStream : Stream;
class Shape;
class Square : Shape;
class Circle : Shape;

Stream::print(Shape * o) { ... }

CLONE Stream:print(Square * o) { ... }
CLONE Stream:print(Circle * o) { ... }
CLONE StringStream:print(Square * o) { ... }
CLONE StringStream:print(Circle * o) { ... }
```

```
main() {
  Object * o = new Circle;
  Stream * s;

  if (...) s = new StringStream;
    else s = new Stream;
  s->print(o);
  o = new Square;
  s->print(o);
  ...
}
```

Figure 6.5: Limitation of Standard Dispatch Mechanism

In Figure 6.5 the `print()` method in the `Stream` class takes a single argument `o` which is either a `Circle` or a `Square`. Since the variable `s` can be either a `Stream` or a `StringStream`, the invocation requires dynamic dispatch. However, the standard dispatch mechanism only dispatches on the selector and the class of the target, and hence cannot select between the versions of `Stream::print()` cloned based on the class of parameter `o` (one for `Square` and one for `Circle`). A more powerful dispatch mechanism is required to handle this case.

To address this problem an invocation site specific dispatch mechanism is used. Each site is given an identifier which is used during dynamic dispatch to distinguish the appropriate callee clone for each selector and target object type pair. In our example, the invocation site information would allow us to select the version of `print` for `Circle` at the first site and that

for `Square` at the second. It should be noted that even multiple-dispatch [32] is not sufficient, since the distinguishing factor could be anything related to the calling environment, including how the return value will be used in the future!

A second problem is that cloning partitions the objects in user defined classes into concrete types for which more specialized information is available. Concrete types are essentially implementation classes describing, for example, a special the memory layout specific to some subset of objects of a particular user class. The new dispatch mechanism may need to distinguish these concrete types. For the example, this case would occur if we had constructed specialized versions of `Circle` such as `BigCircle` and `SmallCircle` with their own memory layouts.

The modified dispatch mechanism uses $< site, selector, concrete\ type >$ to select the method to be executed. Since only a single dimension is added, this mechanism is the smallest extension sufficient to select the correct clone, and, unlike multiple-dispatch, is independent of the number of arguments. This mechanism can be implemented to induce no overhead when the selector is known (i.e. when the selector does not come from a variable), and only one instruction otherwise, since the *site* can be added into the selector to form a single index into a virtual function table.

6.4 Selecting Clones

Clones are selected by partitioning method contours and concrete types by partitioning class contours. The initial set of partitions is determined by optimization criteria such as minimization of dynamic dispatch and maximizing unboxing. These partitions represent potential concrete types and clones (versions of methods) amenable to special optimization which are then iteratively refined until the cloned call graph is realizable by the new dispatch mechanism.

The overall algorithm is presented in Figure 6.6. It is based on two functions, one which determines if two method contours can share a clone (are *equivalent*) and an analogous function for class contours. These functions (shown in Figure 6.7) induce partitions over their respective contours. Then `repartition` computes these partitions by grouping the contours such that all the contours in a partition are equivalent. Initially, this equivalence corresponds to that induced by the optimization criteria. Since finer partition of class contours can induce a finer partition of method contours and vice versa, we repeat the process until a fixed point is reached.

```
clone_selection() {
  // establish initial partitions
  method_partition = new List;
  forall m in method_contours do
    partition(m) = method_partition;
  class_partition = new List;
  forall c in class_contours do
    partition(c) = class_partition;
  // refine for equivalence and realizability
  while (!fixed_point) {
    repartition(method_contours,
                method_contours_equivalent);
    check_class_contours_used_for_dispatch();
    repartition(class_contours,
                class_contours_equivalent);
  }
}
```

```
repartition(part,equivalent){
  result = new List;
  result.add( new List(part.first()));
  forall e in part.rest() do
    forall s in result do
      if (forall r in s do
          equivalent(e,r))
        s.add(e);
      else result.add( new List(e));
}
```

Figure 6.6: Cloning Selection Drivers (pseudocode)

Termination is ensured because the number of contours is finite and the partitioning proceeds monotonically (see Figure 6.7 under the comment `monotonicity`).

The initial partitions are built based on optimization criteria by the contour equivalence functions. To maximize static binding we examine each invocation site in the method for the two contours, and if they would bind to different clones (method contour partitions) or different sets of clones we declare the two contours not equivalent. Similarly for representation optimizations (unboxing), if a variable within two method contours or an instance variable within two class contours has different efficient representations (unboxed or inlined objects) we declare them not equivalent. This is because grouping the contours would prevent optimization. The code to check these optimization criteria appears in Figure 6.7 and is indicated by the `optimization criteria` comment. Standard techniques for profiling or frequency estimation [181] can be applied to maximize the benefits of optimization while limiting code expansion by ignoring optimization of non-performance critical code.

To ensure that the call graph is realizable by the modified dispatch mechanism, further refinement of the partitions may be required, affecting both method and class partitions. This occurs when the dispatch mechanism is not able to resolve a unique method at an invocation site. Figure 6.8 shows graphically for the matrix multiply example, how the decision to specialize `innerproduct()` by partitioning the contours for `Array2Dint` and `Array2Dfloat` induces a repartitioning of contours of (and ultimately the specialization of) `mm()`.

```
boolean method_contours_equivalent(a,b) {
 return
     ((partition(a) == partition(b))                      /* monotonicity */
     && (foreach s in invsites(method(a)) do              /* optimization criteria */
          binding(s,a)==binding(s,b))
     && (foreach v in variables(method(b)) do
          boxing(v,a)==boxing(v,b))
     && (foreach c in creation_points(method(a)) do       /* realizability */
          class_contour(c,a)==class_contour(c,b));
}

boolean class_contours_equivalent(a,b) {
 return
       ((partition(a) == partition(b))                   /* monotonicity */
     && (foreach v in instance_variables(class(b)) do    /* optimization criteria */
          boxing(v,a)==boxing(v,b))
     && (! b in not_equivalent(a)));                     /* realizability */
}

check_class_contours_used_for_dispatch() {
  foreach s in invsites do
    foreach e1,e2 in call_graph_edges(s) do
      if  ((partition(callee(e1)) != (partition(callee(e2))))
        && (selector(e1) == selector(e2))
        && (partition(object(e1)) == (partition(object(e2)))))
        make_not_equivalent(object(e1),object(e2));
}

make_not_equivalent(a,b) {
  not_equivalent(a).add(b);
  not_equivalent(b).add(a);
}
```

Figure 6.7: Contour Equivalence Functions (pseudocode)

Since the dispatch mechanism uses concrete type (class contour partition) to select the target method, if the invocation site and selector are the same, the two class contours must be to be in different partitions in order be able to resolve the appropriate method. Consider the case in Figure 6.9 of optimizing the binding of `print()` in the method `print_contents()` to `Circle::print()` for circle containers and `Square::print()` for square containers at `site 3`. Since the invocation site and selector are identical, the concrete type of `c` must be used to distinguish the correct version. Thus, the method contour partition of `print_contents()` has induced a class contour partition of `Container` to distinguish those instances for which `o` is a

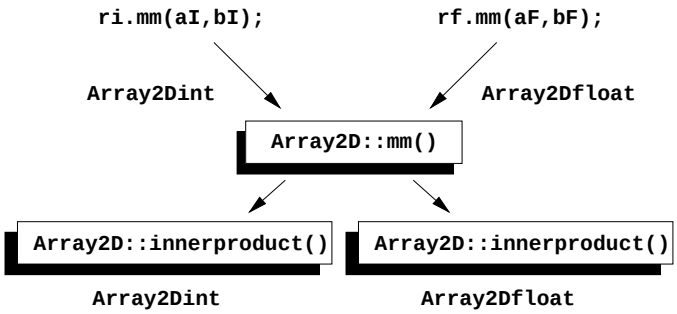


Figure 6.8: Partitioning of `innerproduct()` contours induces repartitioning of `mm()` contours.

`Circle` from those for which `o` is a `Square`. The function which checks this condition and ensures that two class contours will be non-equivalent is `check_class_contours_used_for_dispatch` in Figure 6.7.

```
class Container { Object * o; ... };
void Container::print_contents(){ this->o->print(); }
Container * create() { return new Container; }

main() {
  Container *a = create(); /* site 1 */
  Container *b = create(); /* site 2 */
  a->o = new Circle;
  b->o = new Square;
  Container *c = a;
  if (...) c = b;
  c->print_contents();     /* site 3 */
}
```

Figure 6.9: Example Requiring Repartitioning of Contours

Similarly, class contour partitions can induce method contour partitions. Class contours are defined by their creation point (creating statement and surrounding method contour). Since the partitions of class contours will be the concrete types which are used by the dispatch mechanism, objects must be tagged at their creation points with their concrete type. This means that two method contours cannot be in the same partition if they define different class contour partitions. For example, in Figure 6.9, we have partitioned the class contour for `Container` based on the type of `o` (`Circle` or `Square`). In order to tag `Circle` containers and `Square` containers as different concrete types, enabling the dispatch mechanism to select between them, we must

repartition the method contours for `create()`, separating those called from site **2** from those invoked from site **3**. Thus, the class contour partition of `Container` has induced a method contour partition of `create()`. This is checked by the function `method_contours_equivalent` under the comment `realizability` in Figure 6.7.

6.5 Creating Clones

A method clone is created for each method contour partition and a concrete types for each class contour partition. For each method clone, the code of the original method is duplicated and the data flow information updated to reflect only that for the contours in its partition. The invocation sites and variables have the more precise information dictated by the optimization criteria enabling a wide variety of optimizations (see Chapters 7 and 8 for a detailed discussion and experimental results). In particular, the statically bound invocation sites are connected to the appropriate clone and are amenable to inlining. Methods which contain creation points are modified so that the created objects are tagged with the appropriate concrete type (instead of the original class). Finally, the modified dispatch tables are constructed. Invocations which require dynamic dispatch are assigned identifiers. For each edge in the interprocedural call graph from these sites, an entry is made into the dispatch table mapping the $< site, selector, concrete\ type >$ to the appropriate clone.

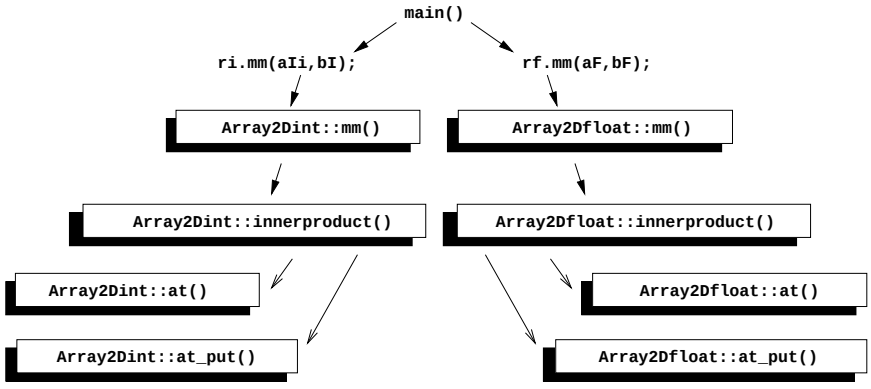


Figure 6.10: Specialization of Matrix Multiply Example

For our example, the specialized classes `Array2Dint` and `Array2Dfloat` are created with the knowledge of the types of `inner`, `outer` and the array elements. The constructors for `aI,bI` and `aF,bF` are specialized to create objects of these new classes. The access methods `inner()`

and `outer()` are specialized to extract unboxed integers. Likewise, the specialized versions of `at()`, `at_put()`, `innerproduct()` and `mm()` are created. Finally, the call graph is updated so that, for instance, `innerproduct` on `Array2Dfloat` invokes `at_put()` on `Array2Dfloat` as in Figure 6.10.

6.6 Performance and Results

In this section we describe the results of applying the cloning algorithm to the test suite of Section 5.7.1. The programs were analyzed with the flow-sensitive interprocedural analysis described in Chapter 5. The number of clones produced and the effects of cloning on dynamic dispatch, procedure calls and code size are reported.

6.6.1 Clone Selection

To evaluate clone selection, initial contour partitions were generated using aggressive optimization criteria. One criteria is to remove as many dynamic dispatches as possible regardless of the number of times the statement is executed. The second criteria was to optimize the representation of as many arrays and local integer and floating point variables by unboxing. We applied these criteria and evaluated the number of concrete types and method clones produced. To demonstrate that clone selection was able to combine contours not required for optimization we also report the number of contours produced by the analysis.

It should be noted that the number of contours produced by an analysis is only superficially related to the quality of information it produces and the difficulty of selecting clones based on that information. In theory, flow analyses produce $O(N)$, $O(N^2)$, $O(N^6)$ or more contours for a program of size N [136, 135, 1, 105] and can require large amounts of space [3]. The adaptive analysis in Chapter 5 creates contours in response to imprecisions discovered in previous iterations. As a result, it is relatively conservative with respect to the number of contours it creates.

6.6.1.1 Selection of Concrete Types (Class Clones)

The number of user classes, analyzed class contours, and the number of concrete types produced by the selection algorithm are reported below:

Program	ion	network	circuit	pic	mandel	tsp	richards	mmult	poly	test
Program Classes	11	30	15	11	11	12	12	7	6	10
Class Contours	64	43	30	27	26	17	27	13	17	18
Concrete Types	11	32	15	11	11	12	13	7	6	10

Figure 6.11: Selection of Concrete Types (Class Clones)

The data in Figure 6.11 show that the number of class contours is much greater than the number of user-defined classes. However, the number of concrete types selected by our cloning algorithm is closer to the number of user classes. This is because not all those distinguished by the analysis are required for optimization. In particular, when all invocations on objects corresponding to some class contour are statically bound, the dispatch mechanism does not need a concrete type for dispatch and no distinct concrete type is created. Methods for such objects are simply specialized for the class contour and statically bound.

6.6.1.2 Selection of Method Clones

The number of reachable user methods used (as opposed to simply defined) in the program, analyzed method contours, clones selected by our algorithm, and the final number of methods after inlining appear in Figure 6.12. The inlining criteria (Section 7.3.3) are based on the size of the source and target methods as well as a static estimation of the invocation frequency. When all invocations on a method are inlined, that method is eliminated from the program.

Program	ion	network	circuit	pic	mandel	tsp	richards	mmult	poly	test
Methods	348	330	143	157	108	103	129	48	42	40
Contours	720	555	511	271	168	153	280	139	189	87
Clones	445	342	173	195	115	108	138	64	54	40
After Inlining	347	181	101	148	63	71	65	42	26	22

Figure 6.12: Selection of Method Clones

Again, the analysis creates many more method contours than user defined methods. However, the selection algorithm chooses only those required for optimization; in most cases ending with only somewhat more than the number of user defined methods. Moreover, since many invocation sites can be statically bound after cloning, many of the smaller methods can be in-

lined at all their callers. Thus, the number of methods which remain after cloning and inlining is actually smaller than the number of methods in the original programs.

6.6.2 Dynamic Dispatch

Dynamic dispatch (virtual function calls) is described in Section 2.1.4. Static binding is the process of transforming dynamic dispatches into regular function calls. Cloning enables static binding by creating versions of code specialized for the classes of data they operate on. We compare three optimizations levels, **unoptimized**, **optimized** and **cloning**. The unoptimized code represents the lower bound on efficiency, indicating the number of methods and messages required by a naive implementation were only accessors (Section 2.3.3) are inlined. The optimized 0CFA version uses customization [26] to create specialized versions of methods for each target object class and statically binds all methods for which there is only one possible target method.

6.6.3 Site Counts

In Figure 6.13 we report the number of dynamic dispatch sites in the final code. Overall, the number of dynamic dispatch sites in the **optimized** codes is almost identical to that in the **unoptimized** code. The differences result from inlining and dead code elimination. On the other hand, very few dynamic dispatch sites remain in the **cloning** codes. As we will see in Chapters 7 and 8, elimination of these sites enables many optimizations.

Without cloning all the programs but two contain a number of dynamic dispatch sites. **mandel** is primarily numerical and does not use polymorphism and in **test** the selectors are unique, enabling invocations to be statically bound even without sophisticated analysis. With cloning, only one program has more than two dynamic dispatch sites. Those dispatches which remain correspond to the true polymorphism in the programs, and cannot be statically bound to single methods. For instance, in **richards** (the OS simulator) the single remaining dispatch is in the task dispatcher, where the simulated tasks are executed. Since the tasks are data dependent, this dynamic dispatch cannot be eliminated.

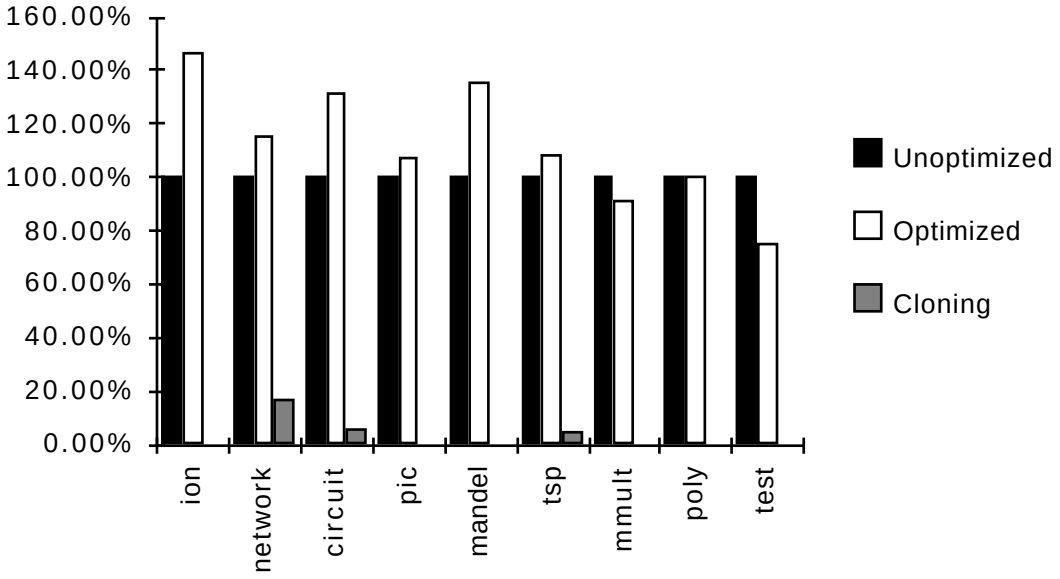


Figure 6.13: Dynamic Dispatch Sites

6.6.3.1 Event Counts

The runtime counts in Figure 6.14 demonstrate the effectiveness of cloning for elimination of dynamic dispatch during program execution. The test suite was optimized and then executed on a sample input and the number of invocations (both dynamically dispatched and statically bound) were collected. The number of dynamic dispatches is reported as a percentage of those occurring in the **unoptimized** code. While global analysis and optimization alone is able to statically bind many invocations, reducing the counts by approximately 8x, cloning is able to statically bind many more. Moreover, once the number of invocations is reduced by inlining, those remaining in the **optimized** case are frequently dynamic dispatches. Figure 6.15 shows the number of dynamic dispatches as a percentage of the remaining invocations. This shows that optimization of the **optimized** code is limited by dynamic dispatches which inhibit inlining. In contrast, cloning keeps dynamic dispatches to a small fraction of the total number of invocations. Note that this graph should not be used to compare the absolute number of dynamic dispatches since the total number of invocations in the cloned version is less than that in the optimized version.

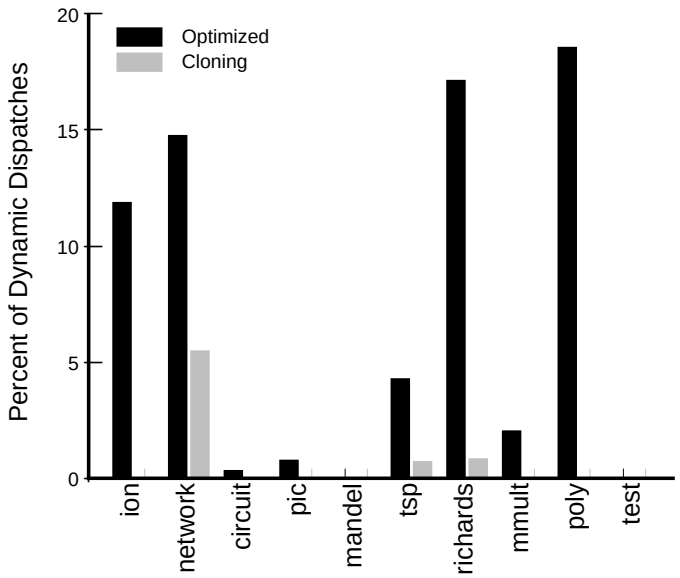


Figure 6.14: Percent of Total Dynamic

6.6.4 Number of Invocations

In Figure 6.16 we report the total number of invocations (static and dynamic) after optimization. For the baseline (100%) we use the number of invocations in the **baseline** version. Global analysis and inlining eliminate between 35% and 99% of the invocations, and in some cases cloning eliminates 20% more. The use of better use of frequency information combined with the greater number of statically bound methods in the **cloning** version might reduce the number of calls even further.

6.6.5 Code Size

One important measure of the effectiveness of clone selection is the final code size. Figure 6.17 compares the resulting code size before and after cloning. The cloned programs usually increase in size by a modest amount, and always by less than 70%. The relatively large increase in **ion** is the result of extensive use of first class selectors (virtual function pointers in C++) for program output. Code size expansion can be reduced by using profiling or frequency estimation to restrict cloning to the parts of the program which execute the most. Since the output phase is only executed once, such restrictions would have helped for **ion**.

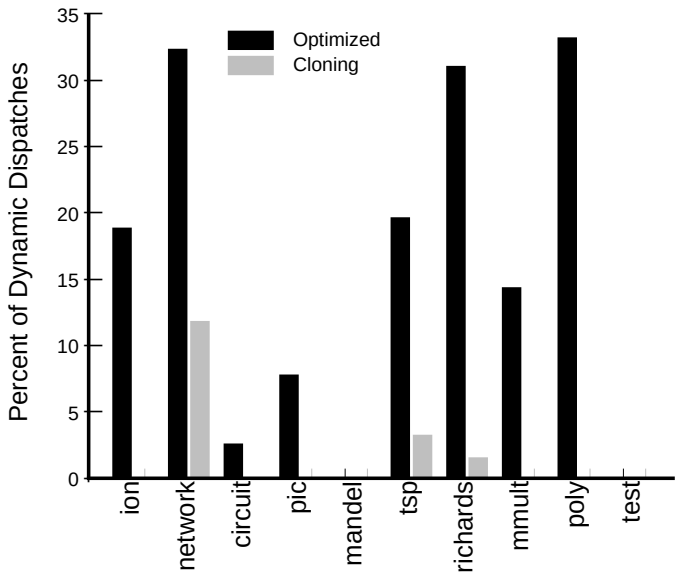


Figure 6.15: Percent of Remaining Dynamic

6.7 Discussion

Pure object-oriented languages rely on polymorphism and dynamic binding to express generic abstractions. In C++, templates [165] give the programmer explicit control over how and when code is replicated and/or shared. In order to avoid code bloat, C++ programmers must use derivation (inheritance) to ensure code sharing among different types; as Bjarne Stroustrup said: "People who do not use a technique like this (in C++ or in other languages with similar facilities for type parameterization) have found that replicated code can cost megabytes of code space even in moderate size programs." [166]. Cloning uses optimization criteria, interprocedural analysis and transformation to automatically generate efficient specialized data structures and code which reflect the actual application structure. However, the generic versions can be used to share code for non-performance critical parts of the application. Moreover, these performance tuning considerations are decoupled from the higher level expression of the program, simplifying coding and increasing the potential for reuse.

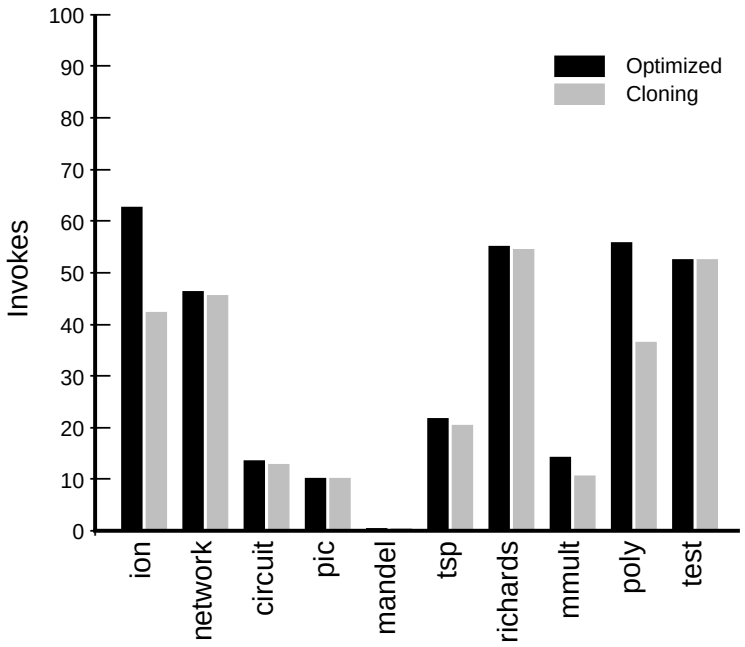


Figure 6.16: Total Number of Invocations

6.8 Related Work

Cooper [47] presents general interprocedural analysis and optimization techniques. Whole program (global) analysis is used to construct the call graph and solve a number of data flow problems. Transformation techniques are described to increase the availability of this information through linkage optimization including cloning. However, this work does not address clone minimization. Cooper and Hall [83, 85, 48, 49, 84, 86] present comprehensive interprocedural compilation techniques and cloning for FORTRAN. This work is general over forward data flow problems, and presents mechanisms for preserving information across clones and minimizing their number. However, concrete types are not a forward data flow problem. Hall determines initial clones by propagation of *clone vectors* containing potentially interesting information which are merged using *state vectors* of important information into the final clones. We handle forward flow problems in a similar manner, but rely on global propagation to determine the final clones for recursive methods.

Several different approaches have been used to reduce the overhead of object-orientation. *Customization* [26] is a simple form of cloning whereby a method is cloned for each subclass

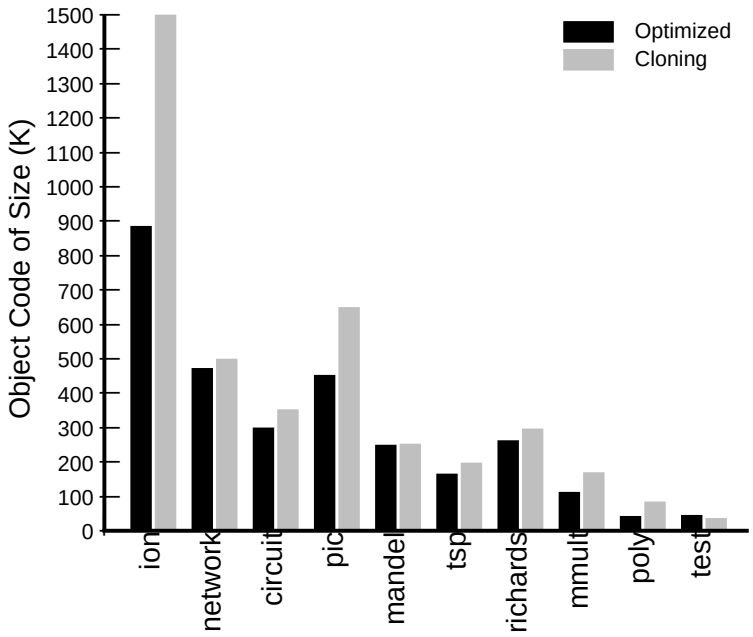


Figure 6.17: Effect of Cloning on Code Size

which inherits it. This enables invocations on `self` (or `this` in C++ terminology) to be statically bound. Another simple approach is to statically bind invocations when there is only one possible method [13]. This idea was extended by Calder and Grunwald [21] through "if conversion," essentially a static version of polymorphic inline caches [96]. This work also shares some similarities with that done for the SELF [176] and Cecil [31] languages. Chambers and Ungar [27], used *splitting*, essentially an intraprocedural cloning of basic blocks, to preserve type information within a function. Early work on Smalltalk used inline caches [61] to exploit type locality. Hölzle and Ungar [97] have shown the information obtained by polymorphic inline caches can be used to speculatively inline methods. While run time tests are still required, various techniques are presented to preserve the resulting type information. None of these approaches uses globally analyzes and transformation to eliminate the run time checks nor to preserve general global data flow information. More recently, Dean, Chambers, and Grove [58] have used information collected at run time to specialize methods with respect to argument types. While this can remove dynamic dispatches across method invocations, it does not handle polymorphic instance variables. Finally, Agesen and Hölzle have recently used the results of

global analysis in the SELF compiler [3]. However, the information for all the contours for each customized method is combined before being used by the optimizer.

The cloning algorithm we have presented is general enough to enable optimization based on any data flow information provided by global flow analysis. All that is required is that the contour equivalence functions be modified to reflect the new optimization criteria. We have used optimization criteria for increasing the availability of interprocedural constants, integrating subobjects and separating algorithm phases successfully with this cloning algorithm. However, efficient cloning for such information requires estimating its potential use for optimization. Interested readers are referred to [83] for a discussion of such issues.

6.9 Summary

Cloning builds specialized versions of classes and methods for optimization purposes. It begins with the results of flow analysis (Chapter 5), the call graph and a set of contours. These contours are partitioned into prototypical clones based on optimization critieria. Object contours are partitioned into concrete types, and method contours are partitioned into method clones. Next, an iterative algorithm is applied which repartitions the contours until the call graph is *realizable*; until the objects can be created of correct concrete types, and the correct clones can be invoked for each invocation site. The standard dynamic dispatch mechanism which selects the desired method based on the selector and class of the target must be modified to be context sensitive. The new dispatch mechanism uses an invocation site identifier during dynamic dispatch. This identifier can typically be folded into the selector. A study of nine object-oriented programs demonstrates that 99% of all invocations can be statically bound to a single method through cloning with modest code size expansion.

Chapter 7

Optimization of Object-Oriented Programs

> One does not know — cannot know — the best that is in one.
>
> *Nietzsche, Beyond Good and Evil*

This chapter describes a range of general and object-orientation specific optimizations and demonstrates, for a set of standard benchmarks, that they are sufficient to enable a pure dynamically-typed object-oriented language to match the performance of C (GCC) and beat that of C++ (G++). The optimizations in this chapter occur after flow analysis (Chapter 5) and cloning (Chapter 6). Section 7.1 maps the potential inefficiencies of the programming and execution models (Section 2.2.4 and Section 3.4) to particular optimization problems. Section 7.2 provides an overview of the solutions to these problems, which are then covered in detail in the remaining sections.

Section 7.3 discusses optimization of invocations; including static binding, if-conversion, inlining and speculative inlining. Section 7.4 is concerned with optimization of low level data access through unboxing, the removal of type tags and the operations which manipulate them. Section 7.5 covers the promotion of instance variables to local variables using interprocedural aliasing information. Finally, in Section 7.7 a suite of general optimizations necessary to extract the final modicum of performance are discussed.

7.1 Efficiency Problems

In Section 2.2.4, we pointed out that abstraction boundaries and polymorphism are potential sources of inefficiency in the programming model. Crossing abstraction boundaries trivially maps to method invocations (virtual function calls as in C++ [166]), which are more expensive than inline operations. Likewise, polymorphic objects must be handled indirectly (i.e. using pointers), increasing potential aliasing. Finally, Section 3.4 points out the impact of control flow ambiguities resulting from dynamic dispatch. Thus, OOP programs have more smaller methods, more data dependent control flow, and more potential aliases than procedural programs [95, 22]. These features decrease performance, and, moreover, their effects compound. For example, the large number of data dependent invocations increases aliasing ambiguity which in turn increases register spill at the large number of invocation sites.

7.1.1 Method Size

Small method size in object-oriented programs is a result of encapsulation and programming by difference which are supported by methods and inheritance respectively. Methods describe the interface to the object, physically embodying the abstraction boundary. Using inheritance, the programmer partitions the program into methods representing a general solution and a set of variation points which are delimited by method boundaries.

The effects of object-orientation on program characteristics have been confirmed empirically by comparision of of C++ and C. Calder et al. [22] found that the instructions to invocation ratio for C++ was less than half that of C. Moreover, the basic block sizes for C++ was slightly smaller than that of C. In addition to the overhead of the method invocation itself, small methods and basic block size make it harder for modern microprocessors to extract the instruction level parallelism they depend on for high performance (see Section 3.1). In CA and other pure object-oriented languages like SELF and Smalltalk the invocation density is even higher [177].

In addition to the direct cost of the method invocations themselves, small method size decreases the effectiveness of register allocation and instruction scheduling, both critical to performance on modern microprocessors (Section 3.1.1). To increase the size of size of methods

the bodies of small method need to be *inlined* (Section 7.3.3), spliced, into their caller where the can be optimized in context.

7.1.2 Data Dependent Control Flow

Since, polymorphic variables may at different times refer to objects several classes, methods invoked on them are dependent on the concrete type of the object. In general, the method executed is determined by a combination of the selector (generic function name) and the actual class of the object, both of which may vary at run time. This data dependence of control flow complicates inlining and increases the cost of method invocations. Much data dependence can be eliminated through a combination of global analysis (Chapter 5) and cloning (Chapter 6). However, transforming the program to take advantage of the additional information, and to eliminate the remaining data dependence requires special optimization. Invocation sites are statically bound (Section 7.3.1) or inlined speculatively (Section 7.3.2), based on the selector and class of the target object.

7.1.3 Aliasing

Since objects are referenced indirectly, they and, consequently, their instance (member) variables, are potentially aliased. As a result, these variables generally cannot be cached in registers across method invocations or assignments through pointers. Since instance variables are implicitly scoped in C++ (they need not be accessed through the `this` pointer), this performance consequence may not be readily apparent to the programmer constructing or using the abstraction. Moreover, the potential alias problem is exacerbated by high method invocation frequency.

```
class Array2D {
  rows;
  cols;
  at(i,j);
}

Array2D::at(i,j) {
  return self[(i * cols) + j];
}
```

```
...
  for (j = 0; j < a.rows ; j++ )
    for (i = 0; i < a.cols ; i++ )
      b.compute( a.at(j,i) );
...
```

Figure 7.1: Aliasing Example

An example of this effect appears in Figure 7.1, which derived from the matrix multiply example of Figure 6.1. In isolation, the method `cols` requires a memory access to retrieve the value of the `cols` instance variable. Inlined into the `for` loop, the load of `cols` cannot be hoisted above the loop as an invariant unless the compiler can prove that `a.cols` cannot be changed by `compute`. The interprocedural call graph which is discussed in Section 7.5 is needed to make this determination. Similarly, approximations of alias information for arrays can also be used for optimization (Section 7.6).

7.2 Optimization Overview

Analysis provides the information, and cloning makes it available so that the compiler can convert method invocations into lower level operations which are amenable to conventional optimizations. The specialized clones and concrete types produced by cloning (Chapter 6) are similar to C++ template instantiations in that the information they contain has been made more precise by code replication. However, high method invocation density and the large number of pointer-based data accesses result in inefficient code.

The problem of invocation density is addressed through a set of invocation optimizations, including static binding, speculation, and inlining. Data access overhead is addressed by unboxing, and the eliminating pointer-based accesses through conversion of instance variables to Static Single Assignment form and array alias analysis. Finally, a suite of standard low level optimizations are performed to eliminate the residue of high level abstractions.

7.2.1 Benchmarks

In order to illustrate and evaluate the optimizations in this chapter, a set of benchmarks are used. The Stanford Integer Benchmarks consist of bubble sort (**bubble**), integer matrix multiply (**intmm**), a permutation generator (**perm**), a 15-puzzle solver (**puzzle**), the N-queens problem (**queens**), the sieve of Erastothenes (**sieve**), the towers of Hanoi (**towers**), and a program to construct a random binary tree (**tree**). The Stanford OOP benchmarks include Richards, an operating system simulator which creates a number of different tasks which are stored in a queue and periodically executed and Delta Blue [151], a constraint solver which builds a network, solves it a number of times and removes the constraints. Two different test cases

are provided for Delta Blue, **Chain** which builds a chain of `Equal` constraints, and **Projection** which builds two sets of variables related by `ScaleOffset` constraints. These benchmarks and the testing methodology are discussed further in Section 7.8.

7.3 Invocation Optimization

Since object-oriented programming produces code with a large number of invocations and uses a relatively expensive calling mechanism, invocation optimization is of particular importance. First, the call mechanism can be optimized by static binding, or the conversion of a dynamic dispatched invocation site to a static call (a normal C style call). This is possible only when it can be proven that only one method may be called from that invocation site. When that is not the case, we can speculate as to which method will be called, insert code to verify the speculation, and use a static call if we are correct. Last, for statically bound sites, the body of the called method can be inserted inline, eliminating the invocation overhead and allowing the code to be specialized for the specific calling context.

7.3.1 Static Binding

General method invocations (dynamically dispatched) can be converted to direct function calls when it can be determined that only one method could possibly be called. In C++ this is trivially the case when a method is not declared `virtual`, is `static` or is never overridden in a subclass. As we will see in Section 7.8, this information in C++ is insufficient, in general, to enable an efficient implementation. In a dynamically typed language (e.g. Smalltalk [76], CA [43], or the language used in this thesis) a dynamic dispatch can only be transformed without analysis when the selector is constant, and there is only one method with that name.

Global analysis and cloning (see Chapters 5 and 6) are capable of resolving many dynamic dispatch sites to a method; the effectiveness of which on a set of general object-oriented programs is discussed in Section 6.6.2. For programs in which all the polymorphism is parametric (determined by static parameterization of classes and methods with respect to their creation or calling environment), these techniques allow static binding of all invocations. Figure 7.2, shows the number of static and dynamic dispatch sites in the Stanford Integer Benchmarks (described

in Section 7.2.1) after analysis and cloning. Since these are procedural codes translated into an object-oriented style, it is not surprising that all invocations can be statically bound.

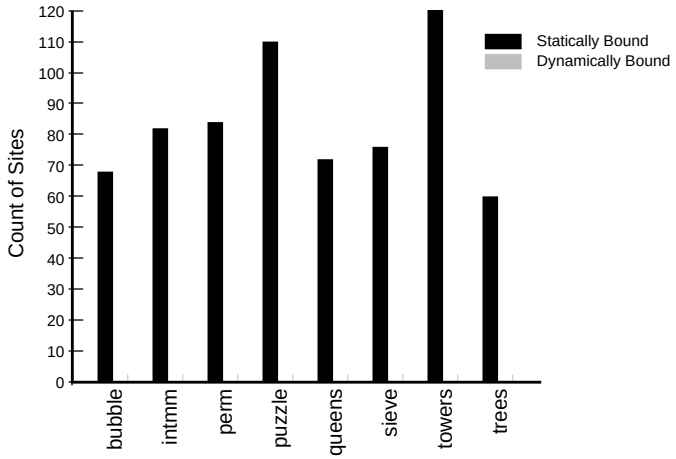


Figure 7.2: Static and Dynamic Invocation Sites in the Optimized Stanford Integer Benchmarks

For the Stanford object-oriented benchmarks, the results show that the programs do require runtime dynamic dispatch. Nevertheless, very few dynamic dispatch sites remain in the generated code. Figure 7.3 shows the number of statically and dynamically bound invocation sites remaining after optimization. In fact, each code only contains a single invocation site which requires dynamic dispatch once cloning has created special versions of classes and methods for each instance of parametric polymorphism. The reason that `proj` and `chain` seem to have a number of dynamic dispatch sites is that the single site is inlined in several places.

Static binding can have a significant impact on performance compared to using a general dispatch mechanism. In Figure 7.4 we compare the performance of the Stanford Benchmarks (Integer and OOP) with and without static binding. In this study, all other optimizations are enabled, in particular inlining is used when possible, however, when a invocation is required, a general dynamic dispatch mechanism is used. In Concert, the general dispatch mechanism requires arguments to be boxed (see Section 7.4 below), and the method lookup uses a bucket hash table. When the correct method is found, a wrapper interfaces the boxed arguments to the unboxed values used in the method.

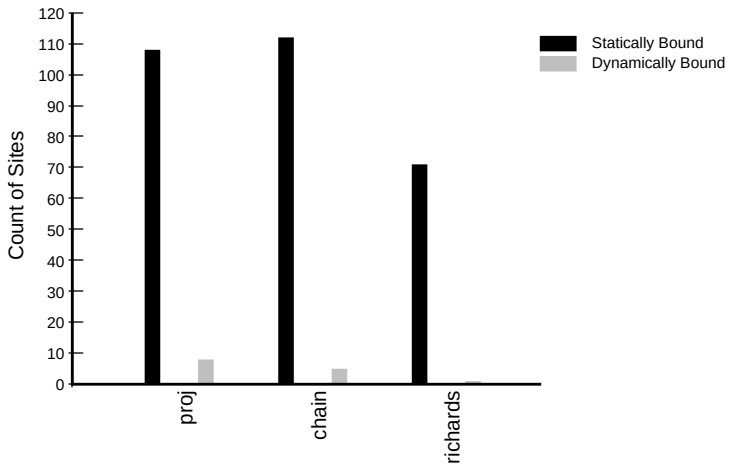


Figure 7.3: Static and Dynamic Invocation Sites in the Optimized Stanford Object-oriented Benchmarks

7.3.2 Speculation

> Even a thought, even a possibility, can shatter us and transform us.
>
> *Nietzsche, Eternal Recurrence*

Speculation is optimization which takes into account probability and relative cost to optimize average case performance. The possibility that an event could occur, for instance a invocation to a particular method at a particular point in the code, is used to transform that code. Instead of using a general calling mechanism (e.g. table driven indirection) to obtain the target method, a conditional is inserted. In the two branches of the conditional, some of the possibilities have been eliminated, refining the information available and (potentially) enabling optimization. This idea will be used extensively in Chapter 8, but first let us examine an example.

In Figure 7.5 on the left the method `foreach` is defined which takes a selector `f`. If it could be determine that the selector argument was `dbl`, all the method invocations could be eliminated, resulting in much more efficient code. While, in general, analysis and cloning cannot always resolve such arguments to a single method, they can reduce also the number of possibilities, both of selectors and target object classes. In such cases, it is often profitable to speculate, for example to insert a conditional to check the value of `f` and if it is `dbl` to execute the optimized code.

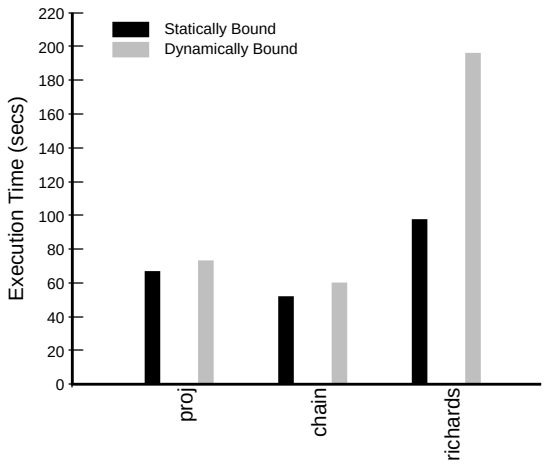


Figure 7.4: Speed Comparison for Static Binding in the Stanford Object-oriented Benchmarks

```
dbl(i) { i + i }

foreach (a,f) {
  for (let i = 0;i<a.size;i++ )
    a[i] = f(a[i]);
}
```

```
foreach (a,f) {
  if (f == dbl)
    for (let i = 0;i<a.size;i++ )
      a[i] += a[i];
  else
    for (let i = 0;i<a.size;i++ )
      a[i] = f(a[i]);
}
```

Figure 7.5: Speculation Example

In Figure 7.6 the average number of target methods which might possibly be invoked at a dispatch site (the static arity) is reported for the Stanford object-oriented benchmarks. The graph on the left presents the counts of invocation sites and possible targets, while the graph on the right presents the ratio of targets to invocations. Inlining (Section 7.3.3) has eliminated many of the statically bound invocation sites which would have an arity of one. Nevertheless, the ratio of approximately 1.25 indicates that the majority of remaining sites are still statically bound.

Figure 7.7 reports the average arity of invocations during execution of the Stanford object-oriented benchmarks (dynamic arity). Since the programs require a dynamic dispatch in the inner loop (on the evaluation selector of the constraint network node in the case of Delta Blue, and on the class of the simulated task in Richards), the dynamic arity is higher than the static arity. In particular, through inlining, the inner loop of Richards has been reduced to a single

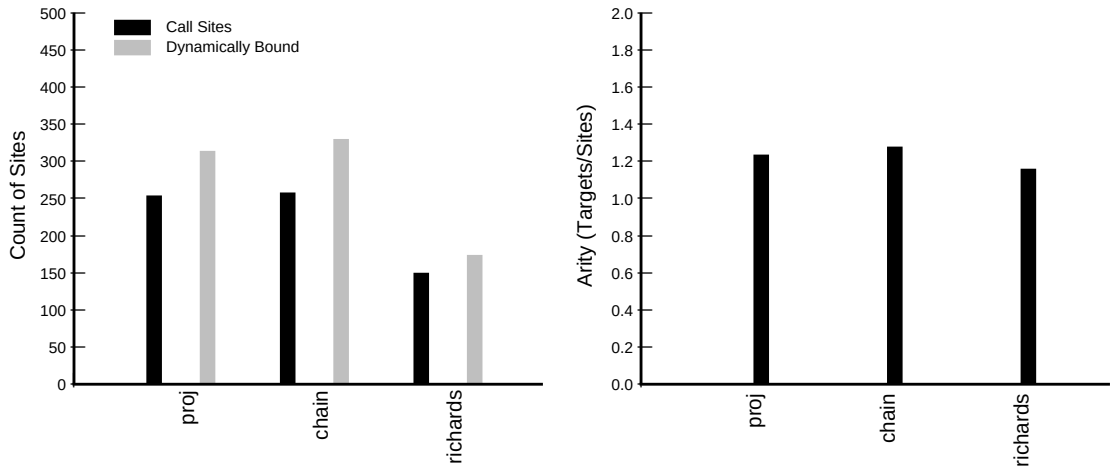


Figure 7.6: Static Arity of Dispatch Sites in the Stanford Object-oriented Benchmarks

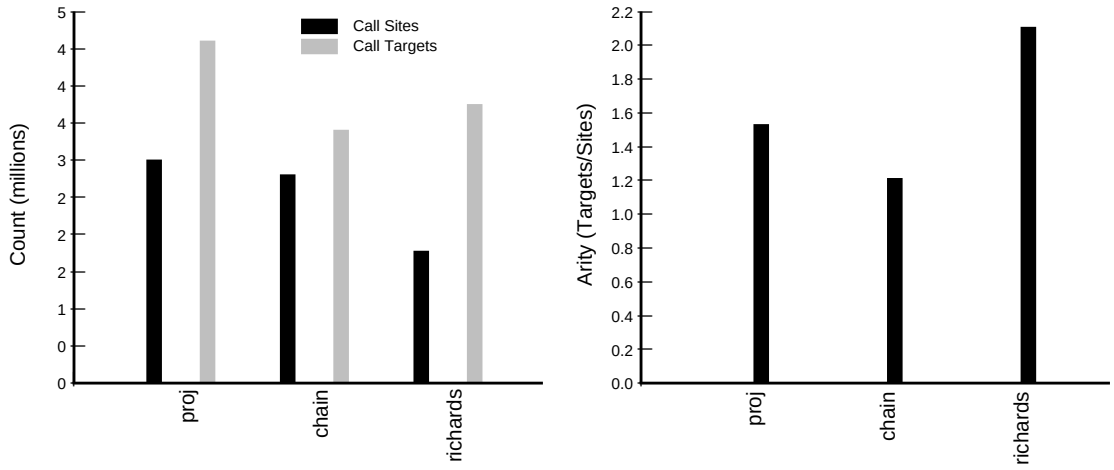


Figure 7.7: Dynamic Arity of Dispatch Sites in the Stanford Object-oriented Benchmarks

method containing a dynamic dispatch at the core. These dynamic dispatches are necessary as they result from non-parametric polymorphism in the data structures.

Through speculation, inserting conditionals in the place of dynamic dispatch, the overhead of the general invocation mechanism can be avoided. As we have demonstrated, the number of such invocations sites tends to be small, as does their arity. Thus, the effect on the overall code size should be small. The main advantage of speculation comes from inlining invocations under the conditionals.

7.3.3 Inlining

Inlining is the process of replacing a statically method invocation with the body of the called method. This can improve the performance of code both by removing the method invocation overhead and by enabling the body of the method to be optimized in context. In order to prevent excessive code expansion, inlining is performed based on heuristics which attempt to balance performance and code size. The Concert compiler uses a combination of static estimation [181] and size constraints to decide when to inline, eliminating the cost of crossing a procedure boundary.

Since many small method bodies contain only a few instructions, inlining is of particular importance for object-oriented programs. Figure 7.8, compares the speed of the fully optimized programs to those for which only accessors (methods which access instance variables) and primitive operations (e.g. integer add) have been inlined. This latter case corresponds roughly to the level of optimization available from simple C++ implementations.

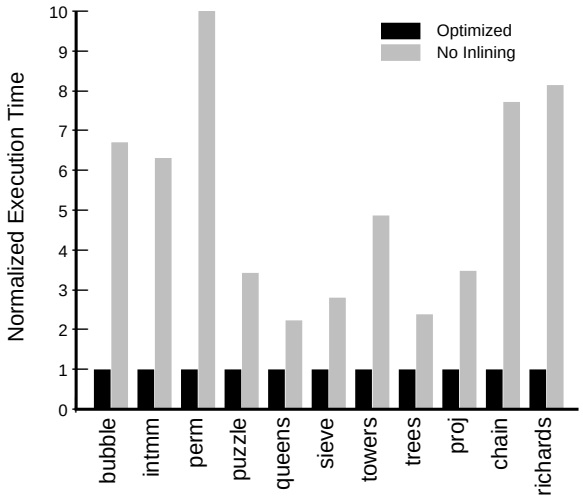


Figure 7.8: Effects of Inlining on Execution Time

The interprocedural call graph can be used to determine when a method has been inlined at all invocation sites and eliminate the method. As we saw in Section 6.6.5, inlining need not result in a large code size increase. For example, both versions of the `innerproduct()` method will be inlined into their corresponding `mm()` invocation sites and the methods will be eliminated since they have no other callers.

When speculation is combined with inlining, the inlined code is placed under a conditional which can prevent some optimizations. Chapter 8 describes optimizations which expand these conditionals to encompass larger portions of code, removing overhead and enabling additional optimization. These transformations, along with dead code elimination (Section 7.7), perform the same function as *splitting* [27] by preserving information obtained by runtime type or selector checks within methods.

7.4 Unboxing

Unboxing converts a tagged slot (Section 3.2.3) into an untagged data location. This decreases memory requirements and eliminates the overhead of tag manipulations. To unbox a piece of data, it is not necessary to know the exact type (e.g. pointer to a `Point` object or pointer to a `Circle` object), only primitive type (e.g. integer, pointer, floating point number). This information is provided by analysis and cloning. There are four types of variables which can be unboxed: local variables, instance variables, array elements and arguments.

Local Variables

Unboxing local variables requires building a new memory map for the context containing the local state (see Section 3.2.1.1). This memory map describes which locations the runtime and garbage collection system can expect will contain pointers, and which will be tagged. Unboxed local variables can be allocated to registers. Chapter 9, considers the trade-off between increasing the active state and decreasing context switch time in a concurrent object-oriented system.

Instance Variables and Arrays

Likewise, unboxing of instance variables requires building a new memory memory map. Cloning can produce several concrete types corresponding to a single class. However, specializing the memory maps, or failing to update tag fields appropriately can result in the inability to share a single version of method code across these concrete types. The same is true for superclass methods, which objects of a subclass may not be able to share if they modify their memory map in an unconformant way (Section 3.4).

Arrays can be unboxed just as instance variables, and entail the same conformance problems. Both CA and ICC++ provide arrays as objects with the array part stored after the instance variables. The array part can then be considered as a final instance variable. Conformance is based on the start location of the array part and whether or not the elements are tagged or packed.

Arguments

Unboxing of arguments requires changing the calling convention. In the presence of dynamic dispatch, it is possible for the caller to not have the precise type information which is implied by the dispatch criteria. For example, in Figure 7.9, the increment `inc` methods can take unboxed arguments, but at the invoking site, the variable x is polymorphic. If the compiler decides to unbox the `self` argument for `inc`, either speculation or a trampoline must be used to map the boxed caller. Calling convention conversion are discussed in Chapter 9.

```
int::inc() { 1 + self }
float::inc() { 1.0 + self }

let x = ... ? 2 : 2.0,
    y = x.inc;
```

Figure 7.9: Calling Convention Conversion

7.5 Instance Variable to Static Single Assignment Conversion

Since instance variables are ubiquitous, accessed by reference and potentially aliased (see Section 7.1.3) in object-oriented programs they must be loaded from and stored into memory frequently, increasing memory hierarchy traffic and representing a large potential overhead. Using the interprocedural call graph and the object creation context information provided by the analysis, we estimate whether a method invocation or instance variable access might alias a given instance variable. Since good encapsulation disallows pointers into objects, only other accesses to the same instance variable of objects created at the same point as the instance variable in question can alias it. The interprocedural call graph enables us to approximate

the statements reached by a method invocation. Instance variables unaliased over a range of statements are transformed into locals and can be allocated to registers.

Likewise, global variables can be transformed into local variables. Since global variables are uniquely named and cannot be aliased, their sharing patterns can be easily determined from the interprocedural call graph. Section 8.4.3 describes related optimizations of global variables for concurrent object-oriented languages.

```
Array2D::mm(a,b) {
  for (i=0;i<b.rows;i++)
    for (j=0;j<a.cols;j++)
      innerproduct(a,b,i,j)
}
```

```
Array2D::mm(a,b) {
  tmp_rows = b.rows;
  tmp_cols = a.cols;
  for (i=0;i<tmp_rows;i++)
    for (j=0;j<tmp_cols;j++)
      innerproduct(a,b,i,j)
}
```

Figure 7.10: Instance Variable Transformation Example

In our example, the `cols` and `rows` instance variables are part of the `Array2D` object which is potentially aliased. However, using the call graph we can determine that for the objects created at `L1` and `L2` (Figure 6.1) these instance variables are not changed within any method invoked from `mm()`. Thus, as in Figure 7.10 we can transform the instance variables to local temporaries `tmp_cols` and `tmp_rows` and hoist them out of the loop.

7.6 Array Aliasing

In the same way that alias information enables transformation of instance variables into locals, it enables optimization of arrays accesses. Since good encapsulation prevents pointers into the middle of arrays, absolute and/or symbolic analysis can be used to determine that array accesses do not conflict. We use a simple creation point test to estimate interprocedural array aliasing and combine it with simple symbolic analysis to enable array references to be lifted and common subexpression eliminated.

For example, in Figure 7.11 from the bubble sort benchmark (see Section 7.8), the inner loop contains two array reads in the conditional and two in the body (left). We can determine by simple analysis over the call tree that the array could not be written between the first invocation to `a.at(i)` and the second. Thus, we can lift and common subexpression eliminate the `a.at(i)` in the loop (right).

```
if (a.at(i) > a.at(i+1)) {
  tmp = a.at(i);
  a.at_put(i,a.at(i+1));
  a.at_put(i+1,tmp);
}
```

```
tmp_i = a.at(i);
tmp_i1 = a.at(i+1);
if (tmp_i > tmp_i1) {
  a.at_put(i,tmp_i1);
  a.at_put(i+1,tmp_i);
}
```

Figure 7.11: Example of Common Subexpression Elimination of Array Operations

7.7 General Optimizations

Since the abstractions of object-oriented programming are often used to hide the representation of data, ostensibly simple operations may require many instructions. For example, the CA language does not support native multi-dimensional arrays. These are constructed out of single dimension arrays with instance variables containing the dimension sizes and linearization methods (as in Figure 6.1). When multi-dimension arrays are used in loops, the instance variables can be transformed to local variables and the linearization operations strength reduced and moved outside the loop. With these optimizations, matrix multiply of multi-dimensional arrays in CA is as fast as C (see Section 7.8), even though the array operations are abstracted and ostensibly require much more work.

Since these general optimizations are a standard part of most optimizing compilers, they are only summarized in Table 7.1. More information can be found in the large body of compiler design literature (i.e. [7]).

7.8 Overall Performance and Results

We use a standard benchmark suite (Section 7.2.1) to evaluate and compare the performance of CA, C and C++. The Stanford Integer Benchmarks, Richards and Delta Blue were used to evaluate the SELF language by Chambers [30] and later by Hölzle [95]. The Stanford Integer Benchmarks are small procedural codes. The CA versions use encapsulated objects for the primary data structures, but otherwise follow the C code structure. Richards and Delta Blue both use polymorphism, some of which can be removed at compile time by templates or cloning (i.e. parametric polymorphism) and some which cannot (the task queue and constraint network respectively). The C++ codes are annotated by declaring functions `virtual` only when nec-

Constant Propagation	The realization that when a variable is assigned only a single value, it must be that value; applied transitively.
Constant Folding	Executing an operation whose inputs are compile time constants at compile time to produce a new constant. Algebraic properties are normally used to enable more constant folding in languages (like CA and ICC++) which allow it.
Dead Code Elimination	Removal of code which cannot be executed, normally because the conditions predicating its execution have been determined not to hold. This is often the result of inlining and specialization.
Calling Convention Optimization	Unboxing of arguments and removal of unused arguments from the method interface.
Common Subexpression Elimination	The recognition that two computations compute the same value, and substitution of the result of one for that of the other, enabling dead code elimination of the other.
Global Value Numbering	The extension of common subexpression elimination across basic blocks.
Invariant Lifting	The removal of a computation from a loop which does not depend on any values computed in the loop, enabling the value to be reused for each iteration.
Strength Reduction	The transformation of a scaling operation in a loop into a set of successive additions.

Table 7.1: Standard Optimizations

essary [95], including inline accessors, and, in Delta Blue, by the use of a `List` template. The CA versions of these codes follow the C++ encapsulation and code structures.

7.8.1 Methodology

We compare the performance of CA codes translated from the Stanford sources.[1] Our compiler uses the GNU compiler [161] as a back end, enabling us to control for instruction selection and scheduling differences by using the same version (2.7.1) for both the back end of the Concert compiler and the C and C++ benchmarks. All tests were conducted on an unloaded 75Mhz SPARCStation-20 with SuperCache running Solaris 2.4. We present both individual results

[1]Our thanks to Craig Chambers and Urs Hölzle for making the codes available.

and summarized results for the procedural and object-oriented benchmarks. The individual results are the average execution time of 10 repetitions of each benchmark at each optimization setting normalized to the execution time of C/C++ at -O2. The summarized results are the geometric means of the normalized times over all the benchmarks at each optimization setting. This effectively constructs a synthetic workload in which each benchmark runs for the same amount of time for C/C++ at -O2.

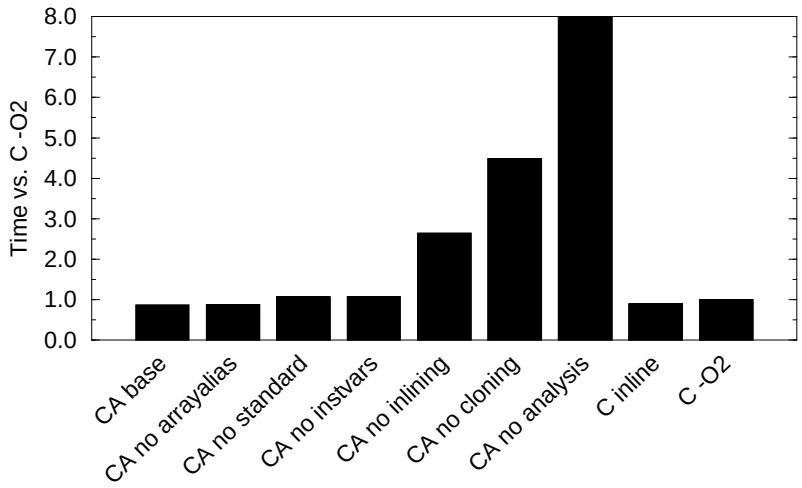


Figure 7.12: Geometric Mean of Execution Times Relative to **C -O2** for the Stanford Integer Benchmarks for a Range of Optimization Settings

7.8.2 Procedural Codes: Overall Results

Figure 7.12 graphically summarizes the execution time of the Stanford Integer Benchmarks under various optimization settings relative to the C optimized at -O2. Overall, the results show that at full optimization the performance of the dynamically-type pure object-oriented codes matches that of C. The different bars report the cumulative effect of disabling optimizations. In order to make a comparison with C++ easier, the **no inlining** bar does not prevent inlining of accessors or operations on primitive data types (e.g. integer add) which would automatically be inlined in C++. Also, the **analysis** bar only disables flow sensitivity. Flow insensitive analysis provides information roughly comparable to type declarations, particularly with regard to C++ primitive data types.

Each optimization contributes to the overall performance which would otherwise be an order of magnitude (9.2 times) less than C. Context sensitive flow analysis alone provides a factor of two by allowing more primitive operations to be inlined, and method invocations to be statically bound. Cloning contributes another factor of two for essentially the same reasons, by making monomorphic versions of polymorphic code. Inlining operates on statically bound invocations, more than doubling performance by eliminating invocation overhead. Transforming instance variables to locals enables many of the **standard** optimizations which, together with array alias analysis provide the last twenty percent of overall performance.

These results demonstrate that for such procedural kernels, the cost of the unused flexibility of object-oriented features can be eliminated. The remaining differences in performance reflect the low level optimizations favored by the GCC compiler and the code structure more than any inherent language advantage. For example, GCC strength reduces array accesses based on `sizeof(int)` using a simple heuristic which is easily confused by the RTL-like output of the Concert compiler. Likewise, computing booleans into intermediates can inhibit direct use of condition codes. On the other hand, GCC only inlines functions which appear previously in the same file, and the standard `malloc` routine is relatively inefficient.

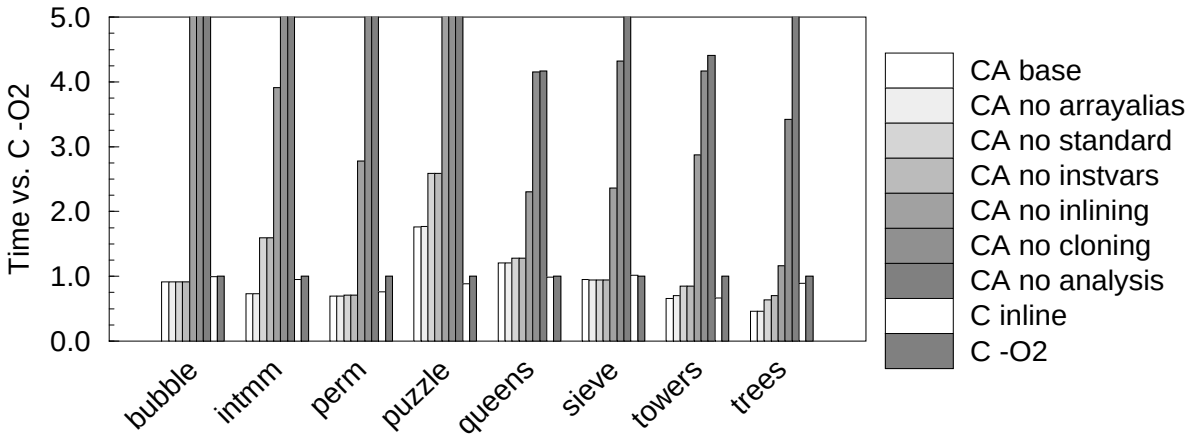


Figure 7.13: Performance relative to **C -O2** on the Stanford Integer Benchmarks

7.8.3 Procedural Codes: Individual Results

Figure 7.13 reports the individual performance of the benchmarks. Overall, the results are split, with **CA base** outperforming **C -O2** in six cases and **C -O2** outperforming **CA base**

in two. **C inline** increases that number by one, adding **towers** (a highly recursive code) to the list for which C is faster. Since these codes are largely monomorphic they can be analyzed easily, and they differ only in their control structures and method boundaries. The CA codes are faster because of relatively more aggressive inlining except **trees** which allocates many objects. Even though the CA code garbage collects eleven times during each run, it is still more efficient than `malloc()`. On the other hand, CA does not support `break`, and the resulting additional condition in `while` loops drops CA performance on **puzzle** by almost a factor of two. Discounting these special cases, individual benchmark performance is nearly identical.

Different optimizations had larger effect on different benchmarks, indicating their individual importance. The **bubble** sort and permutation (**perm**) programs are heavily dependent on inlining (for the swap) which provides most of the performance. The **trees** and **puzzle** programs benefit directly from instance variable promotion, in **trees** case because of the heavy use of the `left` and `right` child instance variables. Matrix multiply (**intmm**) and **puzzle** are loop based, and depend on the standard optimizations, and in particular strength reduction which is enabled by instance variable promotion (for the inner loop dimension). Finally, **towers** and **puzzle** benefit from array alias analysis because the code repeatedly accesses the elements at the same array offsets.

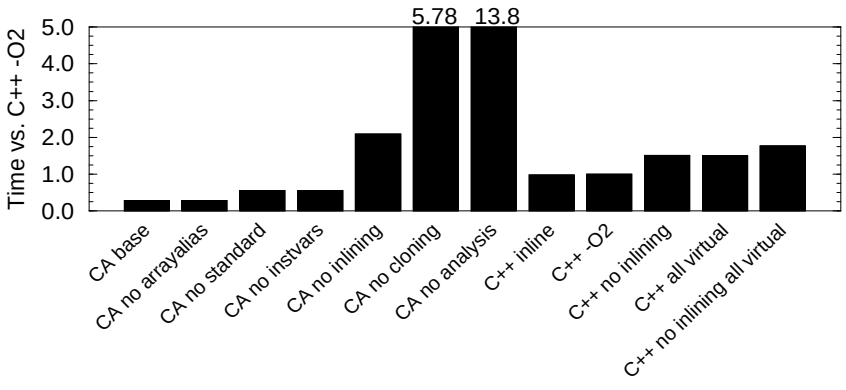


Figure 7.14: Geometric Mean of Execution Times Relative to **C++ -O2** for the Object-oriented Benchmarks for a Range of Optimization Settings

7.8.4 Object-Oriented Codes: Overall Results

Figure 7.14 graphically summarizes the execution time for the object-oriented benchmarks for CA at seven optimization settings and for C++ versions at five. Overall, the Concert

compiler produces much better performance, a factor of four improvement, than the C++ compiler even with user provided annotations and automatic inlining enabled (-O3). Again, the individual optimizations made their contributions starting from a initial performance point an approximately order of magnitude (13.8 times) worse than C++. Context sensitive flow analysis provided a factor of 2.4. This is more than the factor of two for the procedural codes, indicating its relative importance for object-oriented codes. Likewise, cloning was responsible for approximately a factor of 2.5. Inlining contributed a factor of four, again showing its relative importance for OO codes. Finally, standard low level optimizations enabled by instance variable promotion contributed a factor of two.

We evaluated the performance of the benchmarks with the C++ compiler at five optimization settings, including the base (-O2), automatic inlining (-O3), without any inlining, with all virtual functions, and without any inlining and all virtual functions. Automatic inlining improved performance by approximately two percent, indicating that most of the automatically inlinable functions had been annotated. Disabling either inlining or static binding annotations reduced performance by 50 percent, and with both were disabled, performance decreased by 70 percent. Performance of the CA code without inlining was comparable to that of the C++ compiler without inlining. However, as we will see in the next section, the individual results vary, indicating this is probably just coincidental.

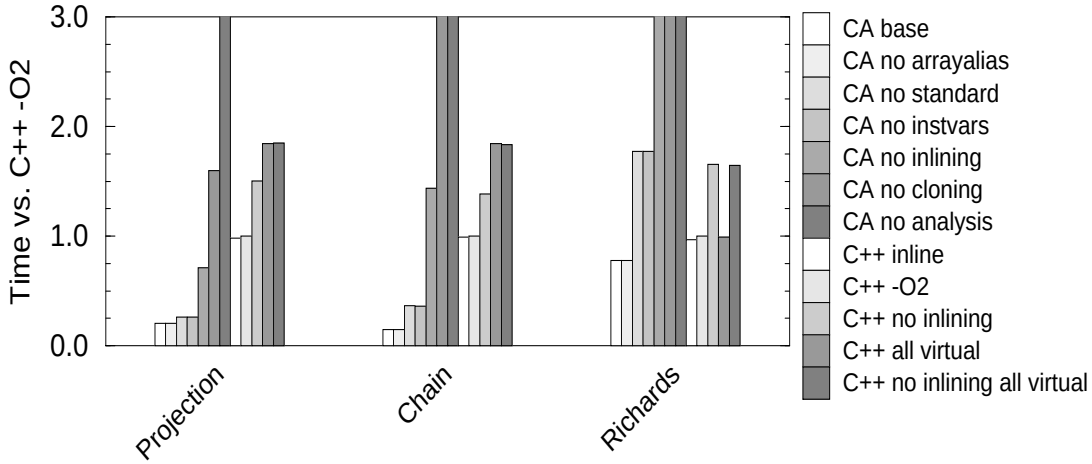


Figure 7.15: Performance of CA and C++ relative to **C++ -O2** on OOP Benchmarks

7.8.5 Object-Oriented Codes: Individual Results

Figure 7.15 reports the results for the **Richards**, Delta Blue **Chain** and **Projection** individual benchmarks under various optimization settings of both the Concert compiler and the C++ compiler. The CA versions vary from almost six times faster (**Chain**) to twenty-five percent faster (**Richards**) than **C++ -O2**. In the case of Delta Blue, this difference is attributable to many invocations to small methods. For example, in object-oriented fashion, Delta Blue uses a general `List` container object with a method which applies a selector (function pointer) across its elements (e.g. `do:` in Smalltalk or `map` in Scheme). The Concert compiler clones and inlines both invocation sites, turning this into a simple C style loop containing operations directly on the elements, while the C++ compiler does not. For **Richards**, the performance difference is primarily a result of optimizations enabled by instance variable promotion. Richards uses an object to encapsulate its current state including the head of the task queue, and manipulation of this state makes up the largest part of the execution time.

The different C++ optimization settings produced different results for the different benchmarks, much as they did for CA. For **Richards**, disabling inlining decreased performance by 65 percent. However, making all methods `virtual` has no effect on performance at all. This is because **Richards** is largely a procedural code where the central switch statement has been replaced with a dyanamic dispatch (the `run` method on `Task` objects). On the other hand, Delta Blue uses accessor methods and other small methods for encapsulation of object state. For **Chain**, disabling inlining decreases performance by 40 percent and for **Projection** the impact is a 50 percent decrease. Furthermore, since Delta Blue does most computation through methods, making all methods `virtual` (which effectively prevents inlining of methods as well) reduces performance by 85 percent, and the performance is unchanged when inlining is disabled as well.

These results show that for these benchmarks, the overhead from object-orientation can be removed automatically. Furthermore, it they show that the annotations provided by the C++ programmer are not sufficient to optimize the programs.

7.9 Related Work

Speculative inlining, and specialization with respect to runtime checks is discussed in detail by Craig Chambers [30] and Urs Hoölzle [95] in their respective theses on the SELF system. This work was based either on guesses as to the type of particular object or on runtime feedback (profiling) instead of interprocedural analysis. The focus of Chambers' work was preserving the information acquired through runtime checks over larger bodies of code, similar in purpose to the access region manipulations described in Chapter 8. Earlier information on these optimization in the SELF system include [27, 28]. Simple speculative optimization by "if-conversion" was discussed by Brad Calder and Dirk Grunwald in [21] as a replacement for table indirected dynamic dispatch in cases only a small set of methods of the same name existed based on examining the class hierarchy. Recently, interest has turned to dynamic compilation using compiler supplied templates [123, 14]. This work uses a combination of static, dynamic information and runtime checks to select optimized versions of code. Again, the empasis is on simple analysis combined with profiling information. Optimization of object-oriented calling mechanisms has been discussed extensively. Two relevant sources include: for the pure object-oriented language SELF, Hölzle [96] and for C++, Stroustrup [166]. The advantage of the system described in this chapter is that flow analysis and cloning can often determine arity of a method invocation site to be a small number (often one). This results in a shift in the focus of optimization toward static binding and inlining, since the general dispatch mechanism is rarely used. Unboxing has also been a popular for many years in the Lisp community. Most recently, researchers in ML have discovered it [153, 90, 125] in the context of polymorphic types. Because these ML systems are type-based and support separate compilation, they have neither complete knowledge nor access to the whole program, preventing them from the sort of global transformations described in this chapter. Finally, the Standard Template Libraray [163] has been proposed as a solution to the problem of optimization of polymorphic code in C++. In this system, the notions of type parameterization and code replication are combined. The result is a system which provides specialized versions of code by massive code replication, largely out the hands of the compiler. Given that modern computers are memory bandwidth limited, this approach is of dubious value.

7.10 Summary

Abstraction boundaries and polymorphism in object-oriented programs are potential sources of inefficiency. Moreover, small method size, high invocation density, data dependent invocations, data access overhead and aliasing problems compound these inefficiencies. Invocation density is addressed through invocation optimizations: static binding, speculation and inlining. Data access overhead is addressed by unboxing and the elimination of pointer-based accesses through conversion of instance variables to Static Single Assignment (SSA) form. Array alias analysis and a suite of standard low level optimizations are also performed. Results for the Stanford Integer Benchmarks demonstrate that a pure dynamically-typed object-oriented language implemented by Concert can be as efficient as C. Moreover, the Concert system implementations of the Stanford object-oriented benchmarks were much more efficient than the C++ implementation G++ at the highest optimization level.

Chapter 8

Optimization of Concurrent Object-Oriented Programs

This chapter describes optimizations which are specific to distributed and concurrent object-oriented programs. Many of these are targeted to aspects of the execution model presented in Chapter 3, and include object access control operations (Section 8.1). Section 8.2 revisits the topic of speculative inlining in the context of locality and lock optimizations. Sections 8.3 and 8.4 are concerned with extending the dynamic range of speculation, and with optimization of the use of the memory hierarchy respectively. Sections 8.5 discusses touch placement. Finally, Section 8.6 demonstrates the effectiveness of many of these transformations on the Livermore Loops.

8.1 Simple Lock Optimization

Our execution model requires locks for every method which accesses instance variables of the target object (Section 3.2.5). While these semantics support data abstractions in a concurrent environment, given the frequency of method invocations in object-oriented programs (Section 7.1.1), a direct implementation of this model implies a frequency of locking operations which is a serious source of inefficiency. Two optimizations which can eliminate unnecessary lock operations in many cases are *access subsumption* and exploitation of *stateless* methods. The latter are also useful since no object is not actually required for the method to execute, allowing such methods to executed anywhere on the target machine.

8.1.1 Access Subsumption

Access subsumption avoids redundant acquisition of locks that must have been previously acquired above in the call graph. So long as the calling method does not complete before the callee (i.e. tree-structured concurrency, Section 2.2), the calling method's access rights will subsume the callee's, making it unnecessary for the callee to acquire locks. Access subsumption optimization uses the call graph provided by analysis and cloning to recognize cases where a method is called from another method which has already acquired the locks required by the first. Alternatively, if not all callers acquire the needed locks, two versions of the method, a flag or alternate entry points can be used to separate the cases or pass along contextual information.

8.1.2 Stateless Methods

Stateless methods do not (on their own) access any state of the target object and therefore need neither execute local to the object nor acquire any locks to ensure correct semantics. In many cases, stateless methods merely validate arguments or add default parameters. Often they were not stateless initially, but after cloning and interprocedural constant propagation they become stateless. For example, consider the `at()` method in Figure 6.1. If this method is cloned for arrays of a particular dimensionality, the `cols` instance variable becomes constant, and the two dimensional `at()` becomes stateless, simply passing a computed value to its inherited `at()` method. Likewise, *aggregates* within CA [43] and *collections* within ICC++ [81] are objects which provide message vectoring and indexing capabilities based on invariant distributed information. While not technically stateless, these methods can be executed anywhere, and do not require locks. Similarly, methods which only read temporally constant globals are "stateless." Stateless methods do not require locks and can be inlined without concern for locality.

8.2 Speculative Inlining Revisited

Speculative inlining uses runtime checks to condition the execution of inlined code. In Section 7.3.2, speculation was used to inline code when the selector or concrete type of the target object was not known at compile time. In this chapter we are concerned with run time properties resulting from distributed and concurrent execution, locality and locking. A method may

only be inlined if any required data is available (the target object is local and any required lock is available).

```
if (selector == SELECTOR_AT)            | runtime checks
  && LOCAL_POINTER(X)                   |
  && CONCRETE_TYPE(X)==CLASS_ARRAY      |
  && LOCK_OBJECT(X))                    |
  inlined method body of at             | access region of X
  UNLOCK_OBJECT(X)                      | release resources
else
  INVOKE(selector, X, i)                | fallback code
```

Figure 8.1: General Form of Speculative Inlined Invocation

In general, a speculative inlined method will have the form in Figure 8.1. The first condition checks to see that the selector is that of the method to be inlined (`at`). The second, `LOCAL_POINTER`, checks to see that the target object is local. Since the state of the object, including its concrete type and the values of its lock fields, will not be available unless the object is local, this check must be conducted first. If the object is local, the concrete type is verified to be that of the method to be inlined (`CLASS_ARRAY`). The last check attempts to acquire the necessary locks. In ICC++, an additional parameter is required giving the lock mask, since the current ICC++ implementation provides individual locks for each instance variable.[1] If these checks succeed, the body of the method is inlined. For a given speculative inline it may be possible to omit some or all of these checks. In this chapter we will assume that methods have been statically bound and omit the selector and concrete type checks. This region of code in which access to the object has been obtained is called an *access region*.

8.3 Access Regions

Access regions are created for each inlined operation on objects which require them. Figure 8.2 shows the twelfth kernel of the Livermore Loops [129] as an example. The C code for the kernel on the left contains a doubly nested loop surrounding three accesses to two objects. On the

[1] In fact, a separate lock need only be provided for sets of instance variables accessed as a unit.

right is a graphical representation of the access regions induced by speculative inlining of the access operations. The form of the graph is the Program Dependence Graph (PDG) [72]. The squares represent basic blocks (sets of statements which are all guaranteed to execute if any one executes), the ovals conditionals, and the circles access regions. The lines represent control decisions which are followed when the condition is true **T** or false **F**. Multiple lines with the same condition value are all followed for that value.

```
for ( l=1 ; l<=loop ; l++ )
  for ( k=0 ; k<n ; k++ )
    x[k] = y[k+1] - y[k];
```

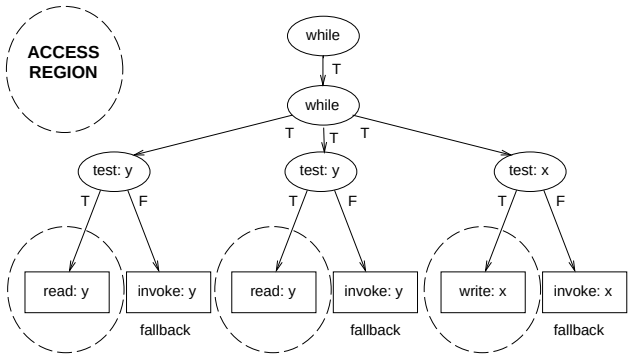


Figure 8.2: Livermore Loops Kernel 12: Code and Access Regions

8.3.1 Extending Access Regions

Entering the access regions introduced by speculative inlining requires costly run time checks. Since methods are often small in unoptimized code, access regions are entered frequently and the overhead can be severe. Consider the loop in Figure 8.2. Three run time checks are issued for each iteration of the inner loop. In order to reduce the overhead of these checks, the dynamic extent of the access regions should be expanded. This not only reduces the runtime check overhead but also produces larger basic blocks for the general optimizations of Section 7.7.

Since the Concert compiler normalizes all loops to be while loops, the PDG forms a tree. Thus, access regions are properly nested, with the locks being acquired and released at the same nesting level. This means that all operations on access regions can be broken down into operations which move statements into a region and those which create new empty regions with some set of conditions (e.g. speculatively acquiring access to a set of objects). The statements which are moved into a region may include other regions, conditionals and loops.

In this section, we first consider correctness issues in extending the dynamic extent of access regions (Section 8.3.2). We the describe three particular transformations, adding statements to

access regions (Section 8.3.3), merging of adjacent access regions (Section 8.3.4.1) and lifting access regions over loops and conditionals (Section 8.3.4.2).

8.3.2 Safety

Optimizations which expand access regions must not change the meaning of the program; they must be safe. In particular, they must preserve the original exclusivity properties and may not introduce deadlock. For example, transformations may not make it possible for the operations given by two separate methods to interfere, reading and writing the same variables over the same period of time. Likewise, transformations cannot hold a resource and then try to acquire that resource again with a blocking operation, inducing deadlock. These properties will be discussed for the individual optimizations.

Only the safety of those transformations which are peculiar to access regions are discussed here. In particular, the safety of moving a statement into both branches of a conditional, breaking a two-sided condition into two one-sided conditionals and eliminating code are not considered as they are covered in standard compiler texts.

8.3.3 Adding Statements to a Region

Entrance criteria for a region condition the enclosed storage accesses by tests for locality and access control conditions. Furthermore, the resources (represented by the locks held on the objects within the region) are held over the region. Hence, there are three types of statements: functional statements, those which do not access storage or hold resource; statements which access storage; and statements which hold resources, additional regions and blocking primitives operations.

Statements which are functional cannot interfere, nor can they capture shared resources (the registers and execution units they require are managed by the compiler). Hence, they can be moved safely into any region. For a storage access to be moved into the region, the tests for the destination region must subsume the tests for the storage access. Furthermore, if storage accesses for the same object from two distinct regions are moved into the region, they must be relatively exclusive [88]. One way to achieve this is to serialize the operations within the region. Finally, statements which hold resources cannot be moved into a region if doing so will induce

a cycle in the resource graph and deadlock. Lock cycles are prevented by merging the access regions (Section 8.3.4.1).

Primitives which hold resources (e.g. input/output) can also give rise to deadlock [94]. However, they are not common in performance critical sections, and can be conservatively excluded from the region. Furthermore, standard deadlock prevention techniques, like ordering resources, can be applied using the conservative call graph and alias information provided by flow analysis (Chapter 5).

Beyond these basic constraints, access-region expanding optimizations must also ensure that we do not move operations into an object's access region which could affect or be affected by the locality or locked status of the object. For example, we cannot move in any operation which might migrate the object, nor any operation which might directly or indirectly require its own lock on the object.

```
if (LOCAL_POINTER(X)
  && LOCK_OBJECT(X))
  inlined method body of at  | access region
  j = i + 1                  |
  UNLOCK_OBJECT(X)           | release resources
else
  INVOKE(at, X, i)           | fallback code
  j = i + 1                  |
```

Figure 8.3: Adding `j = i + 1` to an Access Region

Figure 8.3 shows a simple function statement `j = i + 1` added to the access region from Figure 8.1. Since the statement is functional, no additional conditions are required for entrance into the region. The statement is added both to the access region and to the fallback code, preserving the meaning of the program under different dynamic conditions.

8.3.4 Making a New Region

An empty access region consists solely of a test of access conditions and a temporary acquisition of resources. Such regions do not change the meaning of the program unless they introduce new deadlocks. Such deadlocks would arise from new dependences between locks, and can be prevented by testing and obtaining all required locks atomically. The runtime (Section 3.5)

provides multi-locking atomic primitives. Thus, if a set of resources is available, they are acquired. Since the empty region contains no statements, whether they are acquired or not, the resources are immediately released. Thus, new regions can be created without introducing deadlocks and statements can be moved into the region as above.

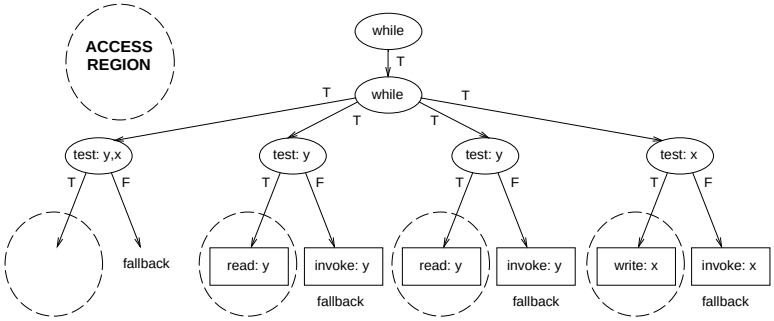


Figure 8.4: Livermore Loops Kernel 12 with New Region

New regions are created with access conditions for a set of operations which can be profitably optimized as a unit. For example, to optimize the statements enclosed in the three regions from Figure 8.2, the access conditions would require testing the properties of both x and y. Figure 8.4, show the creation of such a new region. Note that both the access region and fallback code blocks are empty.

8.3.4.1 Merging Access Regions

Merging access regions combines the access conditions for two regions and merges the access and fallback code blocks. First, a new region is created with the conjunction of the access conditions. Then, using the partial order of execution derived from local data flow and the CFG Data Dependences (Section 4.2.3.1) in the PDG, statements which must execute between the two regions are determined. These statements are moved into both the access region and the fallback blocks for the new region. If all such statements can be moved, the regions can be merged. Finally, the access and fallback statements are moved from the two initial regions into their respective branches of the new region.

The combined access conditions represent the conjunction of those for the original regions. If the new conditions attempt to acquire the locks on a single object twice, the attempt will fail, preserving mutual exclusion. However, if we know the two objects are the same, we

can take out a single set of locks and ensure mutual exclusion through relative atomicity by sequencing the operations from the two regions so that they do not interleave. This requires a must-alias determination which need only be conservative since the fallback code is completely general. The statements from the access and fallback blocks for two such regions are given an order which is consistent with the partial order of execution when they are placed into their respective branches of the new region.

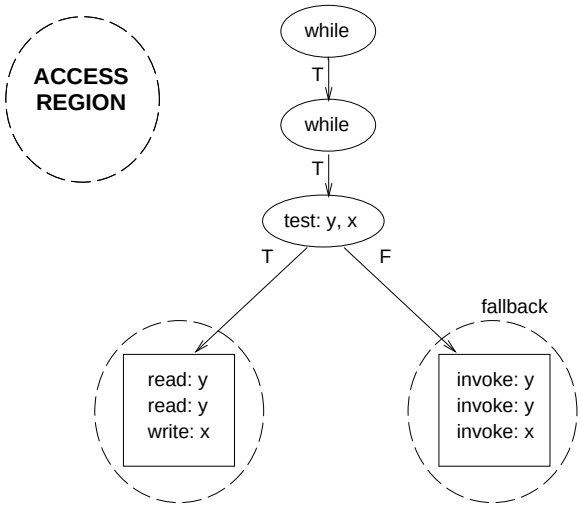


Figure 8.5: Livermore Loops Kernel 12 After Merge

The result of merging the regions in Figure 8.2 appears in Figure 8.5. The new region introduced in Figure 8.4 has absorbed all the statements from the other regions. These regions can then be eliminated safely following the logic by which new empty regions were introduced without changing the meaning of the program. The three conditionals have been merged into a single access condition and the three optimized and fallback blocks into single optimized and fallback blocks.

8.3.4.2 Lifting Access Regions

The PDG is a tree whose the interior nodes are conditionals and while loops. Lifting access regions higher in this tree can improve efficiency by enabling runtime testing overhead to be removed from loop bodies. Lifting access regions over conditionals can enable further merging and lifting operations and which also increase efficiency. The key to lifting access regions is to ensure that all the statements under the PDG node to be lifted over can be safely added

to the region. Using a bottom up traversal of the PDG, at each level we attempt to merge adjacent access regions until only one remains within the control dependence region. We then attempt to move any remaining statements into the single access region. If there is one access region and no other statements in a control dependence region, that region can be lifted over the conditional or while loop.

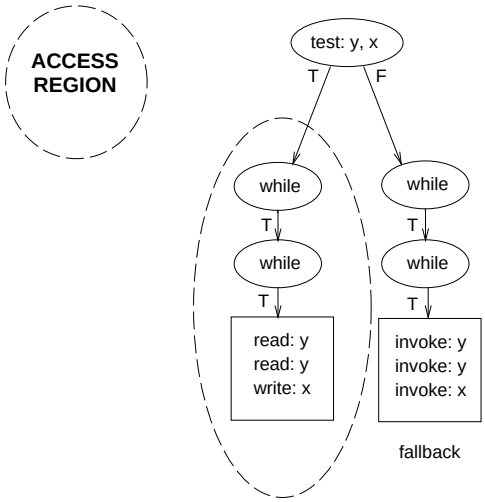


Figure 8.6: Livermore Loops Kernel 12 After Hoist

Consider the two types of control structures in our PDG: while loops and conditionals. While loops have a single control dependence region under them, corresponding to the loop body. If this control dependence region is entirely contained in a single access region, the loop header can be moved into the access region. This results in an access region which is lifted over the loop. For single armed conditionals, the transformation is analogous. So long as loops eventually terminate, the resources acquired by the access region will be released. Fortunately, the notion of fairness supported by Concert requires loops to terminate. A stronger notion of fairness would require periodically releasing the locks at points consistent with the required mutual exclusion of the initial regions.

Conditionals with two arms require the two regions to be merged and lifted simultaneously. Logically, we break such conditionals into two one-armed conditionals, the second with the negation of the original condition. Then, the two access regions are lifted over the conditionals. Next, the two access regions are merged. Finally, the two one-armed conditionals within the access region are recombined in a single two-armed conditionals. The result being that the

```
if( LOCAL_POINTER(x) && LOCAL_POINTER(y)
    && LOCK_OBJECT(x,y) )
    for ( l=1 ; l<=loop ; l++ )
        for ( k=0 ; k<n ; k++ )
            x[k] = y[k+1] - y[k];
    UNLOCK_OBJECT(x,y);
else {
    for ( l=1 ; l<=loop ; l++ )
        for ( k=0 ; k<n ; k++ ) {
            t1 = INVOKE(at, y, k+1);
            t2 = INVOKE(at, y, k);
            INVOKE(putat, x, t1 - t2);
        }
}
```

Figure 8.7: Kernel 12 After Lifting

region has been lifted over the two-armed conditional. In practice, these operations can be combined and the entire transformation done at once.

8.3.4.3 Access Region Optimization

Determining which regions can be most profitably merged requires information about the probability that access conditions will hold. For example, a block of code which operates on three objects, two of which are usually local and one of which is usually remote should be implemented with two access regions; one for the two usually local objects and one for the usually remote. This is because the optimized path is only executed when all the conditions are satisfied at once. Since the cost of the general case (blocking for a lock or remote message) is large, the optimization extracts high efficiency from the optimized path at a relatively small increase in cost along the unoptimized path. However, including a condition which is often unsatisfied prevents the optimization of all affected code. Of course, additional levels of speculation can be used in the fallback block at the cost of additional code expansion.

Once the access regions have been expanded, the code consists of larger regions of optimizable sequential code. If the program spends the majority of its time in these regions it will be very nearly as efficient as a sequential implementation. Applying these transformations to the code on the left of Figure 8.2 results in the code structure in Figure 8.6. The final result is an

optimized loop body under two loops conditioned by a single combined access condition, and the fallback code under a copy of the same two loops. When both x and y are local, the loop nest in the access region is identical to a sequential program. An example of the resulting code appears in Figure 8.7.

8.4 Caching

Caching is the storage of a piece of data higher in the memory hierarchy (Section 3.1.1) where it can be accessed with greater efficiency. There are three classes of data which can be cached under different conditions: globals (Section 8.4.3), local variables and instance variables.

Our programming model disallows pointers to local variables since this breaks encapsulation of the local state (Chapter 2). If this were not the case, aliasing problems combined with concurrency might require a local variable to be reloaded from memory for each operation. Instead, the programming model allows for multiple return arguments which satisfies most uses of such pointers. In the remaining cases, the data element can be represented by an object. Since objects are potentially aliased, by extension, an aliasing problem exists for instance variables. Fortunately locks and the information provided by flow analysis can be used to effectively test and conservatively approximate object aliasing.

8.4.1 Local Variables

Local variables are part of the state of a thread. Because threads can block for indeterminate time, in general, local variable data must be stored in a persistent store like the heap. Since accessing such data can be expensive, local variables are cached into registers where possible.[2] Two pieces of information are required to cache local variables: the type of the data, which determines the sort of register to use; and the lifetime of the value. This lifetime differs from normal lifetime computations because it is dependent on possible context switch points where the data must be stored to the persistent store.

Exploiting lifetimes across basic blocks is discussed in detail in [145]. Essentially the idea is to partition the control flow graph into contiguous sets of instructions delimited by possible

[2]Chapter 9 considers the tradeoff between caching data and saving data back in the heap context.

```
//  retrieve a  and b  from the store
func0(a);
b = a + b;
func1(a);
func2(a);
//  store b  to the store
// POSSIBLE CONTEXT SWITCH
//  if switched, retrieve a  from the store
func1(a);
func2(a);
```

Figure 8.8: Example of Caching and Partitions

context switch points (touches). For each partition, the set of variables active in that partition are cached in registers. At partition boundaries, the compiler generates code mapping the active set of one partition to the other by storing or loading variables between registers and the heap as appropriate. Figure 8.8 shows and example with two partitions and the code to load and store the local variables. If the context switch does occur, all variables will have been stored to the heap, and all required variables will be loaded on resumption.

8.4.2 Instance Variables

In Section 7.5, aliasing information derived from flow analysis is used to convert instance variables into Static Single Assignment form. This enables them to be allocated to registers over parts of their lifetime. Access regions provide aliasing information which can be used for the same purpose. Within an access region, all accesses to the data of locked objects must be through known pointers. Hence, by exploiting access regions, object state can be cached in registers safely eliminating memory accesses and requiring only a single update at the end of the access region or subsequent method invocation on the object in question.[3]

On the left in Figure 8.9 a two-dimensional array is accessed by linearization (Section 6.1). The code on the right uses the properties of access regions to cache `cols` in a local temporary

[3]Specialized versions of methods can also be created which would take the instance variables in registers. Such data structure transformations are made possible through cloning (Chapter 6) which allows specialized versions of methods to be created based on characteristics of the calling environment.

```
if (LOCAL_POINTER(a)&&LOCK_OBJECT(a))
  for (let i = 0; i < a.rows; i++)
    for (let j = 0; j < a.cols; j++)
      ... = a[a.cols * i + j];
  UNLOCK_OBJECT(a);
else
  // fallback
```

```
if (LOCAL_POINTER(a)&&LOCK_OBJECT(a))
  temp = a.cols;
  for (let i = 0; i < a.rows; i++)
    for (let j = 0; j < temp; j++)
      ... = a[temp * i + j];
  UNLOCK_OBJECT(a);
else
  // fallback
```

Figure 8.9: Caching of Instance Variables

`temp`. This optimization saves a memory reference in the innermost loop and enabling other optimizations such as strength reduction. Even if there was an assignment to `cols` within the loop, so long as it did not involve the object `a` this transformation would be safe. The access conditions act as a dynamic alias check, ensuring that no other accesses to the object will occur except through the designated reference.

8.4.3 Globals

In order to reduce the volume of communication in a distributed memory machine, variables which are used frequently but do not change their value over some period of time should be replicated in memory closer to each processing element. This is done by detecting a set of reads delimited by synchronizations from surrounding write operations, inserting a distribution operation and redirecting the reads to local copies.

In a programming model with tree-structured concurrency, time can be defined in terms of the synchronizations at the start and end of a subtree of tasks. Thus, globals which are constant in all concurrent subtrees, are constant over some period of time. For example, in Figure 8.10 the task on the left writes x, but is synchronized with the two concurrent task subtrees on the right, both of which read x. Thus, x is does not change after the synchronizations.

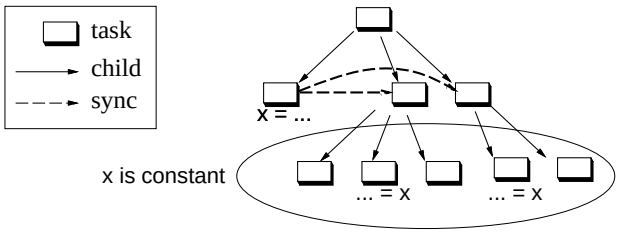


Figure 8.10: Temporally Constant Globals

This situation, of temporally constant globals, can be detected by a specialized interprocedural analysis using the interprocedural call graph produced by flow analysis (Chapter 5). The objective is to detect when some group of reads is delimited by synchronization points from the writes. The initial synchronization is the point where the value becomes fixed and it can be replicated in each local memory. The final synchronization prevents causal inconsistencies.

To see the causal inconsistency more clearly it is necessary to flatten the call tree into a task graph. In Figure 8.11, the write at **A** is separated by a synchronization (shown as an arrow) from the reads at **B** and **C**, however, the write at **D** is unsynchronized. In theory, both **B** and **C** could both use the value produced by **A**. If **D** and **B** are assigned to the hardware so that they share the same local memory, **D** could update the value of y which **B** reads while **C** would read the replicated value from **A**. Because **B** is synchronized with **C**, this forms a temporal inconsistency (**B** sees the world after **D** while, **C**, which should occur after sees the world before **D**). We might try to eliminate this problem by separating the replicated and updatable data, but that leads to a second type of inconsistency.

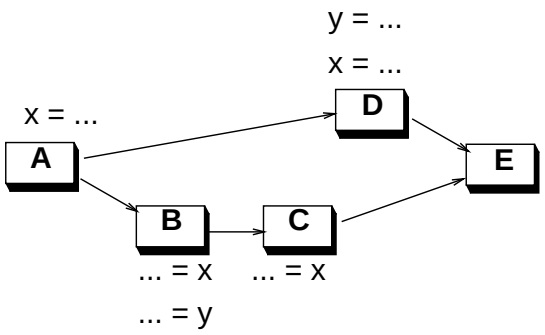


Figure 8.11: Global Temporal Inconsistency

The second sort of inconsistency comes from indirect communication through shared data. Consider the variable y in Figure 8.11. Here we have an analogous situation. The values of y and x are synchronized through **D**, hence **B** cannot use the replicated value of x and the unreplicated y. Clearly there are situations where the requirement of the final synchronization can be avoided. However, this optimization has proven useful even with that restriction. As we will see in Sections 8.3 and 8.4, scheduling and atomicity can be exploited to further reduce communication costs for temporally constant data.

Thus, when a set of delimited read operations is found, a distribution operation is inserted and the reads are redirected to the local copies of data. Figure 8.12 shows a group of read operations optimized in this manner.

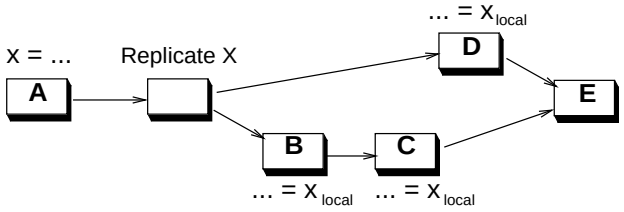


Figure 8.12: Temporally Constant Globals

8.5 Touch Optimization (Futures)

Touches are the points at which a thread synchronizes with the results of asynchronous message sends. Their placement affects the number of context switches and the amount of state which must be saved at each context switch point. In order to minimize the number of context switches and maximize the effectiveness of latency hiding, touches are pushed forward in the program and grouped.

8.5.1 Pushing

The ability of a concurrent program to withstand remote operation latency is dependent on the number of outstanding concurrent asynchronous operations, and the distance in time between when the operations are initiated and when the results are required. Touches are the points where the results are demanded and must occur before the results are used. Placing touches as far from the point where the future was created as possible increases both the number and duration of asynchronous operations.

Figure 8.13 shows two alternative touch placements. On the left, each operation is initiated and the results required in turn (i.e. sequentially). On the right, both operations are initiated first, allowing them to overlap in time, before any result is required. In this way, the program pays the maximum of the two operation latencies instead of the sum.

Touches are inserted along the *data frontier*. The data frontier is the last set of points in the control flow graph which possibly redundantly dominates all of the uses of the data. Consider

```
(xFuture,xCont) = MAKE_FUTURE(x);
INVOKE(meth1, a, xCont);
TOUCH(x);
(yFuture,yCont) = MAKE_FUTURE(y);
INVOKE(meth2, b, yCont);
TOUCH(y);
z = x + y;
```

```
(xFuture,xCont) = MAKE_FUTURE(x);
(yFuture,yCont) = MAKE_FUTURE(y);
INVOKE(meth1, a, xCont);
INVOKE(meth2, b, yCont);
TOUCH(x);
TOUCH(y);
z = x + y;
```

Figure 8.13: Touch Placement for Latency Hiding

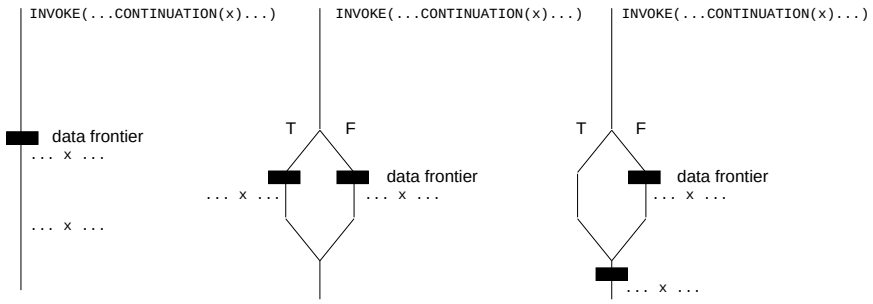


Figure 8.14: Data Frontier for Touch Insertion

the control flow graphs in Figure 8.13. Within a basic block, the frontier is the single point before the data is used (left). If a conditional or loop is encountered the frontier may include several points. Because touches are idempotent it is safe to touch a value which has already been touched. Hence, the data frontier may include more than one point along some paths through the program (right).

When control flows to a higher level in the PDG (i.e. exits a loop or conditional branch) a ϕ-node is encountered, causing the touch to be inserted before the loop or branch is completed. Counting continuations are an exception (Section 3.2.1.5). Counting continuations are used to synchronize the termination of all the bodies of a parallel loop. The creation of a normal continuation for a variable which had not yet been touched would result in the destruction of the future and a failure to synchronize with the first result value. However, since counting continuations record the number of outstanding result values, multiple continuations can be constructed for the same variable and a single touch placed outside the loop is used for synchronization. Figure 8.15 shows a loop with (left) with a standard continuation. The variable is touched before the iteration terminates. In the case of the counting continuation (right), the variable is touched outside the loop, synchronizing once for all iterations.

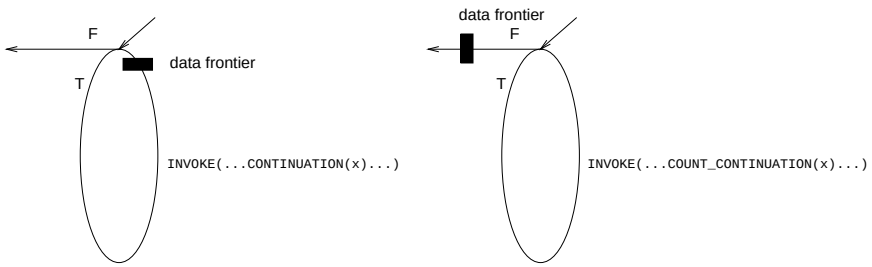


Figure 8.15: Data Frontier for Loops

There is an inherent tradeoff between latency hiding and active thread state. The more asynchronous invocations which are outstanding, the more active state must be maintained. For example, in Figure 8.16, the code on the left touches the variables x and y early. This allows it to resolve the variable q making x and y inactive. Alternatively, the code on the right waits for all of the variables x, y and z to be computed before doing any computation. The code on the left requires less active state, but the code on the right hides latency better.

```
(xFuture,xCont) = MAKE_FUTURE(x);
(yFuture,yCont) = MAKE_FUTURE(y);
INVOKE(meth1, a, xCont);
INVOKE(meth2, b, yCont);
TOUCH(x);
TOUCH(y);
q = x + y;
(zFuture,zCont) = MAKE_FUTURE(z);
INVOKE(meth3, c, zCont);
TOUCH(z);
r = z + q;
```

```
(xFuture,xCont) = MAKE_FUTURE(x);
(yFuture,yCont) = MAKE_FUTURE(y);
(zFuture,zCont) = MAKE_FUTURE(z);
INVOKE(meth1, a, xCont);
INVOKE(meth2, b, yCont);
INVOKE(meth3, c, zCont);
TOUCH(x);
TOUCH(y);
TOUCH(z);
q = x + y;
r = z + q;
```

Figure 8.16: Touches and Active State

8.5.2 Grouping

To minimize the number of times a thread is restarted as a result of the value of a future becoming available, touches are grouped so that the thread restarts once for a set of values. Figure 8.17 illustrates how multiple outstanding messages and a single multi-touch operation are used. In a fine-grained concurrent language, the order of the invocations of `meth()` on `a`, `b` and `c` is not strictly specified. This enables the compiler to pull the messages sends up and push the touches down.

```
v1 = a.meth;        (v1Fut,v1Continuation) = MAKE_FUTURE(v1); ...
v2 = b.meth;        (v2Fut,v2Continuation) = MAKE_FUTURE(v2); ...
v3 = c.meth;        (v3Fut,v3Continuation) = MAKE_FUTURE(v3); ...
...                 ...
func(v1,v2,v3);     if(!MUTIPLE_TOUCH(v1Fut,v2Fut,v3Fut)) SUSPEND;
```

Figure 8.17: Latency Hiding and Multiple Touches

8.6 Experimental Results

To test the effectiveness of these transformation, we compare the performance of our concurrent object-oriented system to C [114]. For comparison, the Livermore Loops, a set of numerical kernels [129] are used to measure efficiency through computation rate. The Livermore Loops are used for three reasons. First, they are a traditional benchmark for sequential compilers. Second, they are extremely sensitive to any inefficiency. Any extra operations in the inner loop can dramatically reduce performance. Third, the granularity of each method (amount of work per method invocation) is very small. Many methods simply compute the array index, or load or store a single element. This makes the elimination of overhead especially important. All reported numbers are for Workload 3 of the Livermore kernels at single precision run on a Sparcstation II. The COOP execution times were collected with the UNIX `time` facility using high iteration counts, and are accurate to within a few percent.

The COOP programs are written in a natural object-oriented style. Multi-dimensional arrays were created by subclassing a single dimensional array and using methods to linearize the indexing operations (see Section 6.1). Since our COOP programming model does not allow pointers, the programmer cannot bypass the encapsulation of the arrays as is typically done in C++ programs to obtain efficiency. We compare our COOP system's performance against the native C version of the Livermore kernels compiled by the same GNU C/C++ compiler as we used for the Concert backend, minimizing differences in low-level optimizations like instruction selection and scheduling.

To illustrate the effect of the different optimizations, we applied each in turn to Kernel 12. As with the results presented in Chapter 7, a full suite of conventional low level optimizations were also applied. These have been separated out. These numbers are presented in Figure 8.18. Each transformation produces significant performance improvement. Traditional optimizations alone (**none**) achieve only several kiloFLOPS (thousands of floating point operations per second).

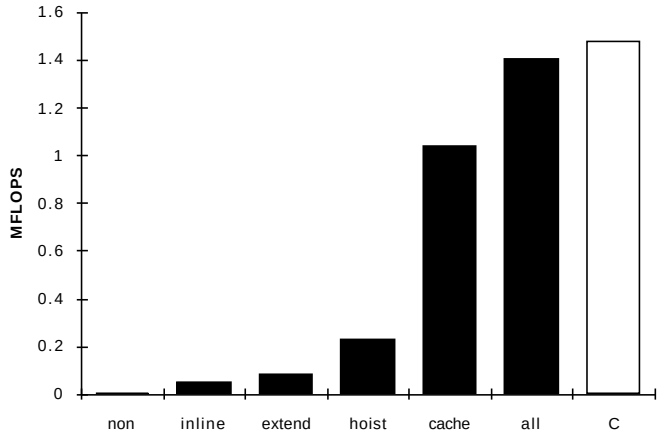


Figure 8.18: Cumulative Effect of Optimizations on Kernel 12

Speculative inlining produced an eighteen-fold performance improvement (**inline**). Expanding access regions by merging increased performance by another 60% while lifting them brings this to 440%. At this point, performance was still quite poor, around 200 kiloFLOPS on a code where the sequential C program achieves 1.5 MFLOPS. Caching (**cache**) resulted in 4.5 times performance improvement, closing on C's performance. The remaining gap is the result of the backend C/C++ compiler's inability to do common optimizations on the code output by the Concert compiler. After applying these standard low level optimizations, the final results (**all**) improve by 40%, essentially matching the native C implementation and nearly 500 times better than that of **none**.

The performance results for all of the Livermore Loops are in Figure 8.19. The performance of the COOP code and that of the native C code are quite close. Furthermore, this performance exceeds that which would be delivered by most C++ compilers on code written in an object-oriented style. We measured the performance of two representative Livermore kernels using the same version of the GNU C++ compiler used for the other experiments. Kernel 12, using virtual functions to access elements in a one-dimensional array, achieves 0.42 MFLOPS, less than a third of the COOP or the C performance. Kernel 21, on two-dimensional arrays, achieves 0.32 MFLOPS, and with non-virtual functions, only 0.45 MFLOPS, less than one fifth of the COOP or C performance.

Figure 8.20 shows the performance of the COOP implementations relative to the C implementations ((COOP-C)/C). Of the 24 kernels, our COOP implementation was more than

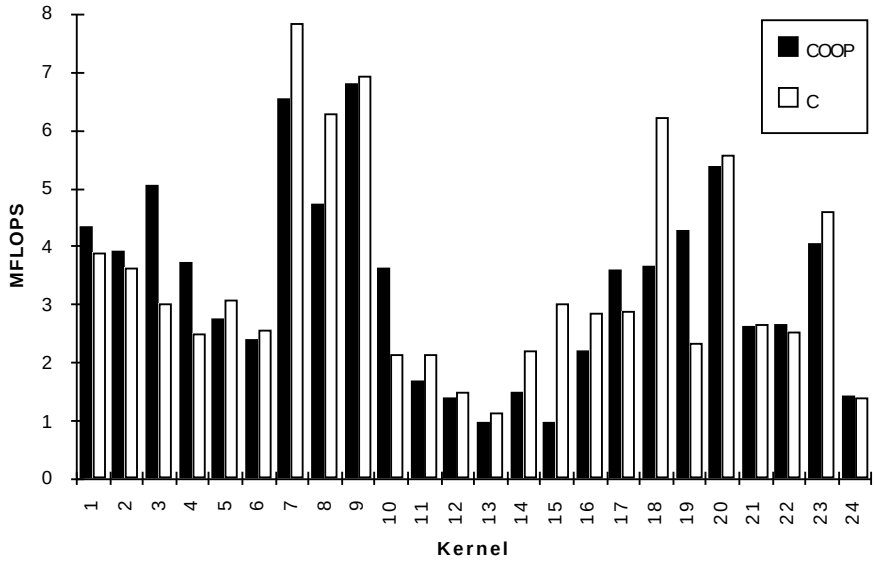


Figure 8.19: Performance on Livermore Loops

20% faster on five, the C implementation was more than 20% faster on six, and the remaining thirteen were essentially the same. For the codes where the C compiler gave superior performance, the differences are the result of idiomatic array manipulation optimizations in the GNU C compiler and some deficiencies in our strength reduction optimization (it uses extra registers and does not work for operations under conditionals in loops as in Kernel 15). The GNU C compiler manipulates arrays without the use of integer multiplication, dramatically improving performance on the SparcStation II which uses a multiply step instruction. Where the COOP system was faster, the major factor was again low level optimizations. By recognizing more general patterns instead of idioms, we were able to apply some optimizations where the GNU C compiler was unable to.

Overall, these results demonstrate that the potential overhead of the COOP model can be eliminated. The remaining differences are the result of differences in standard low level optimization. Furthermore, in some cases, the COOP model has exposed additional opportunities for optimization. For example, dynamic aliasing information is explicit in the access regions and could be used to reorder memory optimizations within the loops, potentially increasing the efficiency even further.

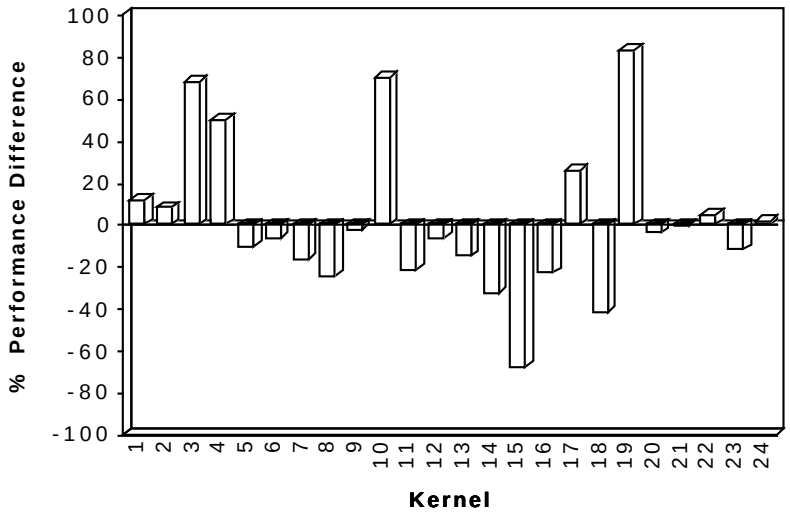


Figure 8.20: Performance Difference ((COOP-C)/C)

8.7 Related Work

There are few high performance fine-grained COOP language implementations: many COOP system use COOP simply as a coordination language for algorithmic cores written in more conventional languages like C or FORTRAN. Nevertheless, there are systems which transform COOP programs for greater efficiency. The HAL system [6, 101, 115] supports Actors style programming, which differs somewhat in synchronization and concurrency introduction from the Concert COOP style, with a high degree of flexibility and efficiency. The family of ABCL implementations use a variety of different techniques and strategies to obtain performance [185, 186, 183, 187, 170, 171], with an emphasis on the efficient implementation of various language features. Exclusivity and deadlock issues appear in the concurrent systems framework of critical regions [88], monitors [20] and deadlock prevention [94]. Our inlining and access region lifting techniques draw on the efforts at Rice on Fortran D [93], and in particular the ideas in [84] and the importance of interprocedural optimizations within loops. Also, combining and lifting access regions resemble invarient lifting and the lifting and blocking of communication in parallel Fortran. However, access region optimizations have special safety requirements, and require managing fallback code and manipulating entrance criteria.

8.8 Summary

This chapter discusses a number of optimizations specific to concurrent object-oriented programming. Lock operations can be optimized by taking advantage of *access subsumption*, when the structure of the call graph necessitates that the access rights required by a method will have already been acquired when the method is called, and by recognizing *stateless methods* which do not required access at all. Speculative inlining introduces *access regions*, regions of code over which access to an object has been granted. These regions can be transformed so to amortize the cost of speculation and to increase the size of access regions for conventional optimizations. Optimization of memory hierarchy traffic is of particular importance for COOP codes where much of the data is access by indirection. Flow analysis and the information provided by obtaining access to objects can be used to cache data at higher levels of the memory hierarchy for efficiency. Distributed global variables can likewise be optimized by using the call graph to detect *temporally constant* globals, and by caching their value on each node. Synchronization of threads is another potential source of inefficiency which can optimized by careful placement of touches. The programmer can also provide locality and locking information which must be propagated to the points of use in the intermediate representation. It is shown that the Livermore Loops, written in a natural COOP style, can be made as as efficient as C when the data is available (local and the required locks are available).

8.9 Acknowledgments

Caching of local variables (summarized in Section 8.4.1) is the work of Xingbin Zhang who contributed greatly to this work in general, and it is included here only for completeness.

Chapter 9

Hybrid Sequential-Parallel Execution

The hybrid parallel-sequential execution model [144] presented in this chapter adapts for parallel or sequential execution and provides a hierarchy of calling schemas of increasingly power and cost. Together, these enable hybrid execution to achieve both high sequential efficiency when the required data is available, and latency hiding and parallelism generation where required. Section 9.1 describes how irregular programs can benefit from the adaptive nature of hybrid execution. Section 9.2 describes hybrid execution, the parallel and sequential versions of methods, four sequential schema and wrapper functions used to match different calling conventions. Finally, Section 9.3 evaluates the effectiveness of hybrid execution for invocation intensive codes, regular and irregular applications.

9.1 Adaptation

Irregular and dynamic programs (such as molecular dynamics, particle simulations and adaptive mesh refinement) have a data distribution which cannot, in general, be predicted statically. In addition, modern algorithms for such problems depend increasingly on sophisticated data structures to achieve high efficiency [15, 79, 25]. Moreover, runtime techniques like dynamic data shipping, for increased data locality, and dynamic function shipping, for load distribution, disrupt the static data locality relationships. As a result, a program implementation must

adapt to the irregular and dynamic structure of the data, exploiting locality where available, to achieve high performance.

Adaptation is a flexible technique which enables the program to take advantage of good data distributions provided by the programmer or the runtime system. The data distribution can even be modified at runtime, based on evolving program information. The execution of the program – including thread creation, messaging, and synchronization – adapts to the new data distribution, providing immediate improvement. Figures 9.1 shows an example of data (the circles) distributed around a parallel machine (the squares). Relationships between the pieces of data are described by lines between them. This is a good data layout since groups of tightly coupled objects are on the same node.

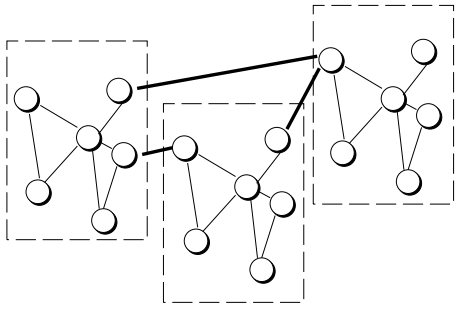


Figure 9.1: Data (Object) Layout Graph

An efficient computation over the data minimizes the communication between nodes. For example, in Figure 9.2 the tree of threads is distributed over the machine with portions at the leaves executing on co-located objects. This structure enables specialized sequential code to be executed for sets of threads near the leaves, and specialized parallel code for those near the root. Such specialized code can be optimized to its task, so that the algorithm core will be as fast as conventional sequential code, whereas the parallel code will efficiently spawn parallelism and hide latency.

The sequential code is optimized by assuming that threads will complete immediately. This enables temporary storage to be reused immediately, expensive scheduling operations to be avoided, and cheap linkage mechanisms to be used. The normal program stack, call and return mechanisms can be used, enabling efficiency matching conventional sequential code. If the thread does not complete, the running program adapts by spinning off a parallel thread lazily.

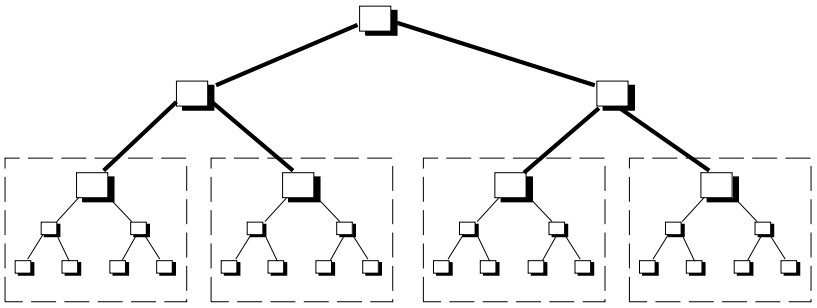


Figure 9.2: Distributed Computation Structure

Since the overhead of creating a thread is high compared to that of a conventional call, this strategy produces low overhead even when fall back to the parallel version is common.

Hybrid execution is the complement of hardware shared memory, which adapts the location of data to suit the location of computation. Hybrid execution can adapt the location of the computation to the location of the data. Moreover, the units of communication and computation are controlled by the compiler and runtime system. Where the shared memory system must migrate a cache line to the requesting processor, hybrid execution can adapt to the movement of data and functions of arbitrary size. Also, while the shared memory hardware allows a limited number of outstanding data requests based on a non-blocking cache, the number of outstanding operations and their latency has no fixed limit with hybrid execution.

While a static execution models might provide good performance for a particular computational structures, the hybrid execution model with its adaptation and hierarchy of invocation schemas can provide good performance for many structures. Using the results of flow analysis (Chapter 5), the compiler specializes the calling conventions based on the synchronization features required by the called method. Furthermore, the runtime provides a hierarchy of runtime primitives of increasing cost and complexity, enabling the compiler to select the most efficient mechanism for a given circumstance.

9.2 Hybrid Execution

Hybrid execution adapts to the computational structure of a running program by providing separately optimized sequential and parallel code. The sequential code executes method invocations representing potentially independent threads in LIFO (last in first out) order and

schedules them immediately. This eliminates the overhead normally associated with thread creation, scheduling and synchronization. The parallel code spins off independent threads, generating parallel work and hiding the latency of concurrent operations.

The goals of the hybrid sequential-parallel execution are efficiency, flexibility, portability, and support for range of data layouts through runtime adaptation. This execution scheme is meant to supplement static compile time techniques such as static data placement and code specialization (Chapter 6). In order to support these goals, hybrid execution uses sequential and parallel versions of methods and four distinct invocation schema. These schemas range from cheap, simple and limited to general, complex and more expensive. To avoid confusion, invocations in the concurrent object-oriented programming model are called *method invocations* and implementation level C calls, *function calls*.

9.2.1 Overview

For each method, there are two versions: a parallel version optimized for latency hiding and parallelism generation using a heap context and a sequential version optimized for efficient sequential execution using the stack.[1] The parallel version is completely general, capable of handling remote invocations and suspension, but can be inefficient when the generality is not required. The sequential version comes in three flavors of increasing generality. These different versions and flavors use different calling conventions to handle synchronization, return values and reclaim activation records. Table 9.1 describes these cases.

Case		Basic Operation
Parallel		Most general schema, all arguments/linkage through the heap; frame reclamation based on function termination
Sequential	Non-blocking	Regular C call/return
	May-Block	Regular call; check return code to either continue computation or peel stack frames to heap
	Continuation Passing	Extension of May-Block which enables forwarding on the stack

Table 9.1: Invocation Schemas

[1]In practice, one of the versions may not actually be generated if it is deemed unnecessary for either correctness or efficiency.

In the remainder of this section, the mapping from method invocations to C function calls is described. Since the Concert compiler backend generates C++, this description roughly parallels the output of the compiler. First the parallel invocation mechanism is described (Section 9.2.2, followed by the flavors of sequential invocation (Section 9.2.3). Finally, Section 9.2.4 discusses the *proxy* contexts and wrapper functions which are used to handle certain boundary cases.

```
method(...args...) {
  Slot a, b, c;
  INVOKE(methodA,&a,...);
  INVOKE(methodB,&b,...);
  INVOKE(methodC,&c,...);
  ...  continue heap execution ...
  if (!touch(&a,&b,&c,...)) {
      ...  store state ...
      ...  suspend ...
   }
  ...  use values in a, b, c ...
  REPLY(continuation, return_value);
}
```

Figure 9.3: Generated Code Structure for a Parallel Method Version

9.2.2 Parallel Invocations

The parallel invocation schema is a conservative implementation of the general case, allocating the activation record on the heap and passing the arguments through the heap as well. Parallel invocations create threads which preserve their state between context switches in the heap activation record. This is the execution model described in Section 3.2.1. By storing inactive temporary values in the heap-based record (context), the cost of suspension is minimized. Such suspensions occur while waiting for the result of:

- A remote invocation,
- An invocation on a locked object,
- A blocking primitive (e.g. I/O), or
- A local invocation which itself has suspended

Suspension and fall back from the stack invocation schemas are described below.

Figure 9.3 shows an example of a parallel version of a method which several parallel method invocations and synchronizes on their return with a single touch. A pointer is provided to the slot corresponding to the return value. If the runtime system decides to create a new thread for the invocation it will create a continuation and convert the slot into a future (Section 3.2.1.3). If the task is scheduled immediately, the continuation/future pair need not be created and the value can be written directly into the return location with minimal overhead.

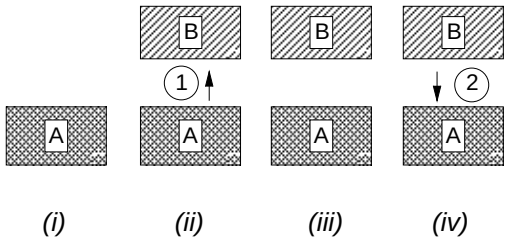


Figure 9.4: Parallel Invocation Schema Graphic

Figure 9.4 gives a graphical representation of the parallel calling schema. On a call to **B**, a thread is created for **B** and a heap-based context. A future/continuation pair are created for the return value. The future is created in **A** and the continuation is stored in **B**. Both threads are now free to execute in parallel. When **B** completes, it uses the continuation to pass the return value to the future in **A**.

Since the invocations execute in parallel the return values can arrive in any order. Touching a set of futures are at one time to avoid unnecessary restarts of the activation when not all of the needed values are available (Section 8.5.2). Concurrency is generated both across parallel calls and between caller and callee and latency is masked by enabling several invocations from the same method to proceed concurrently. Thus, parallel method versions are optimized for concurrency generation and latency hiding.

9.2.3 Sequential Invocations

There are three different schema for sequential method versions, each requiring a different calling convention (see Figure 9.5). Since determining the correct schema can depend on non-local (transitive) properties, the interprocedural call graph provided by flow analysis (Chapter 5) is used to conservatively determines the blocking and continuation requirements of methods and to select the appropriate schema. Since only one sequential version of each method is generated, this classification determines the calling convention used when the method is invoked.

NON-BLOCKING	`return_val = non_blocking_method( ... );`
MAY-BLOCK	`callee_context = may_block_method(&return_val,...);`
CONTINUATION PASSING	`caller_context = cont_passing_method(&return_val,caller_info,...)`

Figure 9.5: Invocation Schema Calling Interfaces

The criteria for selection of the sequential method versions is as follows. If the method and all of its callees cannot block, then the **Non-blocking** version is used. In this case, the function return value can be used to convey the future value. When it cannot be shown that blocking will not occur but the callee does not require a continuation, the **May-block** is used. In this case we optimistically assume the method will not block, and allocate any required context lazily as described in Section 9.2.3.2. Finally, the **Continuation Passing** version is used if the callee may require the continuation of a future in the caller's as yet uncreated context. In this case both context and continuation are created lazily. Lazy creation of continuations is described in Section 9.2.3.3.

9.2.3.1 Non-blocking: Standard Call

When the compiler determines that a method will not block, a standard C procedure invocation is used. Since this situation is determined over the call graph, entire non-blocking subgraphs are executed with no overhead. Thus, those portions of the program which do not require the flexibility of the full concurrent object-oriented programming model are not penalized.

```
int method15(...args...) {
  variable1 = method23(...);
  variable2 = method1(...);
  variable3 = method7(...);
  ...  use values in variable1, variable2, variable3  ...
  return return_value;
}
```

Figure 9.6: Code Structure for a Non-blocking Sequential Method

Figure 9.6 shows the structure of a non-blocking sequential method. Return values are assigned directly to variables which are then used normally. And the result of the method itself is returned directly. Logically the thread representing the invoked methods have been statically scheduled immediately. Since they cannot block, fairness (Section 3.3.2) is not a problem.

9.2.3.2 May-block: Lazy Context Allocation

In the may-block case, the calling schema assumes that the method will complete immediately, and if it blocks, that it will create its context lazily. Linkage is provided by having the caller create future and continuation for the result value and place the continuation in the newly created context.

```
callee_context = may_block_method(&return_val,...);
if (callee_context != NULL) {     // fallback code
   context = create_context();
   callee_context->continuation = make_continuation(context[13]);
   ... {\it save_state_to_heap} context ...
   return context;                // propagate blocking
}
```

Figure 9.7: May-block Calling Schema

Figure 9.7 shows an example of the may-block calling schema. This schema distinguishes two outcomes for the callee: successful completion and blocking. If the callee runs to completion, a NULL value is returned. If the callee blocks, it allocates a heap context, stores its state and return a pointer to that context. Since the C return value is used to indicate completion, the

result of the method is returned through a pointer passed in as an additional argument. A native code implementation would likely use an additional register for the result instead.[2]

In the example, on successful completion the caller extracts the actual return value from `return_val`. If the method blocks, the callee context is returned. This context is used to set up the linkage between caller and callee. The caller creates a continuation for the future value of the result, and places that continuation in the callee context. This process can cascade. The caller may, if necessary, create its own context, revert to its parallel method version, and return its context to its caller.

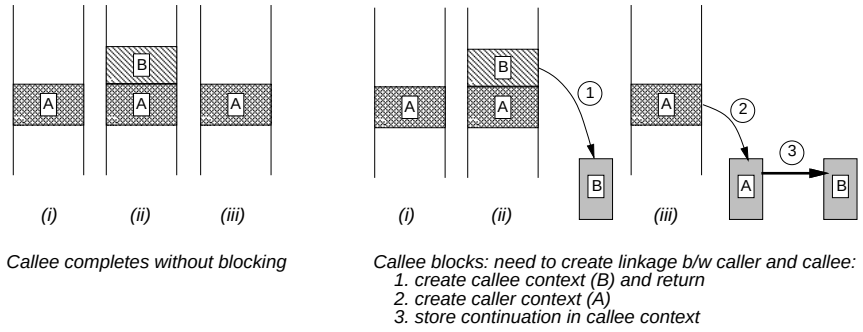


Figure 9.8: May-block Schema Graphic

Figure 9.8 shows an example of the calling schema for the may-block case in action. The figure on the left shows successful completion. In it the local state for thread **B** is stored in a stack frame. Since the thread completes before **A** is rescheduled, the stack frame can be deallocated in normal FIFO order. The figure on the right shows the stack unwinding when the call cannot be completed. In it **B** was allocated on the stack but then blocked. In order to preserve FIFO scheduling order, the stack frame must be flushed to the heap. In this example, **A** also flushes itself to the heap and writes a continuation into the heap context of **B**. If **A** did not require the result of **B** (if the result was to be ignored or forwarded back to **A**'s caller) **A** could fill in the continuation and continue executing off the stack.

Thus, a sequence of may-block method invocations can run to completion on the stack, or unwind off the stack and complete their execution in the heap. The fallback code creates the

[2]Attempting to use `long long` for this purpose in GCC 2.6.3 resulted in unnecessarily inefficient code.

callee's context, saves local state into it, and propagates the fall back by returning this context to its caller which then sets up the linkage.

9.2.3.3 Continuation Passing: Lazy Continuation Creation

Explicit continuation passing (Section 3.2.1.4) can improve the composability of concurrent programs [186, 41]. However, creating continuations and using continuations to return results is expensive. Moreover, continuation passing requires more information to make the linkage. Normally, if an invocation is being executed on the stack, the callee's continuation is implicit. In the may-block case, the callee returns a pointer to its context into which the caller writes the continuation. In the continuation passing case, the callee may require the continuation immediately so that it can pass it on. Furthermore, since one of our goals is to execute forwarded invocations [98] on the stack, lazy allocation of the continuation is essential.

The continuation passing schema (see Figure 9.9) uses an additional parameter, `caller_info`, which, along with the `return_ptr`, encodes the information necessary to determine what to do should the continuation be needed. The `caller_info` field is not used if the callee does not need to manipulate the continuation directly (e.g. store it or pass it off node). In the case of local forwarding the `caller_info` information is simply passed along. If a method needs the continuation the information is used to create it. The `caller_info` indicates whether the context containing the continuation's future has already been created, the context's size if it has not, the location of the return value within the context, and whether the continuation was forwarded. Table 9.2 describes the `caller_info` information in detail.

As with the may-block schema, the result the method invocation is passed back in one of two ways. If the continuation is not needed (i.e. the continuation is not explicitly manipulated), the method invocation result is passed back using the `return_val_ptr`. The method simply writes the result through `return_val_ptr`, and passes NULL return values back to its caller. The caller of the first continuation-passing method (root of the forwarding chain), receives this NULL value and looks in `return_val` for the result. Thus, local continuation passing is executed completely on the stack.

If, on the other hand, the continuation is required, `caller_info` is consulted. There are four cases which are handled by the fallback code. First, if the continuation was initially forwarded, the context must already exist as must the continuation (which is always stored

```
Context* root_method(...,return_val_ptr,...) {
  ...
  caller_context = cont_passing_intermed(&return_val,
                                make_caller_info(root_func),...);
  if (caller_context != NULL) {
    save_state_to_heap(caller_context);
    return caller_context;         // propagate blocking
  }
}

Context* cont_passing_intermed(...,return_val_ptr,caller_info,...) {
  ...
  caller_context = cont_passing_method(return_val_ptr,caller_info,...);
  return caller_context;
}

Context* cont_passing_method(...,return_val_ptr,caller_info,...) {
  ...
  if ( can_return_value ) {
    *return_val_ptr = value;
    return NULL;
  } else {                           // need continuation
    caller_context = create_context_from_caller_info(return_val_ptr,caller_info);
    my_context = create_context();
    my_context->continuation = make_continuation(caller_context, caller_info);
    save_state_to_heap(my_context);
    return caller_context;
  }
}
```

Figure 9.9: Continuation Passing Calling Schema

at a fixed location in heap contexts). The continuation is extracted by subtracting the return location offset in `caller_info` from the `return_val_ptr`, adding on the fixed location offset and dereferencing. Second, if the context already exists but the continuation does not, the continuation is created for a new future at `return_val_ptr`. The future needs to have a reference to the context (in order restart the thread). It computes this reference using the `return_val_ptr` and the the return location offset in `caller_info`. Finally, if the context does not exist, it is created based on the size information from `caller_info`, and the continuation is created for a future at the return value offset. The callee now has the continuation with which it may do what it needs. Figure 9.10 contains pseudo code describing these cases.

context flag	indicates whether or not the heap context to which the result of this method invocation should go has already been created.
forward flag	indicates whether or not the continuation was forwarded from *initial* context pointed to by `return_val_ptr`. **context flag** is always true when **forward flag** is true.
future flag	indicates that the future has already been created. This makes the continuation creation operation to be idempotent, increasing placement flexibility.
count flag	indicates that the continuation is a counting continuation. This allows many continuations to be forwarded on the stack then off-node from a parallel loop.
null flag	indicates that the continuation is null (simply consumes the result).
return location offset	the offset within the heap context where the future for the method invocation result should be created. Along with the `return_val_ptr` can be used to calculate the pointer to the context when **context flag** is true.
method	a descriptor which is used to create the context to which the result will be sent.

Table 9.2: Continuation Passing Caller Information

When the callee completes, it indicates that it required the continuation by passing the continuation's future's context back to its caller. Note that in the case of forwarding, this is *not* the caller's context. When a forwarding invocation returns a non-NULL value, the caller may continue to execute, but must ultimately return the context pointer to its caller regardless of whether or not it completes. This is because the context may be that of its caller which will then fall back to its parallel version.

Figure 9.11 shows a graphical example of the continuation passing schema. The thread **A** (`root_method` from Figure 9.9) is the root of a continuation forwarding chain in which we can imagine that `cont_passing_intermed` is an intermediate function through which the continuation is forwarded to thread **B** (`cont_passing_method`). On the left, normal completion has the `caller_info` and `return_val_ptr` passed as a pair from **A** through to **B** where the `return_val_ptr` is used to return the result directly from **B** to **A**. On the right, **B** requires the continuation, and it must create the context for **A** in order to build it. When **A**'s context pointer is eventually returned to `A`, it stores its state and reverts to its parallel version.

```
make_continuation_passing_continuation( return_val_ptr, caller_info) {
  if (caller_info.forward_flag)
    return extract_continuation(extract_context(return_val_ptr,caller_info));
  else {
    context = NULL;
    if (caller_info.context_flag)
      context = extract_context(return_val_ptr,caller_info));
    else
      context = create_context(caller_info.method);
    return make_continuation(caller_info.count_flag,
                             context,
                             caller_info.return_location_offset);
  }
}
```

Figure 9.10: Pseudo Code for Continuation Creation

9.2.4 Wrapper Functions and Proxy Contexts

Calling the sequential versions of methods from the runtime or a different schema method can require impedance matching, interfacing the available information to the desired interface. For example, when a message arrives at a node it contains a continuation for the result. If the appropriate stack-based schema is non-blocking, a wrapper function is used which calls the sequential version, obtains the result and passes it to the calling method through the continuation. This example is illustrated in Figure 9.12. The result is check to verify that a value was returned (which will not be the case in a purely reactive computation) which is then passed to the waiting future by way of the continuation.

The wrapper function takes either a vector of arguments or a communication buffer and invokes the stack-based version of a method with the appropriate calling convention. In this manner, a remote message can be processed entirely on the stack, and if the continuation is forwarded, it may pass through several nodes, finally respond to the initial caller, all without allocating a heap context.

Figure 9.13 illustrates the case for may-block methods. The result value is checked as in the non-blocking case, and passed through the continuation if available. Furthermore, if the callee blocks, the continuation is placed in the callee's context. Note, both the result value and callee context pointer are checked in any case since the result may or may not be returned independent of whether or not the method blocks.

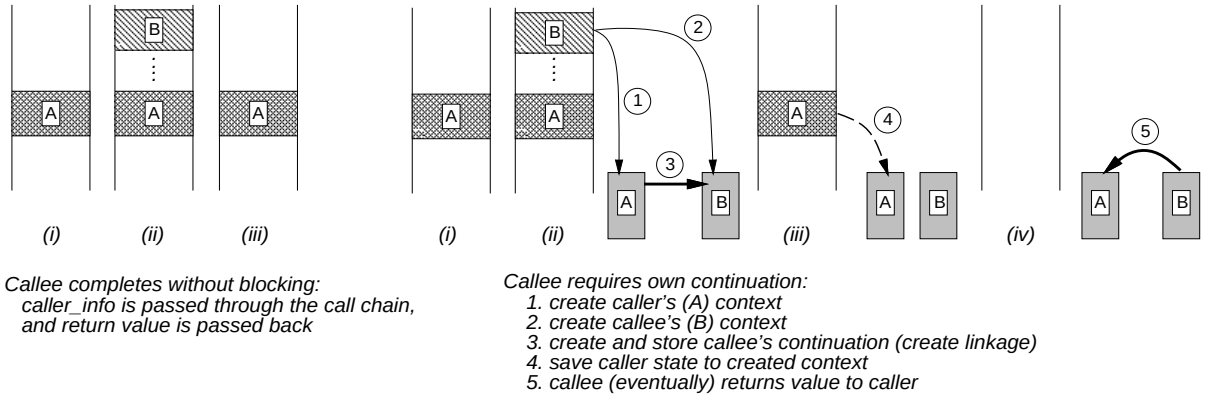


Figure 9.11: Continuation Passing Schema Graphic

```
void non_blocking_msg_wrapper(Slot * buff) { // Non-blocking
  result_val = non_blocking_method(buff[0],...)
  if (!EMPTY(result_val)) reply(buff[CONTINUATION],result_val);
}
```

Figure 9.12: Non-blocking Wrapper

Finally, for continuation passing (Figure 9.14), a proxy context and a `caller_info` parameter are created which indicates that the context exists and that the continuation was forwarded. Thus, if the continuation is required it is extracted from the proxy context (see Section 9.2.3.3). This proxy context technique is also used when an arbitrary continuation (perhaps one stored in a data structure) is passed by a method to another which requires a `return_val` and `caller_info` pair. This can occur with user-defined synchronization structures like barriers.

9.3 Evaluation

This section evaluates the effectiveness of hybrid parallel-sequential execution. First, the performance of hybrid execution for is compared to the performance of straight sequential execution for the same programs written in C. This comparison is made using a variety of synchronization structures. Then, parallel performance is examined by measuring the improvement over straight heap-based execution, and demonstrate the ability of the execution model to adapt

```
void may_blocking_msg_wrapper(Slot * buff) { // May-block
  Slot result_val = EMPTY_SLOT;
  Context * callee_context = may_block_method(&result_val,buff[0],...)
  if (!EMPTY(result_val)) reply(buff[CONTINUATION],result_val);
  if (callee_context != NULL) {
    callee_context->continuation = buff[CONTINUATION];
  }
}
```

Figure 9.13: May-block Wrapper

```
void cont_passing_msg_wrapper(Slot * buff) { // Continuation Passing
  Proxy proxy_context;
  proxy_context.return_val = EMPTY;
  proxy_context.continuation = buff[CONTINUATION];
  Caller_Info caller_info = PROXY_CALLER_INFO;
  Context * caller_context = cont_passing_method(&proxy_context.result_val,...)
  if (!EMPTY(proxy_context.result_val)) reply(buff[CONTINUATION],result_val);
}
```

Figure 9.14: Continuation Passing Wrapper

to different data locality characteristics for both regular and irregular parallel programs. The sequential experiments were conducted on SPARC workstations and the parallel experiments on the CM5 and the T3D (Section 4.3).

9.3.1 Sequential Performance

Sequential performance is evaluated for a set of invocation intensive benchmark programs. Table 9.3 presents the results. The sequential execution times (in seconds) using the hybrid mechanisms are compared with times for a parallel-only version and the original C versions of the programs. The hybrid versions are evaluated under varying degrees of flexibility: *1 interface* uses only the Continuation-passing interface, while *3 interfaces* uses all three interfaces. Finally, *Seq-opt* is a version which eliminates parallelization overheads.

Using the most flexible hybrid version (*3 interfaces*), all programs run significantly faster than than the heap-only versions and achieve close to the performance of a comparable C

Program Description	Parallel Version	Hybrid Parallel-Sequential Versions				C Program
		1 interface	*2 interfaces*	*3 interfaces*	*Seq-opt*	
fib(29)	13.12	1.01	0.95	0.70	0.69	1.10
tak(18,12,6)	152.87	11.37	12.08	6.75	6.71	7.00
qsort(10000)	2.90	0.25	0.28	0.23	0.23	0.16
nqueens(8x8)	3.37	0.77	0.80	0.60	0.56	0.38
list-traversal (128 elements)	1.34	1.05 1.17[a]	0.81 1.00[a]	0.81 0.87[a]	0.81 0.87[a]	0.78

[a]without forwarding optimization supported by Continuation-passing interface.

Table 9.3: Sequential Performance

program despite concurrent semantics.[3] Several programs required all three interfaces to achieve comparable performance. The remaining overhead is due to parallelization. *Seq-opt* shows the performance when this overhead is eliminated.

The parallel and hybrid versions can be run directly on parallel machines. They include parallelization overhead in the form of name translation, locality and concurrency checks (Chapter 8). Since speculative inlining (Section 8.2) is required to obtain good performance from a fine-grained model, it was used as one of the optimizations contributing to these results. Since speculative inlining lowers the overall invocation frequency, it decreases the impact of our hybrid execution model. However, this impact is mitigated by only inlining one level of recursion.

The three hybrid versions provide benefits in different cases. Each level of flexibility enables additional operations to be executed on the stack at some increase on cost. Allowing all three stack interface versions (i.e., the non-blocking, may-block and continuation-passing call schemas) improves performance by up to 30% as compared to when only the most general (continuation passing) interface is always used. The cases where the performance using two interfaces is worse than that using only one interface are an anomaly arises from an improper alignment of invocation arguments causing them to be spilled to stack instead of being passed in registers. In summary, the hybrid mechanism enables C-like performance when data is locally accessible.

[3]The relative performance of `fib` and `tak` is a result of the comparatively aggressive inlining of our compiler.

9.3.2 Parallel Performance

This section considers three application kernels which characterize the parallel performance of the hybrid mechanisms as compared to straight heap-based execution. The performance is characterized with respect to the effect of data locality. First, a regular code, Successive Over Relaxation (SOR), is considered, and the performance improvement resulting from the use of the hybrid is shown to increases with the amount of data locality, reaching close to the theoretical maximum for the given locality. Then two irregular codes, MD-Force and EM3D, are considered, and the ability of the hybrid mechanisms to adapt to available data locality in the presence of irregular computation and communication structures is demonstrated.

9.3.2.1 Regular Parallel Code: SOR

Successive Over Relaxation (SOR) is an indirect method for numerically solving partial differential equations on a grid. The algorithm evaluates the new value of a grid point according to a 5-point stencil and consists of two half-iterations: in the first half-iteration a new value is computed for each grid point, and in the second half-iteration, the grid point is updated with the computed value. To characterize the impact of data locality, the grid size is fixed (1024×1024) and various block sizes are considered for a block-cyclic distribution of the grid on an 8×8 grid of processors. These different data layouts result in different ratios of local to remote method invocations and correspond to differing amounts of data locality.

Data Locality		CM5 Performance			T3D Performance		
Block Size	*Local vs Remote*	*Parallel (secs)*	*Hybrid (secs)*	*Parallel/ Hybrid*	*Parallel (secs)*	*Hybrid) (secs)*	*Parallel/ Hybrid*
8×8	0.083:1	135.58	136.85	0.991	48.99	43.50	1.126
16×16	1.167:1	97.16	88.15	1.102	46.77	28.66	1.632
32×32	3.333:1	83.47	52.15	1.601	43.46	20.70	2.099
64×64	7.667:1	60.33	32.43	1.860	34.94	14.97	2.334
128×128	16.333:1	45.80	19.89	2.303	28.46	12.00	2.372

Table 9.4: Execution Results: SOR

Table 9.4 shows the performance of the hybrid mechanisms on 64-node configurations of the CM5 and T3D for five choices of the block size. The parallel execution times are for SOR on a 1024×1024 grid at 100 iterations. The CM5 and T3D both used 64-node configurations.

The performance of hybrid mechanisms is compared with a parallel-only version for varying amounts of data locality (indicated by *Block Size*) for a block-cyclic distribution of the grid. *Local vs Remote* gives the ratio of local to remote method invocations for the given block size.

The results in Table 9.4 show that improvement from hybrid execution is proportional to data locality. The speedup of hybrid mechanisms over the parallel version increases from ~1.0 when the fraction of local invocations is 0.077 to ~2.4 when the fraction of local invocations is 0.942. These numbers are in the neighborhood of the theoretical peak values which are determined by the relative costs of useful work, invocation overhead and remote communication. For example, factoring out the useful work in the 128×128 SOR block layout on the CM5, the maximum possible speedup is 2.63 given that, on average, a remote invocation incurs 10 times the cost of a local heap invocation. The measured value of 2.3 approaches this maximum, indicating that hybrid execution efficiently adapts to available locality.

9.3.2.2 Irregular Parallel Code: MD-Force

MD-Force is the kernel of the nonbonded force computation phase of a molecular dynamics simulation of proteins [91]. The computation iterates over a set of atom pairs that are within a spatial cutoff radius. Each iteration updates the force fields of neighboring atoms using their current coordinates, resulting in irregular data access patterns because of the spatial nature of data sharing. The implementation reduces the communication demands of the kernel by caching the coordinates of remote atoms and combining force increments.

Data Locality		CM5 Performance			T3D Performance		
Data Layout	*Local vs Remote*	*Parallel (secs)*	*Hybrid (secs)*	*Parallel/ Hybrid*	*Parallel (secs)*	*Hybrid (secs)*	*Parallel/ Hybrid*
Random	0.38:1	10.71	10.41	1.03	3.94	3.82	1.03
Block	6.05:1	1.46	1.02	1.43	1.32	0.87	1.52

Table 9.5: Execution Results: MD-Force

Table 9.5 shows the performance of the MD-Force kernel Parallel execution times for MD-Force kernel (10503 atoms for 1 iteration) on 64-node configurations of the CM5 and T3D. The performance of the hybrid mechanisms is compared with a parallel-only version for low-locality random and high-locality data layouts. The *random* layout uniformly distributes atoms on the

nodes, ignoring the spatial distribution of atoms. The *spatial* layout uses orthogonal recursive bisection to group together spatially proximate atoms.

Because of poor locality the execution time for the *random* distribution is communication overhead dominated. Since communication costs remain unchanged by the choice of the invocation mechanisms the speedup is 1.03 in this case. For the spatially blocked distribution, the hybrid mechanisms enable the computation to adapt dynamically to data locality, yielding speedups of 1.43 and 1.52 on the CM5 and T3D respectively.

As with SOR, when run time checks determine that both atoms of an atom pair are local and the computation is small it is speculatively inlined. When an atom is found to be remote but its coordinates are in the cache, the computation completes on the stack without incurring parallel invocation overhead. When communication is required, and the stack invocation falls back to the parallel version to enable multithreading for latency tolerance. Furthermore, hybrid execution provide a performance improvement for communication because the invoked method can execute directly from the message handler. Thus, hybrid execution adapts to the data layout and synchronization structures of the program.

9.3.2.3 Irregular Parallel Code: EM3D

EM3D is an application kernel which models propagation of electromagnetic waves [70]. The data structure is a graph containing nodes for the electric field and for the magnetic field with edges between nodes of different types. A simple linear function is computed at each node based on the values carried along the edges. Three versions of the EM3D code are used to evaluate the ability of the hybrid model to adapt to different communication and synchronization structures. Since they are intended to examine invocation mechanisms, elaborate communication blocking mechanisms are not used. The first version, `pull`, reads values directly from remote nodes. The second version, `push`, writes values to the computing node, updating from the remote nodes each timestep. Finally, in the `forward` version, the updates were done by forwarding a single message through the nodes requiring the update.

Table 9.6 describes the performance of the three versions of EM3D on a 64-node CM5 and a 16-node T3D. The parallel execution times are reported for 8192 nodes of degree 16 for 100 iterations. The performance for three versions of the algorithm using the hybrid mechanisms is

Data Locality		CM5 Performance			T3D Performance		
Algorithm	*Local vs Remote*	*Parallel (secs)*	*Hybrid (secs)*	*Parallel/ Hybrid*	*Parallel (secs)*	*Hybrid (secs)*	*Parallel/ Hybrid*
EM3D *pull*	0.0156:1	93.93	68.94	1.362	349.04	336.32	1.037
	99:1	7.42	3.34	2.222	29.681	25.96	1.148
EM3D *push*	0.0156:1	543.73	145.36	3.741	494.85	473.59	1.045
	99:1	11.76	10.96	1.073	41.27	29.47	1.400
EM3D *forward*	0.0156:1	180.40	181.6	0.993	602.41	433.65	1.389
	99:1	18.86	15.53	1.214	112.79	39.41	2.262

Table 9.6: Execution Results: EM3D

compared with parallel-only versions for random node placement with low locality (0.0156:1) and placement with high locality (99:1).

The results show that the hybrid scheme is capable of improving performance for different communication and synchronization structures for both cases of high and low data locality. For low locality, efficiency is increased because off-node requests are handled directly from the message buffer, without requiring the allocation of a heap context. When locality is high, hybrid execution executes local portions of the computation entirely on the stack. Hybrid execution yields speedups ranging from unity to nearly four times, achieving superior performance in all but one case where the continuation passing schema is used with extremely low locality on the CM5. In addition to the cost of fallback, this combination produces the worst case for our scheduler on the CM5.

Overall, the *pull* version provides the best absolute performance since it computes directly from the values it retrieves rather than using intermediate storage. The *forward* version requires longer update messages than *push* but fewer replies. On the CM5 replies are inexpensive (a single packet), so the cost of *forward*'s longer messages overwhelms the cost of the larger number of replies required by *push*. However, on the T3D the decrease in overall message count enables *forward* to perform better than *push* for low locality. Also, the CM5 compiler performs better on the unstructured output of our compiler than the T3D compiler. As a result, the cost of the additional operations required by *push* and *forward* has less of an impact on the T3D than messaging overhead. Thus, for high locality, the hybrid mechanism is most beneficial for *pull* on the CM5 and for *forward* on the T3D where local computation and messaging dominate respectively.

9.4 Related Work

Several languages supporting explicit futures on shared-memory machines have focused on restricting concurrency for efficiency [87, 117]. However, unlike our programming model where almost all values define futures, futures occur less frequently in these systems, decreasing the importance of their optimization. More recently, lazy task creation [132], leapfrogging [180] and other schemes [104, 18] have addressed load balancing problems resulting from serialization by stealing work from previously deferred stack frames. However, none of these approaches deals with locality constraints arising from data placements and local address spaces on distributed memory machines.

Several recent thread management systems have been targeted to distributed memory machines. Two of them, Olden [146] and Stacklets [77] use the same mechanism for work generation and communication. Furthermore, they require specialized calling conventions limiting their portability. StackThreads [172] used by ABCL/f has a portable implementation. However, this system also uses a single calling convention, and allocates futures separate from the context. Thus, an additional memory reference is required to touch futures. Also, its single version of each method cannot be fully optimized for both parallel and sequential execution. The HAL system provides specialized calling conventions for different synchronization idioms [115]. However, it is not adaptive and does not provide seperate sequential and parallel versions, concentrating instead on the efficiency of individual operations.

9.5 Summary

Irregular and dynamic programs require an execution model which can adapt to the run time computation and communication structure. Also, adaptation is critical for supporting runtime techniques such as data and function shipping. Hybrid execution uses separately optimized sequential and parallel versions of various levels of flexibility and efficiency to adapt at run time to data layout and computational structure. Sequential efficiency can be achieved for COOP codes by dynamically aggregating the many small threads into larger threads executed in LIFO fashion using a convention stack. Parallel efficiency is achieved by generating parallelism and hiding latency by creating threads using heap-based contexts. Hybrid execution uses specialized parallel and sequential versions of each method and four distinct invocation schemas of varying

complexity and cost to efficiently. For call intensive kernels, it is demonstrated that hybrid execution is very nearly as efficient as straight sequential C code. Furthermore, the different calling schemas contribute to this efficiency. Hybrid execution approaches optimal efficiency for regular problems and effectively adapt to irregular problems with complex communication and synchronization structures. It is demonstrated on the CM5 and T3D that hybrid execution provides performance benefits proportional to the amount of data locality. Hybrid execution provides both sequential efficiency when data is available and parallel efficiency for work generation and latency hiding.

9.6 Acknowledgments

The SOR and MD-Force source code, numbers and results analysis in this chapter are the work of Vijay Karamcheti and Xingbin Zhang respectively. They appear here to illustrate the effectiveness of hybrid execution on a wider range of programs.

Chapter 10

Conclusions

OOP and COOP can be made efficient through interprocedural analysis and transformation.

Thesis

The proof of this thesis has been a long process, spread over many years and involving many parts, the utility of which are not necessarily obvious in a vacuum. The goal of the Concert project is ambitious: to start with a a highly abstract programming model and build a programming system to rival, in absolute efficiency, state of the art systems for conventional languages with decades of high performance tradition. My part of that goal was sequential efficiency; that is, computational efficiency discounting load balance and the availability of data, two enormous areas of research in themselves. This part alone required developing new analysis and transformation techniques which proved useful for tackling the other problems as well. Furthermore, because we chose absolute efficiency as our goal, many small optimizations for common boundary cases were required, as well as a host of standard transformations, available in any optimizing compiler.

Building a complete system was extremely satisfying in the sense that I am confident that we have explored a good part of the problem space and developed an understanding far beyond the sterile and often deceptive lines of theory. But it has been disturbing as well. Much of what we have learned is that there is no panacea, no quick fix; that a workable solution requires many pieces which depend strongly on the capabilities of the others. Very simple programming systems of immense power and efficiency can be constructed, but no short cuts are possible, and the solution is not incremental. It requires rethinking of the interaction between language and

implementation and reformulation of the compilation process. This translation of viewpoint does not fit easily with either the academic nor commercial mindsets. It does not divulge its secrets to simplified problems or yield to incremental solutions. The work presented here only begins to describe what will ultimately be required to fully realize simple, powerful, parallel programming systems. It will be years, and probably decades before such systems become common.

10.1 Thesis Summary

Object-oriented programming (OOP) is the byword of software engineering; promising to increase productivity through abstraction and software reuse. Concurrent object-oriented programming (COOP) applies those tools to parallelism and distribution, with applications from supercomputers to web browsers. These technologies produce programs with very different structures than standard procedural codes, but with the same high demands for efficiency. However, localized compilation techniques are poorly suited to the dynamic nature of OOP and COOP codes. The abstractions which free programmers from implementation details, hide information from the compiler, resulting in conservative implementations and poor performance.

Through interprocedural analysis and transformation, specialized implementations of these abstractions can be constructed so that OOP and COOP can be as efficient as conventional procedural programming. I have developed a compilation framework using novel optimization techniques for context-sensitive analysis and specialization of abstractions. This framework maps flexible dynamic programs to efficient static implementations. For standard benchmarks, including the Stanford Integer, Richards, and Delta-Blue benchmarks and the Livermore Loops, these techniques produce implementations as efficient as those written in C and more efficient than those written in C++ and compiled with standard compilers.

The framework starts with a new interprocedural context-sensitive flow analysis which breaks through abstraction barriers. Classes and functions are then cloned for the contexts in which they are used. An iterative cloning algorithm rebuilds the cloned call graph using a modified dispatch mechanism. Using the information made static by cloning, classes and functions are specialized: member (instance) and local variables are unboxed, virtual functions (methods) are statically bound and inlining is performed speculatively based on the class of

objects or function pointers for OOP and location or lock availability for COOP. Other optimizations include promotion of member and global variables to locals, lifting and merging of access regions, removal of redundant lock and locality check operations, and redundant array operation removal. Finally, the program is mapped to a new hybrid stack and heap-based execution model suitable for distributed memory multicomputers.

The general contribution of this thesis is an optimization framework for object-oriented and fine-grained concurrent languages. Individual contributions are: 1) a new iterative flow analysis for the analysis of object-oriented programs which is both practical and more precise than previous analyses, 2) a new cloning algorithm which extends the state of the art in cloning to general flow problems and object-oriented programs, 3) a set of novel optimizations for removing object-oriented and fine-grained concurrency overhead, and 4) a new hybrid stack-heap execution model which provides two separately optimized versions of code enabling it to adapt to the location of data.

10.2 Final Thoughts

While the work described in this thesis forms a cohesive whole, it is by no means the final closed solution. There is a great deal left to be done; foremost perhaps in the area of interprocedural analysis and transformation. Adaptive analysis techniques with time bounds for particular program structures need to be developed. These must be able to handle the problem of *structure analysis*, the determination of the structure of imperatively manipulated pointer-based data structures directly from the program text [143]. The structure analysis problem subsumes alias analysis, an outstanding problem of great complexity and importance [120, 118, 44, 60, 182, 147].

Transformation must go farther to break the boundaries between compilation, calculation, communication and data. Translation between these different modes of computation should be in the providence of the programming system. For example, data can be migrated, cached, recomputed, and encoded in control flow. Communication can be similarly be transformed, mapped into control flow, cached, replaced with one of the possible results indicated by non-determism. Calculations can be replaced by partially computed, inferred and speculated values. Finally, run time compilation can blur the boundary between static and dynamic, data and code.

Non-determinism needs to be recognized as an essential and desirable property of algorithmic description; a tool for describing a range acceptable behavior. In contrast, deterministic solutions are overspecified, and have no physical analogue in the real world where simultaneity is a reality. Non-determinism is an enormously powerful transformational tool, enabling the implementation to select a path to the answer based on efficiency. Such tradeoffs are organic and natural for the vast majority of computer users. As computing emerges from the cold and brittle world of mechanical logics, into the warmth of art and information age, the nature of computation will change.

On a similar note, consistency models, which have heretofore been relegated to the mechanics of hardware cache consistency, need to be expanded to the level of the programming model. We live in a world where performance is limited by the ability of the system to deliver and maintain the consistency of data; where distribution and simultaneity have obliterated the notions of absolute time and absolute location; and where facts are relative. Internal consistency for a given system, encapsulation of assumptions, predicated answers and answers as of a given time are all common in the natural systems with which the computing fabric is ever more intertwined. The science of computation must embrace these concepts because they are the future of computing.

With the recent explosion of interest in network computing, the world wide web and Java, is clear that at its core, computing is parallel and distributed. It is also clear that the old fork-join-semaphore model has become hopelessly dated. The rise of portable executable formats, just-in-time and whole-program compilation has upset the traditional view of compilers. This area, which had stagnated under endless discussions of "yet another parsing algorithm" and an obsession with speeding translation through clever coding schemes, has been given new life. Unfortuntely, the current climate of compulsive paper chase discourages revolutionary developments and deep thought in general. For this we suffer as the same old tired ideas, cast in new terms and superficially evaluated, clog the literature, numbing minds and wearying spirits. In this age where time-to-market is king, industry, ever with all its shortsightedness, has surpassed academia for innovation and progressiveness. The ivory towers, like the castles of provincial lords at the close of feudal times, are threatened by progress and the rising power of the merchant class. It is my hope that academia can rise to the challange and, using the

towers' height and relative peace for far sight and contemplation, to refine, perfect and lay the groundwork for the next revolution. I hope this thesis makes a contribution to that end.

And what is writ is writ,–
Would it were worthier! Lord Byron

Appendix A

Annotations

In order to instruct the compiler and runtime with information about the placement and accessibility of objects, the Concert system supports, among others, locality and locking annotations [43]. These annotations are not meant to change the meaning of the program, but rather to enable the programmer supply information which might not be easily accessible to the compiler and/or runtime, and to enable the compiler writer to test the usefulness of particular pieces of information.

```
a = new(LOCAL) A;
b = new(LOCAL) B;
c = new C;
a.x = b;
d = flag ? a : b;
e = flag ? a : c;
if (cond) (e $local $nolock)->foo;
```

Figure A.1: Annotations Example

Figure A.1 shows a block of code which uses annotations and allocation directives. Here, `a` and `b` are both allocated local to the current object (`self`). As a result, both `a` and `b` are relatively local to each other and to `self`. Conservatively, we can then infer that `d` will be relatively local to `self` as well. We may not infer that `e` is relatively local to `self` since `c` may be allocated on any node. However, we can infer that `a.x` will be relatively local to `a` (so long as this is the only assignment to `x`. Finally, in the last line, the conditional `cond` is evaluate,

and the compiler is directed that `e` is local and no lock is required to access it (a situation implied by `cond`).

```
b = a $local;
if (...)
  d = c $local;
else
  e = c $local;
```

Figure A.2: Annotations Propagation Example

Annotations propagation is a data flow problem. Annotations are propagated along the local data flow arcs using a conservative merge. When a reference to a local object merges with a reference to a remote object, the resulting reference is to a remote object. Since the annotation specify a property of a value, they flow backward as well. If an annotation has been placed on every value into which another value may flow, the original value must have that property as well. For example, in Figure A.2 `a` and `b` are subject to the same conditions of execution. Therefore, `b` must be local in the block containing `a`. Likewise, `d` and `e` are annotated to be local. Since one or the other branch must be executed, it must be true that `c` is local in the surrounding block. Care must be taken when using data or function migration. For example, if the unspecified condition in Figure A.2 moved the object `c` so that it was local, `c` need not be local before the conditional. In such cases, annotation should not be propagated across operations which can change the properties being annotated.

Appendix B

Raw Flow Analysis Data

The results of analysis for the three algorithms on a variety of complete, kernel and synthetic programs appear in Table B.1. IFA refers to our incremental inference algorithm, OPS refers to the inference algorithm in [135], and 0CFA refers to a standard flow insensitive algorithm which allocates exactly one type variable per static program variable.

The number of **Passes** is determined by the algorithms automatically when it determines that no run time type errors are possible. **Nodes** is the number of flow graph nodes used by the algorithm. **Invokes** is the number of invocations (abstract calls) analyzed. **Contours** is the number of contours. In 0CFA this corresponds to the number of methods. A program can be **Typed?** by an algorithm if it can prove an absence of run time type errors. **Checks** is the number of type checks which must be made to ensure such an absence. The number of imprecisions *Im* indicates number of nodes which were not resolved to a singleon value. The implementation is approximately 2600 lines of largely unoptimized Common Lisp/CLOS and **Time** in seconds is reported for CMU Common Lisp/PCL on a Sparc10/31.

Program	*Lines*	*Passes*	*Nodes*	*Invokes*	*Contours*	*Typed?*	*Checks*	*Im*	*Time*
IFA									
ion	1934	5	50779	3470	760	YES	0	0	713.70
network	1799	3	29090	2228	730	YES	0	31	234.15
circuit	1247	6	34505	1801	430	YES	0	7	289.52
pic	759	6	40284	2128	357	YES	0	0	363.18
mandel	642	1	17257	1011	442	YES	0	0	25.48
tsp	500	3	10290	627	207	YES	0	0	56.24
mmult	139	7	11518	543	147	YES	0	0	78.35
poly	41	4	3819	234	90	YES	0	0	18.12
test	39	7	1581	130	76	YES	0	0	15.11
OPS									
ion	1934	1	115800	7098	2817	NO	19	264	577.51
network	1799	1	73864	6018	2296	YES	0	87	357.47
circuit	1247	1	49849	2646	1097	NO	44	679	136.03
pic	759	1	48420	2783	1068	NO	28	196	144.16
mandel	642	1	26280	1442	562	YES	0	0	60.78
tsp	500	1	18203	1150	472	NO	2	31	40.78
mmult	139	1	10928	595	216	NO	4	104	22.36
poly	41	1	4233	250	137	YES	0	0	8.25
test	39	1	1353	123	100	NO	2	0	2.94
0CFA									
ion	1934	1	34729	3380	396	NO	260	1096	131.16
network	1799	1	18874	1804	407	NO	132	926	58.77
circuit	1247	1	15491	976	190	NO	111	405	28.93
pic	759	1	16065	1300	180	NO	119	390	37.68
mandel	642	1	8755	760	116	NO	59	524	16.52
tsp	500	1	7006	571	130	NO	27	225	15.79
mmult	139	1	3842	231	61	NO	4	89	7.60
poly	41	1	1848	138	48	NO	4	55	3.84
test	39	1	1001	108	44	NO	2	19	2.92

Table B.1: Results of Iterative Flow Analysis

Appendix C

Raw OOP Data

benchmark	CA baseline	CA no arrayalias	CA no standard	CA no instvars
bubble	10.224	10.22	10.23	10.23
intmm	8.12	8.12	17.74	17.74
perm	8.70	8.70	8.89	8.89
puzzle	128.11	128.59	188.15	188.12
queens	10.10	10.10	10.71	10.71
sieve	2.47	2.44	2.45	2.44
towers	12.91	13.78	16.60	16.60
trees	64.85	65.12	91.10	90.40

Table C.1: Raw Data for Stanford Integer Benchmarks

benchmark	CA no inlining	CA no cloning	CA no analysis	C inline	C -O2
bubble	67.07	135.10	135.12	11.19	11.13
intmm	43.45	60.70	487.15	11.12	10.57
perm	34.78	64.47	64.45	12.50	9.46
puzzle	393.60	614.04	6567.59	72.81	64.35
queens	19.25	34.79	34.88	8.37	8.27
sieve	6.09	11.18	57.34	2.59	2.63
towers	56.20	81.618	86.34	19.59	13.01
trees	164.13	480.98	986.44	141.09	125.66

Table C.2: Raw Data for Stanford Integer Benchmarks (cont)

benchmark	CA base	CA no arrayalias	CA no standard	CA no instvars
Projection	66.98	66.02	64.64	64.30
Chain	52.09	52.67	129.22	128.34
Richards	57.72	67.68	223.28	223.37

Table C.3: Raw Data for Stanford OOP Benchmarks)

benchmark	CA no inlining	CA no cloning	CA no analysis	C++ inline	C++ -O2
Projection	232.84	617.65	1599.78	318.6	323.7
Chain	402.00	1320.59	3320.34	350.7	354.1
Richards	795.70	4085.57	7153.84	121.5	125.9

Table C.4: Raw Data for Stanford OOP Benchmarks (cont))

benchmark	C++ no inlining	C++ all virtual	C++ no inlining all virtual
Projection	486.9	586.6	598.4
Chain	491.0	652.5	649.1
Richards	207.9	124.6	207.1

Table C.5: Raw Data for Stanford OOP Benchmarks (cont cont))

Appendix D

CA Standard Prologue

```
;;  Standard Prologue for Concurrent Aggregates
;;  "Builtin" Classes must be declared early as they are used to define
;;  constants.

;;  ROOTCLASS : everything inherits from this

(class rootclass)
(method rootclass primitive ())            ;; *handled internally*
(method rootclass new () :no_exclusion)
(method rootclass new_local () :no_exclusion)
(method rootclass eq (a) :no_exclusion
  (reply (primitive root_equal self a)))
(method rootclass neq (a) :no_exclusion
  (reply (primitive root_not_equal self a)))
(method rootclass assert_class (asserted_class)
  (reply (primitive assert_class self asserted_class)))
(method rootclass check_type (v)
  (seq (primitive check_type v self) (reply done)))
(method rootclass check_null (v)
  (seq (primitive check_null v self) (reply done)))
(function rootclass break () :no_exclusion
  (let ((con (global console)))
    (if (eq self con)
        (reply (primitive debugger_break self))
      (reply (primitive debugger_break con)))))
(function rootclass numproc () :no_exclusion
          (reply (primitive num_proc)))
(function rootclass procid () :no_exclusion
          (reply (primitive proc_id)))
(function rootclass if_local (obj) :no_exclusion
  (reply (primitive check_if_local obj)))

;;  CONTINUATIONCLASS
```

```
(class continuationclass (rootclass))
(method continuationclass reply (value))   ;; *handled internally*

;;  ROOTAGG : all aggregates inherit from this

(aggregate rootagg (rootclass) group groupsize myindex physize
  (parameters sz)
  (initial sz))
;; unoptimized
(method rootagg sibling (i) :no_exclusion :unoptimized
  (if (bounds_check i groupsize)
      (reply (primitive aggregate_to_representative self i))
    (SIBLING_BOUNDS_ERROR (global console) self groupsize i)))
;; optimized
(method rootagg sibling (i) :no_exclusion :optimized
   (reply (primitive aggregate_to_representative self i)))

(method rootagg physical_sibling (i) :no_exclusion
        (reply (primitive agg_to_physical_rep self i)))
(method rootagg physical_groupsize () :no_exclusion
   (reply (primitive physical_groupsize self)))
;; only works on local objects
(method rootagg if_logical () :no_exclusion
   (reply (primitive check_if_logical self)))

;;  NULLCLASS

(class nullclass (rootclass))
(method nullclass eq (a) :no_exclusion
  (reply (primitive null_equal self a)))
(method nullclass neq (a) :no_exclusion
  (reply (primitive null_not_equal self a)))

;; unoptimized
(method nullclass :rest ()  :unoptimized
  (NIL_MESSAGE (global console)))
;; optimized
(method nullclass :rest ()  :optimized )

;;  OSYSTEM

(class osystem (rootclass))

;;  GLOBALCLASS
```

```
(class globalclass (rootclass) state)
(method globalclass global ()
  (reply (state self)))
(method globalclass set_global (val)
  (seq (set_state self val)
    (reply (state self))))

;;  STRING_CONSTANT

(class string_constant (rootclass))

;;  SELECTOR

(class selector (string_constant))

;;  INTEGER

(class integer (rootclass))
(method integer + (i) :no_exclusion
  (reply (primitive integer_add self i)))
(method integer - (i) :no_exclusion
  (reply (primitive integer_subtract self i)))
(method integer * (i) :no_exclusion
  (reply (primitive integer_multiply self i)))
(method integer / (i) :no_exclusion
  (reply (primitive integer_divide self i)))
(method integer > (i) :no_exclusion
  (reply (primitive integer_greater_than self i)))
(method integer < (i) :no_exclusion
  (reply (primitive integer_less_than self i)))
(method integer >= (i) :no_exclusion
  (reply (primitive integer_greater_than_or_equal_to self i)))
(method integer <= (i) :no_exclusion
  (reply (primitive integer_less_than_or_equal_to self i)))
(method integer = (i) :no_exclusion
  (reply (primitive integer_equal_to self i)))
(method integer != (i) :no_exclusion
  (reply (primitive integer_not_equal_to self i)))
(method integer mod (i) :no_exclusion
  (reply (primitive integer_modulo self i)))
(method integer lshift (i) :no_exclusion
  (reply (primitive integer_left_shift self i)))
(method integer rshift (i) :no_exclusion
  (reply (primitive integer_right_shift self i)))
(method integer and (i) :no_exclusion
  (reply (primitive integer_bitwise_and self i)))
```

```
(method integer or (i) :no_exclusion
  (reply (primitive integer_bitwise_or self i)))
(method integer not () :no_exclusion
  (reply (primitive integer_not self)))
(method integer int2float () :no_exclusion
  (reply (primitive integer_to_float self)))
(method integer min (i) :no_exclusion
  (if (> self i) (reply i)
     (reply self)))
(method integer max (i) :no_exclusion
  (if (> self i) (reply self)
    (reply i)))
(method integer ash (i) :no_exclusion
  (if (>= i 0) (reply (lshift self i))
    (reply (rshift self (- 0 i)))))
(method integer +1 () :no_exclusion
  (reply (+ self 1)))
(method integer bounds_check (size) :no_exclusion
  (reply (and (>= self 0) (< self size))))

;;  FLOAT

(class float (rootclass))
(method float + (f)  :no_exclusion
  (reply (primitive float_add self f)))
(method float - (f)  :no_exclusion
  (reply (primitive float_subtract self f)))
(method float * (f)  :no_exclusion
  (reply (primitive float_multiply self f)))
(method float / (f) :no_exclusion
  (reply (primitive float_divide self f)))
(method float > (f) :no_exclusion
  (reply (primitive float_greater_than self f)))
(method float < (f) :no_exclusion
  (reply (primitive float_less_than self f)))
(method float = (f) :no_exclusion
  (reply (primitive float_equal_to self f)))
(method float != (f) :no_exclusion
  (reply (primitive float_not_equal_to self f)))
(method float sin () :no_exclusion
  (reply (primitive float_sin self)))
(method float cos () :no_exclusion
  (reply (primitive float_cos self)))
(method float tan () :no_exclusion
  (reply (primitive float_tan self)))
(method float sqrt () :no_exclusion
```

```
  (reply (primitive float_square_root self)))
(method float exp () :no_exclusion
  (reply (primitive float_exp self)))
(method float pow (f) :no_exclusion
  (reply (primitive float_pow self f)))
(method float log () :no_exclusion
  (reply (primitive float_log self)))
(method float asin () :no_exclusion
  (reply (primitive float_arc_sin self)))
(method float acos () :no_exclusion
  (reply (primitive float_arc_cos self)))
(method float atan () :no_exclusion
  (reply (primitive float_arc_tan self)))
(method float atan2 (f) :no_exclusion
  (reply (primitive float_arc_tan2 self f)))
(method float ceil () :no_exclusion
  (reply (primitive float_ceil self)))
(method float floor () :no_exclusion
  (reply (primitive float_floor self)))
(method float float2int () :no_exclusion
  (reply (primitive float_to_integer self)))
(method float min (i) :no_exclusion
  (if (> self i) (reply i)
    (reply self)))
(method float max (i) :no_exclusion
  (if (> self i) (reply self)
    (reply i)))

;;  ARRAY

(class array (rootclass) size)
;; unoptimized versions
(method array at (index) :unoptimized
  (if (bounds_check index (size self))
      (reply (primitive array_at self index))
    (ARRAY_BOUNDS_ERROR (global console))))
(method array put_at (val index) :unoptimized
  (if (bounds_check index (size self))
      (reply (primitive array_put_at self val index))
    (ARRAY_BOUNDS_ERROR (global console))))
(method array putat (val index) :unoptimized
  (if (bounds_check index (size self))
      (reply (primitive array_put_at self val index))
    (ARRAY_BOUNDS_ERROR (global console))))
;; optimized versions with no bounds check (to match FORTRAN and C)
(method array at (index) :optimized
```

```
        (reply (primitive array_at self index)))
(method array put_at (val index) :optimized
        (reply (primitive array_put_at self val index)))
(method array putat (val index) :optimized
        (reply (primitive array_put_at self val index)))

;; included for backward compatibility
(method array atput (val index)
  (reply (put_at self val index)))

;;  MESSAGECLASS

(class messageclass (array) selector receiver continuation  :no_reader_writer)
(method messageclass msg_at (pos) :no_exclusion
  (reply (primitive message_at self pos)))
(method messageclass msg_putat (val pos) :no_exclusion
  (reply (primitive message_put_at self val pos)))
(method messageclass msg_atput (val pos) :no_exclusion
  (reply (msg_putat self val pos)))
(method messageclass send () :no_exclusion
  (forward (primitive message_send self)))
(method messageclass send_to (to) :no_exclusion
  (forward (primitive message_send_to self to)))
(method messageclass get_requester () :no_exclusion
  (reply (primitive message_requester self)))
(method messageclass set_requester (to) :no_exclusion
  (reply (primitive message_set_requester self to)))
(method messageclass get_receiver () :no_exclusion
  (reply (primitive message_receiver self)))
(method messageclass set_receiver (to) :no_exclusion
  (reply (primitive message_set_receiver self to)))
(method messageclass get_selector () :no_exclusion
  (reply (primitive message_selector self)))
(method messageclass set_selector (to) :no_exclusion
  (reply (primitive message_set_selector self to)))

;;  RAW_ARRAY

(class raw_array (array))

;;  STRING

(class string (raw_array)
       (parameters isize)
       (initial (initial super isize)))
```

```
;;  STREAMCLASS

(class streamclass (raw_array) id status
   (parameters name)
   (initial (set_id self (primitive open_file name self))))
(method streamclass fileopen (name) :no_exclusion
  (reply (new streamclass 520 name)))   ; 512 bytes + NULL

(method streamclass id () :no_exclusion
  (reply id))
(method streamclass close () :no_exclusion
  (forward (primitive close_file self)))
(method streamclass read_int () :no_exclusion
  (forward (primitive read_integer self)))
(method streamclass read_float () :no_exclusion
  (forward (primitive read_float self)))
(method streamclass write_int (i) :no_exclusion
  (forward (primitive write_integer self i)))
(method streamclass write_float (f) :no_exclusion
  (forward (primitive write_float self f)))
(method streamclass write_string (s) :no_exclusion
  (forward (primitive write_string self s)))
(method streamclass write_object (o) :no_exclusion
  (forward (primitive write_object self o)))
(method streamclass end_of_file () :no_exclusion
  (forward (primitive end_of_file self)))

;;  CONSOLECLASS

(class consoleclass (streamclass)
  (parameters name)
  (initial (initial_streamclass super name)))

(method consoleclass id () :no_exclusion
  (reply id))
(method consoleclass :rest () :no_exclusion
        (forward (primitive write_message self msg)))

;; INSTR_COUNTER_AGG (and auxiliary classes)

(function rootclass increment (instr_counter_obj)
  (reply
    (primitive increment_instr_counter
               (storage_obj (sibling instr_counter_obj (procid))) 1)))
(function rootclass increment_by (instr_counter_obj value)
  (reply
```

```
    (primitive increment_instr_counter
               (storage_obj (sibling instr_counter_obj (procid))) value)))
(function rootclass read_count(instr_counter_obj)
  (reply (read instr_counter_obj)))
(function rootclass read_local_count(instr_counter_obj node_id)
  (reply (read_local (sibling instr_counter_obj node_id))))

(aggregate instr_counter_agg summary_obj storage_obj
  (parameters isize summary storage_size)
  (initial isize
    (forall i from 0 below groupsize
       (init (sibling self i) summary storage_size))))
(handler instr_counter_agg create ()
  (let* ((summary_obj (create instr_summary_class))
         (agg_obj (new instr_counter_agg (numproc) summary_obj 32)))
    (reply agg_obj)))
(handler instr_counter_agg init (summary size)
  (seq
    (set_summary_obj self summary)
    (set_storage_obj self (new_local array size))
    (primitive init_instr_counter (storage_obj self))
    (reply DONE)))
(handler instr_counter_agg reset () :no_exclusion
  (seq
    (forall i from 0 below groupsize
      (reset_myself (sibling self i)))
    (reset (summary_obj self))
    (reply DONE)))
(handler instr_counter_agg reset_myself () :no_exclusion
  (seq
    (primitive reset_instr_counter (storage_obj self))
    (reply DONE)))
(handler instr_counter_agg read () :no_exclusion
  (let ((count 0))
    (seq
      (forall i from 0 below groupsize
        (set! count (+ count (read_local (sibling self i)))))
      (reply count))))
(handler instr_counter_agg read_local () :no_exclusion
  (reply (primitive read_instr_counter (storage_obj self))))
(handler instr_counter_agg min () :no_exclusion
  (seq
    (sync_summary self)
    (reply (min (summary_obj self)))))
(handler instr_counter_agg max () :no_exclusion
  (seq
```

```
    (sync_summary self)
    (reply (max (summary_obj self)))))
(handler instr_counter_agg mean () :no_exclusion
  (seq
    (sync_summary self)
    (reply (mean (summary_obj self)))))
(handler instr_counter_agg stdev () :no_exclusion
  (seq
    (sync_summary self)
    (reply (stdev (summary_obj self)))))
(handler instr_counter_agg summarize (stream title) :no_exclusion
  (let ((total_count 0))
    (seq
      (write_string stream title)
      (write_string stream "\tNode\tCount\n")
      (forall i from 0 below groupsize
        (let ((count (read_local_count self i)))
          (seq
            (write_string stream "\t")
            (write_int stream i)
            (write_string stream "\t")
            (write_int stream count)
            (write_string stream "\n"))
          (set! total_count (+ total_count count))
          (inc (summary_obj self) (int2float count))))
      (write_string stream "\tTotal:\t")
      (write_int stream total_count)
      (write_string stream "\n\tMin:\t")
      (write_float stream (min (summary_obj self)))
      (write_string stream "\tMax:\t")
      (write_float stream (max (summary_obj self)))
      (write_string stream "\n\tMean:\t")
      (write_float stream (mean (summary_obj self)))
      (write_string stream "\tStdev:\t")
      (write_float stream (stdev (summary_obj self)))
      (write_string stream "\n")
      (reply DONE))))
(handler instr_counter_agg sync_summary () :no_exclusion
  (seq
    (reset (summary_obj self))
    (forall i from 0 below groupsize
      (inc (summary_obj self) (int2float (read_local (sibling self i)))))
    (reply DONE)))

;; INSTR_SUMMARY_CLASS
```

```
(class instr_summary_class (array)
  (initial (primitive init_instr_summary self)))
(method instr_summary_class create ()
  (reply (new instr_summary_class 32)))
(method instr_summary_class inc (value)
  (reply (primitive inc_instr_summary self value)))
(method instr_summary_class min ()
  (reply (primitive instr_summary_min self)))
(method instr_summary_class max ()
  (reply (primitive instr_summary_max self)))
(method instr_summary_class mean ()
  (reply (primitive instr_summary_mean self)))
(method instr_summary_class stdev ()
  (reply (primitive instr_summary_stdev self)))
(method instr_summary_class reset ()
  (reply (primitive reset_instr_summary self)))

;; INSTR_STOPWATCH_CLASS

(class instr_stopwatch_class (rootclass) start_time elapsed_time
  (initial (set_start_time self 0.0) (set_elapsed_time self 0.0)))
(method instr_stopwatch_class create ()
  (reply (new instr_stopwatch_class)))
(method instr_stopwatch_class start ()
  (set_start_time self (primitive now))
  (reply (elapsed_time self)))
(method instr_stopwatch_class stop ()
  (seq
    (set_elapsed_time self
              (+ (elapsed_time self) (- (primitive now) (start_time self))))
    (reply (elapsed_time self))))
(method instr_stopwatch_class read ()
  (reply (elapsed_time self)))
(method instr_stopwatch_class reset ()
  (let ((old_elapsed (elapsed_time self)))
    (set_start_time self 0.0)
    (set_elapsed_time self 0.0)
    (reply old_elapsed)))

;; TRACECLASS

(class traceclass)
(method traceclass :rest ()
  (reply (primitive write_event msg)))

;;  SYSTEM CONSTANTS
```

```
(constant true -1)
(constant false 0)

;;  PREDEFINED GLOBALS

(global nil (new nullclass))
(global console (new consoleclass 520 "console_stream"))  ; same as streamclass
(global trace_monitor (new traceclass))
```

Bibliography

[1] O. Agesen, J. Palsberg, and M. Schwartzbach. Type inference of SELF: Analysis of objects with dynamic and multiple inheritance. In *Proceedings of ECOOP '93*, 1993.

[2] Ole Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *Proceedings of ECOOP '95*, pages 2–26. Springer-Verlag Lecture Notes in Computer Science No. 952, 1995.

[3] Ole Agesen and Urs Hölzle. Type feedback vs. concrete type analysis: A comparison of optimization techniques for object-oriented languages. In *Proceedings of OOPSLA'95*, pages 91–107, 1995.

[4] G. Agha and C. Hewitt. *Object-Oriented Concurrent Programming*, chapter on Actors: A Conceptual Foundation for Concurrent Object-Oriented Programming, pages 49–74. MIT Press, 1987.

[5] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, 1986.

[6] Gul Agha. The structure and semantics of actor languages. In J.W. de Bakker, W. P. Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages*, pages 1–59. Springer-Verlag, 1991.

[7] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Computer Science. Addison-Wesley, Reading, Massachusetts, 1987.

[8] Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Twenty First Symposium on Principles of Programming Languages*, pages 151–162, Portland, Oregon, January 1994.

[9] P. America. Inheritance and subtyping in a parallel object-oriented language. In *Proceedings of ECOOP*, pages 234–42. Springer-Verlag, June 1987.

[10] Pierre America. POOL-T: A parallel object-oriented language. In Aki Yonezawa and Mario Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 199–220. MIT Press, 1987.

[11] Pierre America. A parallel object-oriented language with inheritance and subtyping. In *Proceedings of ECOOP/OOPSLA '90*, pages 161–8, 1990.

[12] T. Anderson, D. Culler, and D. Patterson. A case for NOW (networks of workstations). *IEEE Micro*, 15(1):54–64, 1995.

[13] Apple Computer, Inc., Cupertino, California. *Object Pascal User's Manual*, 1988.

[14] Joel Auslander, Matthai Philipose, Craig Chambers, Susan J. Eggers, and Brian N. Bershad. Fast, effective dynamic compilation. In *Proceedings of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 149–159, May 1996.

[15] J. Barnes and P. Hut. A hierarchical O(N log N) force calculation algorithm. Technical report, The Institute for Advanced Study, Princeton, New Jersey, 1986.

[16] Gerald Baumgarter and Vince F. Russo. Signatures: A language extension for improving type abstraction and subtype polymorphism in C++. *Software – Practice and Experience*, 25(8):863–889, August 1995.

[17] B.N. Bershad, E.D. Lazowska, and H.M. Levy. Presto: A system for object-oriented parallel programming. *Software — Practice and Experience*, 18(8):713–732, August 1988.

[18] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, Andrew Shaw, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of Principles and Practice of Parallel Programming*, 1995.

[19] Nan J. Boden. A study of fine-grain programming using cantor. Master's thesis, California Institute of Technology, 1988. Caltech-CS-TR-88-11.

[20] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the Association for Computing Machinery*, 17(10):547–557, 1974.

[21] Brad Calder and Dirk Grunwald. Reducing indirect function call overhead in C++ programs. In *Twenty-first Symposium on Principles of Programming Languages*, pages 397–408. ACM SIGPLAN, 1994.

[22] Brad Calder, Dirk Grunwald, and Benjamin Zorn. Quantifying differences between C and C++ programs. Technical Report CU-CS-698-94, University of Colorado, Boulder, January 1994.

[23] Peter S. Canning, William R. Cook, Walter L. Hill, and Walter G. Olthoff. Interfaces for strongly-typed object-oriented programming languages. In *Proceedings of OOPSLA'89, Fourth Annual Conference on Object-Oriented Systems, Languages and Applications*, pages 457–467, October 1989.

[24] Robert Cartwright and Mike Fagan. Soft typing. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 278–292, Ontario, Canada, June 1991.

[25] D. A. Case. Computer simulations of protein dynamics and thermodynamics. *IEEE Computer*, 26:47, 1993.

[26] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation*, pages 146–60, 1989.

[27] C. Chambers and D. Ungar. Iterative type analysis and extended message splitting. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 150–60, 1990.

[28] C. Chambers and D. Ungar. Making pure object-oriented languages practical. In *OOPSLA '91 Conference Proceedings*, pages 1–15, May 1991.

[29] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of SELF, a dynamically-typed object-oriented language based on prototypes. In *OOPSLA '89 Conference Proceedings*, pages 49–70, July 1989.

[30] Craig Chambers. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. PhD thesis, Stanford University, Stanford, CA, March 1992.

[31] Craig Chambers. The Cecil language: Specification and rationale. Technical Report TR 93-03-05, Department of Computer Science and Engineering, University of Washington, Seattle, Washington, March 1993.

[32] Craig Chambers. The Cecil language: Specification and rationale, version 2.0. Technical report, Department of Computer Science and Engineering, University of Washington, Seattle, Washington, March 1995.

[33] Rohit Chandra, Anoop Gupta, and John L. Hennessy. Data locality and load balancing in COOL. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1993.

[34] K. Mani Chandy and Carl Kesselman. Compositional C++: Compositional parallel programming. In *Proceedings of the Fifth Workshop on Compilers and Languages for Parallel Computing*, New Haven, Connecticut, 1992. YALEU/DCS/RR-915, Springer-Verlag Lecture Notes in Computer Science, 1993.

[35] A. A. Chien, U. S. Reddy, J. Plevyak, and J. Dolby. ICC++ – a C++ dialect for high performance parallel computing. In *Proceedings of the 2nd International Symposium on Object Technologies for Advanced Software*. Springer-Verlag, LNCS 742, 1996.

[36] A. A. Chien, M. Straka, J. Dolby, V. Karamcheti, J. Plevyak, and X. Zhang. A case study in irregular parallel programming. In *DIMACS Workshop on Specification of Parallel Algorithms*, May 1994. Also available as Springer-Verlag LNCS.

[37] Andrew Chien, Vijay Karamcheti, and John Plevyak. The Concert system—compiler and runtime support for efficient fine-grained concurrent object-oriented programs. Technical Report UIUCDCS-R-93-1815, Department of Computer Science, University of Illinois, Urbana, Illinois, June 1993.

[38] Andrew Chien and Uday Reddy. ICC++ language definition. Concurrent Systems Architecture Group Memo, Also available from http://www-csag.cs.uiuc.edu/, February 1995.

[39] Andrew A. Chien. Concurrent Aggregates: An object-oriented language for fine-grained message-passing machines. Technical Report MIT-AI-TR-1248, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Cambridge, MA, September 1990.

[40] Andrew A. Chien. Concurrent aggregates: Using multiple-access data abstractions to manage complexity in concurrent programs. In *In Proceedings of the Workshop on Object-Based Concurrent Programming at OOPSLA '90.*, 1990. Appeared in OOPS Messenger, Volume 2, Number 2, April 1991.

[41] Andrew A. Chien. *Concurrent Aggregates: Supporting Modularity in Massively-Parallel Programs*. MIT Press, Cambridge, MA, 1993.

[42] Andrew A. Chien and William J. Dally. Experience with concurrent aggregates (ca): Implementation and programming. In *Proceedings of the Fifth Distributed Memory Computers Conference*, Charleston, South Carolina, April 8-12 1990. SIAM.

[43] Andrew A. Chien, Vijay Karamcheti, John Plevyak, and Xingbin Zhang. Concurrent Aggregates language report 2.0. Available via anonymous ftp from cs.uiuc.edu in /pub/csag or from http://www-csag.cs.uiuc.edu/, September 1993.

[44] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensative interprocedural computation of pointer-induced aliases and side effects. In *Twentieth Symposium on Principles of Programming Languages*, pages 232–245. ACM SIGPLAN, 1993.

[45] W. Clinger and J. Rees (editors). *Revised*^{4} report on the algorithmic lanuguage scheme. *ACM Lisp Pointers IV*, July-September 1991.

[46] William D. Clinger. Foundations of actor semantics. Technical Report AI-TR-633, MIT Artificial Intelligence Laboratory, 1981.

[47] K. Cooper, K. Kennedy, and L. Torczon. The impact of interprocedural analysis and optimization in the R^n environment. *ACM Transactions on Programming Languages and Systems*, 8(4):491–523, October 1986.

[48] K. D. Cooper, M. W. Hall, and K. Kennedy. Procedure cloning. In *Proceedings of the IEEE Computer Society 1992 International Conference on Computer Languages*, pages 96–105, April 1992.

[49] K. D. Cooper, M. W. Hall, and K. Kennedy. A methodology for procedure cloning. *Computer Languages*, 19(2):105–118, April 1993.

[50] Patrick Cousot and Radia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Fourth Symposium on the Principles of Programming Languages*, pages 238–252, 1977.

[51] Cray Research, Inc., Eagan, Minnesota 55121. *CRAY T3D Software Overview Technical Note*, 1992.

[52] Cray Research, Inc. *Cray T3D System Architecture Overview*, March 1993.

[53] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. An efficient method of computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[54] William Dally and Andrew Chien. Object oriented concurrent programming in cst. In *Proceedings of the Third Conference on Hypercube Computers*, pages 434–9, Pasadena, California, 1988. SIAM.

[55] William J. Dally. *A VLSI Architecture for Concurrent Data Structures*. Kluwer Academic Publishers, Boston, Mass., 1987.

[56] William J. Dally, Andrew Chien, Stuart Fiske, Waldemar Horwat, John Keen, Michael Larivee, Rich Lethin, Peter Nuth, Scott Wills, Paul Carrick, and Greg Fyler. The J-Machine: A fine-grain concurrent computer. In *Information Processing 89, Proceedings of the IFIP Congress*, pages 1147–1153, August 1989.

[57] William J. Dally, J. A. Stuart Fiske, John S. Keen, Richard A. Lethin, Michael D. Noakes, Peter R. Nuth, Roy E. Davison, and Gregory A. Fyler. The message-driven processor. *IEEE Micro*, pages 23–39, April 1992.

[58] Jeffrey Dean, Craig Chambers, and David Grove. Selective specialization for object-oriented languages. In *Proceedings of the ACM SIGPLAN '95 Conference on Programmin g Language Design and Implementation*, pages 93–102, La Jolla, CA, June 1995.

[59] Jeffrey Dean, Dave Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. Technical Report TR 94-12-01, Department of Computer Science and Engineering, University of Washington, Seattle, Washington, December 1994.

[60] Alain Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 230–241, 1994.

[61] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the smalltalk-80 system. In *Eleventh Symposium on Principles of Programming Languages*, pages 297–302. ACM, 1984.

[62] J. Dolby. Using the concert debugger. CSA Group Memo, October 1993.

[63] J. Dolby, V. Karamcheti, and J. Plevyak. Using the concert system on sun workstations. CSA Group Memo, September 1993.

[64] Julian Dolby. Parasight: A debugger for concurrent object-oriented programs. Master's thesis, University of Illinois, Department of Computer Science, 1304 W. Springfield Avenue, Urbana, Illinois., August 1995.

[65] Julian Dolby, Vijay Karamcheti, John Plevyak, and Xingbin Zhang. Concert tutorial. CSA Group Memo, March 1994.

[66] Jack J. Dongarra, Roldan Pozo, and David W. Walker. LAPACK++: A design overview of object-oriented extensions for high performance linear algebra. In *Proceedings of Supercomputing'93*, pages 162–171, 1993.

[67] J. Elliot and B. Moss. Managing stack frames in smalltalk. In *Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, pages 229–40, 1987. SIGPLAN NOTICES Volume 22 Number 7 July 1987.

[68] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[69] Ana M. Erosa and Laurie J. Hendren. Taming control flow: A structured approach to eliminating goto statements. Technical Report 76, McGill University, September 1993. ftp wally.cs.mcgill.ca as memo76.

[70] A. Krishnamuthy et al. Parallel programming in Split-C. In *Proceedings of Supercomputing*, pages 262–273, 1993.

[71] C. Leiserson et al. The network architecture of the Connection Machine CM-5. In *Proceedings of the Symposium on Parallel Algorithms and Architectures*, 1992. Available from `ftp://cmns.think.com/doc/Papers/net.ps.Z`.

[72] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Laguages and Systems*, 9(3):319–49, July 1987.

[73] S. Frolund. Inheritance of synchronization constraints in concurrent object-oriented programming languages. In *Proceedings of the European Conference on Object-oriented Programming (ECOOP)*, 1992.

[74] Kildal G. A unified approach to global program optimization. In *ACM Symposium on Principles of Programming Languages*, pages 194–206, 1973.

[75] Richard Gabriel. Lisp: Good news, bad news, how to win big. In *First European Conference on the Practical Applications of Lisp*, Cambridge, England, March 1990. Cambridge University. Available at http://www.ai.mit.edu/good-news/good-=news.

[76] Adele Goldberg and David Robson. *Smalltalk-80: The language and its implementation.* Addison-Wesley, 1985.

[77] Seth Copen Goldstein, Klaus Eric Schauser, and David Culler. Lazy threads, stacklets, and synchronizers: Enabling primitives for parallel languages. In *Proceedings of POOMA'94*, 1994.

[78] J. Graver and R. Johnson. A type system for smalltalk. In *Proceedings of POPL*, pages 136–150, 1990.

[79] L. Greengard and V Rokhlin. A fast algorithm for particle simulations. *Journal of Computational Physics*, 73:325–48, 1987.

[80] A. Grimshaw. Easy-to-use object-oriented parallel processing with Mentat. *IEEE Computer*, 5(26):39–51, May 1993.

[81] Concurrent Systems Architecture Group. The ICC++ reference manual. Concurrent Systems Architecture Group Memo, May 1996.

[82] Linley Gwennap. P6 underscores Intel's lead. *Microprocessor Report*, 9(2), February 1995.

[83] M. W. Hall. *Managing Interprocedural Optimization.* PhD thesis, Rice University, 1991.

[84] M. W. Hall, S. Hiranandani, and K. Kennedy. Interprocedural compilation of Fortran D for MIMD distributed memory machines. In *Supercomputing '92*, pages 522–535, 1992.

[85] Mary W. Hall, Ken Kennedy, and Kathryn S. McKinley. Interprocedural transformations for parallel code generation. In *Proceedings of the* 4^{th} *Annual Conference on High-Performance Computing (Supercomputing '91)*, pages 424–434, November 1991.

[86] Mary W. Hall, John M. Mellor-Crummey, Alan Clarle, and René G. Rodríguez. FIAT: A framework for interprocedural analysis and transformation. In *Proceedings of the Sixth Workshop for Languages and Compilers for Parallel Machines*, pages 522–545, August 1993.

[87] Robert H. Halstead Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[88] P. Brinch Hansen. Structured multiprogramming. *Communications of the ACM*, 15(7):574–590, July 1972.

[89] Samuel P. Harbison. *Modula-3*. Prentice Hall, 1992.

[90] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Twenty-second Annual ACM Symposium on Principles of Programming Languages*, pages 130–141. ACM SIGPLAN, 1995.

[91] J. Hermans and M. Carson. Cedar documentation. Unpublished manual for CEDAR, 1985.

[92] C. Hewitt and H. Baker. Actors and continuous functionals. In *Proceedings of the IFIP Working Conference on Formal Description of Programming Concepts*, pages 367–87, August 1977.

[93] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiler optimizations for FORTRAN D on MIMD distributed-memory machines. *Communications of the ACM*, August 1992.

[94] Richard C. Holt. Some deadlock properties of computer systems. *ACM Computing Surveys*, 4(3):179–196, Sept 1972.

[95] Urs Hölzle. *Adaptive Optimization for* SELF*: Reconciling High Performance with Exploratory Programming*. PhD thesis, Stanford University, Stanford, CA, August 1994.

[96] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *ECOOP'91 Conference Proceedings*. Springer-Verlag, 1991. Lecture Notes in Computer Science 512.

[97] Urs Hölzle and David Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the 1994 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 326–336, June 1994.

[98] W. Horwat, A. Chien, and W. Dally. Experience with CST: Programming and implementation. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 101–9. ACM SIGPLAN, ACM Press, 1989.

[99] W. Horwat, B. Totty, and W. Dally. Cosmos: An operating system for a fine-grain concurrent computer. To appear in *Research in Object-based Concurrency*, G. Agha, Editor, 1990.

[100] Waldemar Horwat. Concurrent Smalltalk on the message-driven processor. Master's thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, June 1989.

[101] C. Houck and G. Agha. HAL: A high-level actor language and its distributed implementation. In *Proceedings of the 21st International Conference on Parallel Processing*, pages 158–165, St. Charles, IL, August 1992.

[102] Christopher James Houck. Run-time system support for distributed actor programs. Master's thesis, University of Illinois at Urbana-Champaign, 1992.

[103] International Organization for Standardization. *Ada 95 Reference Manual*, version 6.0 edition, Dec 1994.

[104] Suresh Jagannathan and Jim Philbin. A foundation for an efficient multi-threaded Scheme system. In *Proc. 1992 ACM Conference on Lisp and Functional Programming*, pages 345–357, June 1992.

[105] Suresh Jagannathan and Stephen Weeks. A unified treatment of flow analysis in higher-order languages. In *Twenty-second Symposium on Principles of Programming Languages*, pages 393–407. ACM SIGPLAN, 1995.

[106] Suresh Jagannathan and Andrew Wright. Effective flow analysis for avoiding run-time checks. Technical Report 95-3, NEC Research Institute, May 1995.

[107] Guy L. Steele Jr. *Common LISP: The Language.* Digital Press, second edition, 1990.

[108] L. V. Kale and Sanjeev Krishnan. CHARM++: A portable concurrent object oriented system based on C++. In *Proceedings of OOPSLA'93*, pages 91–108, 1993.

[109] Vijay Karamcheti and Andrew Chien. Concert – efficient runtime support for concurrent object-oriented programming languages on stock hardware. In *Proceedings of Supercomputing'93*, 1993.

[110] Vijay Karamcheti and Andrew Chien. Do faster routers imply faster communication? In *Proceedings of the Parallel Computer Routing and Communication Workshop (PCRCW'94)*, pages 1–15, Seattle, Washington, 1994. Springer-Verlag Lecture Notes in Computer Science No. 853.

[111] Vijay Karamcheti and Andrew Chien. Software overhead in messaging layers: Where does the time go? In *Proceedings of the Sixth Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, 1994. Available from `http://www-csag.cs.uiuc.edu/papers/asplos94.ps`.

[112] Vijay Karamcheti and Andrew A. Chien. A comparison of architectural support for messaging on the TMC CM-5 and the Cray T3D. In *Proceedings of the International Symposium on Computer Architecture*, 1995. Available from `http://www-csag.cs.uiuc.edu/papers/cm5-t3d-messaging.ps`.

[113] Vijay Karamcheti, John Plevyak, and Andrew A. Chien. Runtime mechanisms for efficient dynamic multithreading. *Journal of Parallel and Distributed Computing*, 1996.

[114] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1978.

[115] Woo Young Kim and Gul Agha. Efficient support for location transparency in concurrent object-oriented programming languages. In *Proceedings of the Supercomputing '95 Conference*, San Diego, CA, December 1995.

[116] H. Konaka. An overview of ocore: A massively parallel object-based language. Technical Report TR-P-93-002, Tsukuba Research Center, Real World Computing Partnership, Tsukuba Mitsui Building 16F, 1-6-1 Takezono, Tsukuba-shi, Ibaraki 305, JAPAN, 1993.

[117] D. Kranz, R. Halstead Jr., and E. Mohr. Mul-T: A high-performance parallel lisp. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 81–90, 1989.

[118] W. Landi and B. Ryder. Pointer-induced aliasing: A problem classification. In *Symposium on Principles of Programming Languages*, 1991.

[119] James Larus. C**: a large-grain, object-oriented, data parallel programming language. In *Proceedings of the Fifth Workshop for Languages and Compilers for Parallel Machines*, pages 326–341. Springer-Verlag, August 1992.

[120] James Richard Larus. Restructuring symbolic programs for concurrent execution on multiprocessors. Technical Report UCB/CSD 89/502, University of California at Berkeley, 1989.

[121] W. J. Leddy and K. S. Smith. The design of the experimental systems kernel. In *Proceedings of the Fourth Conference on Hypercube Computers*, March 1989.

[122] J. Lee and D. Gannon. Object oriented parallel programming. In *Proceedings of the ACM/IEEE Conference on Supercomputing*. IEEE Computer Society Press, 1991.

[123] Peter Lee and Mark Leone. Optimizing ml with run-time code generation. In *Proceedings of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 137–148, May 1996.

[124] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *Transactions on Programming Languages and Systems*, 1(1):121–141, July 1979.

[125] Xavier Leroy. Unboxed objects and polymorphic typing. In *Proc. 19th symp. Principles of Programming Languages*, pages 177–188. ACM press, 1992.

[126] Carl R. Manning. Acore: The design of a core actor language and its compiler. Master's thesis, Massachusetts Institute of Technology, August 1987.

[127] H. Masuhara, S. Matsuoka, T. Watanaba, and A. Yonezawa. Objected-oriented concurrent reflective languages can be implemented efficiently. In *Proceedings of OOPSLA '92*, pages 127–144, 1992.

[128] S. Matsuoka and A. Yonezawa. *Research Directions in Object-Based Concurrency*, chapter "Analysis of Inheritance Anomaly in Object-Oriented Concurrent Languages". MIT Press, 1993.

[129] F. H. McMahon. The Livermore Fortran kernels: a computer test of the numerical performance range. Technical report UCRL-53745, Lawerence Livermore National Laboratory, Livermore, California, 1986.

[130] Bertrand Meyer. *Object-oriented Software Construction.* Prentice Hall, 1988.

[131] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* The MIT Press, 1990.

[132] E. Mohr, D. Kranz, and R. Halstead Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, July 1991.

[133] D. A. Moon. Object-Oriented Programming with Flavors. *SIGPLAN Notices (Proc. Intl. Conf. on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA))*, 21(11), September 1986.

[134] Stephan Murer, Jerome A. Feldman, Chu-Cheow Lim, and Martina-Maria Seidel. pSather: Layered extensions to an object-oriented language for efficient parallel computation. Technical Report TR-93-028, International Computer Science Institute, Berkeley, CA, June 1993 November 1993.

[135] N. Oxhøj, J. Palsberg, and M. Schwartzbach. Making type inference practical. In *Proceedings of ECOOP '92*, pages 329–349. Springer-Verlag, Lecture Notes in Computer Science, No. 615, 1992.

[136] J. Palsberg and M. Schwartzbach. Object-oriented type inference. In *Proceedings of OOPSLA '91*, pages 146–61, 1991.

[137] Jens Palsberg and Michael I. Schwartzbach. Safety analysis versus type inference for partial types. *Information Processing Letters*, 43:175–180, 1992.

[138] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, 1990.

[139] K. A. Pier. A retrospective on the dorado. In *Proceedings of the Tenth Annual Symposium on Computer Architecture*, pages 252–269, June 1983.

[140] John Plevyak and Andrew Chien. Incremental inference of concrete types. Technical Report UIUCDCS-R-93-1829, Department of Computer Science, University of Illinois, Urbana, Illinois, June 1993.

[141] John Plevyak and Andrew A. Chien. Precise concrete type inference of object-oriented programs. In *Proceedings of OOPSLA'94, Object-Oriented Programming Systems, Languages and Architectures*, pages 324–340, 1994.

[142] John Plevyak and Andrew A. Chien. Type directed cloning for object-oriented programs. In *Proceedings of the Workshop for Languages and Compilers for Parallel Computing*, pages 566–580, 1995.

[143] John Plevyak, Vijay Karamcheti, and Andrew Chien. Analysis of dynamic structures for efficient parallel execution. In *Proceedings of the Sixth Workshop for Languages and Compilers for Parallel Machines*, pages 37–56, August 1993.

[144] John Plevyak, Vijay Karamcheti, Xingbin Zhang, and Andrew Chien. A hybrid execution model for fine-grained languages on distributed memory multicomputers. In *Proceedings of Supercomputing'95*, 1995.

[145] John Plevyak, Xingbin Zhang, and Andrew A. Chien. Obtaining sequential efficiency in concurrent object-oriented programs. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, pages 311–321, January 1995.

[146] A. Rogers, M. Carlisle, J. Reppy, and L. Hendren. Supporting dynamic data structures on distributed memory machines. *ACM Transactions on Programming Languages and Systems*, 1995.

[147] Erik Ruf. Context-insensitive alias analysis reconsidered. In *Proceedings of the 1995 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–22, June 1995.

[148] Bernhard Rytz and Marc Gengler. A polyvariant binding time analysis. Technical Report YALEU/DCS/RR-909, Yale University, Department of Computer Science, 1992. Proceedings of the 1992 ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation.

[149] G. Sabot. *The Paralation Model.* MIT Press, Cambridge, Massachusetts, 1988.

[150] A. D. Samples, D. Ungar, and P. Hilfinger. Soar: Smalltalk without bytecodes. In *OOPSLA '86 Prodeedings*, pages 107–18, September 1986.

[151] Michael Sannella, John Maloney, Bjorn Freeman-Benson, and Alan Borning. Multi-way versus one-way constraints in user interfaces: Experience with the deltablue algorithm. *Software – Practice and Experience*, 23(5):529–566, May 1993.

[152] Marcel Schelvis and Eddy Bledoeg. The implementation of a distributed smalltalk. In *Proceedings of the European Conference on Object-Oriented Programming*, 1988.

[153] Zhong Shao and Andrew W. Appel. A type-based compiler for standard ML. In *SIGPLAN Conference on Programming Language Design and Implement ation*, pages 116–129, 1995.

[154] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages.* PhD thesis, Carnegie Mellon University Department of Computer Science, Pittsburgh, PA, May 1991. also CMU-CS-91-145.

[155] Olin Shivers. The semantics of scheme control-flow analysis. In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 190–198, June 1991.

[156] Olin Shivers. *Topics in Advanced Language Implementation*, chapter Data-Flow Analysis and Type Recovery in Scheme, pages 47–88. MIT Press, Cambridge, MA, 1991.

[157] K. Smith and A. Chatterjee. A C++ environment for distributed application execution. Technical Report ACT-ESP-015-91, Microelectronics and Computer Technology Corporation (MCC), November 1990.

[158] K. Smith and R. Smith II. The experimental systems project at the microelectronics and computer technology corporation. In *Proceedings of the Fourth Conference on Hypercube Computers*, March 1989.

[159] SPARC International, NJ. *The SPARC Architecture Manual (Version 8)*, 1993.

[160] Vugranam C. Sreedhar and Guang R. Gao. A linear time algorithm for placing ϕ-nodes. In *Twenty-second Symposium on Principles of Programming Languages*, pages 62–73. ACM SIGPLAN, 1995.

[161] Richard Stallman. *The GNU C Compiler*. Free Software Foundation, 1991.

[162] Dan Stefanescu and Yuli Zhou. An equational framework for the flow analysis of higher-order functional programs. In *Proceedings of ACM Conference on Lisp and Functional Programming*, pages 318–327, 1994.

[163] A. A. Stepanov and M. Lee. The standard template library. ISO programming language C++ project. Technical Report X3J16/94-0095, WG21/NO482, Hewlett-Packard, May 1994.

[164] David Stoutamire and Stephen Omohundro. Sather 1.1, draft. Available from `http://www.icsi.berkeley.edu/~sather/Sather-1.1.ps`, August 1995.

[165] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, second edition, 1991.

[166] Bjarne Stroustrup. *The Design and Evolution of C++*. Addsion-Wesley, 1994.

[167] Mahesh Subramaniam. Using the Concert emulator. CSA Group Memo, Feburary 1994.

[168] Sun Microsystems Computer Corporation. *The Java Language Specification*, March 1995. Available at http://java.sun.com/1.0alpha2/doc/java-whitepaper.ps.

[169] Norihisa Suzuki. Inferring types in Smalltalk. In *Eighth Symposium on Principles of Programming Languages*, pages 187–199, January 1981.

[170] K. Taura, S. Matsuoka, and A. Yonezawa. An efficient implementation scheme of concurrent object-oriented languages on stock multicomputers. In *Proceedings of the Fifth ACM SIGPLAN Symposium on the Principles and Practice of Parallel Programming*, 1993.

[171] Kenjiro Taura. Design and implementation of concurrent object-oriented programming languages on stock multicomputers. Master's thesis, The University of Tokyo, Tokyo, February 1994.

[172] Kenjiro Taura, Satoshi Matsuoka, and Akinori Yonezawa. StackThreads: An abstract machine for scheduling fine-grain threads on stock CPUs. In *Joint Symposium on Parallel Processing*, 1994.

[173] Thinking Machines Corporation, 245 First Street, Cambridge, MA 02154-1264. *The Connection Machine CM-5 Technical Summary*, October 1991.

[174] C. Tomlinson, M. Scheevel, and V. Singh. Report on rosette 1.0. MCC Internal Report, Object-Based Concurrent Systems Project, December 1989.

[175] K. Traub. A dataflow compiler substrate. Computation Structures Group Memo 261, MIT Laboratory for Computer Science, 1986.

[176] David Ungar and Randall B. Smith. SELF: The power of simplicity. In *Proceedings of OOPSLA '87*, pages 227–41. ACM SIGPLAN, ACM Press, 1987.

[177] David M. Ungar. *the Design and Evaluation of a High Performance Smalltalk System.* MIT Press, 1987.

[178] David M. Ungar and David A. Patterson. *Smalltalk-80: Bits of History, Words of Advice*, chapter Berkeley Smalltalk: Who Knows Where the Time Goes? Addison-Wesley series in Computer Science. Addison-Wesley, 1983.

[179] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: A mechanism for intergrated communication and computation. In *International Symposium on Computer Architecture*, 1992. Available from `http://www.cs.cornell.edu/Info/Projects/CAM/isca92.ps`.

[180] David B. Wagner and Bradley G. Calder. Leapfrogging: A portable technique for implementing efficient futures. In *Proceedings of the Fifth ACM SIGPLAN Symposium on the Principles and Practice of Parallel Programming*, pages 208–217, 1993.

[181] Tim A. Wagner, Vance Maverick, Susan L. Graham, and Michael A. Harrison. Accurate static estimators for program optimization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–96, Orlando, Florida USA, June 1994.

[182] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for c programs. In *Proceedings of the 1995 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, June 1995.

[183] M. Yasugi, S. Matsuoka, and A. Yonezawa. ABCL/onEM-4: A new software/hardware architecture for object-oriented concurrent computing on an extended dataflow supercomputer. In *Proceedings of the ACM Conference on Supercomputing '92*, 1992.

[184] Y. Yokote and M. Tokoro. Concurrent programming in ConcurrentSmalltalk. In Aki Yonezawa and Mario Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 129–158. MIT Press, 1987.

[185] A. Yonezawa, E. Shibayama, T. Takada, and Y. Honda. Object-oriented concurrent programming – modelling and programming in an object-oriented concurrent language abcl/1. In Aki Yonezawa and Mario Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 55–89. MIT Press, 1987.

[186] Akinori Yonezawa, editor. *ABCL: An Object-Oriented Concurrent System*. MIT Press, 1990. ISBN 0-262-24029-7.

[187] Akinori Yonezawa, Satoshi Matsuoka, Masahiro Yasugi, and Kenjiro Taura. Implementing concurrent object-oriented languages on multicomputers. *IEEE Parallel and Distributed Technology*, pages 49–61, May 1993.

Index

Vita

Date of Birth: December 13, 1965

Citizenship: USA

Address: Department of Computer Science
University of Illinois at Urbana-Champaign
1304 West Springfield
Urbana, Illinois 61801

W Phone: +1 217 244 7116
H Phone: +1 217 328 5031
Email: jplevyak@cs.uiuc.edu

WWW: http://www-csag.cs.uiuc.edu/individual/jplevyak

Research Interests: Programming languages, compilers and computer architecture. Particularly, object-oriented and fine-grained concurrent programming languages, interprocedural flow analysis and optimization, and parallel and distributed computers.

Academic History:

Ph.D.	Computer Science	University of Illinois at Urbana-Champaign	1996
B.A.	Computer Science	University of California at Berkeley	1988

Professional History:

Research Assistant, Concurrent Systems Architecture Group, 1992-present
Research Assistant, National Center for Supercomputing Applications, 1991-1992
Programmer, Ampex Corporation, 1988-1991
Programmer, Ilex Corporation, 1984-1988

Awards: C.W. Gear Outstanding Graduate Student Award, 1994

Professional Societies:

Member, Association for Computing Machinery (ACM)
Member, Institute of Electrical and Electronics Engineers (IEEE)

Professional Activities:

Referee (partial list):
Object-oriented Programming Languages and Systems (OOPSLA)
Static Analysis Symposium (SAS)
Journal of Parallel and Distributed Computing (JPDC)
International Parallel Processing Symposium (IPPS)
International Symposium on Object Technologies for Advanced Software (ISOTAS)

System/Software Releases:

Illinois Concert System, Versions 1.0,1.1,2.0,3.0 from CSAG
The Concert System is a complete development environment for fine-grained concurrent object-oriented programs. I am the architect and principal author of the Concert compiler which embodies the research described in my thesis. It is available from http://www-csag.cs.uiuc.edu.
DTM, Version 2.3 from NCSA
The Data Transfer Mechanism (DTM) is a message passing facility designed to facilitate the creation of sophisticated distributed applications in a heterogeneous environment. It is available from http::://wwww.ncsa.uiuc.edu.
ACE 10/25, Versions 1.1,2.0 from Ampex Corporation
A commercial video edit controller, available as a product.

Additional Information:

Classes:
Computer Systems Organization (CS333), Introduction to VLSI Design (CS335), Program Verification (CS376), Fine-grained Systems: Design and Programming (CS497), Actor Systems (CS397), Object-Oriented Programming and Design (CS497), Programming Language Semantics (CS425), Topics in Graph and Geometric Algorithms (CS474), Genetic Programming (GE493), State in Programming Languages (CS497), Topics in Compiler Construction (CS426)

Seminar Classes:
Research Issues in Massively Parallel Machines (3 semesters)
Programming Languages Seminar (2 semester)
Actor Systems (1 semester)
Parallelizing Compilers and Program Optimization Seminar (1 semester)
Seminar on Languages and Compilers for High-Performance Computing (1 semester)

Lecture Series:
Center for Supercomputing Research and Development Colloquia (periodic)
Computer Science Department Seminars and Lectures (periodic)
Department of Electrical and Computer Engineering Graduate Seminars (periodic)

Teaching/Lecturing:

I have lectured in many of the seminar classes, every semester in group meetings, and twice for my advisor's classes. I am comfortable preparing and lecturing on my own and others' material.

Other Classes Attended:

Art History (ARTHI335,ARTHI343) (Baroque,Art Nouveau)
English (ENGL362,ENGL362) (Topics in Modern American Literature (2))

Other Occasional Audits:

Linguistics, History, Molecular Dynamics

Other Activities:

Tennis, Wallyball