Design and Implementation of a Compiler and Runtime System for Composite Tree Parallelism

Mark C. Chu-Carroll and Lori L. Pollock
carroll@cis.udel.edu pollock@cis.udel.edu
Department of Computer and Information Sciences
University of Delaware
Newark, DE 19716
(302) 831-1953, fax (302) 831-8458

Abstract

In the field of scientific computing, the use of parallelism has led to widespread improvements in the performance of complex computations. Unfortunately, outside of the domain of computational science, the promise of parallelism has been largely unfulfilled. Workstation and PC networks have become common in most offices, and cluster computing is receiving much attention, but the lack of programming support tools directed towards non-scientific programmers has inhibited the use of parallelism to increase the performance of general purpose applications such as spreadsheets, word processors, compilers, and databases. We have developed a new model of parallel programming for general purpose applications, called *composite tree parallelism*, which was designed by making a small number of simple alterations to the basic data parallel programming model. Composite tree parallelism allows programmers to write coarse grained tree-based parallel data structures, in a form that retains the power and simplicity of data parallelism. In this paper, we briefly present the composite tree parallel model, and then describe the design and implementation of the compiler and runtime system for effectively supporting composite parallel programming on workstation clusters.

Keywords: parallel programming, data parallelism, cluster computing.

1 Introduction

There are many general purpose applications that are computationally intensive and could benefit from the modest use of parallelism. These applications include such basic *general purpose* tools as spreadsheets, databases, word processors, and compilers. Workstation and PC networks, present in nearly every modern office, represent a large quantity of parallel computational power which could be used to reap significant performance benefits for a wide variety of applications. If it were possible to implement software that allowed applications to use the parallelism available in these clusters without adding significant complexity to the programming task, then the users of these applications could take advantage of computational power that exceeds that of some parallel supercomputers. Unfortunately, current tools for parallel programming are primarily directed toward scientific applications on supercomputers, and are not particularly suitable for use in general purpose applications, due to their restrictions on the use of data structures, and the difficulty of their use by non-expert parallel programmers.

We have designed and implemented a set of programming language extensions that allows programmers to implement a wide variety of general purpose parallel data structures, without causing significant changes in the underlying semantics of the base language, and without sacrificing compiler and runtime analyzability. Our goals throughout this project have been to provide (1) ease of use for the general purpose parallel programmer, (2) explicit parallelism to allow the programmer to use their knowledge of the problem to exploit parallelism in the solution, (3) portability of these constructs to a wide variety of base languages, and (4) a reasonable increase in performance over the sequential codes on a network cluster.

In particular, we have designed a programming construct called a *composite tree* [CP94], which allows programmers to implement parallel data structures in terms of a hierarchical data parallel programming abstraction. Data parallelism is a programming model that achieves massive parallelism through the lock-step execution of individual operations simultaneously on all members of a parallel aggregate. Composite tree parallelism retains the power and simplicity of data parallelism, but also adds the capability to easily program more general data structures, and effectively compile these parallel programs for performance gains on network clusters.

In this paper, we describe the design and implementation of our compiler and runtime system support for composite tree parallelism. The compiler translates an object oriented variant of Scheme extended with support for composite trees into an SPMD program in object oriented Scheme that makes calls to a message passing library. This compiler uses a new intermediate representation which we have designed, called the parallel continuation graph (PCG). The PCG is a general representation for dynamic parallel code which allows extensive optimization of dynamic parallel programs. The PCG code generation mechanism that we have developed properly handles the use of dynamic code structures, including properly preserving side effects between calls of anonymous functions. In this fashion, we provide language and compiler support which allows general purpose programmers to write efficient, general purpose parallel programs that take advantage of the attractive features of data parallelism, while not losing the ability to write programs that make use of dynamic control and data structures.

The rest of this paper is organized as follows. In section 2, we present a discussion of related work. In section 3, we describe the composite tree model of parallel programming. In section 4, we describe its implementation as a set of language extensions to object oriented Scheme, and give an example composite parallel program. Section 5 describes the compiler in detail, while Section 6 discusses the runtime support system. Section 7 presents performance results on three applications. We conclude with a discussion of ongoing and future work in section 8.

2 Related Work

Parallel programming is receiving more attention with the increase in availability of multiple processor architectures, and the realization of the possibility of using network clusters for executing a single parallel program. As a result, there is an incredible variety of different projects researching the best ways to take advantage of parallel computation. Within this broad realm, there are a number of different directions that may be applicable to general purpose parallel programming. These systems can be categorized into actor based systems, coarse grained data parallel systems, and parallel functional programming systems.

Actor systems are based on the actor model first proposed by Hewitt [HB77] and later refined by Agha [Agh86]. The actor programming model is based on having a computation performed by multiple simultaneously active computational entities, which communicate through message passing in order to cooperatively perform a single task. Actor systems attain massive parallelism by nearly eliminating any serialization in computations. Actor systems for workstation clusters have been proposed in various publications, including [AH92] and [GKSK94]. However, Actor programs are very communication intensive, which makes them difficult to execute efficiently on architectures like workstation clusters that have low communication bandwidth.

Chien [Chi92] proposed an extension to the actor model, called Concurrent Aggregates (CA). CA allows a collection of agents to work cooperatively as if they were a single special actor called an Aggregate. The CA model is very flexible, and can be used to implement nearly any other computational model, including standard actors, data parallelism, or even tree structures similar to composites. But like standard actor systems, they require a fast communication medium to perform effectively. Further, the flexibility of the system can become a liability, because while it can be used to implement any other parallel model, it does not provide direct support for any of them. This makes CA programs difficult to write, and difficult to optimize for effective execution.

The CA project has also recently produced a closely related language called ICC++[CD95], which provides aggregate programming support for general purpose programs written in C++. ICC++ implements parallelism through a merger of more traditional functional parallel methods with the Aggregrate model. It has the goal of allowing a programmer to implement a single source program which can generate efficient code for both sequential and parallel systems. The primary downfall of ICC++ is its complexity; the model of synchronization used in its programs is based on data dependence analysis. Non-expert users of ICC++ are likely to be surprised by the seemingly unpredictable results caused by the intricacies of data dependences in their programs.

A group at Indiana University developed an object-based extension of Concurrent Aggregates, called Distributed Collection Model, and implemented the model in a language called pC++ based on C++ [GL92]. pC++ programs run very effectively on high performance parallel architectures such as the CM-5. pC++ programs could be compiled to execute effectively on workstation networks, but the language model currently is not capable of implementing recursive data structures or nested parallelism.

Grimshaw at the University of Virginia has developed a programming language called Mentat [Gri93a, Gri93b], which is C++ extended by an actors variant called coarse grained dataflow. Mentat programs consist of a sequential program which makes calls on special objects called *Mentat Objects*. Mentat objects are essentially actors, which operate in parallel. When a member function is invoked on a Mentat object, it does not wait until the called function is completed to return results. The results are returned immediately when available, and the function continues performing computations after the result has been returned. The result of this is a programming model which is very similar to the traditional dataflow model.

Data parallel programming originated with the sequence based operators of languages like APL and SETL. These were adapted into a parallel programming model, which was implemented in a variety of data parallel languages, including C* [TMC90] and High Performance Fortran [Zos93], and a huge number of others. These languages are all very similar, and suffer from the same flaws when applied to general purpose programming; they restrict programmers to array based parallel data structures and require low cost communication and synchronization.

A variety of projects have coarsened the grain of data parallel computation to make it work more effectively on workstation clusters. These include dpSather [Sch92], Dino [RSW91], DataParallel C [HQ92], C** [LRV92] and many others. These systems all restrict the use of parallel data structures to arrays, and lack support for general purpose computing. PSather [MFL93] provides support for parallelism in a fashion that resembles data parallelism, and can be used to build nested, data parallel data structures in terms of a non-uniform memory access distributed shared memory. However, it does not provide compiler support for analyzing and optimizing those structures, but requires the programmers to optimize their code manually.

A group at CMU has built a data parallel language called NESL [Ble90], which allows programmers to implement functional programs in terms of a nested data parallel programming model. This allows programmers to write programs that operate on recursive data structures, and to take advantage of nested parallelism. NESL compiles its programs by translating into an intermediate language called Vcode [BC90], and performing optimizations on this intermediate form. The Vcode translation reduces all tree-based parallel structures to a pair of single dimensional arrays. Optimizations are performed on this array representation by translating parallel calls into two forms: data and task parallel. A function invoked by task parallelism may invoke either data parallel or task parallel operations on its inputs, but a data parallel call cannot invoke further parallelism. NESL is currently very oriented towards scientific programming, requires a fine grained parallel architecture for efficiency, and does not provide support for general purpose applications.

Functional programming languages have traditionally had the advantage that, due to their lack of ordering and side effects, programs do not need to be written in explicitly parallel form; any two expressions can be executed independently in parallel. However, recently, many people in the functional programming language community have decided that functional languages need explicit support for parallelism so that functional programmers can write portable, effective parallel programs. This has led to the development of a variety of different functional parallel languages [BMT92, Nik93, FCO90]. Most of these, like their imperative counterparts, are oriented towards the implementation of scientific or numeric programs.

One of these projects in particular seems to have potential in the area of building systems for implementing general purpose parallel programs. Hudak designed a construct for functional programming languages called a Mutable

Abstract Datatype (MAD) [Hud92], based on a functional construct called a Monad. This group has also developed a parallel programming model based on these constructs that allows explicitly parallel programs to be written that operate on these functional mutable data types [JH93]. This model uses a programming structure that is equivalent to our intermediate form to describe parallel computations in a referentially transparent, easily analyzable form. The drawback of this approach is that it requires the programmer to understand the underlying monadic formalism in order to write effective programs. The parallel mechanisms provided by this mechanism are also currently primitive message passing structures, and are difficult to assemble into high level parallel data structures.

3 The Composite Tree Model

Data parallel programming languages are sequential languages that have been extended to access a parallel memory, in which parallel data structures can be stored. The conventional data parallel languages provide a parallel array data type, and a set of three basic operations that can be performed on the parallel array: *selection*, which produces a new parallel array from the subportion of the original array that meets some condition, *mapping*, which allows some primitive (usually arithmetic) operation to be performed on all elements of an array simultaneously, and *reduction*, which uses some arithmetic operation to combine the elements of an array into a scalar value.

The basic model for our language was developed by taking the basic features of the data parallel model, and altering them in ways that make them better suited to general purpose programming on common coarse grained architectures. We make the following changes to the data parallel model:

1. **Increase the granularity:** Instead of using single arithmetic operations as the primitive grain of parallelism, we use entire object methods (member functions).

2. **Make parallel data structures more flexible:** Instead of restricting programmers to a single parallel array structure, we allow programmers to implement new parallel data structures in terms of a parallel tree abstraction. We do this by extending the standard single level host process/parallel memory model of most data parallel languages and architectures into a hierarchy. Tree structure is used, rather than a more general graph structure,

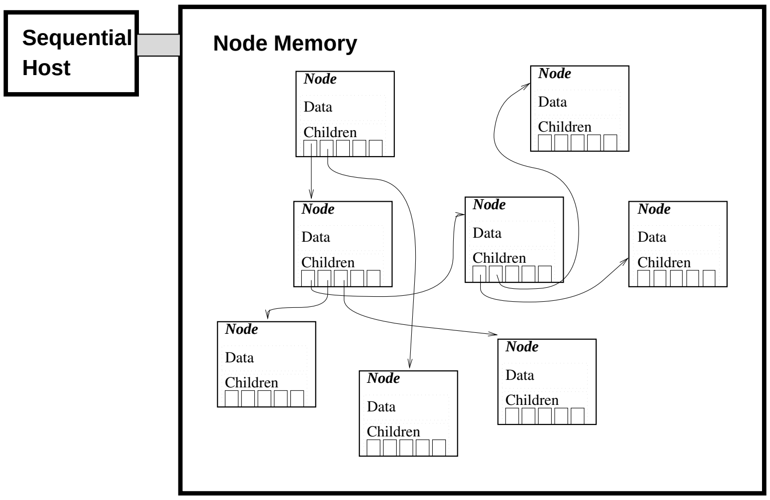


Figure 1: The Composite Virtual Machine

for several reasons. First, when using the standard unmodified control flow, which resembles a fanout process, deadlocks are impossible. Second, the simplicity of a tree-based structure increases the precision of code analysis at compile time. Finally, graph-based structures can be built using trees with rendezvous communication, so that there is no advantage to be gained by adopting the more complex graph structure.

3. **Make parallel control flow more flexible:** Instead of using strict synchronous parallel control flow, we allow programs to alter the flow of data and control through parallel data structures by inserting synchronous remote calls from one tree node to another. This gives programs the ability to alter the tree-based parallel control flow to suit their algorithms and structures. For example, this enables communication of data or synchronization between siblings in the tree. The rendezvous call mechanism was selected for providing this mechanism because it provides both synchronization and data exchange in a single, easy to comprehend construct which can be easily utilized in an object oriented programming language. Further, it provides these important capabilities in a form which does not sacrifice as much analyzability as a more general mechanism.

The result of these changes is what we call *composite tree parallelism*. The composite parallel model, like the data parallel model, consists of an abstract machine model, and a programming model which provides methods for programming the abstract machine. The composite model uses a virtual machine, which consists of a sequential computer, augmented with access to a parallel machine. But in the case of composite parallelism, that parallel machine consists of a *node memory*. Each element of the node memory consists of a private memory space, a private processing unit, a set of operations, and a children collection, which contains a collection of pointers to other nodes in the node memory. An illustration of the composite machine model is shown in figure 1.

Parallelism in the composite virtual machine is never directly invoked by the sequential computer. The sequential computer can access the node memory in two ways: accessing fields of a node, and invoking any of its operations. Parallelism can only be invoked in the node memory by an operation belonging to a node. The operations belonging to a node can perform operations involving other nodes. Parallelism in the composite model is achieved with the use of these operations. A node can perform three types of operation that allow it to interact with other nodes.

- **Child Invocation:** A node can invoke an operation simultaneously on all nodes stored in its children collection;
- **Rendezvous Accept:** A node can accept a rendezvous call of one of its operations from any other node;
- **Rendezvous Call:** A node can initiate a rendezvous call of an operation on another specific node.

In order to program the composite virtual machine, the composite programming model provides the following capabilities: (1) the ability to define types and methods on the elements of the node memory, (2) the ability to use the composite virtual machine operations in node methods, (3) the ability to invoke a method on a single composite node from a sequential program running on the sequential host machine, and (4) the ability to perform structural alterations on elements of the node memory in order to alter hierarchical structures built from collections of nodes.

4 Cabal - Object Oriented Scheme with Composites

The composite parallel model can be implemented as an extension to any suitable sequential base language. The new constructs that are required are *portable*, in the sense that they are not specific to a particular programming language. A sequential base language is suitable for an implementation of composites if: (1) it provides standard imperative programming structures (variables with mutable state), (2) it allows the programmer to define new data types, and (3) it provides the programmer with the ability to define operations on user defined data types. This set of requirements will

composite-definition ::= `(define-composite` *type-name* `(` *supertypes*$^+$ `)` `(` *slot-definition** `))`
par-stmt ::= `(par` *body-fun combining-fun node*$^+$ `)`
remote-call ::= `(remote-call` *method-name arg** `)`
call-accept ::= `(accept-call` *methodname*$^+$ `)`

(a) Syntax for Cabal Composite Extensions

```
(define-composite <par-list> ( <Composite> )
   ((values :accessor values-of)
      (is-leaf :init form #f :accessor is-leaf?)))
```

(b) A composite type definition in Cabal

```
(define-method search ((pl <par-list>) v)
   (if (is-leaf?  pl)
      (hash-table-get (values pl) v)
      (par (lambda (x) (search x v)) or pl)
```

(c) An example of a method using par

Caller:	**Acceptor:**
`...`	`...`
`(remote-call procA node 3 6)`	`(accept-call procA procB)`
`...`	`...`

(d) An example of a remote call

Figure 2: Composite Extensions in Cabal

be met by any object oriented programming language. In addition, since the object oriented capabilities of inheritance and polymorphism are not strictly required; any language which provides support for abstract data types, such as CLU[Lis81] or Fortran-90[Bra90], can also be used as a suitable base language for a composite implementation.

This section presents *Cabal*[1], which is an extension of an object oriented Scheme for composites. We have developed language extensions for composites with Java [CC96], but our implementation is based on Scheme due to the lack of a stable Java environment at the time this work was started. Cabal is based on a variant of Scheme which

[1] A Cabal is a scheme with multiple participants.

has been extended with a variant of the Common Lisp Object System[Ste84], called Tiny-CLOS[Kic91].

4.1 Cabal Composite Declarations

Cabal provides a new type definition syntax in the form of the `define-composite` construct. This is a new definition which looks nearly identical to the standard `define-class` form. This is semantically equivalent to a standard class definition, except that (i) the default superclass is `<Composite>`, (ii) there is an implicit definition of the children collection inherited from `<Composite>`, and (iii) instances of the object will possess their own control threads and copies of the global state. The syntax for Cabal composite definitions, and an example are shown in figure 2. The example in part b of the figure shows a declaration for a composite list structure. Each node of the composite list contains a flag indicating whether or not it is a leaf node of the tree, and a list of values which are stored in the nodes. The `:accessor` form in the list specifies a function which will be used to access that field.

4.2 Cabal Composite Statements

The composite model requires three operations that can be performed within composite methods: children invocation, rendezvous accept, and rendezvous call. The language construct used for children invocation must specify the method to be invoked over the children collection, and a *combining function*, to be used to combine the collection of return values from the invocations into a single value. This children invocation construct subsumes all three of the primitive data parallel operations. *Mapping* is performed by simultaneously invoking the operation over the children collection. *Selection* is performed by the use of conditions within the mapped function. *Reduction* is performed by the combining function.

Data parallel calls are performed in Cabal using a function `par`, derived from the `map` function which is used for most structure based iteration in Scheme. The `par` statement has three parts: (i) the body, which contains a function to be executed on each child node, (ii) a combining function, which is used to combine the results from the parallel calls into a single value, and (iii) the composite node(s) over which the statement will iterate. The Cabal `par` statement is shown in part c of figure 2. This example demonstrates the use of `par` for parallelizing a tree based search mechanism.

The `par` statement searches all of the children of the node `p1` in parallel, merging the results using `or`, which will return the first non-null result from the child invocations.

The rendezvous statements must allow the initiator to specify the node and the operation to invoke, and the rendezvous accept must allow the receiver to specify a list of operations whose invocations are to be accepted. Rendezvous calls provide programmers with the capability of altering the conventional parallel control and data flow through the program. Through the use of rendezvous communication, programmers can implement directed acyclic graph structures using composite trees. To support rendezvous calling, two constructs are provided: (i) one to transmit a remote procedure call request, and (ii) one to accept a remote call on a particular procedure. The remote call request specifies the remote procedure to call, and blocks until the call is performed and the result returned. The remote call accept specifies a set of calls to accept, and blocks until a call on one of the specified procedures is received. Part d of figure 2 illustrates Cabal's rendezvous call constructs. This example shows a sender making a rendezvous call to `node` on the method named `procA`, with the argument list `(node 3 6)`, and the receiver accepting rendezvous calls on either its `procA` or `procB` methods.

4.3 Cabal Library Support

In addition to the language extensions that are needed to support the composite model, a small library of functions must be provided. These functions provide necessary functionality to the system without requiring any alterations to the base language. For Cabal, the composite support library consists of a collection of functions that operate on composite nodes. This library consists of the functions for node creation, insertion of a node into a parent node's children collection, removal of a particular child node from a parent's children collection, and reference to a particular node within a children collection. Since Scheme is a garbage collected language which does not provide programmer control over finalization, the programmer does not specify any destructor method.

4.4 An Example Cabal Program

In this section, the flexibility and simplicity of the composite model is illustrated by presenting an example of a simple program implemented using composites. Although the example could be implemented in explicit message passing languages, the resulting program would be more difficult to read and write and more difficult for a compiler to analyze and optimize. The example given in this section is kept as short and simple as possible in order to demonstrate the features of composite parallelism with a minimum of complexity. Programs that make use of composites will generally be much larger and more complicated, due to the inherent complexity of significant general purpose programs.

This section demonstrates the use of composite parallelism by a compiler for utilizing parallelism to increase the efficiency of the register allocation phase. In order to demonstrate the simplicity of translating a divide and conquer based algorithm into composite parallel form, the sequential version of this register allocator is shown in contrast to the composite parallel form.

Register allocation is a phase of an optimizing compiler that attempts to assign registers to variables in a segment of code in order to effectively utilize the machine's register set. The standard technique for register allocation builds an *interference graph* for the program, based on variable liveness[CAC$^+$81]. In an interference graph, each live range in the code segment is represented by a node, where a live range is defined as the region between the first assignment to a variable, and the last use of that variable in a section of the program. There is an edge between two nodes if the live ranges represented by the nodes overlap. The interference graph is colored. In the graph coloring, each color represents a register in the physical machine. If the minimal coloring has more colors than there are available registers, the compiler must insert spill code (i.e., memory loads and stores) in order to reduce the number of required registers. There are many algorithms for register allocation based on graph coloring[CAC$^+$81, CH84, BCT92]. Variations on the standard graph coloring algorithms include hierarchical graph coloring [CK91], coloring using clique separators[GSS89], and register allocation over the program dependence graph[NP94].

This phase of compilation tends to be very time consuming, and so is an excellent candidate for parallelization, provided an algorithm can be found which provides opportunities for parallel execution. Further, there is an efficient divide and conquer algorithm for performing graph coloring over straight line code segments. The limitation to straight

```
class Graph extends Object {
    void simple_color();
    void split_on_clique();
    void recolor_from(Graph g);
    void splitting_color(int depth) {
        if (depth == 1)
            simple_color();
        else {
            Graph subgraphs[2] = split_on_clique();
            // Sequentially iterate over the subgraphs, coloring
            // and combining
            for (int ind = 0; ind < 2; ind++) {
                subgraph[ind]->splitting_color(depth - 1);
                recolor_from(subgraph[ind]);
            }
        }
    }
}
```

(a) Sequential version

```
composite Graph extends Composite {
    void simple_color();
    void split_on_clique();
    void recolor_from(Graph g);
    void splitting_color(int depth) {
        if (depth == 1)
            simple_color();
        else {
            Graph subgraphs[2] = split_on_clique();
            // Insert the subgraphs as children nodes
            add_children(subgraphs);
            // Color children in parallel, and combine using the
            // combining body
            par (sg = ->splitting_color(depth - 1) ) {
                recolor_from(sg);
            }
        }
    }
}
```

(b) Parallel version

Figure 3: Comparison of code for a graph coloring register allocator

line code segments may seem limiting, but Zobel showed that it is often possible to transform a register interference graph into an interval graph corresponding to straight line code [Zob92a, Zob92b]. In the study done by Zobel, 33 out of 34 example programs were translatable to interval graphs.

The parallel algorithm used for this example was proposed in [MP95]. This technique uses *clique separators* to divide a register interference graph into two subgraphs, which can be colored independently. Once the two subgraphs are colored, their colorings can be combined using simple color renaming. The renaming is performed as a part of the `recolor_from` method, which applies color information from a child graph to the parent graph. This process can be used recursively to break the interference graph into smaller pieces, each of which can be colored simultaneously. This is based on a theorem by Tarjan which states that if two subgraphs connected by a clique separator can each be colored in n colors, then the original graph can also be colored in n colors by combining the colors of the subgraphs[Tar85].

Parallel and sequential versions of the code for this algorithm are presented in figure 3. The two versions are virtually identical, differing only in the body of the `splitting_color` method. Within this function, the differences are also quite trivial. The separate recursive calls to `splitting_color` are replaced with parallel invocation, and the calls to `recolor_from` in the sequential version are moved into the combining body of the parallel invocation.

5 Compiling Composites

The compiler for a composite programming system translates from a program containing composite constructs to an SPMD (Single Program Multiple Data) program in the base language which uses calls to a message passing library. This source to source translation process, illustrated in figure 4, consists of three phases: (i) programs are first translated into an intermediate representation, (ii) analysis and optimization is then performed on the intermediate form of the program, and (iii) the optimized intermediate form is translated into the target language (i.e., the base language with message passing calls).

The choice of an intermediate representation to use for this compilation process is highly language dependent, because the intermediate representation needs to capture the high level structure of the source language in an analyzable

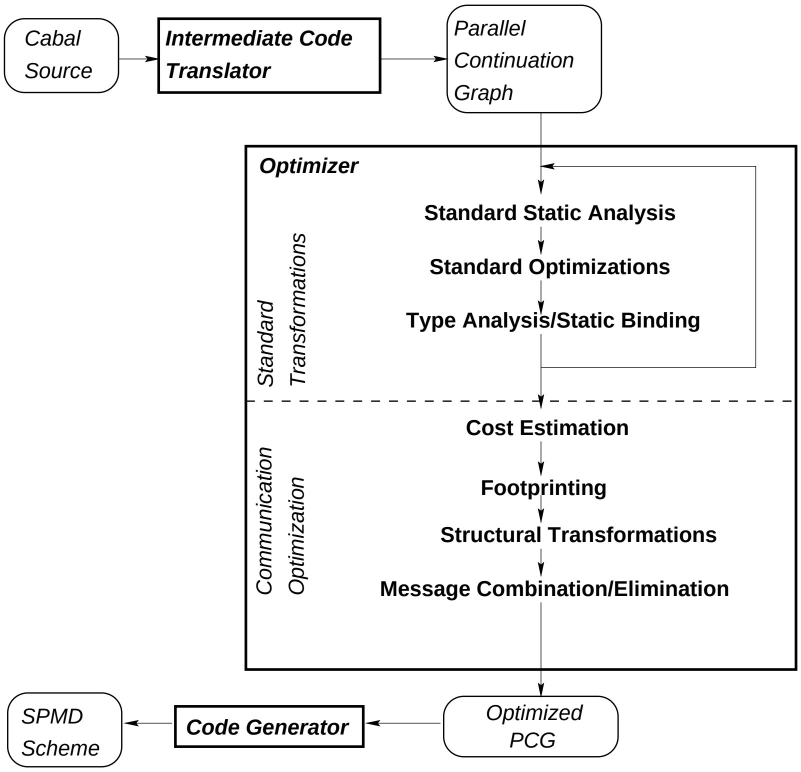


Figure 4: The Cabal compilation process

form. The implementation of composites in this paper is based on Cabal. For the Cabal compiler, a continuation passing style intermediate language called the *parallel continuation graph* was designed, to capture the dynamic and applicative features of the base language. Once the language is translated into intermediate form, a variety of optimizations are performed. These optimizations are *not* dependent on the base language being used, but rely on information which is generated using base language dependent analyses. The rest of this section presents the Cabal compilation system, beginning with a description of the parallel continuation graph, and then presenting in detail each of the analyses and optimizations performed by the Cabal compiler.

5.1 The Parallel Continuation Graph Representation

The PCG is based on an intermediate representation for sequential languages called the continuation passing style representation (CPS), used in applicative languages like Scheme and SML. A complete introduction to the history and usage of CPS based intermediate representations in compilers can be found in [App92]. The usefulness of the CPS form is examined in great theoretical detail in [**?**, **?**, **?**].

Briefly, CPS is a programming style in which: (1) all arguments to functions are either variables or manifest constants, (2) called functions never return, and (3) most functions take an extra parameter of function type, called a *continuation.*

Programs written in direct style (the term used in CPS literature for programs before the CPS translation), can be translated into CPS using a simple recursive procedure [App92, chapter 5]. Performing this simple translation results in a program in which all parameters, even function return addresses, are made explicit; every movement of values from one code segment to another, even the return of values from a function, is explicitly captured as parameter passing; and every dynamically created value has a statically unique[2] name. An example of this transformation, shown using Scheme syntax, is shown in figure 5. This example demonstrates the basic transformations that are performed in the translation to CPS. First, the parameter list of the function is altered to include a continuation parameter (figure 5(b), line 1). Next, the statements that returned a value are altered to call the continuation parameter with the return value as its argument (figure 5(b), lines 4 and 7). Next, each intermediate value is replaced by a local value declaration, so that all function parameters are either variables or manifest constants (figure 5(b), lines 2, 6, and 8). Finally, in place of the complex recursive call in line 4 of the original program, the CPS form must create a continuation argument that uses the return value of the recursive call to generate its own result. This is done by creating a function which performs the same computations as the code that uses the factorial function in the direct style program. This function is defined in the `let` statement in line 5 of figure 5(b). Once these changes are complete, the program is in CPS form. MARK FIX FIGURE

5.1.1 The PCG Intermediate Representation

Unfortunately, a conventional CPS form is not applicable to composite parallel program compilation, because it does not provide for representing multiple processes or inter-process communication. In order to address these deficiencies,

[2] *Statically unique* means that if you consider the program as a static entity, where there is no concept of looping or repetition, the result of each value creating statement has a unique name.

```
(define (fact n)
   (if (= n 0)
      1
      (* n (fact (- n 1)))))
```

(a) Factorial in direct style

```
(define (fact' n c)
   (let(t1 (= n 0))
      (if t1
         (c 1)
         (let((t2 (lambda
                     (let ((t3 (* n k))
                          (c t3)))))
                     (t4 (- n 1)))
            (fact' a t2)))))
```

(b) Factorial in CPS

Figure 5: An example of CPS translation

a new CPS based representation, which we call the *Parallel Continuation Graph (PCG)*, was designed.

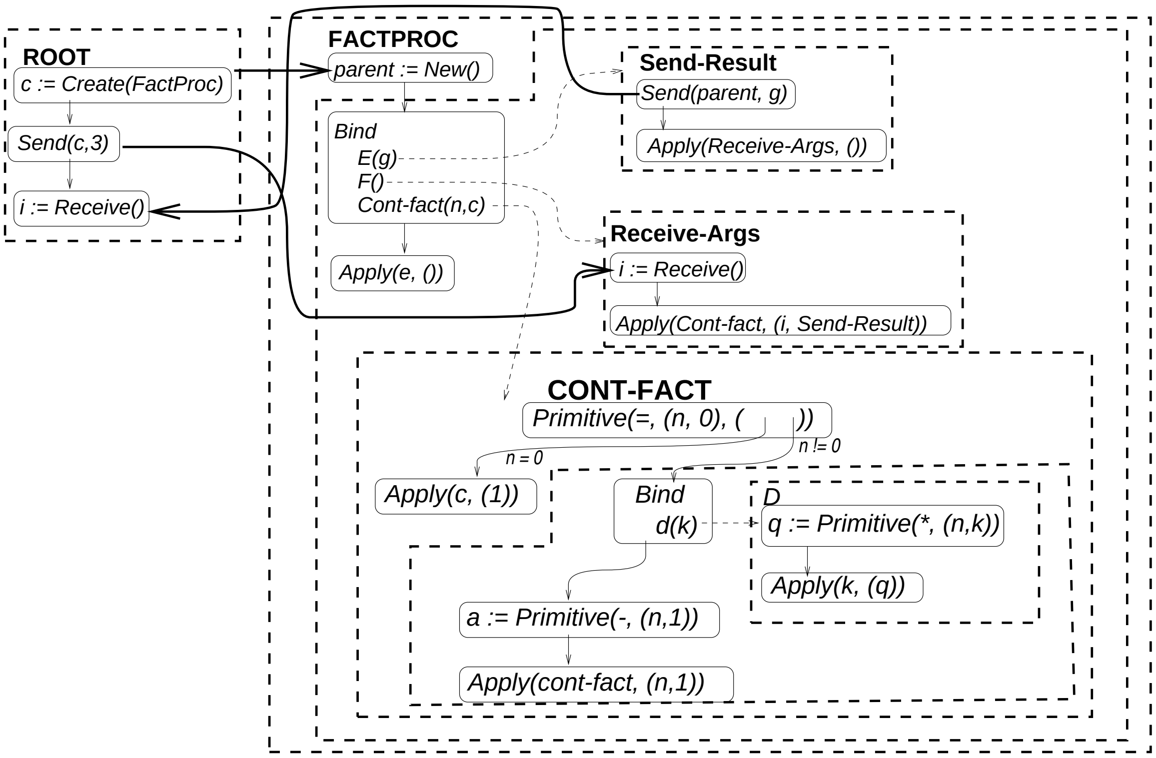
The PCG is a graph based CPS representation, which provides features for representing multiple control threads (processes), and communication between the various threads. Formally, a PCG consists of a set of nodes, each of which represents a single continuation expression (*cexp*). These nodes are interconnected by two different types of edges: continuation edges *(c-edges)*, and message edges *(m-edges)*. A c-edge from a cexp A to a cexp B denotes that A might call B as its continuation. C-edges are labeled with the conditions under which the edge will be followed. An m-edge from a cexp A to a cexp B denotes that A might send a message to B. (A and B must be in subgraphs that are disjoint if m-edges are not considered.) There are several different types of PCG nodes, for representing different programming constructs. The node types, and their parameters are as follows:

- `Apply(var, args)`: apply the continuation function stored in `var` to the arguments `args`.
- `Bind(funcs, vars, cont)`: create a new set of mutually recursive functions from `funcs`, assigned to the variables in `vars`, and call the cexp stored in `cont` in the scope of the new definitions.

```
(define root (lambda ()
   (let ((c (create-process factproc)))
      (send c 3)
      (let ((result (recv c)))
         ...))

(define (factproc parent)
   (define (receive-args)
      (let ((arg (receive parent)))
         (send-result (fact arg))))
   (define (send-result arg)
      (send parent arg)
      (receive-args))
   (define (fact n) ...)
   (receive-args))
```

(a) Example program in direct style



(b) PCG for the example program

Figure 6: A multiprocess Scheme program and its PCG

- Create(var, child-cexp, cont): create a new child process, which will call the cexp stored in child-cexp, store a reference to the communication port for the child process in the variable var, and call the cexp stored in cont in the scope of the new variable.
- New(var, cont): initialize a new process, storing a communication port for communication with its parent process in var, and call the cexp stored in cont in the scope of the new variable. (This node is always the root of the PCG for the child-cexp of a Create node.)
- Send(var, arg, cont): send the value of arg in a message to the communication port stored in var, and call the cexp stored in cont.
- Receive(portvar, var, cont): receive a value on the communication port stored in portvar, and assign it to the new variable var, and call the cexp stored in cont in the new scope.
- Primitive(op, args, vars, conts): perform a primitive operation op of the base language on the arguments stored in args, put the results into the new variables in vars, and call one of the cexps stored in conts. Primitive operations include operations such as comparisons, calls to direct functions not translated into CPS, object creations, and tests.
- Switch(var, cexp-list): use the value assigned to var as an integer index into the list of cexps cexp-list, and call the selected cexp. Switch nodes are essentially syntactic sugar for cascaded Primitive nodes containing comparisons. The Switch node was added to the PCG because it is easier for the analyzer to recognize a multiway branch represented by a single node than to discover a series of cascaded conditionals.

5.1.2 An Example PCG

A simple program that uses multiple processes and an equivalent PCG are shown in figure 6. In this figure, scope regions are illustrated with dotted boxes. The ROOT area is the translation of the root function from the Scheme program. The root function creates a new process, sends a message to that process, and then receives a result. In the PCG, this is represented by a CREATE node, to create the new process; a SEND node to send the message; and a RECEIVE node to receive the result.

The process created by the root function is shown in the subgraph labeled FACTPROC. FACTPROC starts with a NEW node to initialize the new process. Once the process is initialized, a BIND node, which corresponds to the nested definitions in lines 8 and 11 of the original Scheme source, is used to define a new scope containing the definitions of the receive-args, send-result, and cont-fact functions. Each of the defined functions has its own scope region, nested in the scope of the BIND node. Once the functions have been defined by the BIND node, the body of the factproc function is translated into an APPLY node which calls the CPS form of the receive-args function, and this node is used as the continuation to the BIND node.

The graphs for the `receive-args` and `send-result` functions are each simple; each performs a communication, and then calls a continuation. In the case of `receive-args`, the continuation is the CPS form of the `fact` function (which in turn is given `send-result` as *its* continuation), and in the case of `send-result`, the continuation is a call to `receive-args`. This creates a simple tail recursive loop, which causes `FACTPROC` to act as a server for the factorial function. It repeatedly receives a message containing an integer, using `receive-args`, computes its factorial, using `CONT-FACT`, returns the results back to the caller, and then starts over. The graph for `CONT-FACT` is a graphical form of the standard CPS `cont-fact` function shown in figure 5.

5.1.3 Translation of Cabal to the PCG

Programs written in Cabal are translated into the PCG form using a process derived from the standard CPS conversion algorithm. Most functions and statements are translated exactly as in the standard CPS algorithm. This section concentrates on the new statements that support composite programming, which are translated as follows:

For each **composite declaration** in the program, a composite PCG is generated, which is the root of a new process. Later in the translation, the PCG of each method of the composite type will be attached to this PCG as a subgraph. This PCG consists of a `New` node for the process, which calls as its continuation the `initialize` method for the composite class. The `initialize` method, in turn, is given the composite process body as its continuation. The process body consists of a message loop, like the one created by the `receive-arg` and `send-result` functions in figure 6. This message loop receives an argument list. The first argument is a number that will be used to select one of the method subgraphs, and the remaining arguments are to be passed to the selected subgraph.

For each **composite type**, a constructor method is automatically generated. It consists of a `CREATE` node which is attached to the root of the composite PCG. This `CREATE` node is embedded in the PCG representing the `make` method for the composite type.

Each **composite method** is translated into a PCG which will be attached to the composite PCG under a unique index. The generated subgraph is precisely the CPS form generated by the standard CPS translation algorithm for the method body.

For each composite method, an additional stub function is generated. This stub function consists of a PCG that contains the `Send` and `Receive` nodes for transmitting a call, and receiving the return value. This stub is generated to support separate compilation; with the stub in place, calls to composite methods from outside the immediate compilation unit will generate correct results without being modified by the compiler.

`par` **statements** are translated into graphs consisting of two loops. The first loop iterates over the children of the specified composite node. The body of this loop consists of a `Send` node, which transmits the `par` body method call to the children nodes. The second loop iterates over the children nodes again, this time using a `Receive` node to collect results, and calls the combining function on these results.

Rendezvous Call accepts are translated into a primitive node which calls a runtime function to register which method calls are to be accepted, followed by a `RECEIVE` node receiving the message, and finally an `Apply` node which calls a runtime function which selects the appropriate method to call.

Rendezvous Call initiations are translated into a `SEND` node sending the message for the call, followed by a `RECEIVE` node to receive the result of the call.

5.1.4 Critique of the PCG

Most constructs of Scheme are easily translated into the PCG. The control flow of the original Scheme program is maintained in a very clear, easy to analyze form. The dynamic features of Scheme are represented explicitly, in a clear, easily analyzable form. Communication events in the original program are represented explicitly, and relationships between communication events are maintained through the optimization process. The translation to PCG is $\mathcal{O}(n)$ in both time and space, where n is the number of expressions in the original Scheme program. We refer the reader to the dissertation for the proof[CC96].

5.1.5 Translating the PCG to SPMD Message Passing Code

Once the optimization of a program represented by the PCG is complete, it can be translated into SPMD code in Scheme very easily. Because Scheme is required by the Scheme standard to be time and space correct on tail recursion[**?**], there is no need to translate to direct style. Further, each PCG node type can be trivially translated to a Scheme expression:

1. `Apply` nodes are translated into normal function applications.
2. `Bind` nodes are translated into `letrec` expressions.
3. `Create` nodes are translated into calls to the runtime functions for node creation.
4. `New` nodes are translated into calls to the runtime process initialization function.
5. `Send` nodes are translated into calls to the runtime message transmission function.
6. `Receive` nodes are translated into calls to the runtime message receive function.
7. `Primitive` nodes correspond to primitive language operations of Scheme programs. Each type of primitive node is translated into an invocation of the appropriate primitive operation.
8. `Switch` nodes are translated into `case` expressions.

5.2 Static Analysis

The compiler for a composite programming language will make extensive use of dataflow analysis[ASU86] in order to safely optimize programs. The exact set of analyses will vary slightly depending on the nature of the base language. In the case of the compiler for Cabal, the compiler performs dataflow analysis to determine the following kinds of information:

Reaching Definitions. Reaching definitions is a standard dataflow analysis which determines which assignments to a variable may affect particular uses of that variable. In Cabal, this is particularly important, because a call to a generic function (member function) can actually call any of a number of different functions. This results in imprecision in the control flow information. Reaching definitions information is used in the Cabal compiler to produce a more precise control flow graph (by determining which functions may have been assigned to the target of an `App` node), and to reduce the number of *m-edges* in the PCG (by determining which PCGs may be the target of a particular `Send` statement).

Constant propagation. Constant propagation is a dataflow analysis which determines which variables in the program will contain constant values. Constant propagation is used by the Cabal compiler in order to determine when the target of a particular `App` node must be a particular function, as well as to increase the precision of other analyses.

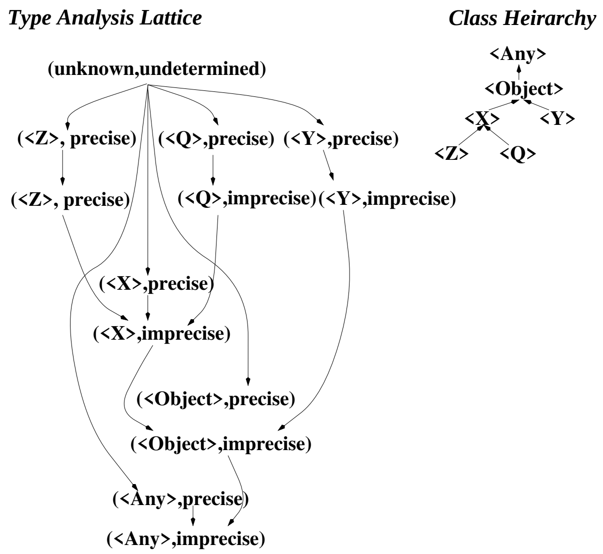


Figure 7: An example class hierarchy and type lattice

Variable liveness. Variable liveness analysis determines the region of a program over which each definition of a variable may be used. A variable is considered to be live at a particular program point if the variable is defined before that point, and then used before its next definition. Variable liveness is used by the Cabal compiler during the extended footprinting optimization, described in section 5.3.1.

5.2.1 Type Analysis in Cabal

In addition to the standard dataflow analyses, the Cabal compiler performs a dataflow based type analysis, based on a simplified form of type hierarchy analysis[DGC94]. This analysis attempts to derive precise types for objects in the program. This analysis is needed, because variables in an object oriented Scheme program do not have declared types (except for arguments to methods). This is complicated by the fact that the type system in object oriented Scheme consists of a type heterarchy (a directed acyclic graph of subtype relationships). In a type heterarchy, even when it is known that a variable has a particular type (as in the case of a method argument), the exact type of the value stored in that variable may actually be any subtype of the known type. In order to generate a precise control flow graph, it is highly desirable to know the precise type of program variables.

This is done by performing a dataflow analysis closely modeled on constant propagation[ASU86, Section 10.11].

Instead of assignments to a variable being treated as assignments of either constant or non-constant values, they are treated as assignments of a given type, which may be precise (if the exact type is known), or imprecise. These types are then propagated through the program in essentially the same manner as constant status in the constant propagation analysis.

To be more precise, type analysis can be described in terms of the standard dataflow analysis framework. A dataflow analysis framework consists of a lattice L of values to be propagated through the control flow graph, a set of transfer functions from L to L representing the flow of information through basic blocks in the control flow graph, and a binary meet operator $\leq$ on L to represent the confluence of flow at join points in the control flow graph[ASU86].

In type analysis, the lattice L is the set of all mappings from variables in the program to pairs of the form *(t,precision)*, where t is either a type or unknown, and *precision* is either precise (p), imprecise (i), or undetermined (u). The top of the lattice is the pair (unknown, undetermined), and the bottom of the lattice is the pair (any, imprecise). Let u and v be two members of the lattice L; that is, u and v both map each of the variables in the program to a pair in the form described above. Then $\rho = u \wedge v$ is given in the following table, where $\rho(x)$ is in terms of $u(x)$ and $v(x)$ for each variable x:

	(unknown, u)	(t,p)	(t,i)	(s,p), $s \neq t$	(s,i), $s \neq t$
(unknown, u)	(unknown, u)	(t,p)	(t,i)	(s,p)	(s,i)
(t,i)	(t, i)	(t,i)	(t,i)	(lub(s,t), i)	(lub(s,t), i)
(t,p)	(t, p)	(t,p)	(t, i)	(lub(s,t), i)	(lub(s,t), i)

The $\leq$ operator is defined by the property that $u \leq v \iff u \wedge v = u$. This implies that $(t,*) \leq (unknown, u)$, $(t,i) \leq (t,p)$, and if $lub(s,t) = s$, then $(s,i) \leq (t,*)$. A class hierarchy and corresponding type analysis lattice are shown in figure 7.

We give a "basis" for the set of transfer functions by describing the functions that represent single definition statements. The entire set of transfer functions can be constructed by composing functions from this basis set to reflect blocks with more than one definition statement.

The basis set $\mathcal{F}$ contains the following functions:

1. The identity function. If u is a mapping from variables in the program to pairs of type information, $I(u) = u$.

This function reflects a statement that has no effect on type information.

2. For each variable v, and type t, there is a function $f \in \mathcal{F}$, such that for every mapping u, we have $f(u) = v$, where $v(w) = u(w)$ for every w other than x, and $v(x) = (t, p)$. This reflects statements that generate precise type information about the variable being defined. For example, in the statement (set! p (make <A>)), (make <A>) returns a precise type.

3. For each variable v and type t, there is a function $f \in \mathcal{F}$, such that for every mapping u, we have $f(u) = v$, where $v(w) = u(w)$ for every w other than x, and $v(x) = (t, i)$. This reflects statements that generate imprecise type information about the variable being defined. For example, in the statement (set! x (cdr l)), x could be assigned either a value of type <Pair>, or a value of type <Null>. In this case, the assigned type information is the least upper bound of these two types, which is (<List>, imprecise).

In summary, the basic transfer functions reflect the possible type assignments that statements can have, and the identity function for the remainder of the variables. The variables for which the identity function does not apply have a function that is the same for all flow into the block.

The iterative dataflow algorithm will terminate on this dataflow problem. This type analysis is also a distributive problem. We refer the reader to the dissertation for the detailed proofs[CC96]. The fact that type analysis is distributive means that the results are *precise*, meaning that no statically available information is lost by the transfer functions used by the analysis. This precision makes the type analysis framework very valuable to the Cabal compiler, since it provides accurate results from the statically available information.

5.2.2 Cost Estimation Analysis

Cost estimation analysis examines the intermediate form of a program, and attempts to determine how long certain parts of the program will take to execute. This is useful in the compilation of Cabal because it provides the compiler with a metric that can be used to determine when certain structural transformations should be performed for improved execution. These estimates are not used in determining the *safety* of an optimization.

Determining cost estimate information precisely is clearly an impossibility, both for the standard theoretical reasons, as well as because of the fact that variations in the input data will cause variations in the dynamic behavior of the program. However, it is possible to come to an estimate of the performance behavior, which with the help of program annotations added by the programmer, should be sufficient for the targeting of optimizations. Since our fundamental approach is to allow the programmer and the compiler to act as partners in the process of generating an efficient program, the use of programmer annotations to allow the compiler to generate better code is an appropriate solution. These annotations take the form of *pragmas*, which are stylized comments that provide the compiler with extra semantic information. In the Cabal compiler, a pragma is a comment starting with "`;;-*-pragma`".

The cost estimation analysis used for Cabal compilation is based on counting the invocations of primitive Scheme procedures. The number of simple invocations is estimated using the following approximations:

- Simple Scheme procedures like + or `cons` are assigned a cost of 1.
- Invocations of loops or mapping procedures cost the estimate of the number of iterations times the cost estimate of the body. If the number of iterations can be determined by constant folding, then the exact number is used. Otherwise, if the loop is annotated, the annotation is used to generate the estimate. If no annotation is provided, then the estimate is based on a default value, which can be set by the programmer.
- Invocations of unknown functions have a cost which is set by pragma, with a default if no pragma is supplied.
- Sequences of statements have a cost which is the sum of the costs of the component sequences.
- The cost of a conditional is the sum of the cost of evaluating the condition, plus the *minimum* of the costs of the branches. Minimum is used because the compiler will optimize if the cost estimate of a method body is *too small* to be executed in parallel. Therefore, it is better to err by optimizing unnecessarily than to not optimize at all. This can be altered by pragma, by indicating that one of the conditions is the default branch (i.e., the branch most likely to be executed), in which case the cost of the default branch is used.
- The cost of an invocation of a known function is generated by applying the analysis, after constant propagation has been used to propagate information from any parameters of the invocation known to be constant. This process is applied on recursive definitions to a depth set by the user. (By default, the depth is only 1.)
- `par` statements are treated as having a cost estimate based on a single invocation of the body, combined with a cost estimate of communication time. The communication cost estimate is set by pragma.

For example, consider the code fragment shown in figure 8. This fragment contains the definition of a function which could be called by a composite method. It makes use of pragmas in lines 2 and 3 in order to inform the compiler of expected runtime costs. The cost estimate for this function called on unknown arguments would be computed as

```
(define (func x y z)
   ;;-*-pragma (cost y) 30
   ;;-*-pragma (length x) 10
   (map
      (lambda (a)
         (if (y a) ;;-*-pragma default false
            a
            (z a)))))
```

Figure 8: Example code for cost estimation analysis

follows: (1) The cost of the entire function is the expected number of iterations of the map function (set by pragma to have a default value of 10), times the cost of the body of the map. (2) The time estimate for the body of the map is the time estimate for the loop condition (set to 30 by pragma), plus the cost of the default case of the loop body. (3) The default case of the loop is set by pragma to be the false alternative. Since this function has no cost estimate, a default estimate is used. For this example, assume that the default cost is 50. (This default would be set by either pragma, or by the compiler, based on observed averages.) Thus, the total cost for the code segment is computed to be 10 * (30 + 50) = 800.

This cost estimation analysis is obviously not very precise. It is based on a large number of pragmatic, but still arbitrary, assumptions. However, for the purposes for which it is used, greater precision is not necessary. The simple analysis provided here is simply a framework that provides a programmer with the capability of specifying cost estimates where necessary. With cooperation from the programmer, the information generated by this simple algorithm can generate results as precise as necessary.

5.3 Optimizing Composite Programs

Once composite code has been thoroughly analyzed, the compiler can perform composite-specific transformations on the code, in order to enhance the parallel performance of the program. The Cabal compiler uses two methods to increase the performance of the program: communication optimization, to reduce the overall quantity of communication within the program, and extended footprinting, to reduce the effective cost of the communications that are performed.

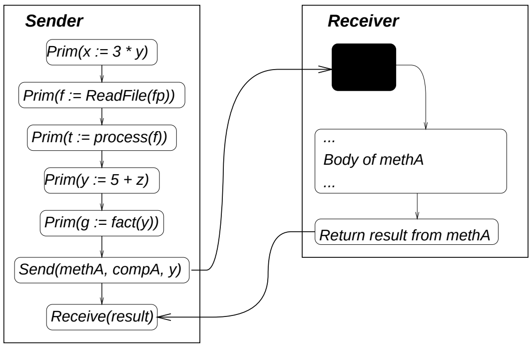
5.3.1 Extended Footprinting

Footprinting[LA94] is an optimization that reduces the effective cost of communication in parallel programs by overlapping computation with communication. This is done by separating a single communication event into two distinct events: an initiation and a completion. In Cabal, the initiation is the transmission of a request from the body of a `par` statement, and the completion is the receipt of the results of that call. The actual code transformation called footprinting is the movement of statements from before the initiation and after the completion into the region between the initiation and completion events. Statements can be moved within the region between the initiation and the completion so long as the statements reside within the *footprint* of the communication, which is the region of code between the definitions of variables transmitted in the initiation, and the use of variables defined by return values in the completion.

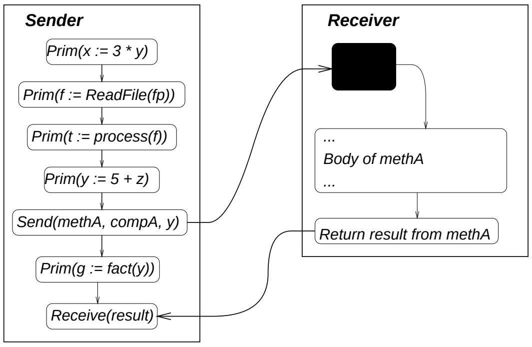
Within the PCG, it is possible to extend the standard footprinting optimization by actually copying code across m-edges of the graph. In conventional footprinting, when the statement which contains the definition of a parameter to the `Send` is reached, the system assumes that the limit of the footprint has been reached, and code motion transformations are halted. This is illustrated in figure 9b, where the footprinting optimization halts when it sees the definition of `y`, which is a parameter to the `Send` statement. The optimizer concludes that the limit of the footprint has been reached.

Extended footprinting allows the code to be moved past the `Send` node when there is exactly one m-edge exiting the `Send`, and when the definition meets the condition of *replicability*. A statement S is replicable if, and only if, S does not produce any side effects which will produce a visible effect on processes executing in a different address space. Examples of *non-replicable* statements include input/output statements (which affect the state of the file system, and are thus visible to other processes), interprocess communications, and operating system calls. In order to remain conservative, the composite optimizer assumes that a statement is *not* replicable unless static analysis can guarantee that it is replicable.

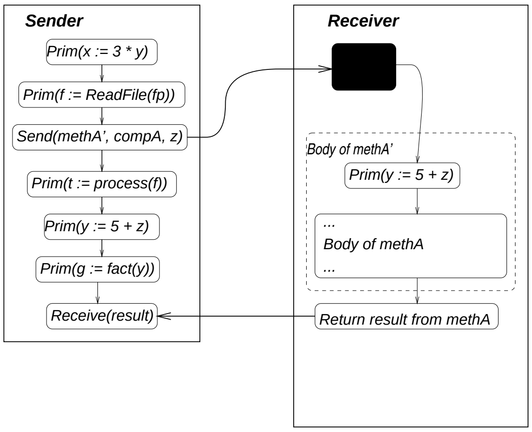
If a definition of a parameter to a `Send` is replicable, the definition node may be moved past the `Send` by performing the following steps:



(a) Before footprinting



(b) After standard footprinting



(c) After extended footprinting

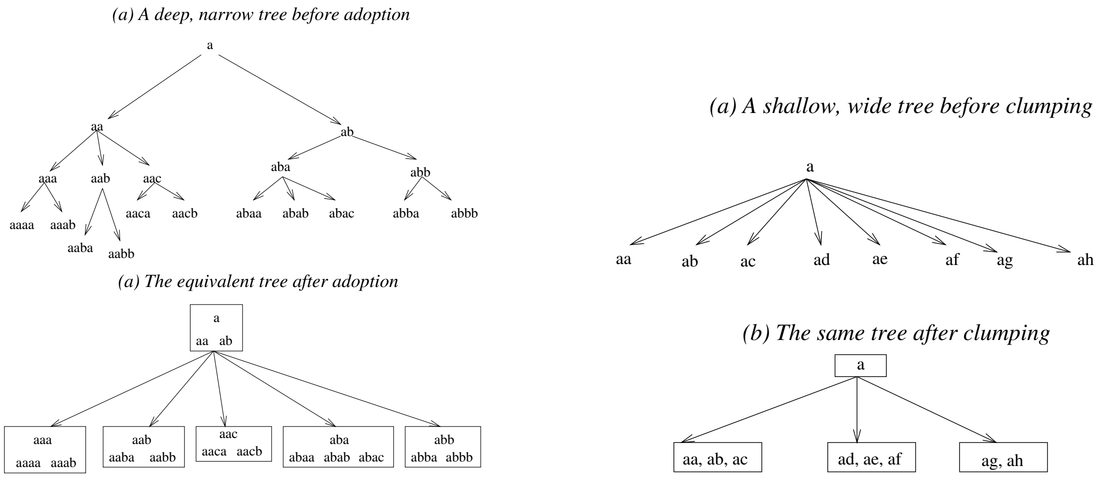
Figure 9: An example of extended footprinting on a composite PCG

1. Create a copy of the method to be called at the target of the `Send` node, and insert the copy under the composite PCG with a new ID.
2. Alter the parameter list of the `Send` by:
   (a) Replacing the method ID parameter of the `Send` with the ID of the new copied method.
   (b) Removing the defined variable from the `Send` node, and replacing it with the values used by the definition, but not already in the `Send`.
3. Alter the parameter list of the method PCG by removing the defined variable, and adding the same new parameters as in the `Send`.
4. Insert a copy of the definition statement at the beginning of the new method PCG.

Figure 9c illustrates how the footprint can be extended by altering the body of the sending method, and copying code across the communication link into the body of the target. Before extended footprinting is performed, the sending process (`SENDER`) sends a call to `compAs` method `methA`, with the parameter `y`. After standard footprinting in figure 9b, one statement has been moved into the region between the initiation and the completion. Once that statement has been moved, standard footprinting cannot proceed any further, because it cannot move the definition of a parameter of the `Send`. Part (c) of the figure illustrates how extended footprinting can continue to move code into the region between the initiation and completion by copying the definition of `y` into a new copy of `methA`, called `methA`, and altering the send statement. Once this statement has been copied, standard footprinting can continue to move code, until a non-replicable statement is reached.

5.3.2 Structural Transformations

Structural transformations are a class of optimizations developed to reduce communication by conceptually altering the communication structure of a composite tree. The structure can be altered by combining tree nodes, altering or combining messages, or collapsing levels of the tree. Structural transformations are used to allow programmers to use the natural algorithmic structure of their program, even when that natural structure is more fine grained than is desirable for executing composite programs. Rather than expecting programmers to go through the tedious and error prone process of manually rewriting their program to coarsen the grain size, the compiler and runtime system can use structural transformations to perform this process automatically.



(a) An example of node adoption

(b) An example of node clumping

Figure 10: Examples of tree transformations

The Cabal system uses a number of different structural transformations, including:

Child adoption. Adoption combines levels of the tree, by combining one level of children into the parent node, effectively reducing the number of tree levels by a factor of two, and eliminating all messages between the combined levels of the tree. An example of the adoption transformation is given in figure 10. In this example, each tree node before adoption is assigned a name based on a traversal of the tree. The figure after adoption shows the new tree, with each node containing the labels of the nodes in the original tree which were combined by adoption. Child adoption is actually performed whenever cost estimation analysis or pragma indicates that the grain size of a composite's methods is too small, and the number of children is statically known to be under some threshold. The cost estimate threshold and maximum number of children is dependent on the particular target architecture, and can be set either by pragma or compiler option.

Node clumping. Node clumping combines several sibling nodes into a single node. Method requests and result returns from the combined nodes are combined into single messages. Clumping is a desirable optimization on shallow, wide trees. An example of the clumping transformation can be found in figure 10. Clumping is performed when the cost estimate of a composite's methods is below a minimum threshold, and the number of children is statically known to be greater than some minimum value. Clumping improves performance by reducing the number of communications sent from a parent to its children, and increasing the quantity of computation performed in response to each message.

Left child elimination. Left child elimination embeds the leftmost child of a composite node into the composite node itself. This eliminates two messages (the request and the reply) for each parallel call. Left child elimination is nearly always desirable, and is always performed when adoption is not used.

Broadcast method call. Normally, a parent node sends a separate message to each of its children. Broadcast calls combine those messages into a single request, which is broadcast to all of the runtime system of all Cabal processes simultaneously. The single broadcast message contains the contents of all of the original requests. This eliminates a number of messages at the cost of replacing them with one larger, more expensive message. This is desirable when the latency of creating many messages is very expensive in comparison to the cost of sending a single larger message, and the cost of using a broadcast message is not significantly larger than using several directed messages[3].

The applicability of these optimizations can be very difficult to determine at compile time, because the information needed to make the decision of which, if any, structural transformation to apply is dynamic in nature, and often cannot be determined statically. In order to address this problem, these optimizations are performed in the Cabal implementation using a dynamic approach that relies on the runtime support system, which uses a variety of runtime techniques to achieve the same effect as performing the transformation on the source code. The effect of combining nodes in the graph can be achieved without source code alteration by ensuring that the combined nodes will be located on the same processor at runtime, and by instructing the runtime system to replace messages with direct method calls when the source and target of a message are on the same processor. When either pragmas or the cost estimation analysis of a composite indicates that adoption or clumping should be performed, the Cabal compiler generates code that directs the runtime system to lay out the composite in a fashion equivalent to the adopted or clumped structure. When static information is not available, the compiler allows the runtime system to use a default layout, and then perform dynamic optimization when necessary.

[3] This is often highly operating system dependent. For example, using identical runtime software, broadcast messages are less expensive than multiple directed messages on DEC Alphas, but more expensive than directed messages on Sun Sparcs running SunOS4.1.3.

In order to make this approach work, the Cabal compiler generates extra code to provide the runtime system with enough information to simulate these transformations. The types of generated code include:

Layout directives. The compiler can insert calls describing what kind of layout strategy to use for allocating nodes to processors. Structural transformations can be performed by specifying layouts that result in nodes residing on the same processor. For instance, adoption can be performed for a given composite tree by providing a layout directive that puts even level nodes of the tree on the same processor as their parent node.

Blocked messages. When a `par` call is performed, instead of sending each parent to child message separately, the compiler generates code that informs the runtime system that the following `send`s are a *block* that should be sent simultaneously. The runtime system does not actually send *any* of the messages until the block is complete. Once a complete set of messages is ready for transmission, the generated code calls a function informing the runtime system that the block is complete, and the runtime system can send the collected block of messages. Given a complete block, the runtime system can either merge messages being sent to nodes on the same target processor, or combine all of the messages into a broadcast, depending on how the runtime system is configured.

Single-use. When the compiler notes that a composite tree is never used after the parallel operation that created it (as occurs in the case of the register allocator example used earlier), the compiler can insert calls to the runtime system instructing it to *not* attempt to optimize the tree by altering layouts.

Frequent-use. When the compiler notes that a tree is used for multiple parallel operations without any tree alterations between calls, it can insert a call into the program informing the runtime system that it is highly desirable to rearrange the nodes of this tree to reduce communication. When the system is idle, the runtime system will rearrange nodes to either clump or adopt, depending on the observed dynamic structure of the tree.

Details on how the runtime system performs these transformations dynamically are given in the next section.

5.3.3 Message Combination and Elimination

The final class of communication optimization consists of eliminating redundant messages. Messages in Cabal composite programs can be eliminated under two different conditions. First, dataflow analysis can show that the return

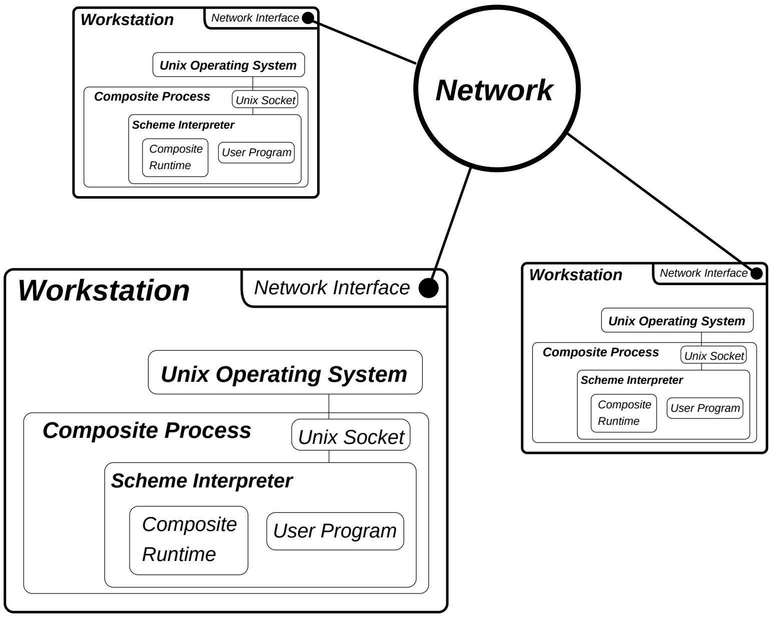


Figure 11: The structure of an executing composite program

value from a particular `par` call is never used. In this case, the messages that transmit return values back to the parent can be eliminated. Second, footprinting can move the communication initiation of a `par` statement to a position which is adjacent to another `par` statement. In this case, the communications from the two `par` statements can be combined. These transformations and their detection are straightforward.

6 Runtime Support for Composites

In most languages, the runtime support includes such things as runtime data representations, memory allocation, process and resource management, and other similar features[ASU86, chapter 7]. In addition to these standard runtime services, composite programs require several additional services in order to support the distributed execution of composite parallel programs. These runtime support services are highly dependent on the the language and target environment for which they provide support, and so the discussion in this section necessarily focuses on the implementation of one particular runtime system, the runtime support for Cabal on a network of Unix workstations.

The overall structure of a composite program at execution time is illustrated in figure 11, and the structure of

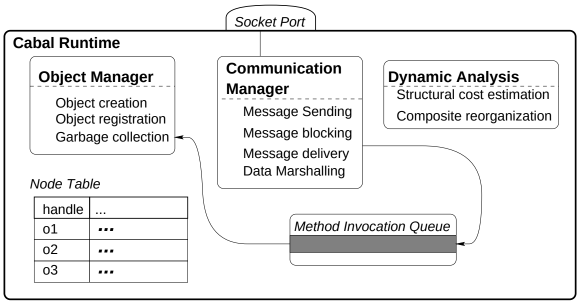


Figure 12: A diagram of the runtime structure of Cabal programs

the composite runtime system is illustrated in figure 12. On each workstation that is actively running the composite program, the system runs a copy of the Cabal program that has been linked with the Cabal runtime. The runtime consists of three main units: (i) the **object manager**, which controls the allocation, registration, and tracking of composite node objects; (ii) the **communications manager** which actually sends the messages to perform communications between composite nodes; and (iii) the **dynamic optimizer**, which dynamically gathers information about communication in the composite system, and uses it to reconfigure the layout of composites to reduce communication costs. These units make use of two shared data structures: (i) the node table, which maintains information about nodes, and relationships between them, and (ii) the method invocation queue, which is used to provide correct synchronization between the object manager and the communications manager.

6.1 Object Management in Cabal

The object manager (OM) of the runtime is responsible for actually maintaining the nodes of composite trees. It actually performs the creation and destruction of nodes, maintains necessary information about each composite node, destroys nodes when a tree is garbage collected, and ensures the structural validity of composite trees.

The node table is a data structure maintained by the OM to store the information concerning nodes in composite programs, including its parent, children, tree-handle, location, and message log. When a composite node is created, two objects are created: the actual composite node structure, and a composite node handle. The node handle is used to reference the object, and is the index into the node table to access information about the node. The node structure itself is only accessible through its node table entry. The node table is maintained consistently on all of the processors participating in the execution of a composite program. Any change to the node table will be broadcast to all composite runtimes.

In Cabal, Composite nodes are created by calling the `make` method for a composite class. The `make` method creates a *handle* for the root node of the system, to be used to reference the node. It then calls a runtime function that actually creates the composite node. The creation call may optionally include information about where to locate the node (i.e., the allocation to a specific processor). If location information is not provided, the initial location of the node is determined by using the *layout strategy* for the composite containing the node.

The layout strategy specifies how nodes are assigned to new locations. The layout strategies permitted by the Cabal implementation are round robin, block(n), and adopted(n). In round robin allocation, the allocator places one node on each processor, in sequence. Given P processors, the ith node of a tree will be placed on processor $i \ mod \ p$ Blocked allocation resembles round robin, except nodes are assigned to processors in blocks. During allocation, n consecutive nodes are created on a processor before going on to the next. (Given P processors, the ith node of a tree will be placed on processor $\lfloor i/n \rfloor \ mod \ P$.) Adopted allocation combines levels of the tree. Nodes of levels not evenly divisible by n are allocated on the same level as their parent. Nodes of levels evenly divisible by n are allocated round robin.

When an initial location for a new node has been selected, the runtime on the processor holding the root node of the tree broadcasts a message containing the handle for the node, and the location at which it should be created, and the other information required for the node table entry to the runtimes of all other processors. The runtime on the processor where the node should be created actually creates the object in response to this message, and all of the runtimes record the location of the node in their copy of the node table.

Composite nodes are also deallocated by the composite runtime. On the processor containing the root of a composite tree, the node table maintains a *weak pointer*[4] to the handle for the root of the tree. This weak pointer is intermittently checked. When it becomes invalid, this indicates that the composite tree is no longer referenced by the program, and should be collected. The runtime on the processor containing the root of the composite tree then broadcasts a message

[4] A weak pointer is a pointer which is ignored by the garbage collector in determining if a block of allocated memory should be collected.

to the other processors indicating that they should deallocate the composite nodes of that tree.

The OM performs synchronization management by using method registration. A composite node registers a list of available methods with the OM. When method invocations are received by the OM, the method availability list of the target node is examined. If the requested method is in the availability list, the method is performed. Otherwise, the method is inserted into the method invocation queue, where it will await a change in the availability registration of the target node. The system implements correct synchronization as follows. Before any method has been invoked on a node, the node's availability list consists of a list of all of its available methods. When one of those methods is invoked, *all* of the methods are removed from the node's availability list, so that any incoming calls will be deferred until the current invocation is complete. When the invocation executes a synchronous call receive, the methods listed in the receive statement are inserted into the node's availability list, allowing the execution of only those methods. Once a synchronous call send requests one of the appropriate methods, the node's availability list is again cleared. This simple scheme provides exactly the synchronization guaranteed by the composite model.

Since composite node structures can only be accessed through the node table, invocations in user code are actually performed on the node handles. These invocations call a OM function which invokes the method implementation on the actual node structure on the appropriate processor. The invocation function checks the node table to determine the processor location of the node. If the target node is located on a different processor than the invoking node, the OM calls a communication manager function to transmit a method invocation message to the target node. Otherwise, the method is invoked directly.

Method invocation messages are received by the communication manager, and inserted into a method invocation queue. Whenever new messages are inserted into the queue, the communication manager calls an OM function which examines each call in the queue, and performs the method if the target node has registered its availability to perform the requested method. Otherwise, the request is reinserted at the tail of the queue. Once all of the requests in the method invocation queue have been processed, the function returns control to the communication management system.

In addition to creating and destroying objects, the runtime is responsible for verifying the structural invariants of a composite tree. These structural invariants are: (1) A given node has exactly one parent node. (2) A given node is an element of exactly one composite tree. (3) A given node exists on exactly one processor. By using the information recorded in the node table, verifying these invariants is trivial. When a misuse of a composite library function results in the violating these invariants, the composite tree is in a condition which can easily result in deadlock, and an error is signaled using the error handling mechanism of the base Scheme interpreter.

6.2 Cabal Communication Management

The communication manager is a relatively simple part of the Cabal runtime. It provides functionality for actually passing messages between nodes, and for marshalling the arguments into transmissible form. In the Cabal implementation, the message passing functionality is provided by interfacing with the Unix socket interprocedure communication system. The actual system used to pass messages can be replaced by simply replacing this component with another message passing system, without requiring any alteration or recompilation of Cabal programs.

Data marshalling is the process of translating an object stored as a pointer graph in memory into a form which can be transmitted across a communication link. Marshalling in many languages is a complex process which requires extensive annotation of data structure declarations (for an example, see [AGG+93]). However, these problems are caused primarily by the use of pointers to untagged data types. In Scheme, the fact that every value is tagged with information that identifies its data type makes marshalling quite simple.

The Cabal marshaller is closely based on the interpretive marshaller of Concert/C. The Concert/C marshalling strategy uses *marshall plans* for marshalling. For each data structure, a marshall plan describes how to determine the type and size of each field of the structure, and the number of elements in any array. A data structure can be marshalled using its marshall plan by consulting the marshall plan for the type and size of each field of the structure, and using the information about that field to determine how to marshall it.

In Concert/C, the compiler can perform marshalling in two different fashions: interpretive marshalling, and compiled marshalling. In interpretive marshalling, the compiler emits code that generates marshall plans in executing code for each data structure, and then interprets those plans in order to marshall the structure. In Scheme, the lack of declared typing prevents the use of compiled marshalling, and intepretive marshalling requires no compiler support.

For each primitive data type, marshall plans are built into the marshaller, and for user-defined classes, the class object itself (which has a runtime representation) can be used as a marshall plan for objects of the type. Given these marshall plans for each data type, the marshaller works in exactly the same fashion as the Concert/C interpretive marshaller.

6.3 Dynamic Optimization in Cabal

For some composite programs, it is impossible to statically determine enough information about the structure of a particular type of composite tree to choose an appropriate layout. In this case, the system will initially lay out the composite round robin, and will dynamically analyze communication patterns to determine whether or not a different layout will achieve better performance.

The compiler can enable this dynamic analysis by inserting a call to the dynamic optimization system instructing it to begin logging of all communications between nodes of a particular tree. This will cause the system to log all send events to the node entries in the node table. The dynamic optimization system dispatches a low priority background thread, which will scan the node table, and compute the communication cost of each of the standard layouts for the communication performed thus far. If any layout produces a significantly smaller number of network communications, then the dynamic optimization system will instruct the object manager to relocate the nodes of the composite according to the improved layout.

7 Performance Results

Three test applications were compiled and executed using the Cabal implementation: a graph coloring system, matrix multiplication, and a sorting tree. Each of the applications was executed on a network of Sparcstations running Solaris on varying numbers of processors, with a variety of input sizes and tree depths.

7.1 Example 1: Matrix Multiplication

The first sample program used to test the Cabal compiler is a composite matrix multiply. The performance was

		Number of Processors			
Size	**Depth**	**1**	**2**	**4**	**8**
1000×1000	2	0.8	0.6	0.4	0.3
10000×10000	2	0.9	1.1	0.8	0.6
50000×50000	2	0.9	1.1	1.2	1.3
100000×100000	2	0.9	1.2	1.4	1.7
100000×100000	3	0.8	1.1	1.2	1.1

Figure 13: Speedups for Matrix Multiply

	Number of Processors			
Graph Size	1	2	4	8
100	1	1.1	1.4	1.4
200	1	1.1	1.6	1.6
300	1	1.2	1.8	2.1
400	1	1.2	1.9	2.4

Figure 14: Speedups for Graph Coloring

measured on a set of randomly generated matrices, with a tree branching factor of 8, with the runtime using blocking node allocation. The execution time was measured using system calls, and measured the time to perform the actual matrix multiplication, *excluding* the time to boot the system and initialize the matrices. The performance results are presented as a ratio of the execution time of a sequential matrix multiply written in Scheme against the time for the parallel execution using a varying number of processors. Performance was measured for a variety of matrix sizes, and for the largest matrix, with two different tree heights. The performance results are shown in figure 13.

In both cases, the performance of the composite program is worse on a single processor. This is expected, because the composite program does more work than a standard sequential matrix multiply. With additional processors, the composite program achieves modest speedup against the sequential program, with the two-level composite tree performing far better than the three-level tree. The maximum speedup measured in trials was approximately 2.

7.2 Example 2: Graph Coloring

The second sample program used to test the Cabal compiler is the register graph coloring system described earlier.

The performance was measured on several semi-randomly generated graphs of varying sizes,[5] with the runtime using round-robin allocation of nodes. The performance results were tested against a sequential version of the same graph coloring algorithm, and the results are presented as a ratio of the sequential performance against the performance of the composite program. The observed performance is shown in figure 14. The performance on this example was much better than matrix multiplication. With eight processors, the composite program achieved speedups between 2 and 3.

7.3 Example 3: Restructuring Sort Trees

This example program is based on a heuristic search tree example. A rebalance procedure rearranges values between the tree nodes so that the tree is balanced. The algorithm used to rebalance the tree is based on mergesort. When the rebalance method is called, each leaf node returns its list of values to its parent in sorted order. The parent merges the lists using the same merge procedure as mergesort, and then returns the list of values to *its* parent, until a full sorted list is assembled on the root node. Then the root node splits this list among its children, which split their list among their children, and so on.

This algorithm is a typical example of restructuring operations that programmers would want to parallelize. For a real world example of this kind of operation, consider a word processor with a tree based representation for a large document. One of the stages of preprocessing which must be performed before printing is performing size computations on all of the elements of the tree. This causes operations to be performed throughout the tree structure of the document, without ever generating a top-level return value from the procedure.

This sort of operation performs extraordinarily well in the Cabal system. The optimizer takes advantage of the fact that the restructuring operation has no combining function, and thus returns no value. Thus the optimizer can transform the method call to the restructuring operation into a single one-way message transmission from the sequential control process to the root of the tree, and then immediately returns. The tree restructuring process continues in the background, with no observable time delay or performance degradation in the sequential control process.

[5] By semi-randomly, I mean that the graphs where generated by creating a collection of random graphs, and then manually generating cliques to join them together.

The actual performance time of the restructuring process was difficult to measure, because with this optimization, there is no way to determine when the process completed according to the system clock on the processor running the sequential control process. The effective performance time was negligible, because the parallel call immediately returned. The only way to observe the actual time to perform the restructuring is to perform other operations on the composite tree, which may block while waiting for the restructuring to complete. However, in the trials, no measurable delay was detected.

7.4 Discussion

The Cabal system was able to achieve modest performance improvements on two of the tested programs, and was able to completely mask the entire execution time of the third, producing an effectively instantaneous completion. In the first two examples, the performance was not substantially improved by the optimizer, but the masking of the execution time of the third was produced exclusively by the optimizer taking advantage of a property of the program.

In all three examples, the implementation of the program as a composite parallel program was not significantly more difficult than the implementation of a sequential equivalent. For this set of examples, our primary goal of achieving modest speedups with very little effort was achieved.

On the tested examples, the performance of the Cabal compiler's footprinting optimization was disappointing. Extended footprinting failed to provide any measurable performance improvements on the tested programs. This poor performance of the footprinting optimization was due to two factors. First, the Cabal compiler currently does not examine entire programs. Due to limits of the implementation, the compiler only examines the fragments of programs containing composite code. The lack of information about the non-composite code results in the footprinting system not having sufficient information to move large sections of code. On average test programs, the footprinting system was able to move only one statement before reaching a barrier. Second, Programs in applicative languages like Scheme tend to consist of many very small functions. Rather than having large blocks of sequentially executed statements, programs consist of cascades of function calls. There are very few statements in positions where the footprinting system could move them.

Manual analysis of Composite Java programs suggests that footprinting will succeed far better in an imperative language like Java. Java programs tend to have large blocks of sequential instructions, which provide significant opportunities for footprinting.

Structual optimization provided good results in the matrix multiplication example. The runtime supoprt which performed clumping without source code modification was able to exactly match the performance of code which has clumping performed manually in the source code. In the case of the register allocation example, the algorithm performed so well without any optimization that structural optimization was not needed.

8 Summary and Future Work

The main goal of this work has been to produce a *portable* programming model which will allow programmers to take advantage of available, cluster parallelism to achieve modest performance improvements in computationally intensive general purpose applications, without significantly increasing the complexity of the programming task. In this paper, we have presented our work on the design and implementation of the compiler and runtime system for Cabal, an implementation of the composite parallel programming model. The prototype implementation of the Cabal system was used to implement several simple test programs, in order to test its efficiency and ease of use. The composite programming model allowed parallel programs to be implemented quite easily, without significantly increasing the complexity of the programming task. The Cabal compiler produced executables that ran with modest performance improvements on workstation clusters.

The composite model provides support for data parallel style programming for general purpose applications. However, many general purpose applications may also contain computations that are better expressed using task parallelism. Future work on the model includes integrating task parallelism with composite-style data parallelism. In addition, more composite programs need to be implemented.

References

[AGG$^+$93] Joshua Auerbach, Arthur Goldberg, Germán Goldszmidt, Ajei Gopal, Mark Kennedy, and James Russel. *Concert/C Manual: A Programmers Guide to a Language for Distributed C Programming*. IBM T.J. Watson Research Center, July 1993.

[Agh86] G. Agha. *Actors: A model of concurrent computation in distributed systems*. MIT Press, 1986.

[AH92] G. Agha and C. Houck. Hal: A high level actor language and its distributed implementation. In *Proceedings of the 21st International Conference on Parallel Processing*, pages 158–165, 1992.

[App92] A. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[ASU86] Alfred Aho, Ravi Sethi, and Jeffrey Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986.

[BC90] G. Blelloch and S. Chatterjee. Vcode: A data parallel intermediate language. In *Proceedings of the 3rd Symposium on the Frontiers of Massively Parallel Computation*, pages 471–480, 1990.

[BCT92] Preston Briggs, Keith D. Cooper, and Linda Torczon. Rematerialization. In *Proceedings of the ACM SIGPLAN 92 Conference on Programming Lang uage Design and Implementation*. ACM, June 1992.

[Ble90] G. Blelloch. Nesl: A nested data parallel language. Technical Report CMU-CS-92-103, CMU, 1990.

[BMT92] D. Berry, R. Milner, and D. Turner. A semantics for ML concurrency primitives. In *Proceedings, 19th Annual Symposium on Principles of Programming Languages*, 1992.

[Bra90] W. Brainerd. *Programmer's Guide to Fortran 90*. McGraw-Hill Book Co., 1990.

[CAC$^+$81] Gregory Chaitin, Marc Auslander, Ashok Chandra, John Cocke, Martin Hopkins, and Peter Markstein. Register allocation via coloring. *Computer Languages*, pages 47–57, January 1981.

[CC96] Mark C. Chu-Carroll. *Programming Language and Compiler Support for General Purpose Parallelism*. PhD thesis, Department of Computer and Information Sciences, University of Delaware, August 1996.

[CD95] A. Chien and J. Dolby. Parallel programming using c++. In *Chapter ICC++: A C++ Dialect for High Performance Parallel Computing*. MIT Press, 1995.

[CH84] Frederick Chow and John Hennessy. Register allocation by priority-based coloring. In *Proceedings of the ACM SIGPLAN 84 Symposium on Compiler Construct ion*. ACM, 1984.

[Chi92] Andrew Chien. *Concurrent Aggregates: Supporting Modularity in Massively Parallel Programs*. MIT Press, 2nd edition, 1992.

[CK91] David Callahan and Brian Koblenz. Register allocation via heirarchical graph coloring. In *Proceeedings of the ACM SIGPLAN '91 Conference on Programming La nguage Design and Implementation*. ACM, 1991.

[CP94] M. Carroll and L. Pollock. Composites: trees for data parallel programming. In *Proceedings of the 1994 International Conference on Computer Languages*, pages 43–54. IEEE, 1994.

[DGC94] J. Dean, D. Grove, and C. Chambers. Optimizationn of object oriented programs using static class hierarchy analysis. Technical Report UW-CSTR94-12-01, University of Washington, 1994.

[FCO90] J. T. Feo, D. C. Cann, and R. R. Oldehoeft. A report on the sisal language project. *Journal of Parallel and Distributed Computing*, 10:349–366, 1990.

[GKSK94] A. Gursoy, S. Krishnan, A. Sinha, and L. Kale. *The Charm(4.3) Programming Language Manual*. University of Illinois, 1994.

[GL92] Dennis Gannon and Jeng-Kuen Lee. Object-oriented parallel programming: experiments and results. In *Proceedings of Supercomputing '91*. IEEE, 1992.

[Gri93a] A. Grimshaw. Easy to use object-oriented parallel programming with mentat. *IEEE Computer*, pages 39–51, May 1993.

[Gri93b] Andrew S. Grimshaw. The mentat computation model: data-driven support for object-oriented parallel processing. Technical Report CS-93-30, University of Virginia, 1993.

[GSS89] Rajiv Gupta, Mary Lou Soffa, and Tim Steele. Register allocation via clique separators. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Lan guage Design and Implementation*. ACM, 1989.

[HB77] C. Hewitt and H. Baker. Actors and continuous functionals. In *Proceedings IFIP Working Conference on Formal Description of Programming Concepts*, pages 367–387, 1977.

[HQ92] P. Hatcher and M. Quinn. *Data parallel programming on MIMD computers*. MIT Press, 1992.

[Hud92] P. Hudak. Mutable abstract datatypes -or- how to have your state and munge it too. Technical Report YALEU/DCS/RR-914, Yale University, 1992.

[JH93] M. Jones and P. Hudak. Implicit and explicit parallel programming in haskell. Technical Report YALEU/DCS/RR-982, Yale University, 1993.

[Kic91] Gregor Kiczales. *The art of the metaobject protocol*. MIT Press, 1991.

[LA94] K. Landry and J. Arthur. Achieving asynchronous speedup while preserving synchronous semantics: an implementation of instruction footprinting in linda. In *Proceedings of the 1994 International Conference on Computer Languages*, pages 55–63. IEEE, 1994.

[Lis81] Barbera Liskov. *CLU Reference Manual*. Springer-Verlag, 1981.

[LRV92] J. Larus, B. Richards, and G. Viwsanathan. C**: A large grain, object-oriented, data-parallel programming language. Technical report, University of Wisconsin, Madison, 1992.

[MFL93] S. Murer, J. Feldman, and C. Lim. psather: Layered extensions to an object-oriented language for efficient parallel computation. Technical report, ICSI, University of California at Berkeley, 1993.

[MP95] Christine Makowski and Lori Pollock. Efficient register allocation via parallel graph coloring. In *Proceedings of the ACM Symposium on Applied Computing, Programming Languages Track*, February 1995.

[Nik93] R. S. Nikhil. An overview of the parallel language id. Technical report, DEC, Cambridge Research Lab, 1993.

[NP94] Cindy Norris and Lori Pollock. Register allocation over the program dependence graph. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Lan guage Design and Implementation*, 1994.

[RSW91] M. Rosing, R. Schnabel, and R. Weaver. The dino parallel programming language. *Journal of Parallel and Distributed Computing*, 13(1):30–42, Sept 1991.

[Sch92] H. Schmidt. Data-parallel object-oriented programming. In *Proceedings of the Fifth Australian Supercomputing Conference*, pages 263–272, 1992.

[Ste84] Guy Steele. *Common Lisp: the Language*. Digital Press, 1984.

[Tar85] R. E. Tarjan. Decomposition by clique separators. *Discrete Mathematics*, (55):221–231, 1985.

[TMC90] *C* Reference Manual*. Thinking Machines Corp., 1990.

[Zob92a] Angelika Zobel. Program structure as a basis for parallelizing global register alloc ation. In *Proceedings of the International Conference on Computer Language*. IEEE, 1992.

[Zob92b] Angelika Zobel. *Program structure as a basis for the parallelization of global compi ler optimizations*. PhD thesis, Department of Computer Science, Carnegie Mellon University, May 1992.

[Zos93] M. Zosel. High performance fortran: an overview. In *Proceedings, IEEE COMPCON '93*, pages 132–136. IEEE, 1993.