Compilation to a Queue-Based Architecture

by

Benjamin Mead Vandiver

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science at the MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2002

© Benjamin Mead Vandiver, MMII. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part.

Certified by.. ........................................

Tom Knight
Senior Research Scientist
Thesis Supervisor

Accepted by.............. ...................................

Arthur C. Smith
Chairman, Department Committee on Graduate Theses
Compilation to a Queue-Based Architecture

by

Benjamin Mead Vandiver

Submitted to the Department of Electrical Engineering and Computer Science on May 23, 2002, in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science

Abstract

With processor speeds improving exponentially while the speed of light remains constant, the latency associated with remote access will become a more serious problem. The Aries Decentralized Abstract Machine (ADAM) addresses this issue with a focus on simple hardware methods of communication, and migration of computation and data. This thesis presents a programming language and compiler that capitalize on these strengths. The People programming language supports linking computational elements together using first-class stream values, all set in an object-based framework. Due to the unique structure of the ADAM, the compiler uses a number of new techniques and optimizations when compiling People programs to ADAM assembly. The compiler output was tested on a cycle-accurate ADAM simulator that simulates processors, memory, and network. Programs that use streaming constructs have demonstrated performance double that of the naive non-streaming implementation.

Thesis Supervisor: Tom Knight
Title: Senior Research Scientist
Acknowledgments

I would like to thank Tom Knight, Andrew Huang (bunnie), Jeremy Brown, and JP Grossman, and the other members of the Aries research group at the MIT AI Laboratory. Bunnie deserves particular mention, as he is the designer of the architecture that my thesis targets. We had many long discussions about the merits of various software models; from which I drew many of the conclusions that my thesis is based on. His ADAM simulator was essential in producing all of my numerical results.

My girlfriend Jen McKeehan has been an essential part of support network that allowed me to work long long hours. Her support has been invaluable.

My sister, Amy Vandiver, helped me stay on track by meeting with me every Friday for months for our Thesis Lunches. She has been encouraging the whole time and provided a great peer review.
Contents

1 Introduction 11

2 Description of ADAM 13
  2.1 Queues .................................................. 15
  2.2 Lazy Instructions ........................................ 15
  2.3 Types ..................................................... 15
  2.4 Capabilities .............................................. 16
  2.5 Multithreading .......................................... 16
  2.6 Queue Mappings ......................................... 16
  2.7 Migration and Load Balancing ............................. 17

3 Related Work 19
  3.1 Fine-grain parallel Architectures .......................... 19
    3.1.1 Decoupled Access/Execute ............................ 19
    3.1.2 Imagine .............................................. 20
    3.1.3 J-Machine ............................................ 20
  3.2 Parallel Languages ...................................... 21
    3.2.1 Objects ............................................... 21
    3.2.2 Streams ............................................... 22

4 Language Design 25
  4.1 Objectives ............................................... 25
  4.2 Design of Couatl ........................................ 25
  4.3 People Overview ........................................ 26
    4.3.1 Method Call .......................................... 26
List of Figures

2-1 Programming model of ADAM ...................................... 14
4-1 Method Call Example .................................................... 27
4-2 Simple Class and Interface ............................................. 29
4-3 Streaming Example Program ........................................... 32
5-1 Method lookup code for ArrayList from figure 4-2 .................. 42
5-2 Localization of sinks .................................................... 49
6-1 Distribution tree built by TreeNode ................................. 52
6-2 Comparison between Hardware and Software queues ............... 54
6-3 Layout of streaming matrix multiply .................................. 55
B-1 Flowgraph of TreeNode, autogenerated by compiler ............... 72
B-2 Sample compiler-generated webpage of package Activation ....... 72
Chapter 1

Introduction

In massively parallel supercomputers, notions of locality and data transmission are very important. The sheer size of the machine and its data network make transmission speeds slow in comparison to processor speed, thus encouraging computational designs which focus on managing communication. The Aries Decentralized Abstract Machine (ADAM)[4] is such an architecture. It manages communication by providing explicit user-level assembly instructions for sending data between processes. It reduces communication by employing a number of migration and load-balancing schemes which serve to bring communicating parties closer together. Finally, it hides communication cost by providing transmission buffering through the use of architecturally-visible queues and latency hiding through hardware multithreading.

However, without software designed to take advantage of these features, the ADAM architecture is useless. The programmer needs a system that guides them into expressing their programs in such a way that the tools can produce code that leverages the advantages of the ADAM. The system should comprise both a language that encourages locality and simple inter-process communication and a compiler that compiles the language into ADAM assembly.

This thesis presents a language and compiler that meet the above requirements. The language is object-based to manage locality, and includes first-class streams to allow simple inter-thread communication. The compiler is first presented as an existence proof that the ADAM architecture can be targeted by a compiler. This result is interesting due to a number of fundamental differences between ADAM and more conventional architectures.
Secondly, the compiler takes advantage of ADAM’s strengths in order to produce efficient output code.

This thesis opens in chapter 2 with a presentation of the key pieces of the ADAM architecture. The architecture is different from most modern architectures, and thus provides a host of pitfalls and possibilities for language design and implementation. However, only features pertaining to the compiler are presented.

Background on the architecture being targeted and the design of similar parallel programming languages is presented in chapter 3. A number of the ideas which similar architectures are built around can be simulated by appropriate ADAM code. Similarly, People drew many of its features from other parallel languages.

Next, the design of the People language is discussed in chapter 4. The chapter opens with a critique of the predecessor language to People, called Couatl. An outline is given of the syntax and semantics of the People. Example source code is presented.

Chapter 5 dives into the implementation of the compiler and the language. A brief moment is devoted to implementation of Couatl. The chapter then presents the details of how each language construct is compiled down to the ADAM assembly level. Finally, the optimizations performed by the compiler are described.

Chapter 6 describes the results obtained from running People programs on the ADAM simulator. The usability of People is discussed, and a number of relative benchmarks presented to show performance improvements from the various optimizations.

Finally, chapter 7 summarizes the results, draws some take-home messages, and details what future directions could prove fruitful.
Chapter 2

Description of ADAM

In order to understand the needs of the language and compiler, the architecture they target must be explored. The Aries Decentralized Abstract Machine is a specification for a programming model of a parallel computing environment. The model consists of an instruction set and a context in which the code is executed. Each ADAM processor node is paired with a memory node, and the whole machine may encompass from one processor-memory pair to millions of pairs. The programmer sees a non-uniform access time shared memory space, and no data caches are allowed. Each processor core is relatively simple, with almost none of the modern instruction-level parallelism extraction complexities. The processor exploits thread-level parallelism by keeping a large number of runnable thread contexts in core and switching between them to avoid stalling. Lastly, fast migration and load-balancing schemes serve to correct for resource bottlenecks and excessive latency issues.

An executing thread uses 128 arbitrarily deep queues instead of registers. All instructions work on queues, including a set of operations that transform a queue into a method of communicating between threads and memory.

When compiling to ADAM, many of its features require treatment which is different from a regular architecture. More specifically the features that require attention are: the use of queues instead of registers, lazy instructions, typed values, capability-based addressing, hardware multi-threading, and hardware-supported intra-thread communication.
Figure 2-1: Programming model of ADAM
2.1 Queues

The most noticeable difference is the substitution of queues for registers; a change that has far-reaching effects. The queues are arbitrarily deep, but the ADAM implementation presumably switches from a fast implementation to a slow one once a certain depth is exceeded. Whereas registers support only two operations: read and write, queues support four: enqueue, dequeue, copy, and clobber. A copy operation copies the value at the head of the queue, leaving the queue state unchanged. A clobber changes the value at the tail of the queue if the queue contains any elements and acts like an enqueue if the queue is empty.

By using nothing but copy and clobber, the queues will behave like registers, but this does not take advantage of the strength of queues. The standard implementation of queues is a named-state queue-file, which acts like a kind of cache. Most importantly, empty queues do not take up any space in the Qfile, so code that keeps the maximal number of queues empty reduces competition for space in the Qfile. However, to use enqueue/clobber and dequeue/copy correctly requires that the compiler understand the data flow of the program being compiled.

2.2 Lazy Instructions

Some instructions take a variable number of cycles to complete and they do not appear to be atomic at an assembler level. The memory allocation and thread spawn instructions are the key ones that have this property. Care must be taken to ensure that the values written upon the completion of these instructions are not overrun by subsequent writes to their output queue. For example, a thread spawn dropping the new context into q6 followed by enqueuing the number 1 onto q6 may result in either order of values. Placing the result of one of these operations directly into a mapped queue may cause similar ordering issues.

2.3 Types

The architecture maintains an extra set of bits for each value which encode its type. These bits are checked when operations are performed, and exceptions are generated when the type does not support the given operation. For example, using a floating point add on a character will yield a type exception. The type bits can also encode type information like
value is immutable, uncopyable, read-only, etc.

2.4 Capabilities

The ADAM architecture uses a capability-based addressing scheme. A capability is a guarded pointer that includes the base and bounds of the memory range that the holder of the capability is allowed to access. Capabilities can only be generated by the system or by sub-dividing an existing capability. Any value marked as a capability is only permitted to be operated on by a small set of operations, and furthermore, any attempt to modify the base and bounds encoded in the capability generates an error. When a thread wishes to access memory, it sends the capability and an offset to the memory system. If the offset falls outside the bounds of the capability, a protection error is raised. This neatly catches all buffer-overflow issues. Since capabilities are unforgeable, code can only access memory it has been given access to. Thus, globally accessible storage is hard to implement.

2.5 Multithreading

Another important feature is the hardware simultaneous multi-threading (SMT) support provided by the ADAM processors. The processor core contains a number of threads and can switch between running one and running another very quickly. When a thread blocks, waiting for either a remote thread or memory, the processor just runs another thread. With a large enough set of threads, the processor spends very little time stalled waiting; it always has something to do. Thus, the objective of a compiler is to supply the architecture with enough threads to keep each processor busy.

2.6 Queue Mappings

Queue mappings are the links between threads. By using a MAPQ or MAPQC instruction, a queue in the local thread context may be mapped to any queue in another thread context. When data is written into the locally mapped queue, the network interface intercedes and sends the data to the destination of the queue mapping. All direct inter-thread communication occurs in this manner: map a queue, write some data, and finally unmap the queue. Since a queue mapping can only be constructed by using the capability of the
target thread, mappings can only be constructed to threads the mapping thread has “heard about.” Additionally, the MAPQ instruction accepts a number for the queue to map to, so any arbitrary queue can have data delivered to it by using the thread context id and the queue number. This enables methods to return values to any queue (and thus variable) in a thread without any synchronous action by the receiving thread. These queue-mappings establish an abstraction of communication because the mapped queue is used the same way, regardless of whether the target of the mapping is a local or remote thread. Because of this separation, queue-mappings can be redirected relatively easily by the migration and load balancing methods.

One limitation of the queue-mapping strategy is that synchronization is provided on a single-word basis. Two threads, both with mappings that target the same destination queue in the same thread context, have no guarantees about the relative ordering of the values arriving in the destination queue. That is, there is no way to ensure that if process A writes 1 then 2 to process C while process B writes 3 and 4 to the same queue in process C, that the ordering 1, 3, 2, 4 is never possible. Software techniques are required for this task. The architecture does provide one bit of functionality to make this task easier, which is the ability to collect the sender thread contexts for each word delivered to a given queue.

Additionally, threads interact with memory through mapped queues. The MMS, MML, and EXCH instructions map queues to memory, providing a conduit for addresses and data to be sent to memory and data to return. The queues provide an explicit load/store buffer for each thread, and some under-the-hood interlocks serve to ensure ordering. Addresses sent to the memory system may involve two values: a capability and an offset. The memory system remembers the last capability input from each address memory mapping. A non-capability sent as an address specifies an offset from the last capability supplied. Thus, performing many accesses into the same capability in a row reduces the number of capabilities that need to be sent to the memory system.

2.7 Migration and Load Balancing

The ADAM architecture assumes that the implementing machine can be very large. Thus, adjusting thread and memory allocation to reduce communication latency is very important. The machine supports automatic migration of threads and memory. The migration system
watches remote accesses and can intercede to move either the accessor closer to the data or the data closer to the accessor. The ADAM also provides a work-stealing load balancer which moves extra computation out of overloaded nodes and into less crowded neighbors. To aid the machine in thread/data placement, the allocation primitives support a set of hints. These hints describe the tolerable latency from creator to the new object, the bandwidth, and the expected amount of resources to be consumed by the new object. Accurate usage or analysis of the effectiveness of these metrics is beyond the scope of this thesis.

With these features in mind, a compiler must generate multi-threaded code that distributes itself across the machine and communicates using queue mappings. Additionally, the queue environment makes most of common wisdom on how to perform procedure calls and other standard operations relatively useless. New techniques are necessary in such an environment.
Chapter 3

Related Work

In order to put the ADAM architecture and language objectives in perspective, some of the parallel architectures and languages that they evolved from need to be examined.

3.1 Fine-grain parallel Architectures

A number of previous architectures have used a queue or stream based model of communication between their nodes. All of them have very narrow specifications for how the interaction is supposed to proceed and which entities are allowed to engage in it. ADAM has intentionally looser constraints on communication, but care should be taken to remember which specific types of communication have paid off in architecture implementation.

3.1.1 Decoupled Access/Execute

A decoupled access/execute (DAE) architecture[16] attempts to separate the tasks of memory interaction from execution. Two processors are used; one performs all address calculation and memory loads and stores and the other actually performs the computation. The processors are linked by two pairs of queues: one pair of queues is used to communicate load and store data, while the other pair is used to exchange control flow information. The queues are accessed as a special set of registers which dequeue when read and enqueue when written. Coordinated control flow decisions result in a loss of decoupling (LOD), as the processors sync up to make the decision. An additional method of decoupling, called control decoupling [2] has been suggested as a way of reducing LOD events. Furthermore, hardware multi-threading can be used to cover the latency associated with LODs [13, 18].
A DAE architecture can demonstrate a super-linear speedup due to the elimination of dependencies between the two instruction streams. The access processor leads the execute processor, eliminating observed memory latency from the execute processor's point of view. The decoupling has the effect of a good caching system and semi-out-of-order execution [9, 5]. One compiler has been written for a DCAE (Decoupled Control/Access/Execute) machine [20] and it focuses on eliminating LODs, using several optimizations to remove them. The key element that makes DAE possible is the hardware queues that connect the two processors. By utilizing the programming language discussed in this thesis, decoupled computation can be achieved by compiled code.

3.1.2 Imagine

Imagine is a stream processor [15] developed at Stanford which focuses on single-processor performance. Imagine's concept of a stream is a stream of data pulled from memory, heading through the processor, and either back through the processor, to memory, or over the network. The objective is to apply computation to many elements of a stream simultaneously. Programming the Imagine involves developing two separate sets of code: applications and operators. Applications are written in a variant of C and they specify streams to move around and operators to apply to them. An operator or kernel is also written in a variant of C and performs an operation on an element of a stream until the stream has finished passing by. Kernels are not composable into larger kernels; there is a clear distinction between the kernel level and the application level. Therefore, nesting of kernels cannot be done, which demonstrates a lack of the proper means of abstraction.

3.1.3 J-Machine

The J-machine is a distributed machine built at MIT and designed with small simple processors communicating using low latency network operations [12]. The processors were designed with hardware network send and receive operations as well as synchronization primitives. Futures were base types that could be hardwired to a register or passed around using a more heavyweight mechanism. Essentially, reference to a empty future caused the thread to block and a new thread scheduled. Two major languages targeted the J-machine, Message-Driven C (MDC) [10] and Concurrent Smalltalk [7]. MDC uses explicit "spawn thread on processor X for computing method Y" semantics. It uses a global shared memory
so pointer passing is valid. Concurrent Smalltalk is tasteful implementation of Smalltalk that supports distributed objects and directives for explicit and implicit parallelism.

### 3.2 Parallel Languages

There are more parallel languages than you can shake a stick at [1, 14, 17], and they all attempt to tackle the problem from different angles. The problem is that writing code that takes advantage of a multiprocessor environment while still running correctly is very hard. For specific cases and benchmarks, thought applied to the problem has produced good solutions, but generally efficient and correct results remain elusive, probably due to the inability of people to think parallel. An alternative is to not expose the parallelism to the programmer and implement it all in the compiler or runtime. However, compilers are not much smarter than people when parallelizing code. All parallel programming languages give the programmer a model to conceptualize their program running in and two common models are active object-oriented languages and stream-based languages.

#### 3.2.1 Objects

As object-oriented languages are popular in the sequential area, so they are popular in the parallel arena. By making objects active entities that perform computation in response to messages, distributing objects across the machine produces a program that executes in parallel. Additionally, objects provide an encapsulation of computation and data that should be performed locally with respect to each other. They also provide obvious synchronization points.

Unlike the sequential world, many concurrent languages are object-based instead of object-oriented. Object-based languages lack inheritance, which is a feature that encourages code reuse. The reason for the lack of inheritance in concurrent languages is that code-reuse is much more dangerous. An issue that has been named inheritance anomaly[11] characterizes the conflict between synchronization and subclassing, resulting in almost complete re-implementation of the superclass in order to satisfy synchronization constraints in the subclass.
Emerald

Emerald[3, 8] provides an object-based concurrent programming language designed to simplify the construction of distributed programs. The language supports strong static typing to aid the programmer and compiler. Objects can be active or passive, where active objects have their own thread of control. Invoking an operation on an object creates a new thread to perform the operation. Abstract types specify a set of operations which objects of that type are supposed to implement. Emerald objects are location independent, but not location invisible. Objects interactions are the same regardless of the location of the object, but the programmer has direct control over object location should they so choose. Emerald allows objects to be explicitly moved to the position of another object, fixed in location, and unfixed. When performing a method call, the arguments can be explicitly migrated to the location of the call invocation. Lastly, a migration/load balancing scheme exists to re-distribute load at runtime.

3.2.2 Streams

Stream-based programming languages glue together a number of processing components into a computational pipeline. Each component is optimized to handle a particular stage of the overall computation. By distributing the components over the machine, parallel computation is achieved. Furthermore, since the communication between the components is regular, communication network can be limited to or optimized for this type of communication. The Imagine application and kernel languages mentioned earlier fall into this category.

StreamIt

The StreamIt language [19] adds a number of classes to Java to support streaming computation. These classes are supported by the compiler [6], which compiles the source into a program that runs on the RAW architecture [21]. Streams are syntactically heavyweight objects that fit into a structured base of stream types. The programmer subclasses the appropriate type (normally a Filter) to specialize the streams to the particular application. The main procedure builds a set of the stream blocks and links them together. The compiler capitalizes on this structure to perform flow analysis on the code in order to generate code
for the static network routers in the RAW architecture. Additional optimizations move blocks of the stream in order to reduce communication overhead and enable better work distribution.
Chapter 4

Language Design

4.1 Objectives

The programming language should be reasonably familiar to a good programmer and provide support for natural constructs for expressing parallelism. In this case reasonably familiar means that an existing application could be rewritten in the language without an undue amount of pain. The parallelism constructs provided by the language should be angled to take advantage of the abstract nature of the machine as well as the architecture’s streaming support.

4.2 Design of Couatl

On the way to the creation of the language People, another language called Couatl was implemented. Couatl is similar to People in that it is an object oriented, declarative programming language with support for streams. However, Couatl supported inheritance and its streaming constructs were a lot closer to implementation level. The language also lacked a number of shortcuts that programmers are used to. Lastly, it provided durasive methods, which were methods for which there was only one invocation thread and this thread was shared between all callers.

The streaming construct consisted of defining a set of “meanders” (pieces of a stream) and a set of variables that expressed the intercommunication. A meander was a snippet of code that executed in an infinite loop, which imported the values of some interconnect variables, and exported others. The interconnect could specify loops, self-loops, and even
initial values for the self-loops. It provided an efficient mechanism for doing small, stream-intensive operations. In addition to the syntax being rather obscure and confusing, it did not provide a means of abstraction over streams. The inability to compose streaming computations was a show-stopper.

4.3 People Overview

Drawing from the experience with Couatl, a second language was designed: People\(^1\). The major objective of People over Couatl was to draw the idea of streaming directly into the language. Objects are a pairing of code and data. Arrays are the storage of similar bits of data. Streams represent the motion of data from one area of the program to another. These three structures form the base for People, upon which a structured process environment is built that controls how the structures are connected and used. The key components of the language are method calls, objects, and streams.

4.3.1 Method Call

In an ordinary language, method call is the means by which data values are delivered to a different context for processing, whether the language is a parallel one or not. In this manner, method call provides a mechanism for streaming data to some method call entity, and then streaming the return value(s) back. Moreover, the return stream could be used to implement multiple-value return from methods; a problem that has yet to be tastefully solved. However, method call has been around for so long that co-opting it to be the source of streams is difficult. Any new method call syntax had better be backwards compatible to an older well-known implementation, or it will horribly confuse programmers. Due to the lack of a tasteful syntax, and an aversion to further overloading the method call mechanism, People does not use method call as the standard process for creating streams.

On the other hand, acquiring parallelism through method call is a much more accepted practice. Additionally, the ADAM encourages thread spawning for this case, as it supplies the new method with a clean context in which to perform its work. Thus, every method call in People results in a new thread being spawned to handle the method. The arguments are sent to the method, along with the return value's destination. For example, in figure

\(^1\)From PPL, a Parallel Programming Language
4-1, the two recursive calls to \texttt{fib} are done in parallel. The first call results in a thread
spawn, with the result delivered to the \texttt{a} variable, and the second to the \texttt{b} variable. When
the following \texttt{return} statement attempts to add the two variables, the method thread will
block until the recursive calls complete and deliver their return values to the variables. The
\texttt{fib} code could have been written to directly return the sum of the two recursive calls, but
it would still wait on two (temporary) variables to obtain values before returning. The
current way was chosen for explicitness.

\begin{verbatim}
int fib(int n) {
    if (n < 2) {
        return n;
    } else {
        int a,b;
        a = fib(n-1);
        b = fib(n-2);
        return a+b;
    }
}
\end{verbatim}

Figure 4-1: Method Call Example

The only way to ascertain whether a method has completed or not is to verify that it
has returned a value. Since void methods do not return a value, they cannot be checked
this way. The \texttt{touch} primitive forces the execution to wait until a value appears in a given
variable, and thus it can be used to ensure that a method call has completed. Unfortunately,
it does not work on unique values; as the variable will be emptied by the \texttt{touch}.

4.3.2 Objects

Objects are logical groupings of data with the code that acts upon it. The programmer
gives names to these groupings, which serve to communicate the intent or intended usage
of the objects by the programmer or other colleagues. A strongly-typed language enables
the compiler to verify these explicit specifications which the programmer has written into
their code; catching numerous bugs at compile time. In addition to being logical to the
programmer, objects are also wonderful sources of locality, as the threads that perform the
methods can be placed near the data they will most likely access merely by placing them
near the object they were invoked on. By distributing the objects across the machine, the
computation and storage requirements are also distributed. Finally, migrating the object
changes the creation point for any method calls invoked on that object, which can increase the impact of migration.

For these reasons, People is an object based language, however it lacks inheritance. The extra complexity of implementing inheritance may not pay off, as noted in the previous chapter as inheritance anomaly. One reason for implementing inheritance is that all sub-classes of a given class are guaranteed to implement at least the methods that their super class does. This allows for a set of classes, all of which may be named with the same type: the superclass. Without inheritance, another class grouping mechanism is necessary, so People has interfaces. An interface specifies a list of method signatures that must be provided by any class that implements the interface.

A class is a grouping of instance variables and methods. Instance variables are values stored with an instance of the object. Most object-oriented models have some protection mechanism which allows for the enforcement of abstraction barriers, but People lacks this feature due to time constraints. The instance variables are accessible by anyone who has a pointer to the object. A method is procedure that performs computation on the object’s behalf. The method named constructor is called whenever an object is instantiated. Unlike Java, People supports standalone methods which are procedures not associated with any object (like C procedures).

Figure 4-2 presents a simple List interface and a class that implements it. The interface List specifies which methods ArrayList must implement. ArrayList declares a pair of instance variables, vals and size. It has four methods, one of which is a constructor. The set method uses a membar to ensure that the array store completes before it returns. It returns a value so that a caller can tell when the set has completed. The print method shows how to invoke a standalone method, even from inside a method of the same name on an object (e.g. the preceding .). In this case, the standalone method being invoked is a built-in printing procedure. Lastly, the main method is a standalone method (not part of any class), which first constructs an ArrayList object, invokes a couple of methods on it, and ensures that they complete in order using touch.

4.3.3 Streams

Traditionally, data motion is implied by either performing many procedure calls or by passing arrays. Multiple procedure calls have the overhead of procedure call setup and return,
package objects;

interface List {
    int get(int i);
    boolean set(int i, int val);
    void print();
}

class ArrayList implements List {
    array[int] vals;
    int size;

    constructor(int sz) {
        size = sz;
        vals = new int[size];
    }

    int get(int i) {
        return vals[i];
    }

    boolean set(int i, int val) {
        vals[i] = val;
        membarO;
        return true;
    }

    void print() {
        int i=0;
        while (i < size) {
            .print(vals[i]);
            i = i + 1;
        }
    }
}

void main() {
    List al = new ArrayList(2);
    touch(al.set(0,4));
    touch(al.set(1,7));
    al.print();
}

Figure 4-2: Simple Class and Interface
which can be very expensive if the procedure call is remote. Passing arrays introduces questions of sharing, and producer/consumer operations require delicate synchronization. When People was designed, a major objective was to integrate streams directly into the language as first class values, but preferably in a way that did not generate special cases. With the decision not to attempt to use method call as the mechanism, the objective was to design a new language construct that shared the good features of method call. One of the most important features of method call is the abstraction barrier that separates the caller from the callee. Thus, stream creation should connect two pieces of code with streams that bridge the abstraction barrier.

The programmatic representation of streams was chosen to maximize the amount of compile-time checking that could be performed. Data flows one direction down a stream, and thus representing a stream as a single entity would require run-time exceptions when a program attempted to cause data to change directions. Instead, a stream is represented by two values, a source and a sink, which are explicitly typed in the language. This allows the compiler to easily ascertain which direction data is flowing and to generate compile-time errors when the stream is used incorrectly. Unfortunately, in place stream construction would now require multiple-value return, which places more restrictions on stream creation.

The last major entity in People is the module, which is the unit of streaming computation. When a module is constructed, a new thread is spawned to execute the module code, and a set of user-declared streams are created to connect the constructor and the module instance. The ends of these streams, called sources and sinks are first-class values that can be passed around. Stream creation is wrapped into module creation to ensure that streams are used as communication elements between entities in the program. This helps the compiler determine where to create the machine objects used to implement the streams. The declaration for a module contains the exported sources or sinks, documentation on such, and the internal other-ends of the external streams. There is no explicit handle returned to the constructor that encapsulates the module; there is no Module type.

Sources and Sinks are typed values that represent the ends of streams. Data enqueued onto a source can be dequeued off the corresponding sink. Source/sink pairs are created when a module is constructed. One end of the stream is available to the module code while the other is returned to the module constructor. Sources and sinks are first-class values that can be passed between procedures as well as stored into instance variables and arrays.
A sink can be permanently connected to source using a `connect` statement, which causes all data arriving at the sink to be forwarded to the source.

Standard source/sink pairs denote a one-to-one communication pattern; they aren't safe for multiple readers and writers by default. In order to preserve the notion of single ends, all stream variables are declared to be `unique`. When a unique variable is copied (such as being passed to a procedure), the variable is emptied. The variable will need to be assigned a new value before references to it will succeed. Thus, splitting or combining a stream requires the programmer to explicitly include an appropriate stream transform. The `unique` keyword may also be applied to other variables, including instance variables.

An example which demonstrates the streaming structures of `People` can be found in figure 4-3. The `main` method constructs a pair of modules, connects them together into a pipeline, sends a set of values into the top, and finally prints out the results as they emerge from the end of the pipe. The `summation` module exports a source and sink to a constructor. Any values the user enqueues onto the source arrive at the sink the module names in. Data that the module enqueues onto its local source named `out` will arrive at the sink the module exports. Neither module ever terminates; when no more data is available the module will block and eventually get garbage collected by the hardware.

### 4.3.4 Placement

People has no explicit placement directives through which the programmer indicates where code should run and where memory should be allocated. Mostly, this is because such analysis was beyond the scope of the thesis. However, the language itself provides guidance as to common access patterns. Method threads are spawned near the object they are associated with because they will likely need to interact with the object's allocated memory or other method invocations. Standalone methods are spawned close to the calling thread, as no other information is available about what the method code will interact with. Objects and modules are distributed randomly over the machine, with the expectation that the migration system will move them as needed.

### 4.3.5 Synchronization

Most synchronization is handled by the underlying architecture through the use of queues. Producer/consumer synchronization requires no programmer effort as long as there is only
package default;

module squarer has source[int], sink[int]
  as "input data", "squares of input values"
  internally sink[int] in, source[int] sqs {

    while (true) {
      int i = dq(in);
      nq(sqs, i*i);
    }
  }

module summation has source[int], sink[int]
  as "input data", "sum of all input values"
  internally sink[int] in, source[int] sum {

    int total=0;
    while (true) {
      total = total + dq(in);
      nq(sum, total);
    }
  }

void main() {

  source[int] values,presquares;
  sink[int] squares,output;

  construct squarer with values,squares;
  construct summation with presquares,output;
  connect(squares,presquares);

  int i=0;
  while (i < 5) {
    nq(values,i);
    i = i + 1;
  }

  i=0;
  while (i < 5) {
    print(dq(output));
    i = i + 1;
  }

}

Figure 4-3: Streaming Example Program
one producer and one consumer. A simple locking mechanism can be constructed by using unique instance variables: reading the value of the variable leaves it empty, which will cause the next reader to block until a new value is written. Additional primitives could be added to allow explicit use of ADAM's EXCH(ange) mechanism. The ADAM architecture supports the idea of transactions and rollback, but it has not fledged enough to provide a real solution.
Chapter 5

Implementation

The primary objective was to have a compiler that translated a high-level language into correct ADAM assembly. A secondary objective was to produce output code which take advantage of the ADAM queues and streaming architecture. A tertiary objective was to produce optimized output code. A Couatl compiler was written that satisfied the primary objective, so the focus turned to taking advantage of ADAM features. First the implementation of the People Compiler is presented, then the details of how the language features of People are mapped onto ADAM by the compiler, and finally the ADAM specific optimizations performed.

5.1 The People Compiler

5.1.1 Organization

The People compiler has 5 main passes. First, it tokenizes and parses the input, using the JLex and CUP tools. The output from the first pass is an intermediate representation (IR) with a tree structure. The second pass performs all the static semantic checks as well as translating names into actual references into the IR. The following pass converts the tree structure into a graph of basic blocks connected by flow edges. The third pass performs dataflow optimization and annotation. The final pass constructs the output assembly. An additional pass generates something similar to Javadoc of the code (See figures 6-1, B-1).

The first pass uses a pair of compiler-writer tools called JLex and JavaCUP. JLex is a scanner-generator which takes a specification for how to tokenize the input stream and
outputs a state machine written in Java that implements the specification. The tokenizer is
coupled to a stack-based parser generated by CUP. CUP reads a grammar which specifies
acceptable arrangements of tokens, called productions, and actions to be performed when
productions are matched. The output is a stack-based LALR(1) parser which recognizes the
input grammar. This first pass concentrates on generating an IR which accurately reflects
the input program, while reporting syntax errors as precisely as possible. It also implements
as much of lexical scoping as possible.

The second pass performs semantic checking as well as coercing the IR into its fully-
linked high-level form. The majority of the work is type checking. This ensures that types
match during assignment, method calls, etc. An additional piece of type checking is ensuring
that all objects that declare they implement an interface actually do. Since the language
allows for the use of instance variables, methods, classes, interfaces, and modules before
they are declared, any unresolved references are fixed during this pass. At this time, free
variables are reported to the user as errors. After the semantic checking pass, the code is
assumed to adhere to the specification for People and thus the code generator should be
able to produce code that implements it. Only in exceptional cases will errors related to
user input be produced after this point.

Next, the high-level IR is converted to the middle-level IR by flattening the tree structure
into a directed graph. Dataflow analysis and code generation works much better on control-
flow graph. All IF, WHILE, and BLOCK constructs are broken out into Blocks connected
by edges. These edges are named or annotated to specify information about when the
edge is taken (True result, False result, etc). On a finer scale, short-circuit conditionals
also produce blocks and edges. When completed, each block in the graph is a basic block.
That is, if it starts executing, every statement in the block is guaranteed to be executed,
and only those statements will be executed until the completion of the block. Basic blocks
are the units of dataflow analysis. In addition to flattening the control flow, temporaries
are inserted to flatten complicated statements down into statements that embody a single
operation. The temporary allocator is not very intelligent, but it does re-use temporaries
when it can.

The fourth pass of the People compiler performs a number of optimizations on the
code. Most of these optimizations provide information to the code-generator to enable it
to produce more efficient code. For this reason, in depth discussion of most of them is left
until after discussion of the code generation pass. The two optimizations worth mentioning now are Live Variable and Data Presence. Live variable analysis is a common analysis that discovers when variable values are never used again (after the last use, the value is considered “dead”). This information is communicated to the code generator so that the queue holding the value is dequeued when the value becomes dead. Additionally, variables marked as unique are forced dead after being read. After Live variable completes, Data presence analysis discovers when variables have values. When a variable lacks a value, a read from it will block. The user should be warned about this case. When a variable has a value and it’s being set again, the code generator should use a clobber instead of an enqueue.

The code generation pass generates assembly instructions for each basic block. These assembly instructions are stored as Statements in a new Block which is attached to the block that it was produced from. When the IR is printed by the assembly printer, each of these blocks is printed. Printing an assembly block prints it’s name as a label, all the instructions in it, and a branch instruction to jump to the block’s successor if necessary. The blocks are printed using a greedy algorithm that minimizes the number of branches that are needed. A code generation object manages the pass and performs a number of useful functions. It performs rudimentary queue (register) allocation on a per-method basis. Since no interesting def-use web analysis is performed, each variable is allocated a queue. Thus, an overly complicated method can run the allocator out of queues (and it will issue a warning that this has occurred). In practice, hand-written methods that have 110 variables are rare, and not worth the time of implementing a real queue allocator in the scope of this thesis.

Should a method have more than 110 live values at any time, the compiler will fail to generate appropriate code due to the lack of any spilling mechanism. A number of spilling mechanisms are possible, including one not available to regular architectures. Passing a stack pointer to a procedure is non-trivial due to the spawn involved. While the base address in the capability can be advanced and marked increment-only, this only supports one procedure performing procedure calls at a time. As soon as the execution splits into two threads running simultaneously, each of which call a procedure, either the capability needs to be sectioned or a new one allocated. Unless queue spilling is very common, it is unlikely that this effort will be worth the cost.

Spilling can be reduced or avoided entirely by making better use of the queue-file. By
realizing that value A will be consumed before value B is next needed, the value of B can be enqueued behind A in the same queue. By doubling or tripling up on queues, many more than 128 values can all be present in the context. This implementation is only beneficial if the underlying implementation of the ADAM supports more than one queue value in a queue. Since ADAM does not specify the required depth of the hardware implementation of the queue, performance will drop sharply at some depth as the underlying architecture switches from the fast medium out to memory. Thus, queue-stacking mechanisms must be careful not to stack too deep.

Another spilling mechanism is to pick one queue and stack into it all the spilled registers. The underlying architecture handles memory allocation, and the worst case situation involves a rotate operation linear in the number of registers spilled. Better performance could be acquired by implementing a scheduler that tried to enqueue in the order the values were used. Lastly, memory could be allocated explicitly to spill queues to on a per thread basis.

5.1.2 Intermediate Representation

The intermediate representation is the heart of the compiler. It models everything the compiler knows about the source code in a way that makes this information accessible when it is needed. In order to make the IR flexible, every object supports annotations. These annotations are key-value associations that name pieces of information the compiler knows about the object. For example, optimizations report their results by placing annotations on the variables and statements that they analyze. Another example is the SYMBOL annotation, which denotes what part of the source program is responsible for the creation of the object. This annotation is used when reporting errors. Unfortunately, certain stages of the compiler depend on various annotations existing in a consistent manner, making the compiler hard to update without total knowledge of the annotation dependencies.

At the top level, every program has an instance of the Source object. The Source keeps track of the packages of source code the user supplied, as well as dangling references, object type descriptors, and the place where output should be saved. Beneath the Source object are the Package containers. These objects are named by the programmer at the top of their source with the package name; statement. They contain all the classes, interfaces, modules, and standalone methods declared in a particular package.
The Library contains primitive commands that the programmer has access to. When the compiler initializes, it reads a library file which contains the names and definitions of the primitives. Primitives can either be inlined or called. Either way, they have access to a very restricted set of queues and are limited in what end-states are allowed. The default library provides documentation on the specifics. The programmer is allowed to specify their own library file, but it must contain a working implementation of sink_code. Typical library routines are those that use the debugging print and cycle count procedures, as well as those that deal with generating random numbers, simple mathematical functions, etc. When a non-inlined function is included, it’s text is marked for inclusion at the end of the output assembly.

The Class object contains the handles of the interfaces the class implements, the instance variables, and the methods. The Interface contains the handles of the method signatures that an implementing class is required to implement. A Module contains the list of streams to be built when the module is constructed, but it actually acts much more like a Method.

A Method contains the argument list, the return type, and the Blocks that implement the code. The argument list is an ordered list of variables that the method expects as input and that are bound in the method body. As the level of the IR is lowered, the Method picks up annotations about the entry block, the list of all blocks, the list of bottom blocks, the list of all variables used in the method, and whether the method performs loads and/or stores.

A Variable is a named temporary typed storage location. It is used as a key into the queue allocation table during code generation. In some ways, a variable is a degenerate stream that does not leave the current thread. Earlier visions of the compiler/architecture involved using this realization to automatically splice user code into separate threads by turning the degenerate streams into actual streams.

A Block is a named ordered collection of statements. The name serves as a label during code generation, and is selected to reflect what operation created the block. For the high-level IR, Blocks can be nested by having blocks contain BLOCK statements. For the lower-level IRs, blocks are basic blocks of code and are connected by edges. An edge is a named object which represents control flow.

A Statement consists of an operation and an ordered list of arguments which can be statements or constants. Statements are categorized by their operations, most of which
are generated by the programmer. Statements can have all sorts of annotations, but most have a TYPE annotation that denotes the type of the value resulting from evaluating the statement.

5.2 Language Implementation

5.2.1 Couatl Implementation

In Couatl, each object had a server thread that handled method invocation requests. Since all threads spawned by the server thread were kept local, distributed processing was accomplished by spreading the server threads over the machine. However, server threads created a bottleneck when many methods were being called on the same object. The server threads also created additional computational overhead and resulted in a large number of “eternal” threads; most of which would have been garbage collected if such a collector existed.

Another feature of Couatl was durasive methods. These were eternal methods which upon reaching the end of user code would branch back to the top and execute again with the next set of arguments. The object server threads provided a gating mechanism to ensure that only one thread was using the durasive method at a time. Durasive methods not only reduced the number of threads spawned, but also provided a synchronization method. Since there was exactly one thread executing the durasive method code, a critical section could be placed within. The gating mechanism in the server thread depended on handing out the durasive method's thread id and having the caller hand it back. If a second request arrived for the same durasive method, the server thread would block until it was returned. Unfortunately, no method invocation requests would be handled during this down-time.

5.2.2 Classes

A object instance is implemented as an allocated chunk of memory and a method-lookup procedure. The class’ constructor method is responsible for building the data structure. Having a pointer to the object means that having a pointer to the memory store for the object. All methods invoked on the object will have the this value pointing to the memory store. The memory store is allocated local to the constructor thread. Since future method calls on the object will be located near the memory store, the object’s location on the machine is determined by where the constructor thread is spawned. The first value in the
object's memory store is the address of the method-lookup procedure, while the rest is devoted to instance variable value storage.

The method-lookup procedure is used to implement dynamic method calls (methods invoked on Interface-typed values). Due to the lack of inheritance, all method calls on class-typed values can be resolved at compile time. In a language with inheritance, this would not be true, because someone could create B, a subclass of A and override the method M. When invoking a method named M on an object of type A, the compiler cannot tell if at run time, it will have an object of type A or type B. In People, a method call on an Interface value could actually be a method call on any object that implements that interface.

At compile time, the compiler knows all the interfaces each class implements, and thus it can generate a lookup procedure that translates an (interface id, method id) pair into a method on the class. Each Interface is assigned its own 32 bit unique id computed as a hash of the package name and the interface name. A method id is composed of the interface id, paired with its 32 bit offset within the interface, i.e. the first method has the number 0, the second is 1, the third 2, and so on. The class' method-lookup procedure extracts the interface bits and switches on them to determine the context in which the method offset should be interpreted. This lookup procedure is called by the caller, and does not generate a thread spawn for efficiency. The jump and link instruction (JMPL) is used to transfer control to the procedure, and it uses only the temporaries q0 and q1 for its computation. See Figure 5-1 for an example method-lookup procedure.

The instance variables for an object are stored after the method-lookup procedure in the object's memory store. At creation time, all instance variables are left uninitialized. Each variable is assigned an offset into the capability and these offsets are publicly available at compile time to any methods.

5.2.3 Interfaces

Interfaces have no direct representation in the output assembly unless they implement a constructor. This constructor is compiled like any standalone method, except that it is semantically required to return an object that implements the interface.
objects_ArrayList__lookup:
    SHRC @q0, 32, q1            ; extract high order bits
    BRNZ q1, objects_ArrayList__lookup_int ; interface?
    SHLC q0, 1, q1               ; compute jump offset
    BREL q1                      ; branch rel based on method number
    MOVECL objects_ArrayList_get, q1 ; move method addr to temp
    JMP q0                       ; return to caller
    MOVECL objects_ArrayList_set, q1 ; move method addr to temp
    JMP q0                       ; return to caller
    MOVECL objects_ArrayList_print, q1 ; move method addr to temp
    JMP q0                       ; return to caller

objects_ArrayList__lookup_int:
    SHRC @q0, 32, q1            ; grab high bits
    SUBC q1, 0x2136D6AA, q1      ; caller thinks I am a List
    BRNZ q1, objects_ArrayList__lookup_int1 ; if not, go on
    ANDC q0, 0x7FFFFFFF, q1      ; grab low bits
    SHLC q1, 1, q1               ; compute jump offset
    BREL q1                      ; branch rel based on method number
    MOVECL objects_ArrayList_get, q1 ; move method addr to temp
    JMP q0                       ; return to caller
    MOVECL objects_ArrayList_set, q1 ; move method addr to temp
    JMP q0                       ; return to caller
    MOVECL objects_ArrayList_print, q1 ; move method addr to temp
    JMP q0                       ; return to caller

objects_ArrayList__lookup_int1:
    HALT                         ; Whoops; did not match anything!

Figure 5-1: Method lookup code for ArrayList from figure 4-2
5.2.4 Methods

A direct consequence of using queues instead of registers is that they are harder to save to memory. Thus, reclaiming queue namespace for use by a called method can be non-trivial with queues of arbitrary depth. Instead, the hardware multi-threading mechanism is used to produce new namespaces. A thread spawn produces 128 new queues to use, and the mapping mechanism allows for argument and return value delivery.

Methods have a preamble, a body, and a closing section. The preamble of a method pulls the arguments off the argument queue (q0) and opens up the load and store mappings. If the method is not standalone, the first argument sent is the value of this, and it is moved to a queue allocated for it. The next possible argument is the return point, which is sent if the method is declared to return a value. A valid return point consists of a thread id and a queue number. If the caller does not desire the return value, a 0 is sent instead. This is distinguishable from a thread id because a thread id is a capability. Both values of the return point are saved in a single queue. Next, mappings to memory are created with the MML and MMS instructions. These are constructed only if the method performs loads and stores. The final piece of the preamble moves the declared arguments from the argument queue to their allocated queues. This ensures that all the arguments arrive before the method code attempts to use q0 again.

The closing of a method attempts to free up any resources it may be using. Before branching to the closing block, any variables that the data presence analysis indicated were full have their queues flushed. This reduces the size of the thread’s footprint in the Qcache. The closing block closes the load and store mapping after ensuring that all data in the store queues have been committed. The this value and return point queues are flushed. If the this value had been treated like a regular variable in the analyses, a slightly more efficient implementation would be possible. The final operation is to HALT the thread execution, which enables the thread garbage collector to reclaim the thread id and backing store.

5.2.5 Modules

Modules are compiled in a similar manner to methods. The only major difference is that they create source/sink pairs in the preamble and return the appropriate ends to the caller. The module construction method is specialized to deal with multiple return values.
A sink is constructed by spawning a new thread that runs the sink server code. This code accepts commands in \texttt{q0} and expects data to arrive in \texttt{q1}. If the command is a capability, it is used as a thread to map back to. The next value is which queue in the requester thread to deliver the dequeued value to. The first value in \texttt{q1} is dequeued and send to the requester. A non-capability specifies a special command. A command of 0 indicates that the sink thread should halt. This command is currently unused, as it can be hard to prove that no new data will be sent. A command of 1 indicates to permanently forward the data in \texttt{q1} on to the following point (thread id and queue number). After receiving this command, the sink thread ceases to accept commands and merely forwards any data arriving in the data queue on to the given point. It does not exit to ensure that the forwarding continues to occur.

5.2.6 Statements

What follows is a description of all top-level statements and how they are compiled.

- **ASSIGN**
  The left-hand side of the expression, which is a variable \texttt{DEF}, an instance variable set, or an array set, is evaluated to acquire a destination queue. Then the right-hand side of the expression is compiled with the destination queue as the target. For example, \texttt{x = y + z} would compile \texttt{x} to it's allocated queue, and then produce an \texttt{ADD} instruction which used the allocated queues of \texttt{y} and \texttt{z} with the destination of \texttt{x}'s queue.

- **CALL**
  There are five types of method calls: object constructors, class object calls, interface object calls, standalone method calls, and library calls. Object constructors are only distinct from class object calls because of how constructors are spawned. The start addresses of constructors, class object calls, standalone method calls, and library calls are determined statically. Only interface object calls require the method lookup procedure to be fetched from the object capability and invoked on the method id. Library calls are special in that most of them are automatically inlined, and even those that are not are executed in the current thread using the branch-and-link instruction. After spawning the thread to run the method at the correct start address, the this
value, return point, and arguments are sent if necessary. A method is never allowed to return a value into a queue that has ordering constraints (load/store, argument queues, etc).

- **NQ**
The source argument to NQ specifies a thread context and a queue number to write the data value to. A queue is mapped to this location, and the value enqueued onto it. After the value is sent, the queue is unmapped. Since a source is a pair of values that are stored in the same queue, a COPY directive during the NQ operation requires a couple of extra steps to return the values to the original queue in the right order.

- **DQ**
The sink argument to DQ specifies the thread context to request a value from. A queue is mapped to the target context and the local thread context id (obtained with PROCID) sent, along with the queue number to return the result in. The sink code reads the return point, and sends a value back to it. Since sinks are unique and uncopyable, only one thread can have the sink capability at a time. This ensures that the context and queue number arrive synchronously with respect to other DQ requests.

- **CONSTRUCT**
Module construction works similarly to calling an object constructor with no arguments, but multiple return values. A new thread is spawned remotely to execute the module code and a return point is sent to indicate where to return the stream values. When the stream values arrive, they are moved to the queues allocated to the variables that receive them.

- **CONNECT**
A connect compiles to sending a forward instruction to the sink to forward to the target of the source. The argument queue is mapped to the sink, the forward command issued, and the source values sent. When the sink receives these values, it recognizes the command, and switches into forward mode. It no longer listens to the command queue, and instead moves every element in the data queue into a queue mapped to the context to forward to. The alternative, mapping the data queue directly, does not
work as well because it depends on at ADAM implementation to do the right thing with a passively mapped queue.

- **TEST**
  The test statement can only occur at the end of a block. It compiles the given expression into a temp queue, and then branches on the result. Tests can be biased true or false depending on what the compiler guesses is the most common case. For example, while-generated tests which determine if the loop should stop are biased false.

- **RETURN**
  The return statement attempts to map to the return point and send the specified value. If the method was instructed not to return a value, the return point will contain a non-capability, and the return step will be skipped. Immediately after attempting to return, the method branches to the closing block.

  The tail call optimization has been implemented. When the value returned is the immediate result of a method call, the current method’s return point is provided to the callee and the current method halts. This prevents the return value from having to propagate from the callee through the current method to reach the current method’s caller. Since arguments are not passed on the stack, the tail call optimization can be performed for any tail call.

### 5.3 Optimizations

Two interesting optimizations are performed by the People compiler. The compiler also performs the live variable, data presence analyses, as well as tail call optimization, but these are not specific to the ADAM architecture. The LastMemoryAccess optimization determines when capabilities need to be sent to the memory system and when they would be redundant. The LocalStream and LocalAccess pair up to detect and then distribute information about where source/sink operations can be localized to the current thread.
5.3.1 LastMemoryAccess

Since the memory mapped queues remember the last capability passed to them, doing a load or a store does not require always sending a capability. The LastMemoryAccess optimization computes the last capability that was issued to the load or store queue at each point in the program. When doing a load or store, if the last capability matches the one about to be used, the current capability is not sent. Additionally, this pass records whether or not the method performs any loads or stores. If it does not perform any of one type or another, the method preamble and closing is adjusted to no bother setting or cleaning up that type of memory operation.

The LastMemoryAccess optimization is implemented as a conservative forward dataflow analysis. The cost of sending an extra capability is negligible compared to using an incorrect one. Since it tracks only the last capability in load and store queues, it needs only remember two values per block. For blocks with multiple predecessors, intersection is used as the combiner. That is, if all preceding blocks have the same value, that is the value entering the current block. If any differ, the value entering the block is set to null to indicate no positive information. The statements in the block are examined in order while keeping track of the last capabilities sent. Those accesses with match are marked as such for the code generation phase to apply. Finally, the output values are written to the block as annotations for the block’s successors. Since the code may contain loops, it make take a number of passes for the information to flow around the loop. If the output values of a block change from the value in the previous iteration, a least one iteration will be run after the current.

The results of applying this optimization are rather nice. In tight loops, it can save a large number of instructions as well as bandwidth to memory. It affects array reference, instance variable access, and interface method calls. However, it would be well combined with loop invariant code motion. Often in loops a capability should be sent to the load address queue right before the loop starts up, but never during the loop. By itself, the LastMemoryAccess optimization does not discover this because the top of the loop has two predecessors: the code above the loop and the bottom of the loop body. The code above the loop has not sent the appropriate capability, so the conservative analysis sends the capability at the beginning of the loop body, and thus every time around the loop. Additionally, a pair
of nested loops with accesses before the loops and inside the innermost loop body will fail to resolve correctly. The optimization fails to realize that the outer access reaches the inner access because the information about last access cannot propagate out of the inner loop or in from outside the outer loop. A more aggressive implementation of reaching definitions which partially simulates the program could resolve this issue.

5.3.2 Localization

Localization is aimed at recovering the efficiency of stream interaction that was present in Couatl. With Couatl's more limited streams implementation, the end points of the streams were known at construction time, so the source could be mapped directly to the sink, which resided in context of the thread that would use the values. Thus, the normal enqueue and dequeue operations combined with queue mapping to produce a simple efficient solution.

In unoptimized People code, dequeuing from a sink requires a map, a network trip to the sink thread, the sink thread being scheduled, a network trip back from the sink thread, and an unmap. Enqueuing onto a source requires mapping a queue to the destination, sending the value, and then unmapping the queue. Both of these operations are blocking, which defeats the intent to hide latency with queues. The Localization analyses discover situations where the programmer has ceased passing the stream end around and wishes to use it. At this point, the stream end point can be tied directly to the thread acting on it and the performance recouped.

Sources and sinks have different characteristics, necessitating different treatment. Since sinks are unique values, as soon as a sink is consumed via dequeue, no thread has access to it again. If this is the case for a sink that arrives in a method, it can be tied permanently to the method's thread context. To accomplish this, a forward message is sent to the sink which instructs it to forward it to the local queue allocated to hold the sink value. If the sink is able to "escape" the thread, this mapping would need to be undone, which is impossible to do safely. Another option would be to forward to local queue to wherever the sink escapes to, but this would introduce another hop for the data to pass along, tie up a queue in the local context permanently, and possibly cause odd behavior when the thread halted. Thus, the localization option is only a good idea when the sink value never escapes from the thread.

Moreover, the compiler must be able to statically determine which dequeues are targeted
at the localized sink. For this reason, copying the sink around will cause it not to be localized until the last copy. An example of this can be found in figure 5-2. \(a,b\) cannot be localized, because the compiler cannot statically determine which sink the \(dq\)s on \(x\) and \(y\) target. \(c\) is localized in the method preamble. \(x,y\) are localized when assigned.

```c
void foo(sink[int] a, sink[int] b, sink[int] c) {
    sink[int] x, y;
    print(dq(a));
    print(dq(b));
    if (dq(c)) {
        x = a;
        y = b;
    } else {
        x = b;
        y = a;
    }
    print(dq(x));
    print(dq(y));
}
```

Figure 5-2: Localization of sinks

Sources are easier to deal with because they do not contain any data. Unlike sinks, sources can be unlocalized by just unmapping the local queue. However, its not worth the effort of localizing the source unless it is going to be used. Due to time constraints, the optimization only localizes sources in the same circumstances that it would localize a sink. However, most code does not move the source value around once it starts using it in earnest.

LocalStream

The LocalStream optimization is a conservative backwards dataflow analysis that tracks values leaving back to their definition. Each block exports the list of variables whose values escape to it’s immediate predecessors. At the bottom of a block, all the blocks successors are examined, and if a variable’s value escapes in one of them, it ends the block escaping. The optimization then scans through the block from end to beginning looking for escaping values and value definition. If the value has not escaped by the time its definition is reached, the definition is marked as local.

A value can escape by being stored to an array or instance variable, passed to a method, moved to another variable, enqueued onto a source, or overwritten by another value. Clob-
bered values are considered to have escaped because the queue allocated to the variable is needed to hold the following sink. If a sink is localized onto it, the queue cannot be used for this purpose. With better queue allocator this problem could be avoided. A value definition is when a value enters the thread context. It can arrive by being passed as an argument, read from an array, local variable, or instance variable, dequeued from a sink, or built by a module construction.

**LocalAccess**

Since LocalStream works backwards to discover which streams can be local, another forward pass is necessary to convey the knowledge of which accesses should be local to the operations that perform the accesses. LocalStream is implemented as a forward dataflow analysis that maintains the list of variables which are local. When it encounters a variable definition, it adds it to the list if the definition is marked local. A USE is marked local if the variable is part of the list of local variables. Once a variable becomes local, it is guaranteed to remain local because of the definition of escaping that LocalStream uses. A variable is guaranteed to be local on all predecessors of a node if it’s local on one of them, so the combiner method does not really matter.

These two optimizations generate a couple of delicate cases for the code generation phase. First, when modules construct source/sink pairs during their preamble, streams marked local do not have a thread spawned to run the sink code. Instead, the sink is allocated locally right from the beginning, providing good performance for simple module code. When a method is called with a localized argument, the value is localized during the method preamble. At every point where a source or sink is assigned to a variable and the assignment is marked local, the stream is localized. For a sink, it is directed to forward to the queue assigned to localized sink variable. For a source, the queue assigned to the source variable is mapped to the queue destination. To use a localized stream, the value is just enqueued or dequeued normally from the queue assigned to the variable. Upon termination, localized sinks are flushed and localized sources are forced empty and then unmapped.
Chapter 6

Results

The primary result is that high level programs written in People can be compiled into code that runs on the ADAM architecture. Moreover, the resultant code does not deadlock, it runs to completion and produces the expected results. In addition to this result, the language was tested for programmability and the compiler tested for performance.

6.1 Programmability

A programming language is successful if the computational model it presents encourages simple direct solutions to the types of problems the programmer is trying to solve. Moreover, the solutions it encourages should translate efficiently to the underlying architecture, resulting in good performance. In practice, a good programming language is tested through use, but People has not existed long enough for a number of programmers to try it out. To demonstrate its usability a couple of test cases are presented, along with some notice about the language’s shortcomings.

A simple implementation of the N-body gravitational simulation was implemented in People. For each step in the simulation, each body needs to be told to update it’s position. Once every body has completed a step, the next step can be started. The mathematics of the simulation were easily implemented, but simple math is generally not hard to program. However, implementing the communication between the central simulation and all of the Body objects that perform the step computations requires a bit of thought. Iterating over the array of Bodies sending step messages is a linear operation, when a logarithmic operation is desired. To this end, the TreeNode module was written to implement a binary distribution
tree (See figure 6-1 for image, appendix B for source). Writing the code took under an hour and yielded a result that not only builds the distribution tree, but the tree is a persistent structure that can distribute an arbitrary number of messages and responses. Streams are used to deliver the message through the tree of TreeNodes to the objects and to return the success value to the original invoker. The handle at the top of the tree is a pair of streams, and since streams are first class, they can be passed around instead of the array. Interfaces are used to provide a class of objects suitable for invoking with the TreeNode, as well as a value to deliver as part of the activation.

Figure 6-1: Distribution tree built by TreeNode

When building a streaming computation, a number of modules are constructed and their streams linked together. The process requires a large number of variable declarations before module construction and a lot of connect statements afterwards. These statements are irritating to code, and hard do correctly. However, the general case requires this level of functionality: not all streaming computations construct the stream statically. It appears that static stream construction is a common enough case that the language should provide a short-cut for doing so succinctly. Another possibility would be to have a graphical stream construction tool which allowed the programmer to graphically string together the streaming elements and the tool would produce the code. In any case, providing a slightly irritating
but general mechanism means the programmer can get disgruntled, not stymied.

One issue the language does not address at all is placement and allocation. Objects and modules are allocated randomly on the machine inside a given latency radius from their creator, while everything else is allocated locally. If elements are placed badly, the ADAM migration and load balancing engine is responsible for correcting the placement. Unfortunately, the programmer can communicate none of their high-level knowledge about the program to the architecture. However, it is not clear that the programmer has useful knowledge to communicate, and anything they communicate could be wrong. Programmers appear to be notoriously bad at indicating bottlenecks and resource contention, probably due to the complexity of the system they are trying to make guesses about.

However, it is very frustrating to observe the machine performing stupid allocation mistakes and having no way to suggest a better solution. The main issue is what language to use to express placement on the machine. In an environment which is not laid out in a grid, which machines may vary in size, which processors may fail and later be fixed, and which may running two programs at once, having the programmer directly indicate which the processor should handle a task is unlikely to be fruitful. Clearly some abstract notion of allocation is necessary. Instead of having the programmer micro-manage allocation, have the programmer help define the resource needs of each piece of the program. Then the issue of placement becomes an informed decision by the compiler and machine. However, what linguistic structure to use to extract this information from the programmer in a palatable way is beyond the scope of this thesis.

6.2 Performance

Since the ADAM architecture is brand new, it is hard to determine what good performance is. Thus, most performance measurement is done in terms of comparisons between optimized and unoptimized versions of the same code.

The first interesting performance result is a comparison between a somewhat naive software queue implementation and the hardware queue implementation. The software queue is implemented as an array, whose first three values contain a lock, the queue head, and the queue length. To enqueue or dequeue, a library routine acquires the lock by using a set of exchange queues as a test-and-set, and yielding if the lock is not acquired. This
mechanism can result in starvation and extra cycle usage in general, but for simple single
producer/consumer activity, it does not produce much overhead.

Figure 6-2: Comparison between Hardware and Software queues

To produce the comparison, a trivial case was used: the producer(s) enqueue sequential
numbers, and the remote consumer dequeues and prints them. For the single producer,
single consumer case, hardware queues are a factor of 34 faster than the software implementa-
tion. Even if the software queue performance is improved by an order of magnitude, hard-
ware queues still offer significant performance gains. However, improving software queue
performance may not be easy, as the memory backing the queue will always be non-local to
one of the threads, and caching will yield little benefit. Hardware queues are optimized for
the single producer, single consumer case, so moving to two producers incurs an additional
software synchronization overhead. Continuing to increase the number of producers allows
the producers to lead the consumer and covering some of the synchronization overhead.
Software queues evidence the opposite effect, as more contention for the lock causes the
consumer to battle producers for the opportunity to access the queue. The two hardware
queue lines represent each producer sending each value by itself (size 1) or sending pairs of
values (size 2). Increasing the “packet” size reduces the number of synchronization events, but can mean longer stalls as more time is spent waiting while another thread sends data.

6.2.1 Streams

By transforming a standard \(N^3\) matrix multiply into two modules which stream the array values directly to the multiplier thread, an access/execute model is produced (see figure 6-3). Since the array indices are entirely predictable, no communication is necessary except sending the initial array capability and streaming out the array values. The multiplier loop merely pulls the values off the streams, does the multiply-add, stores the value in the output array and loops back to do it again.

![Diagram of streaming matrix multiply](image)

Figure 6-3: Layout of streaming matrix multiply
For a 15x15 matrix, standard matrix multiply takes 4915 cycles to initialize, uses 510 cycles per iteration, and ends on cycle 119253. Streaming matrix multiply takes 6880 cycles to initialize, uses around 350 cycles to iteration after migration, and ends on cycle 91101. The memory migration routines kick in on the 8th iteration and move the array memory to the processors where the access threads are running. Without the migration routines, it uses 1100 cycles per iteration and ends on cycle 238815. Clearly the migration routines are essential when the arrays are allocated away from their point of use. A larger matrix size only widens the gap between the regular and streaming version, as an 100x100 matrix requires 3315 cycles per iteration normally and only 1600 cycles per iteration for the streaming implementation.

The loop body execution time was reduced by a factor of approximately two. The access/execute model produces a large benefit in this example, but not one that could have been realized by a normal DAE machine as it required two access processors. In this manner, the ADAM architecture allows for dynamic allocation of access and execute processors as the code requires them.

6.2.2 Optimizations

LocalStream and LocalAccess

The simplest test of stream location involves the first benchmark, where values are being produced in one thread and consumed in another. With localization turned off, initialization takes 139 cycles and each value takes 21 cycles to produce. With localization turned on, initialization takes only 124 cycles, and each value takes 7 cycles. Localization demonstrates a 3x performance increase in this simple case. Even more critically, the producer is leading the consumer, so the extra time per value is all overhead of contacting the sink thread and getting sent the response.

For applications that are more computation-bounded, the performance improvement is not as great. For a program that constructs a seven module pipeline for detecting edges in the input stream and then collecting the edges into features, stream localization produces a 25% speedup over a one feature stream. Once the data stream is extended to 50 features, the speedup is only 10%. The latency-hiding feature of queues is absorbing most of the impact.
If the language were extended to include passing already-created sinks into modules as they were constructed, an additional layer of indirection could be removed. When a module initializes, for sinks it is going to export, it creates a sink thread and sends it to the constructor. When collections of modules are created and linked up, all these sink threads are connected to the sources of the downstream modules in the pipeline. Thus, they are never passed anywhere, and only serve to forward data on. If the downstream module’s source could be passed into the upstream module, the indirection through the upstream’s created sink could be avoided.

**LastMemoryAccess**

The effects of the LastMemoryAccess optimization are minimal without a number of other optimizations which were not implemented due to time constraints. Additionally, LastMemoryAccess should have been handled as a reaching definition analysis, where the definitions are simulated assignment to load and store variables. The current dataflow analysis fails to correctly propagate access information into deep loop nests, and thus it fails on programs like matrix multiply. For programs without loop nests, after applying optimizations like loop-invariant code motion and common sub-expression elimination by hand, the LastMemoryAccess optimization was able to show a factor of four speedup on the initialization of N-body. In addition to improving cycle count, it also significantly reduced the number of words sent to memory as well as the output code size. This optimization, while not independently powerful, is an integral part of the optimization suite for the ADAM.
Chapter 7

Conclusions

The ADAM architecture is not too compiler unfriendly. However, it does require new techniques and thought in order to produce workable, efficient code. The use of queues instead of registers is easily manageable in procedure bodies by simple techniques. Using a thread spawn for every method call is expensive, but appears manageable. By applying an inlining optimization, some of this cost could be recouped. Objects provided a better guess for where method threads should be located, which gave the migration engine better start conditions. The lack of inheritance did not cause large issues due to the existence of interfaces, but such problems may only arise in the presence of large code-bases and many developers.

The introduction of first-class streams to the language was a success. They present a simple communication model to the programmer and the model is very general. With some additional libraries, the language will support any operation a programmer might want to do with a stream. The act of connecting up ends of streams to form a chain of modules can be cumbersome, but it does provide a general enough mechanism to allow the programmer to implement dynamically constructed chains. The implementation of these first-class objects defaults to a slightly inefficient server model. However, the localization optimizations detect where the stream can be directly implemented by the underlying hardware. Localization can be improved by treating the localization of sources separately from sinks, as localizing a source is not nearly as restrictive. By altering the language to allow stream elements to be passed into module constructors, the programmer could use a less awkward syntax and the compiler could avoid generating extra sink threads.
The implicit thread spawn for every procedure call requires a mental shift on the part of the programmer. A fair number of bugs were caused by forgetting this feature. To force in-order computation, a number of methods were declared to return dummy values, which were merely touch'd by the caller. When doing this, the programmer is explicitly managing the parallelism by choosing when and where threads join up. Similar issues exist for memory, as stores are not guaranteed to have completed until after a membar. Loads and stores are guaranteed to complete in-order internal to a thread, but returning to the caller may occur before all the stores complete. This implicit asynchronous operation is an invisible pitfall for the programmer to fall into and might deserve being made explicit.

People does a moderately terrible job of locating elements on the machine. Methods are located near the objects they are invoked on, which is normally a good decision. However, object and module placement is completely random. Even when it would be simple to determine where an array is going to end up, it’s always allocated locally. Standalone methods are always allocated locally, thus quick-sort would run entirely on a single node if the load balancer was turned off. Clearly, the compiler needs to be more intelligent and the language must let the programmer communicate placement information to the hardware.

Before People can become a solid tool for writing parallel programs, a number of library objects and methods need to be developed. The lack of data-parallel operations is crippling. Similarly, implementing a synchronized work queue where multiple threads can safely dequeue values from common sink is needed. Finally, the compiler needs to be updated to support type-checking and linking against pre-compiled libraries.

The People Compiler lacks a large number of standard optimizations that would dovetail nicely with the localization and memory access optimizations it currently does support. Software pipelining is one optimization that the ADAM architecture should make both easy to implement and effective in practice. For ADAM specific optimizations, analysis that extracts access and execute threads from a single original thread could save the programmer a lot of headache. Additionally, data flow analysis to stack values into a smaller number of queues would reduce each thread’s footprint in the qcache, and causing less contention. Finally, detecting statically constructed streams would allow the compiler to avoid the use of abstraction-carrying indirection mechanisms.
Appendix A

People Specification

A.1 Formal Syntax

- **bold** indicates a terminal. The following terminals have special representations:
  - **ID** is any string of characters: alphabetic, followed by alpha-numeric.
  - **INT** is a base ten or hexadecimal number. Hexadecimal prefixed by 0x.
  - **FLOAT** is floating point number.
  - **STRING**s are denoted by "s.

- **italics** denotes a non-terminal.

- [ ]s denote optional values. ']' denotes the character [.

- { }s denote grouping. '{' denotes the character {.

- $x^*$ means $x$ may occur 0 or more times.

- $x^+$ means $x$ may appear 1 or more times.

\[
\begin{align*}
\text{program} & \rightarrow \text{package ID} ; \text{decl}^* \\
\text{decl} & \rightarrow \text{class}_\text{decl} | \text{interface}_\text{decl} | \text{module}_\text{decl} | \text{method}_\text{defn} \\
\text{class}_\text{decl} & \rightarrow \text{class ID} [\text{implements ID},^+] '{\text{ defn}^* }'] \\
\text{defn} & \rightarrow \text{var}_\text{defn} | \text{class}_\text{method}_\text{defn}
\end{align*}
\]
\[
\begin{align*}
\text{var\_defn} \rightarrow & \quad [\text{unique}] \text{ type } \text{ID},^+ ; \\
\text{class\_method\_defn} \rightarrow & \quad [\text{persistent}] \text{ method\_defn} \\
& \quad | \text{constructor } (' \text{param},^+ ') \text{ block} \\
\text{method\_defn} \rightarrow & \quad \{ \text{type} | \text{void} \} \text{ID} (' \text{param},^+ ') \text{ block} \\
\text{param} \rightarrow & \quad \text{type} \text{ID} \\
\text{interface\_decl} \rightarrow & \quad \text{interface ID} \{ \{ \text{method\_decl} \mid \text{constructor\_decl} \}^+ \}' \\
\text{method\_decl} \rightarrow & \quad [\text{persistent}] \{ \text{type} | \text{void} \} \text{ID} (' \text{param},^+ ') ; \\
\text{constructor\_decl} \rightarrow & \quad \text{constructor } (' \text{param},^+ ') [ \text{block} | ; ] \\
\text{module\_decl} \rightarrow & \quad \text{module ID has stream\_type,}^+ \text{ as STRING,}^+ \\
& \quad \text{internally stream\_decl,}^+ \text{ block} \\
\text{stream\_type} \rightarrow & \quad \{ \text{source} | \text{sink} \} \[ ' \text{type} ' ] \\
\text{stream\_decl} \rightarrow & \quad \text{stream\_type} \text{ID} \\
\text{block} \rightarrow & \quad \{ \text{stmt}^* \} \\
\text{stmt} \rightarrow & \quad \text{var\_decl} \\
& \quad | \text{location} = \text{expr} ; \\
& \quad | \text{method\_call} ; \\
& \quad | \text{stream\_op} ; \\
& \quad | \text{construct ID with ID,}^+ ; \\
& \quad | \text{connect } (' \text{ID}, \text{ID} ') ; \\
& \quad | \text{return} [ \text{expr} ] ; \\
& \quad | \text{if } (' \text{expr} ') \text{ block} [ \text{else block} ] \\
& \quad | \text{while } (' \text{expr} ') \text{ block} \\
& \quad | \text{block} \\
\text{var\_defn} \rightarrow & \quad \text{type} \{ \text{ID [=expr] ,}^+ \} ; \\
\text{location} \rightarrow & \quad \text{ID} | \{ \text{simple\_expr} '!' \text{ID} \} | \{ \text{simple\_expr} '[' \text{expr} ']' \} \\
\text{method\_call} \rightarrow & \quad \text{method\_name} (' \text{expr},^+ ') \\
\text{method\_name} \rightarrow & \quad \text{ID} | \{ '.' \text{ID} \} | \{ \text{simple\_expr} '!' \text{ID} \} \\
\text{stream\_op} \rightarrow & \quad \{ \text{nq} '(' \text{ID}, \text{expr} ')' \} | \{ \text{dq} '(' \text{ID} ')' \} \\
\end{align*}
\]
\[ \text{expr} \rightarrow \text{simple_expr} \\
| \quad \text{literal} \\
| \quad \text{new type} \['\ expr \']' \\
| \quad \text{simple_expr} \['\ expr \']' \\
| \quad '" type '" expr \\
| \quad \text{nq} \('\ ID, \ expr\)' \\
| \quad \('\ expr \')' \\
| \quad \text{expr binop expr} \\
| \quad \text{unop expr} \\
\]

\[ \text{type} \rightarrow \text{int} | \text{float} | \text{boolean} | 'ANY' | \text{ID} \\
| \quad \{ \text{array} '" type '" \} | \text{stream_type} \\
\]

\[ \text{simple_expr} \rightarrow \text{ID} | \text{location} | \text{this} | \text{method_call} | \text{new type} \(' \ expr, \* \')' \\
| \quad \text{dq} \('\ ID \')' \\
\]

\[ \text{binop} \rightarrow \text{arithop} | \text{compop} | \text{bitop} | \text{condop} \\
\]

\[ \text{arithop} \rightarrow \'+' | '-' | '*' | '/' \\
\]

\[ \text{compop} \rightarrow '==' | '!=' | '<' | '<=' | '>' | '>=' \\
\]

\[ \text{bitop} \rightarrow '\&' | '||' | '<<' | '<>' | '>>' \\
\]

\[ \text{condop} \rightarrow '\&\&' | '||' \\
\]

\[ \text{unop} \rightarrow '-' | '+' \\
\]

\[ \text{literal} \rightarrow \text{INT} | \text{FLOAT} | \text{true} | \text{false} | \text{null} \\
\]

A.2 Semantics

- Execution starts at the standalone main method.

- Classes
  - Classes lacking a constructor are provided with a default that takes no arguments and does nothing.
  - Instance variables are unprotected; they may be accessed from both inside and outside of the class.
- Java's "static" variables and methods to not exist, as they depend on a global lookup.
- At most one constructor may be declared for a class.
- Classes may be instantiated with the new expression, and the resulting object may be allocated anywhere on the machine.

• Interfaces

- The interface may supply a constructor. If the constructor has no body (e.g. a declaration), it specifies the signature of all implementing class’ constructors. If it does have a body, this body is expected to instantiate an object which implements the interface and return it. This allows for a "factory" paradigm as is sometimes used in Java.
- Interfaces may be instantiated with the new expression if the interface supplies a factory constructor.

• Modules

- The opening line of the module definition describes the exterior interface; the streams that are returned when the module is constructed.
- The second line gives short descriptions of each of the streams, so the user knows how each should be used.
- The third line declares the internal variables which will be assigned the other ends of streams declared on the opening line.
- Each of the lines must have the same number of items, and where the first line declares a source, the third must declare a sink, and vice versa.
- Currently, singleton "initialization" arguments to a module must be passed on a stream.
- Modules run in their own thread.
- Modules may be created with the construct statement, where the variables supplied must match the types on the first line of the module declaration.

• Types
The base types are int, float, and boolean.

All classes and interfaces are valid types. A class type also matches any interface that it implements.

Composite types (array, source, and sink) can be made from any type, including other composite types.

A composite type matches a composite type of the same form (e.g. array, etc) if the derivative types match.

The ANY type matches any type.

A Cast ([type] expr) can be used to force a type change. For base types, this actually coerces the value to the new type on the hardware level.

** Constructors

- The constructor is implicitly declared to return a value with the same type as the container the constructor is declared in.
- The object is valid (eg. this) as soon as execution of user code starts, assuming the constructor is not a factory constructor.
- An implicit “return this” is added to the end of the method.

** Methods

- Argument variables may be of any type, but their names must be unique.
- If the method is declared in a class, the this variable is bound to object the method was invoked on.
- Base types are passed by value, all other types are passed by reference.
- Memory writes (instance variable assignment and array store) appear in order relative to loads in the method, but are not guaranteed to complete before the method returns. Use membar() to force writes to complete.
- For standard methods, a new thread is spawned for each method call, and the thread is located “close” to the location of the object the method is being invoked on.
- **persistent** methods are allocated a thread upon object creation, and this thread loops through the method code each time another thread delivers a set of arguments to it. Recursive invocation will result in deadlock.

- The **return** statement sends the return value (if any) back to the caller and causes the method thread to exit.

**Variables**

- Variables may be of any type.

- Instance variables (declared in a **class**), are associated with a particular instance of the class (an object).

- Variables are lexically scoped, and covering argument and instance variables is allowed.

- Instance variables may be implicitly referenced in methods declared in the same class as the instance variable. They may be explicitly referenced using **this.var**. Instance variables on another object may be referenced with **obj.var**.

- Reading or writing an instance variable will result in a memory interaction; these variables are never implicitly moved to temporaries for multiple operations. This strategy is not recommended for implementing inter-thread communication.

- Reference to an uninitialized local variable will result in the method thread blocking. A warning will be generated by the compiler in these circumstances.

- Reference to an uninitialized instance variable will result in a runtime error when the value is used.

- Local variables may be initialized when they are declared.

- **unique** variables cannot have their value copied; the variable will be emptied when the variable is read. Subsequent access to the variable will block until a value is written into the variable.

- All **source** and **sink** variables are implicitly unique.

- The only exceptions to the uniqueness rule are **nq** and **dq** operations on stream variables; these do not empty the stream variable.

**Streams**
– Streams are only created when modules are constructed.
– The ends of a stream are named separately: the source and the sink.
– The source is where data is placed onto the stream, using nq.
– The sink is where data is removed from the stream, using dq.
– All stream values are unique; they cannot be copied. The semantics of a copy operation would not be well-defined.
– To cause all data arriving at a given sink to be forwarded on to a given source, use the connect statement.

• Arrays
– Arrays are located “close” to the thread performing the allocation.
– Array references are bounds-checked in hardware.
– Uninitialized values loaded from an array will generate a runtime error when used.

• Expressions
– The following operators are supported: (in order of precedence, least to greatest)

  ||
  &&
  &, | (bitwise)
  ==, !=
  <, >, <=, >=
  <<, >>
  +, -
  *, /
  !
  ~ (complement)
  - (unary)

– Method-calls are not guaranteed to have completed until the return value of the method has been used. No guarantees are provided about void methods.
Method-calls without an explicit object to invoke the method on are resolved by first looking in the local object (if any), then for standalone methods. To force the method-call to use a standalone method in the presence of a method of the same name on the class, use `.method-name`.

A.3 Programming Tips

- Copy instance variables to local variables if they are going to be used a lot.

- Use streaming operations to prefetch values when the value sequence is irrespective of the values themselves (like matrix multiply).

- The less “weirdly” sources and sinks are used, the better the localization optimizations work, and the better the code will perform.
Appendix B

TreeNode

//
// TreeNode implements automatic construction of distribution
// tree to do parallel operations on an array of objects.
//
// Setup:
// construct a TreeNode, and send it 0, arraysize-1, and the array.
// The array is required to be an array of objects that implement
// the ActivatableObject interface.
//
// Usage:
// Build an Activation and send it to the Activation source.
// The activation will be delivered to every element of the
// array in log N time. Each element of the array computes on the
// activation and returns a boolean result value. These results
// are collected up and returned back to the top level. The top
// level returns true only if all the results were true.
// This process may be repeated as many times as desired.

package activation;

module TreeNode has source[int],
    source[array[ActivatableObject]],
    source[Activation],
    sink[boolean]
as "start and end of array range",
"array to access"
"stream of activations",
"completion signal"
internally sink[int] ranges,
sink[array[ActivatableObject]] arr,
sink[Activation] acts,
source[boolean] result {

int start,end;
start = dq(ranges);
end = dq(ranges);

// down to a single element of the array,
// become a manager for that element.
if (start == end) {
    ActivatableObject ao = dq(arr)[start]; // get the element
    while (true) {
        // get activation, invoke method, return result to parent.
        nq(result, ao.activate(dq(acts)));
    }
} else {
    // calculate split point
    int median = (start + end) / 2;
    array[ActivatableObject] thearray = dq(arr);

    source[int] range1,range2;
    source[array[ActivatableObject]] arr1, arr2;
    source[Activation] acts1,acts2;
    sink[boolean] res1, res2;

    // build children.
    construct TreeNode with range1, arr1, acts1, res1;
    construct TreeNode with range2, arr2, acts2, res2;

    // initialize
    nq(range1,start); // left child gets start..median
    nq(range1,median);
    nq(range2,median+1); // right child gets median+1..end
    nq(range2,end);
    nq(arr1,thearray); // both get the array
}
nq(arr2,thearray);

// split Activations we receive to children
// important that this spawns new thread to do this.
activationSplitter(acts,acts1,acts2);

while (true) {
    boolean first = dq(res1);
    boolean second = dq(res2);
    // ensure solid values for both results.
touch(first); touch(second);
    // return true if both true.
nq(result,first && second);
}

interface ActivatableObject {
    boolean activate(Activation a);
}

interface Activation {
}

void activationSplitter(sink[Activation] acts,
                        source[Activation] acts1,
                        source[Activation] acts2) {
    while (true) {
        Activation a = dq(acts);
        nq(acts1,a);
        nq(acts2,a);
    }
}
Figure B-1: Flowgraph of TreeNode, autogenerated by compiler

Figure B-2: Sample compiler-generated webpage of package Activation
Bibliography


