Compiling Array Computations for the Fresh Breeze Parallel Processor

by

Igor Arkadiy Ginzburg
B.S. Computer Science and Engineering, Physics
Massachusetts Institute of Technology, 2006

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 2007
[June 2007]

Author .....
Department of Electrical Engineering and Computer Science
May 23, 2007

Certified by .....
Jack Dennis
Professor Emeritus
Thesis Supervisor

Accepted by.....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Optimizing Array Manipulation for the Fresh Breeze Parallel Processor

by

Igor Arkadiy Ginzburg

Submitted to the
Department of Electrical Engineering and Computer Science

May 18, 2007
In partial fulfillment of the requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Fresh Breeze is a highly parallel architecture currently under development, which strives to provide high performance scientific computing with simple programmability. The architecture provides for multithreaded determinate execution with a write-once shared memory system. In particular, Fresh Breeze data structures must be constructed from directed acyclic graphs of immutable fixed-size chunks of memory, rather than laid out in a mutable linear memory. While this model is well suited for executing functional programs, the goal of this thesis is to see if conventional programs can be efficiently compiled for this novel memory system and parallelization model, focusing specifically on array-based linear algebra computations. We compile a subset of Java, targeting the Fresh Breeze instruction set. The compiler, using a static data-flow graph intermediate representation, performs analysis and transformations which reduce communication with the shared memory and identify opportunities for parallelization.

Thesis Supervisor: Professor Jack Dennis
Title: Professor Emeritus

Acknowledgements

I would like to thank Professor Jack Dennis for his advice and guidance when choosing the topic and implementing this project, and for his invaluable help and feedback with this document. I would also like to thank the National Science Foundation for supporting this work. Thank you to my family for your continual support.

Igor Ginzburg

Table of Contents

List of Figures

1. Introduction

While feature sizes on silicon processor chips continue to shrink, allowing continued growth in the number of transistors per chip, concerns over power consumption have slowed the growth in clock speeds. These trends suggest the creation of parallel systems. Exploiting parallelism is essential, but difficult due to concerns over programmability. Fresh Breeze is a new multi-processor architecture designed in light of these trends, promising greater performance for scientific computing while improving programmability.

Computations involving multi-dimensional arrays are important in scientific computation. In "The Landscape of Parallel Computing Research: A View from Berkeley", [2] identify 12 common patterns of computation that have wide applications and serve as benchmarks for parallel architectures. Three of these, including Dense Linear Algebra, Sparse Linear Algebra, and Spectral Methods, are commonly expressed as operations on arrays. Traditional implementations of these algorithms may manipulate these arrays in place, and in general assume a mutable, linear memory model.

One key aspect of the Fresh Breeze architecture is a write-once shared memory system. In this model, all data structures, including arrays, are represented by directed acyclic graphs of immutable fixed-size chunks of memory. So, Fresh Breeze arrays are neither mutable nor linear.

The goal of this work is to reconcile these two observations, allowing array-intensive programs written in a syntax that assumes the traditional memory model, to be efficiently executed on a Fresh Breeze computer. To that end, we have created a Fresh Breeze compiler, which compiles a subset of Java to the Fresh Breeze instruction set. The compiler is structured in several phases, with the majority of the analysis and transformations being performed on

intermediate representations that are independent of the specifics of either Java or the Fresh Breeze instructions set. The first phase converts Java byte code to a flat 3-operand intermediate representation (IR), enforcing some limitations on possible Java expressions. The second phase constructs Static Data Flow Graphs from the flat IR. This Data Flow Graph Representation is inspired by the intermediate representation of the Sisal compiler [8], and serves as the basis for a series of transformations. These include transformations that normalize the representation, standard optimizing transformations common to most compilers, and Fresh Breeze specific transformations that introduce nodes that make it easier to produce efficient, parallel Fresh Breeze code. The final phase of the compiler, which is still under development, will generate Fresh Breeze machine code from the Data Flow Graphs. The main contribution of this work is the design of the special Data Flow Graph nodes which describe opportunities for parallelization, and the implementation of Fresh Breeze-specific transformations which introduce these special nodes.

The remainder of this thesis is organized in six chapters. Chapter 2 gives background information on the Fresh Breeze architecture, the structure of Fresh Breeze arrays, and a strategy for parallelizing the execution of computations on Fresh Breeze arrays. Chapter 3 describes the design of the compiler, specifying the supported subset of Java, and detailing the intermediate representations. Chapter 4 describes the implementation of the compiler, including the construction of Data Flow Graphs, and the specific analysis and transformations that are applied to generate the special nodes. In Chapter 5 we evaluate the compiler's output on several representative linear-algebra routines. In Chapter 6 we propose future extensions to the compiler and discuss their implementation. Finally, in Chapter 7 we discuss the general conclusions of this work.

2. Background

2.1. Fresh Breeze Architecture

Fresh Breeze is a multiprocessor chip architecture which uses the ever-increasing numbers of available transistors for parallelism rather than complexity. By using a RISC instruction set and a simplified pipeline, multiple Multithreaded Processors (MP) can be placed on one chip. To create even more powerful systems, multiple MP chips may be connected together to create a highly parallel system [3].

Fresh Breeze differs from conventional computer architectures in several ways. A global shared address space is used to simplify communication between processors, and eliminate the need for a separation between "memory" and the file system. The shared memory holds immutable 128-byte chunks, each addressed by a 64-bit Unique Identifier (UID). Chunks can be created and read, but not modified once created. This eliminates the cache-coherence problems that can add complexity to traditional multi-processor designs. A reference-count based garbage collector, implemented in hardware, is used to reclaim unused chunks, freeing the associated UIDs. This mechanism can be very efficient, but requires the heap to be cycle-free.

The Fresh Breeze instruction set facilitates the creation and use of these chunks. A ChunkCreate instruction allocates a new chunk returning its UID. New chunks are considered unsealed, meaning that they are mutable, and can be updated using the ChunkSet instruction. To prevent the formation of heap cycles, the ChunkSet instruction cannot place the UID of an unsealed chunk into another unsealed chunk. A chunk can be sealed, or made immutable, by a ChunkSeal instruction. Directed Acyclic Graphs (DAGs) of chunks can be created by placing UIDs of sealed chunks into newly-created unsealed chunks using the ChunkSet instruction. Data

structures in Fresh Breeze programs are represented using these chunk DAGs. Arithmetic operations cannot be performed on UIDs, preventing functions from accessing chunks that where not passed in as parameters and aren't reachable from pointers in the parameter chunks. Chunk data can be read using the ChunkGet instruction. Both the ChunkGet and ChunkSet instructions provide for bulk memory transfers by operating on a range of registers, rather than being restricted to moving one word at a time.

This memory architecture allows for a novel model of execution, which provides for simultaneous multithreading while guaranteeing determinate behavior. The immutable nature of the memory guarantees that functions cannot modify their parameters, preventing them from exhibiting side-effects. This enforces the secure-arguments principle of modular programming and eliminates race conditions, the major source of non-determinacy in traditional parallel computation. Basic parallelism is achieved through a Spawn instruction, which starts the execution of a slave thread, passing it some parameters, and leaving a "join ticket" as a handle on that thread. UIDs of unsealed chunks cannot be passed as parameters through Spawn calls, preventing multiple threads from modifying the same chunk. When the slave thread completes its computation it returns a value using the EnterJoinResult instruction. The master thread can issue a ReadJoinResult instruction to retrieve the value returned by the slave. If the slave has not finished, the ReadJoinResult instruction stalls until the slave thread is done. By issuing several Spawn instructions before reading the join results, multiple parallel slave threads can be instantiated. Other forms of parallelism, such as producer/consumer parallelism, as well as explicitly non-determinate computations are also supported [4], but are not relevant to the present work.

2.2. Writing Software for Fresh Breeze

Software written for a Fresh Breeze system must be able to utilize the massive parallelism provided by the architecture, without creating cycles in the shared memory or mutating existing chunks. These requirements are met by functional programming languages. Purely functional programs do not allow mutation of data, and are therefore easy to parallelize, without creating race conditions or cycles in memory. Functional procedures create new output data based on some input data, rather than updating the input data in place.

Although many functional languages exist, they have some drawbacks. Most, in an effort to support "stateful" programming, are biased towards execution on a sequential computer. Others, like Val [7] and Sisal [6] have no available working front ends. Additionally, our main goal is to examine whether functional programs written within the constraints of a popular syntax, which assumes the standard linear, mutable memory model, can be efficiently compiled for Fresh Breeze. Therefore, the Fresh Breeze compiler is written to translate a new functional language, based on the Java language, called Functional Java.

Java is attractive for several reasons. Its primitive types are well suited for the Fresh Breeze RISC architecture. Its strong type system is helpful both for writing modular code, and for creating program graph models of that code. In addition, Java has a popular syntax, derived from C, and existing compilers and development environments. By using a language which is compatible with existing Java compilers, we avoid the task of building a compiler front end.

2.3. Fresh Breeze Arrays

Arrays are data structures mapping indices to values. On a conventional architecture with a linear, mutable memory, an array with n t-byte elements is usually represented by a contiguous region of $n * t$ bytes in memory, starting at some memory address l. The value at index i can be read or written by manipulating the t bytes at memory address $l + i * t$. This representation

affords fixed size, mutable arrays, matching the properties of Java's built in array class. Java arrays are intended to be implemented as a contiguous sequence of memory cells, providing constant time reads and writes to individual elements.

This representation is not possible on a Fresh Breeze system, where data is stored in chunks whose addresses are not subject to arithmetic operations. Fresh Breeze arrays, like all other Fresh Breeze data structures, must be represented as directed acyclic graphs of chunks. In [3], a representation of Fresh Breeze arrays as fixed depth trees is proposed, where the elements are stored in leaf chunks, and the order of the elements is specified by an in-order traversal of the tree.

An array with n 16-byte elements can be represented by a tree of depth $k = \text{floor}(\log_{16} n)$. While an individual element can be read in such an array (by traversing k chunks), its value cannot be mutated. Changing a value requires creating k new chunks, and produces a new array. The main challenge of the Fresh Breeze compiler is to achieve good performance in light of these properties.

2.4. Parallelizing Array Computation

In order to achieve good performance, we need to take advantage of the parallelism available on Fresh Breeze. This can be done by partitioning computation among different threads. Computations involving arrays are often performed in loops that read through existing arrays or construct new ones. If we can assign different iterations of a loop to different threads, we can greatly reduce computation time.

While not all loops in a Java program can be parallelized, many parallelizable loops are fairly easy to identify. In one such class of loops, each iteration computes a set of values which do not depend on the results of previous iterations, and then the values are combined to produce

the loop's outputs. These values can be combined as elements of a new array, or accumulated to produce a scalar result.

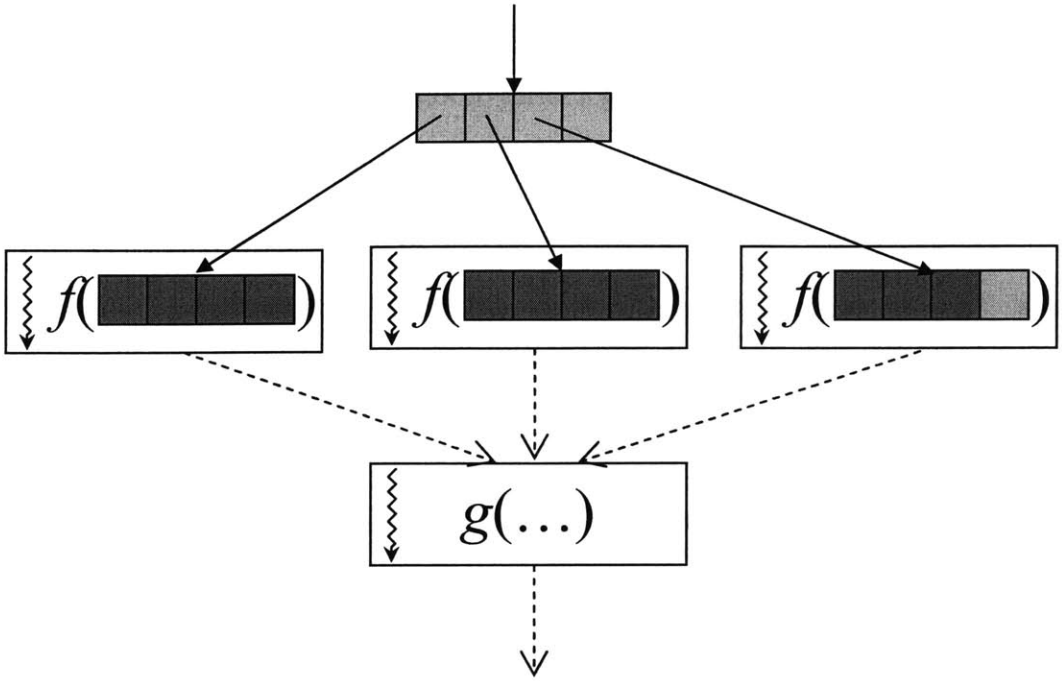

Figure 1: Executing Parallelizable Loops

Our plan for parallelization utilizes the Fresh Breeze primitives discussed in Section 2.1., and is depicted in ***Figure 1***. When encountering a parallelizable loop, a master thread will partition the loop's iterations into a set of intervals, and issue a series of Spawn instructions creating a slave thread for each interval. Each slave thread will be passed its interval, and the inputs to the loop. If a slave thread will only need a subset of an input array's elements, then it can just be passed a sub-tree of the array, possibly just one leaf chunk, containing those elements. The slave will then compute a set of output values. If the loop is constructing an array, then the slave's output will include a sub-tree of the array, or even just one leaf chunk of the array, depending on the granularity of the parallelization. The slave will place the output values on a chunk, and issue an EnterJoinResult instruction with the UID of that chunk. If there

is only one output value, then the slave can just enter that value directly. These computations by the slave threads are identified by the function *f* in ***Figure 1***.

After issuing all the Spawn instructions, the master thread will then retrieve the values computed by the slave threads by issuing a series of ReadJoinResult instructions. For scalar outputs, the master thread will use the corresponding operator to accumulate each result. For array outputs, the thread will build up an array tree out of the sub-trees returned by the slave threads. This computation by the master thread is denoted by the function *g* in ***Figure 1***.

This parallelization plan has several consequences. First, the operators used to combine scalar values into a scalar output must be associative and commutative, to guarantee correctness when iterations are partitioned among slave threads. This is a general parallelization requirement not specific to Fresh Breeze. Second, when an output is a new array, the loop's iterations must produce the array elements in order, either first to last or last to first. Otherwise, the sub-trees returned by the slave threads might have overlapping index ranges, and the master thread will not be able to easily combine these overlapping sub-trees into a tree for the final array. In the standard mutable memory model, this requirement can be relaxed to specifying that each array element is set by exactly one iteration, but they can be set in any order.

```
public static double dotProduct(double[] a, double[] b) {
   double sum = 0;
   for (int i = 0; i < a.length; i++) {
      sum += a[i] * b[i];
   }
   return sum;
}

public static double[] arraySum(double a[], double[] b) {
   double[] c = new double[a.length];
   for (int i = 0; i < a.length; i++) {
      c[i] = a[i] + b[i];
   }
   return c;
}
```

Figure 2: Parallelizable Loops

In ***Figure 2*** we see two examples of loops that meet these requirements. In the *dotProduct()* example, the loop takes in two arrays and outputs a scalar value. The values can be combined using double-precision floating point addition. While this operation is not strictly associative due to rounding errors, for any particular grouping it guarantees a consistent, repeatable result. This is sufficient for us to treat float point addition like an associative operator. If we describe the execution of *dotProduct()* in terms of ***Figure 1***, the function f computed by each slave thread would be a dot product of the vectors defined by a subinterval of the array indices. The function g computed by the master thread would be the summation of the values returned by the slaves.

In the *arraySum()* example a new array is constructed by adding the corresponding elements of two input arrays. Here, the slaves threads would be handed sub-trees of the input arrays, a and b. They would each construct and return the corresponding sub-tree of the output array, c. The master thread would then combine the returned sub-trees into a full tree to represent the output array.

In this discussion we have limited ourselves to the parallelization primitives described in Section 2.1. Fresh Breeze may also support additional kinds of spawn and join instructions which perform the associative accumulations or array constructions directly in hardware. Having the master thread issue one of these instructions, rather than a multiplicity of Spawns followed by a series of ReadJoinResults, may greatly improve performance.

Specific details on how the compiler identifies and represents these parallelizable loops are found in Section 4.3.3, which describes the Fresh Breeze-specific analyses and transformations performed by the compiler on the intermediate representation.

3. Compiler Design

The compiler is designed such that the majority of the work is independent of the specifics of Java Bytecode and the Fresh Breeze instruction set, and will therefore be useful in other settings. To that end, the compiler consists of several phases which operate on different intermediate representations. The initial input is Java Bytecode, which can be generated using an existing Java compiler. In the first phase, the Bytecode is checked to satisfy the requirements of Functional Java and converted to a 3-Operand flat intermediate representation with explicit array manipulations operations. In the next phase Static Data Flow Graphs are constructed from the 3-Operand IR. The Data Flow Graph Representation is then subject to a series of analysis and transformations, including transformations that normalize the representation, standard optimizing transformations common to many compilers, and Fresh Breeze specific transformations that introduce nodes that make it easier to produce efficient, parallel Fresh Breeze code. These transformations are repeated until a fixed point is reached. The following phase flattens the Data Flow Graphs, producing a 3-Operand representation which manipulates chunks rather than arrays. Finally, in the last phase, Fresh Breeze instructions are generated from the 3-Operand representation. This structure can be seen in ***Figure 3***.

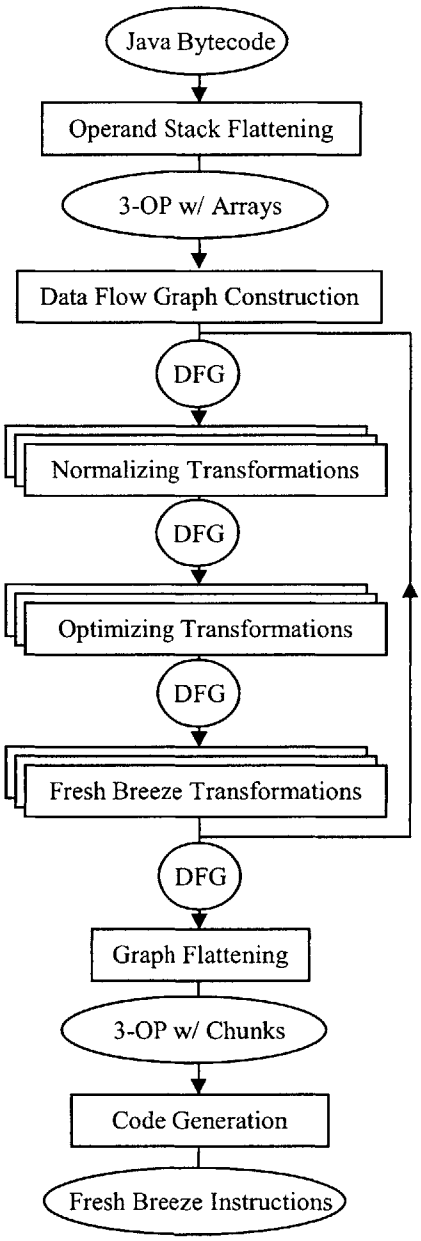

Figure 3: Compiler Design

The remainder of this chapter describes the compiler's input language and intermediate representations. These include the Functional Java language, the 3-Operand Flat Representation, and the Data Flow Graph Representation.

3.1. Functional Subset of Java

Due to the reliance on existing Java compilers, Functional Java must be a subset of Java. In particular, it should be described by a set of rules stating which existing Java expressions are allowed and which are not. To give a better understanding of the constraints imposed by the Fresh Breeze memory architecture, we describe in this section a specification for Functional Java which includes Object types, even though our present compiler implementation does not allow their use.

The main requirements placed on Functional Java are that the generated code cannot mutate sealed Chunks or create reference cycles of Chunks. We will make several assumptions about how the generated code maps Java fields and Objects to Fresh Breeze memory structures. First, each Java object will reside on one or more Chunks. So, any Java object reference can be considered a reference to a Chunk. Additionally, any other globally accessible data will also reside on Chunks. These assumptions, when combined with the stated requirements, imply some general rules about Functional Java. First, all Object fields, of both reference and primitive type must be immutable, since they reside on immutable chunks. Similarly, static fields must also be immutable. Furthermore, it should not be possible to create reference cycles between Objects or allow Objects to contain a self-references, since this will create reference cycles in the shared memory. While a purely functional language may also require local variables to be immutable, this is not required to make Functional Java implementable on Fresh Breeze.

Java's *final* keyword can be used enforce the immutability of both object fields and static fields. However, this alone does not prevent the formation of reference cycles and even some mutations. This is because a constructor can reference the generated object before setting its fields, and static field initializers can access the field before initializing its value. The code in ***Figure 4*** demonstrates both of these phenomena, showing the class *MutatingField* which

mutates a *static final* field, and the class SelfReference which creates a self-reference using a final field.

```
class MutatingField {
   static final int foo = initialFoo();

   static int initialFoo() {
      System.out.println(foo);
      return 1;
   }

   public static void main(String[] args) {
      System.out.println(foo);
   }
}

class SelfReference {
   final SelfRefernce other;
   SelfReference() {
   other = this;
   }
}
```

Figure 4: Limitations of final *Keyword*

To overcome these two limitations we impose additional constraints on Functional Java programs. First, we require that static fields are assigned solely constant (literal) values. Second, we enforce a default form for the constructor where each field value is passed in as a parameter. So, a class with N fields would have a constructor of the form shown in ***Figure 5***.

```
Constructor(Type1 field1, Type2 field2, … , TypeN fieldN) {
   this.field1 = field1;
   this.field2 = field2;
   …
   this.fieldN = fieldN;
}
```

Figure 5: Default Functional Java Constructor

The Java API is integrated with the language. For example, a large portion of the API can be reached by following the methods of *java.lang.Object*. The specifications of many of the Java API classes do not conform with the Functional Java requirements and would be difficult if not impossible to implement on Fresh Breeze. So, a separate designed Fresh Breeze API would be necessary to support a Fresh Breeze compiler which allowed for object types.

3.2. Functional Java with Arrays

Since this work is focused on compiling array manipulations, we ignore the aspects of Functional Java that deal with object types, and instead focus on arrays. Without object types, we are limited to a language which contains static methods, primitive types, and multidimensional arrays of primitives. Fortunately these constructs are sufficient for expressing the types of computations that we wish to compile (Linear Algebra and Spectral Methods).

Now, since Fresh Breeze arrays are immutable, it may be difficult to generate code from programs that contain arbitrary Java array manipulations. For example, in the Java procedure in ***Figure 6*** an array is passed in as an argument and then mutated. Generating Fresh Breeze code for this procedure while maintaining the semantics of Java would not be possible, since the chunks containing the array cannot be mutated.

```
void bad_negate(int[] a) {
   for (int i = 0; i < a.length; i++) {
      a[i] = -a[i];
   }
}
```

Figure 6: Procedure Violating Secure Arguments Principle

By mutating its arguments the above procedure violates the Secure Arguments principle of modular programming [3]. Such procedures can be prohibited from Functional Java, although in some cases, due to aliasing, additional analysis may be required to determine if a procedure is actually modifying its arguments. But, even if this restriction were enforced, it may not be sufficient to ensure simple code generation. ***Figure 7*** contains a procedure that does not violate the Secure Arguments principle, but still makes Fresh Breeze code generation difficult.

```
int[] example(boolean x) {
   int a[] = new int[1];
   int b[] = new int[1];
   if (x) {
      a = b;
   }
   a[0] = 1;
   return b;
}
```

Figure 7: Satisfying Secure Arguments Principle but relying on Array Mutation

By copying array references, the procedure makes it difficult to determine at code generation which array references are affected by an assignment to an array element. One way to avoid this problem is to preclude array reference re-assignment, statements of the form "a = b;" where a and b are arrays. This would be equivalent to marking all array declarations "final". An alternative would be to allow array reference re-assignment, but disallow array element assignments, or statements of the form "a[i] = x;". Then Functional Java arrays would truly be immutable, closely matching the underlying Fresh Breeze arrays. A static library method would be provided for creating new arrays based on existing ones. ***Figure 8*** contains a trivial example which exercises this syntax:

```
boolean alwaysTrue() {
   long[] a = new long[10];
   a = FBArray.set(a, a.length-1, 5);
   return a[9] == 5;
}
```

Figure 8: Array Syntax using Static Library methods

While explicit in its interpretation and closely related to the underlying representation, the syntax in ***Figure 8*** would be unfamiliar and awkward to programmers. So, we simplify array syntax by reinterpreting array element assignments to include a reassignment of the array reference. This reinterpretation also resolves the parameter problem we exposed in ***Figure 6***, since without mutation arrays are in essence passed by value.

```
int[] negate(int[] a) {
   for (int i = 0; i < a.length; i++) {
      a[i] = -a[i];
   }
   return a;
}
```

Figure 9: Functional Java Array Syntax

While reinterpreting the meaning of array-element assignment preserves the general style of the code, in some cases it can make Functional Java execution inconsistent with Java. It should be possible to create an analyzer which statically determines if a particular program may give different results when interpreted as Functional Java rather than Java, and reject such programs from compilation. We did not implement such an analysis, leaving it up to the programmer to determine if the code is correct under the Functional Java interpretation of array element assignment. Designing such an analysis is an interesting avenue for future work. In practice, this array syntax and semantics worked well when implementing the tests and benchmarks for the compiler. ***Figure 9*** demonstrates how array negation may be written using the Functional Java array syntax.

3.3. Flat Representation

We use a generic flat representation of Functional Java methods to separate our work from the specifics of both Java Bytecode and the Fresh Breeze instruction set. This makes the compiler more general and useful, making it possible to add support for additional source language feature or accommodate changes to the Fresh Breeze instruction set with minimal changes to the compiler. Additionally, by abstracting away irrelevant aspects of the source and destination languages, such as Bytecode's operand stack or Fresh Breeze's register layout, we can simplify the Data Flow Graph Construction and Graph Flattening phases of the compiler.

The Flat IR is a three operand representation, with statements that contain two input operands and 1 output operand. This matches Fresh Breeze's RISC instruction set, where most

instructions generally have two sources and one destination register. The operands can be constant values or references to local variables. Local variables are typed, with types corresponding to the arithmetic types supported by Fresh Breeze (Int, Long, Float, Double). When constructed from Bytecode the Flat IR can contain Array types. The type of an array's elements is not included in the array's type information, but rather specified by the statements that construct and access arrays. Similarly, before being used to generate Fresh Breeze instructions, the Flat IR can contain Chunk types, whose sealed or unsealed status can be specified in the Type information.

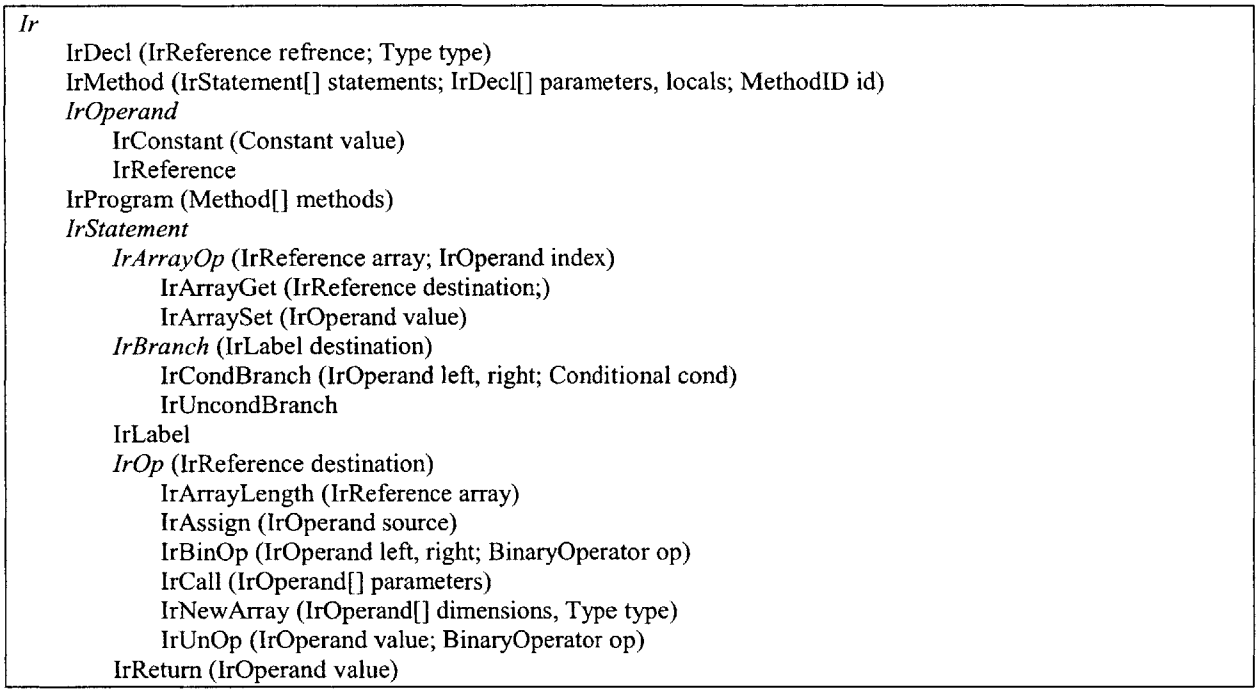

Figure 10: Flat IR Class Hierarchy and Structure

Figure 10 depicts the class hierarchy of the Flat IR. A program consists of a list of methods. Each method is uniquely identified by package, class, name, parameter types, and return type. A method contains an ordered list of statements, and defines the types of its parameters and local variables. The statements take in operands, which are either constants or reference to temporaries. While most statements take in two operands, some like method calls

have a variable number of operands. Control flow is facilitated by branch statements that point to label statements. Conditional branches take two input operands and specify a comparison operator. The Flat IR is immutable and implements the visitor design pattern [5] to facilitate traversal.

3.4. Data Flow Graph Representation

The bulk of the analysis and transformations performed by the compiler are done on a Data Flow Graph Representation. This representation makes data dependencies explicit, simplifying many analyses. Data Flow Graphs assume that functions produce new output data from input data without mutation, complementing the Fresh Breeze memory model. This representation is inspired by the intermediate representation used in the Sisal compiler [8].

3.4.1. Data Flow Graphs

A Data Flow Graph is a labeled directed acyclic graph. The edges carry values, and the nodes are labeled by functions which take in values from their incoming edges and compute new values for the outgoing edges. A set of edges are designated as input edges, and correspond to the inputs of the graph. Similarly, a different set of edges corresponds to the outputs.

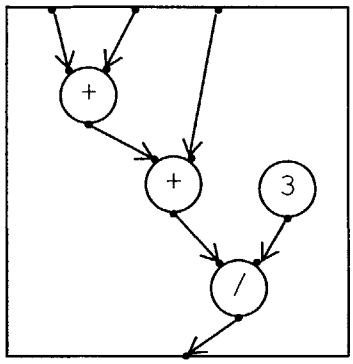

Figure 11: Averaging 3 Inputs

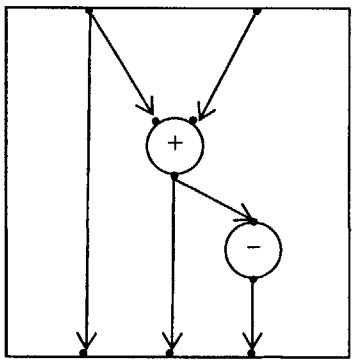

Figure 12: Interesting DFG

Figure 11 depicts a Data Flow Graph which computes the average of three numbers. The graph contains three input edges and one output edge. The inputs are summed using two-input

addition nodes and then divided by three, a value produced by a no-input constant node. In general, Data Flow Graphs can have arbitrary numbers of inputs and outputs and the value computed by a single node can be used in multiple places. Values can also be carried from the graph's inputs to outputs directly, bypassing any intermediate nodes. ***Figure 12*** depicts these possibilities.

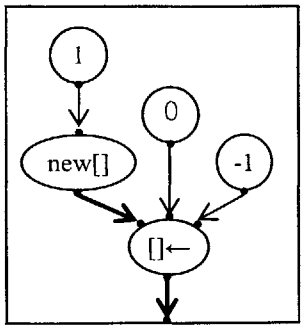

Figure 13: Constructing 1-Element Array

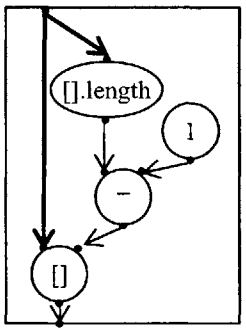

Figure 14: Retrieving Last Array Element

In addition to computing standard arithmetic operations, Data Flow Graphs can represent operations on arrays. We utilize four kinds of nodes that operate on arrays: creating a new array given the size as input, outputting the length of an array, getting the value of an element at a particular index, and constructing a new array from an existing one by changing the value of one element at a given index. These nodes are shown in ***Figure 13*** which depicts a graph that outputs a new one-element array containing the value "-1", and in ***Figure 14*** with a graph that retrieves the last element of an input array. We use heavy lines for edges that contain array pointers.

3.4.2. Compound Nodes

In the Data Flow Graphs presented so far, each node is evaluated exactly once, corresponding to straight line code within a program. To represent computation with control flow, we introduce compound nodes. These nodes contain, as part of their labeling, internal Data Flow Graphs, giving the Data Flow Graph representation a recursive tree-like structure.

Different kinds of compound nodes have different rules for which internal graph is evaluated and how values are passed into and out of that graph, allowing for conditionals, loops and other kinds of control flow.

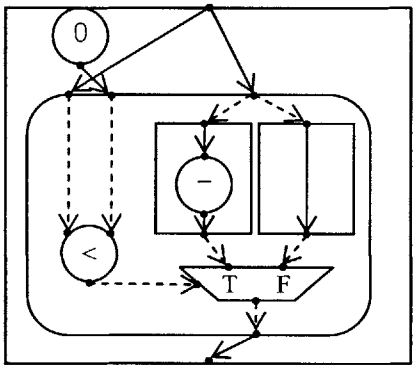

Figure 15: DFG for Absolute Value

```
int abs(int x) {
   if (x < 0) {
      return -x;
   } else {
      return x;
   }
}
```

Figure 16: Code for Absolute Value

In general, the graphs within a compound node contain identical sets of input edges and identical sets output edges, both in number and type. The inputs to the node flow as inputs to the graphs, and the outputs of the graphs flow as outputs of the node. A Conditional node is a kind of compound node that can be used to represent simple *if/else* statements. It contains two graphs, one for the true case and one for the false case. The node has two special inputs which do not flow into its graphs, but are compared using a conditional operator to determine which graph gets evaluated. ***Figure 15*** contains an example of a conditional node which computes absolute value (the corresponding Java code is shown in ***Figure 16***). The node contains two one-input, one-output data flow graphs, one which negates its input and one which does not. The node has three inputs, two of which are compared using the "less than" operator, and one which is passed into both graphs. It has one output which is chosen from the graph corresponding to the result of the "less than" comparison. The figure uses dotted arrows to represent this evaluation process.

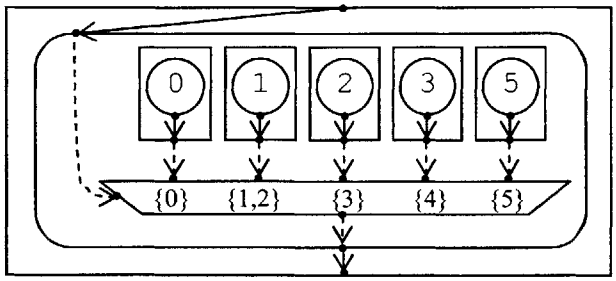

Figure 17: DFG with Switch Node

```
int fibonacci(int x) {
   switch(x) {
   case 0: return 0;
   case 1:
   case 2: return 1;
   case 3: return 2;
   case 4: return 3;
   case 5: return 5;
   }
}
```

Figure 18: Code with Switch Statement

While Conditional nodes are sufficient for representing any computation which does not contain loops in the control flow, other kinds of compound nodes can represent that computation more directly, making it simpler to generate quality code. A switch node is one such compound node that can be used to better represent Java switch statements. Each of a switch node's sub-graphs is associated with a disjoint set of integers, and the node takes one special input, an integer which selects the graph that is evaluated. ***Figure 17*** contains a switch node which evaluates the first few terms of the Fibonacci sequence, and corresponds to the code in ***Figure 18***.

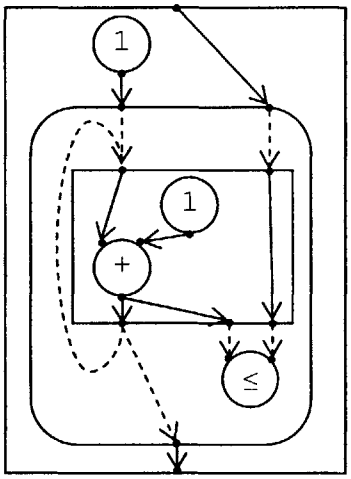

Figure 19: DFG with Loop Node

```
int count(int x) {
   int i = 0;
   do {
      i = i + 1;
   } while (i <= x);
   return i;
}
```

Figure 20: Code with Control Loop

In order to represent loops in control flow we need a different kind of compound node. A Loop node corresponds closely to Java's *do{}while()* statement. A loop node contains one sub-graph, which is reevaluated for each iteration of the loop. The sub-graph has two special outputs

which are compared with a conditional operator to determine loop terminations. Other sub-graph outputs, designated as loop edges, are fed back into the inputs for the next iteration. The sub graph can have additional inputs which are not designated as loop edges, whose values always flow from the loop node's inputs. For the first iteration values for the sub-graph's loop inputs flow from the node's inputs, and after the final iteration the sub-graph's loop outputs flow as the loop node's outputs. ***Figure 19*** contains a Data Flow Graph with a loop node which corresponds to the simple *do{}while()* statement in ***Figure 20***.

Loop nodes and Conditional nodes are sufficient for representing the control flow of arbitrary Functional Java programs. Logical operators like ANDs and ORs can be constructed by combining Conditional nodes tied with edges carrying boolean values. Other kinds of loop statements, like *for(){}* and *while(){}*, can be simulated by wrapping a Loop node in a Conditional node to condition the first iteration.

3.4.3. Concrete Representation

The specific Data Flow Graph Representation used by the compiler is based on the conceptual Data Flow Graphs defined above. The representation uses special nodes to identify a graph's input and output edges. A Data Flow Graph contains an input-less Source Node whose outputs determine the inputs to the graph, and an output-less Sink Node whose inputs determine the outputs of the graph. This design decision makes it possible to write code which analyzes or transforms data flow graphs without including special cases for a graph's input and output edges.

The nodes within a Data Flow Graph are ordered such that the Source Node is first, the Sink Node is last, and other nodes appear after the nodes they depend on. This invariant simplifies both the analysis of the graphs, and the graph flattening phase. Additionally, this invariant guarantees that the graphs are acyclic. While the conceptual Data Flow Graph

drawings omit type information, nodes in the concrete Data Flow Graph representation specify the types of their output ports. By enforcing consistency of this type information across compound nodes it is possible to sanity-check the correctness of a Data Flow Graph after a transformation.

The edge information is stored in Use-Def format, where each node contains a description of its input edges. So in the concrete representation, unlike the conceptual drawings, edges point from the node that uses a value to the node that defines it. Both nodes and their output ports are numbers, such that the start of an edge in the graph can be specified by a pair of numbers, the node number and port number.

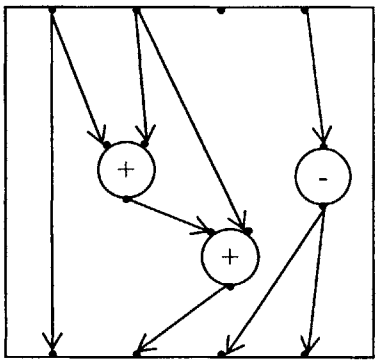

Figure 21: Conceptual DFG

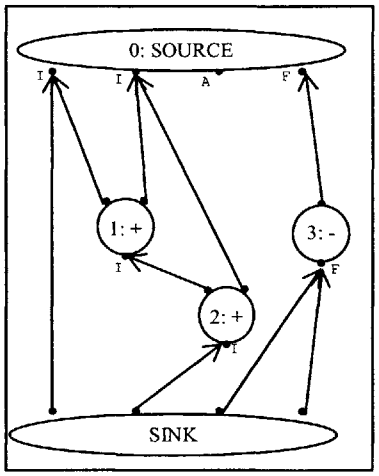

Figure 22: Concrete DFG

The two figures above demonstrate the difference between the conceptual description of Data Flow Graphs and our concrete implementation. ***Figure 21*** shows a sample conceptual Data Flow Graph, while ***Figure 22*** show how it is represented by the compiler.

To maintain uniformity, compound nodes identify their special ports by specifying an input offset and an output offset. The input offset specifies a prefix of inputs that are not passed into the sub-graphs, while the output offset specifies a prefix of sink ports that are not passed from the sub-graphs to the compound node's output ports. So, for a Conditional node the input

offset is 2, for the two special inputs that are compared, and the output offset is 0. Similarly, for a Switch node the output offset is also 0, and the inputs offset is 1, for the integer input that selects the sub-graph. The input offset for a loop node is 0 since all the inputs flow to the sub-graph, but the output offset is 2, for the two sink ports that are compared during each iteration. In addition to the input offset and output offset, loop nodes specify a loop port count identifying the number of sink edges whose values wrap around during each iteration of the loop.

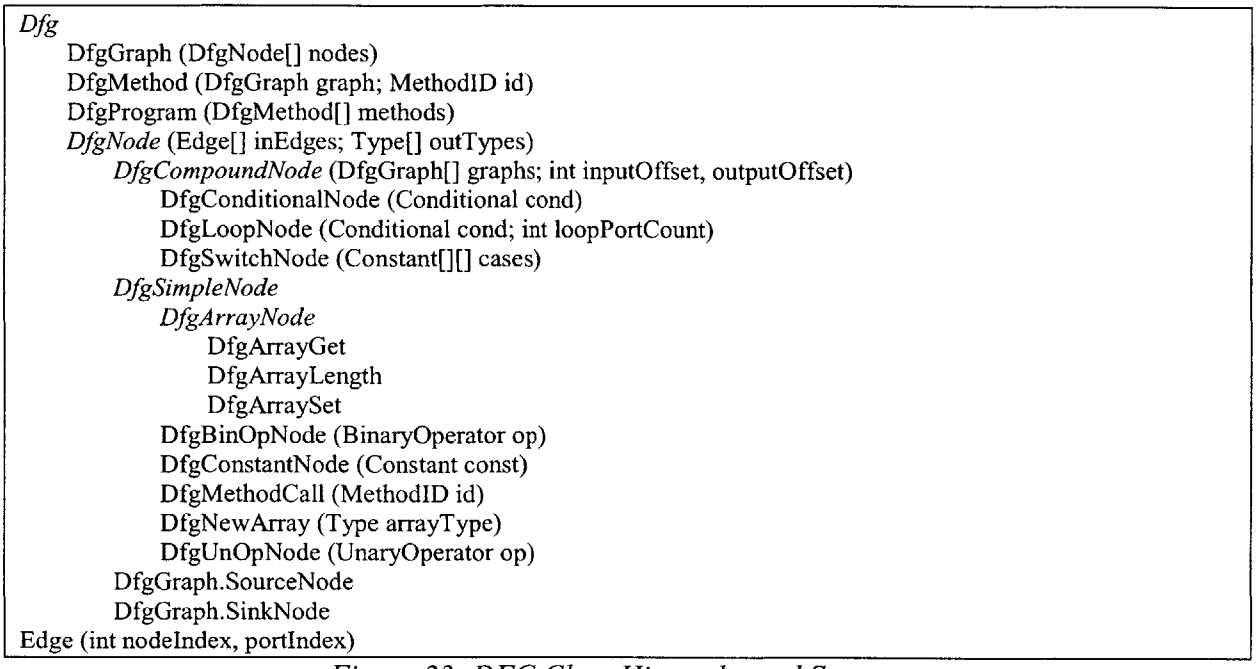

Figure 23: DFG Class Hierarchy and Structure

Figure 23 depicts the class hierarchy and structure of the Data Flow Graph Representation. Like the Flat IR, a program consists of a list of named methods. A method contains a Data Flow Graph with a number of input edges corresponding to the method's parameters, and one output edge corresponding to the method's return value. A Data Flow Graph consists of a list of nodes, starting with a Source Node and ending with Sink Node. Each node has a list of incoming edges, each specified by a node index and a port index, and a list of output types. Simple nodes have one output port, while compound nodes contain a list of Data

Flow Graphs. The unary and binary operators, as well as conditionals, supported by the Data Flow Graph Representation are identical to those in the Flat IR. Data Flow Graphs and their nodes are immutable, and are traversed by dispatching on type using the *instanceof* operator.

The nodes listed in ***Figure 23*** are sufficient for representing the structure of Functional Java programs. The compiler supports additional kinds of nodes which are created during the analysis and transformation phase. These additional nodes represent the information gathered by the analysis, and make it possible to generate improved Fresh Breeze code which better utilizes the Fresh Breeze memory model and takes advantage of the parallelism available in the Fresh Breeze architecture. These nodes are described in conjunction with their associated transformations in Section 4.3.3.

4. Compiler Implementation

The goal of our compiler implementation is to demonstrate the feasibility of effectively compiling Functional Java for Fresh Breeze. While the implementation sets up a framework for a robust compiler, we focused our development efforts on the novel aspects of the compiler, specifically the non-standard transformations in the Data Flow Graph Transformation phase. More common phases, such as the Operand Stack Flattening and Data Flow Graph construction phases, have more skeletal implementation, which are sufficient for testing the Transformation phase, but are not necessarily robust or comprehensive enough for extensive use. Finally, the last few phases of the compiler, including the graph flattening and code generation phase have not been implement, and are described in Future Work (Chapter 6).

4.1. Flattening Byte Code

Rather than compiling Functional Java source files directly, we first use an existing Java compiler, such as Javac or the Eclipse Incremental Compiler, to generate Java class files. Class files contain Java Bytecode, a flat stack-based representation. The first phase of the compiler converts these Java class files into the 3-Operand Flat Intermediate Representation, while checking that the input confirms to the Functional Java specifications. This approach simplifies development effort, and provides compatibility for a wide range of Java language constructs.

The conversion from Bytecode to the Flat IR is based on previous work by Scott Beamer, who wrote a direct translator from Bytecode to Fresh Breeze instructions. Java Bytecode assumes a fixed-sized array of local variables, and utilizes a stack of operands. Most Bytecode instructions pop their parameters from the operand stack, and push their output back onto the stack. Instructions are provided for moving operands to and from the local variable array and for

rearranging operands on the stack. Control flow is accomplished by branch instructions that specify the address of their destination. To simplify the conversion we assume that the operand stack is empty at the start and end of basic blocks. While this invariant is not required by the class file format, the assumption is mostly valid for Bytecode generated by Javac and the Eclipse Incremental Compiler. Without this assumption we would need to perform data-flow analysis on the Bytecode to determine the status of the operand stack.

Given the simplifying assumption, the conversion can proceed in one linear pass over the Bytecode. Each entry in the local variable table is assigned a temporary in the Flat IR, so a load or a store to a local variable becomes an access or an assignment to the temporary. The entries on the operand stack cannot be handled in the same way, since the same stack location can hold different data-types during execution. So, during the conversion, we statically simulate the stack contents, maintaining a stack of operands, which are either constants or references to temporaries in the Flat IR. While some Bytecode conditional branches specify the comparison operator, others rely on an earlier comparison instruction. Since the Flat IR does not have separate compare and branch instructions, we use static analysis to merge the separate Bytecode instructions into one conditional branch. This requires allowing an additional kind of stack entry on our simulated operand stack, which stores the two operands of a comparison.

When converting a Bytecode instruction which loads a local variable onto the stack, either a reference to the variable's associated temporary can be pushed on the stack, or a new temporary can be pushed on the stack while an assignment to the new temporary is added to the generated Flat IR. If the local variable holds an array then creating a new temporary will change the meaning of the code, since the interpretation of array element assignments discussed in Section 2.3. implies that subsequent array element assignment operations on the stack entry will

not affect the value of the array in the local variable. On the other hand, there are several Bytecode instructions, such as *iinc*, which manipulate the values in the local variable table directly, bypassing the operand stack. If new temporaries are not created during loads, then these instructions will have the unintended consequence of changing the value of operands already pushed on the stack. Correct conversion is possible even in light of these contradicting observations, because none of the instructions that manipulate the local variable table directly operate on arrays. So, we create new temporaries only when loading primitive-typed local variables.

Our assumption about the contents of the operand stack at the end of basic blocks does not hold in the case when Java's ternary operator (*condition* ? *trueValue* : *falseValue*) is compiled. To deal with this special case, we create a new IR temporary for each such conditional, maintaining a mapping from the Bytecode index where the two conditional cases merge to the associated temporary.

This implementation of the conversion from Bytecode to Flat IR meets ours goals. It is simple and reasonably efficient. The conversion is correct given that the Bytecode was generated by the specific Java compiler it was designed for. While a more robust implementation would be possible, and might at some point be added to the Fresh Breeze compiler, it is not necessary for this work, since the main focus of the thesis is on the Data Flow Graph Representation rather than the Flat Intermediate Representation.

4.2. Constructing Data Flow Graphs

The second phase of the compiler constructs Data Flow Graphs from the Flat Intermediate Representation. After this point, the Flat IR is discarded and all transformations are done on the Data Flow Graph Representation directly. While the representation is general

enough to represent arbitrary computation, our goal was to develop a simple routine for converting most basic Functional Java programs.

```
boolean hasTrue(boolean[][] bitMatrix) {
   for (int x = 0; x < bitMatrix.length; x++) {
      for (int y = 0; y < bitMatrix[x].length; y++) {
         if (bitMatrix[x][y]) {
            return true;
         }
      }
   }
   return false;
}

boolean hasTrue_equivalent(boolean[][] bitMatrix) {
   boolean returnFlag = false;;
   for (int x = 0; !returnFlag && x < bitMatrix.length; x++) {
      for (int y = 0; !returnFlag && y < bitMatrix[x].length; y++) {
         if (bitMatrix[x][y]) {
            returnFlag = true;
         }
      }
   }
   return returnFlag;
}
```

Figure 24: Simplifying Control Flow by introducing Boolean Variables

The conversion is similar to decompilation, since we are imposing structure on the control flow of low level instructions. But the conversion is more involved than just decompilation, since some java constructs, such as return statements nested within loops, do not have strictly corresponding data flow graphs. Equivalent programs that do have directly corresponding data flow graphs can be created by introducing additional boolean variables that carry information which is implicit in the original control flow. ***Figure 24*** contains an example of such a Java method, *hasTrue*, which iterates over a 2-dimensional array and has a return statement nested inside two loops. By introducing the *returnFlag* boolean variable we can convert *hasTrue* to an equivalent method which does not have this property. While this type of transformation is possible for all Functional Java constructs that do not have directly corresponding Data Flow Graph nodes, we did not implement support for most of these

constructs since they can be easily avoided. In addition to nested returns we don't handle nested continues or breaks, and compound logical operators. Future versions of the compiler may provide more complete support.

Data Flow Graph construction is implemented as a recursive procedure. The procedure takes in an index range over an array of Flat IR statements, and returns a Data Flow Graph corresponding to the statements within the range. The procedure fails if it encounters branch statements whose targets labels are outside of the range. A mapping is maintained from Flat IR temporaries to the Data Flow Graph edge that contains the temporary's current value. When control flow corresponding to an *if(){}else{}* statement is encountered, two recursive calls are made to construct a corresponding Conditional Node, one to generate each of the Conditional Node's sub-graphs. Similarly, when encountering a backwards branch corresponding to a Loop Node, a recursive call is made to generate a Data Flow Graph for the body of the loop. In both of these cases, it is assumed that any of the defined temporaries may be used by the recursive calls, so the edges corresponding to all defined temporaries are given as inputs to the compound node under construction.

4.3. Data Flow Graph Transformations

In the third phase of the compiler a series of transformations are applied to the Data Flow Graph representation of the Functional Java program. Since the representation is immutable, each transformation creates a new program graph from a given program graph. There is a framework for applying the transformations, which cycles through a given set of transformations until a fixed point is reached and the program graph stops changing. The transformations are designed in such a way that a fix point will be reached for any input Functional Java program.

The compiler implements three kinds of transformations. Normalizing transformations reduce syntactic redundancy in the data flow graphs without changing their semantic meaning. Standard Optimizing transformations apply optimizations common to most compilers, improving the eventual performance of the generated code, and simplifying further analysis of the data flow graphs. Fresh Breeze-Specific Transformations identify opportunities for generating code which takes advantage of the Fresh Breeze memory model and model of execution, and introduce new kinds of nodes which describe these opportunities.

4.3.1. Normalizing the Representation

The Data Flow Graph representation defined in Section 3.4. allows for multiple representations of the same underlying computation. When performing analysis on Data Flow Graphs it may be useful to assume certain properties which are not guaranteed by the specification of Data Flow Graphs. For example, it might be useful to assume that any value passed into a compound node is used within that node. The normalizing transformations presented in this section provide these kinds of properties, by changing the syntactic structures of the graphs without affecting their semantic meaning.

4.3.1.1. Dead Input Ports

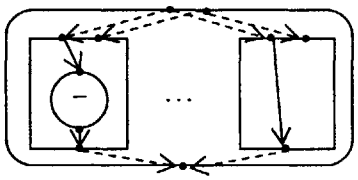

Figure 25: Compound Node with Dead Input

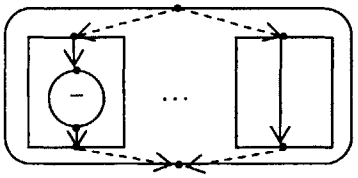

Figure 26: Dead Input Port Removed

The specification of Data Flow Graphs allows compound nodes to have inputs that are not used by any of their sub-graphs. In fact, the Data Flow Graph construction described in Section 4.2. assumes that all available values are going to be needed, and generated compound nodes that take in every available edge. These unused, or dead, input ports generate unnecessary

dependencies in the Data Flow Graph, making further analysis of the graph less productive. The Dead Input Port Removal transformation identifies and removes input ports that are not used by any of a compound node's graphs. The transformation performs a traversal of each sub-graph to identify unused inputs. For Loop Nodes all loop ports are assumed to be used, or live, and are not removed by this transformation. ***Figure 25*** shows a sample compound node with a dead input. ***Figure 26*** has the node with the input removed.

4.3.1.2. Dead Output Ports

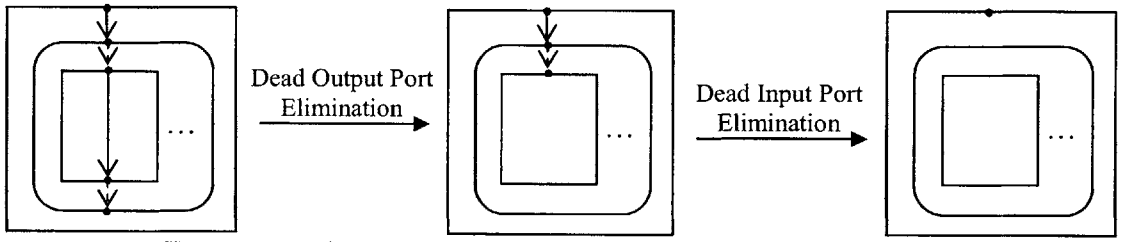

Figure 27: Eliminating Unused Outputs to Eliminate Unused Inputs

In addition to unnecessary inputs, compound nodes can have unnecessary outputs. These are output ports that do not have any outgoing edges in the surrounding graph. Unnecessary, or dead, outputs can prevent the compiler from recognizing dead inputs, since the dead outputs preserve unnecessary edges in a compound node's sub-graphs. ***Figure 27*** has an example of such a compound node, where Dead Output Port Elimination is necessary, before Dead Input Port Elimination can proceed.

4.3.1.3. Copy Propagation

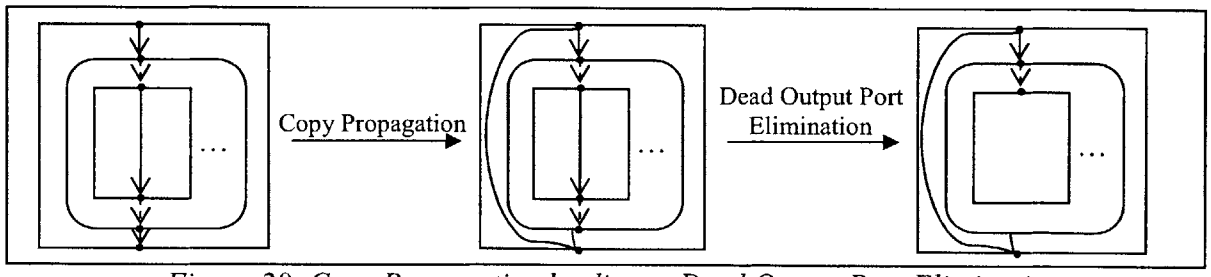

Figure 28: Copy Propagation leading to Dead Output Port Elimination

In traditional compilers, copy propagation replaced targets of direct assignments with their values. While Data Flow Graphs do not have assignment operators, and therefore don't need most types of copy propagation, there are circumstances under which a form of copy propagation can be applied. In particular, when an output from a compound is just a copy of one of the inputs, we can bypass the compound node all together for that output value. This transformation is necessary because the Data Flow Graph Construction phase of the compiler passes all locals, even those that are not modified, through compound nodes. ***Figure 28*** shows simple Data Flow Graph where Dead Output Port elimination cannot be performed without Copy Propagation. When combined with Dead Input Port elimination and Dead Output Port elimination, Copy Propagation guarantees that inputs to compounds nodes are actually used within the nodes.

4.3.1.4. Unnecessary Loop Ports

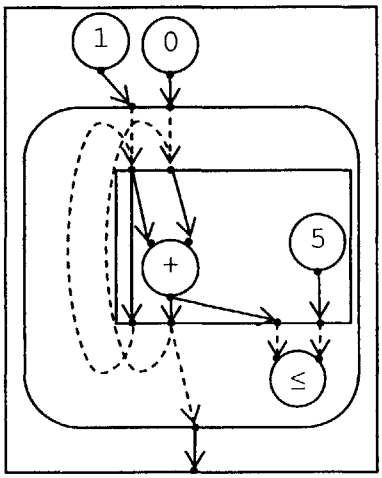

Figure 29: Unnecessary Loop Port

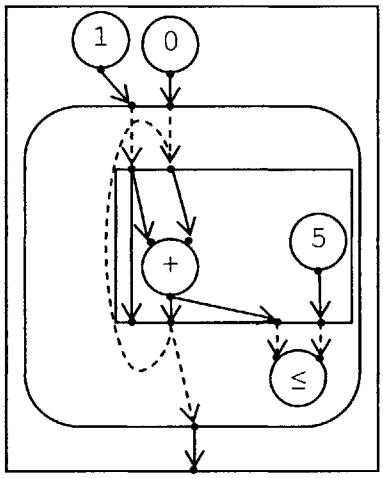

Figure 30: Unnecessary Loop Port Relabeled

Loop nodes specify a set of output ports whose values wrap around during each iteration of the loop. The Data Flow Graph construction phase conservatively labels all inputs to a Loop Node to be such loop ports. Many variables within a loop body do not change their value from one iteration to another, and therefore do not need to be labeled as loop ports. Unnecessary loop

ports stifle additional analysis and transformation, since they cannot be treated like normal inputs and outputs. For example, Dead Output Port elimination ignores loop ports, since their values may be reused within the Loop Node's sub-graph. The compiler identifies two types of unnecessary loop ports: those whose output value is a copy of the corresponding input value, and those whose output is a copy of input which is not a loop port. During Unnecessary Loop Port Elimination, the compiler removes the loop port label from input/output pairs that satisfy one of these conditions, leaving additional cleanup to the other normalizing transformations. The Loop Node in ***Figure 29*** counts up from 0 to 5 in increments of 1, and contains an unnecessary loop port. The loop port is relabeled in ***Figure 30***.

4.3.1.5. Subtraction

Another redundancy in the Data Flow Graph representation comes from the inclusion of both one-input negation operators and two-input subtraction operators. Excluding edge effects and overflows, subtraction is equivalent to addition with one operand negated. So, the first transformation applied by the compiler is to replace all binary subtraction nodes with equivalent combinations of binary addition and unary negation nodes. This transformation simplifies the implementation of subsequent analysis and transformation by reducing the number of binary operators. After all other transformations are complete, but before Data Flow Graph Flattening, the compiler reintroduces binary negation nodes, turning all additions where one of the operands is negated into a subtraction. These simple transformations go a long way to simplifying the implementation of Constant Folding (Section 4.3.2.1), Induction Node Creation (Section 4.3.3.1.) and Accumulation Node Creation (Section 4.3.3.4.).

4.3.2. Standard Optimizing Transformations

The compiler implements a set of optimizing transformations common to most optimizing compilers [1]. There are several reasons for implementing these transformations. First, we demonstrate that the transformation framework and the Data Flow Graph Representation are well suited for implementing standard optimizations. Second, many of these transformations generate more opportunities for applying the Fresh Breeze-Specific Transformations, described in Section 4.3.3., which are the core of this work. Finally, these optimizations will improve the overall performance of the code generated by this compiler, making any future benchmarks more meaningful.

The normalizing transformations in Section 4.3.1. while guaranteeing that the Data Flow Graph representation satisfied certain useful invariants, did not change the structure of the computation or effecting the generated Fresh Breeze code. The transformations presented in this section do change the semantics of the graphs, but preserve the output produced by the Functional Java program.

4.3.2.1. Constant Folding and Propagation

Constant folding is a common optimization which statically evaluates or simplifies expressions whose term values are known at compile time. Applying this optimization has several advantages, including reducing the number of instructions produced for a given program, and enabling other optimizations that depend on constant values.

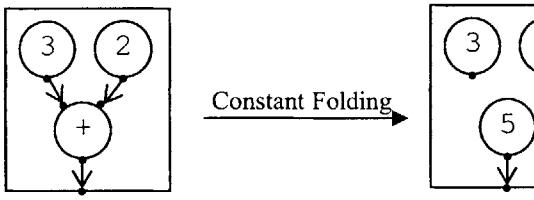

Figure 31: Constant Folding Constant Inputs

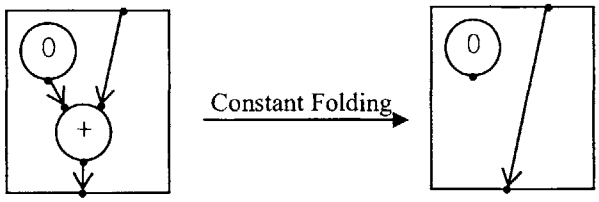

Figure 32: Partial Constant Folding

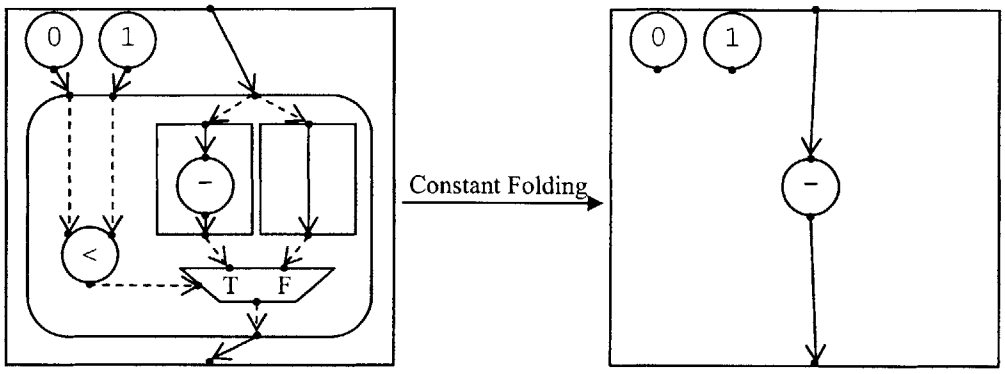

Figure 33: Constant Folding Compound Nodes

Applying this optimization to Data Flow Graphs is very straightforward and efficient, since expressions, or nodes, point directly to their inputs. While evaluating some nodes requires all their inputs to be constant, others can be evaluated when a subset of the inputs are known. ***Figure 31*** shows Constant Folding applied when all inputs are constant, while in ***Figure 32*** only one of the inputs are constant. This transformation can also be applied to compound nodes, such as conditional nodes whose conditional can be pre-evaluated. In ***Figure 33*** a conditional node is replace with one of its sub-graphs.

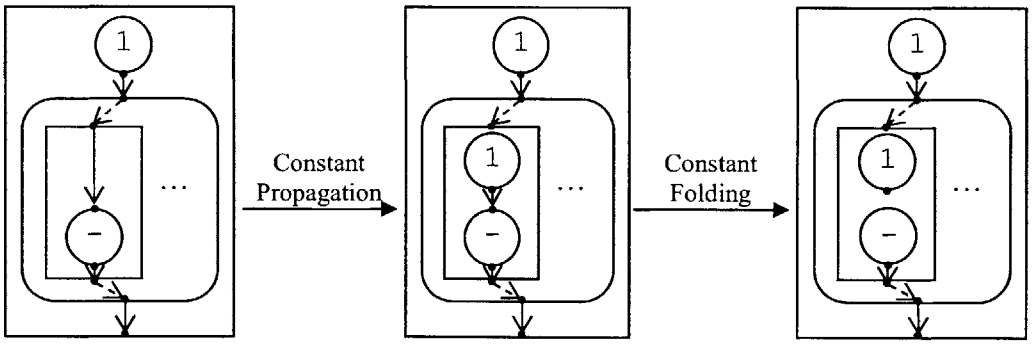

Figure 34: Propagating a Constant into a Compound Node enables Constant Folding

A constant input to a compound node can also create opportunities for Constant Folding within the node's sub-graphs, but these opportunities cannot be identified by checking whether the input edges of a node point to a constant node. This problem is solved by propagating constant inputs into compound nodes. We must be careful to only propagate the constants into sub-graphs where they will be used, since otherwise we cannot guarantee that alternating

Constant Propagation with Dead Code Elimination (see Section 4.3.2.2.) will reach a fix point. ***Figure 34*** contains an example of Constant Propagation into a Compound Node which leads to additional Constant Folding.

4.3.2.2. Dead Code

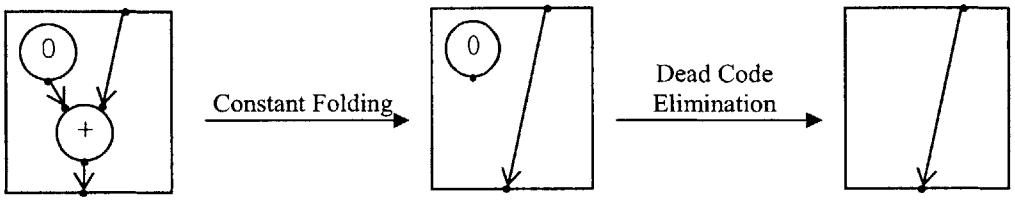

Figure 35: Constant Folding generating opportunities for Dead Code Elimination

The goal of Dead Code Elimination is to eliminate expressions whose outputs are never used, reducing the size of the generated code and eliminating unnecessary instructions. While a well written program should not have much dead code, other transformations like Constant Folding can generate opportunities for Dead Code Elimination. ***Figure 35*** has such an example. The compiler implements this transformation by performing a reverse scan of a graph's nodes, assuming the sink node is used, and marking all the inputs to a used node as used. This simple analysis is sufficient in most cases, since the normalizing transformations of Section 4.3.1. guarantee that inputs to compound nodes are used within those nodes.

4.3.2.3. Common Sub-Expressions

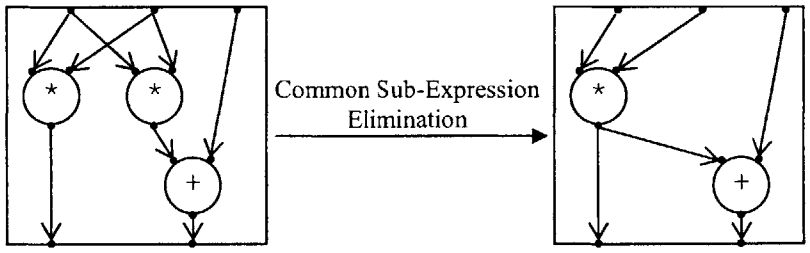

Figure 36: Common Sub-Expression Elimination

Common Sub-Expression Elimination is an optimization which reduces repeated computation. This is done by storing previously computed values in temporaries, rather than re-

computing them. We implement a local form of Common Sub-Expression Elimination, which eliminates redundant computation within a single Data Flow Graph. We do a linear pass over the nodes, using a hash table to identify identical nodes. Since checking compound nodes for equality is both expensive and complicated, due to redundancies in the representation, this transformation only eliminates simple nodes. Two nodes are considered equal if their input edges and labels are equal. To make the transformation more general, we ignore the order of the inputs for two-input nodes computing commutative operators. ***Figure 36*** includes an example of Common Sub-Expression elimination. Since methods in Functional Java do not have side effects and depend solely on their inputs, we can safely eliminate redundant method calls without performing cross-procedural analysis.

4.3.2.4. Inlining

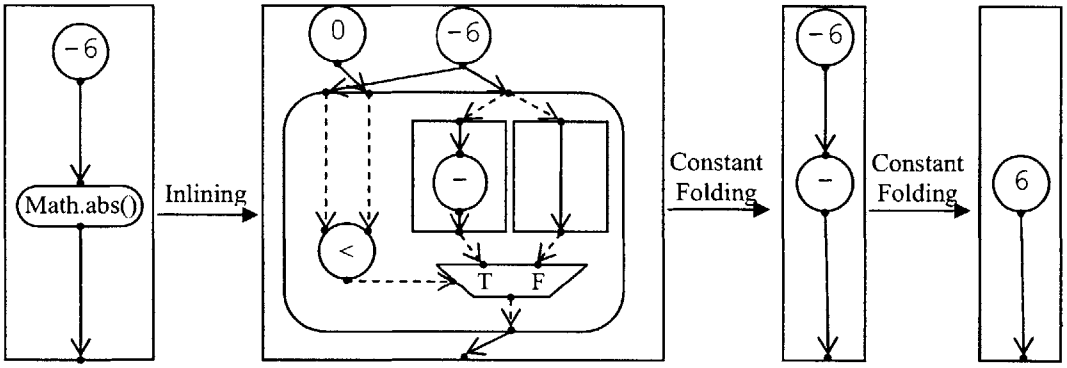

Figure 37: Inlining leading to Constant Folding

Inlining is another standard optimization, which replaces a method call with the body of the method. While increasing code size, and possibly adding register pressure, this optimization can greatly reduce the number of method calls. Additionally, inlining can greatly increase the effectiveness of other transformations, in particular constant propagation and folding, without requiring a cross-procedural analysis framework. We inline a method when it doesn't contain

any method calls. This simple heuristic is sufficient for our needs, and guarantees that inlining reaches a fixed point. ***Figure 37*** contains an example of Inlining leading to Constant Folding.

4.3.2.5. Loop Invariant Code

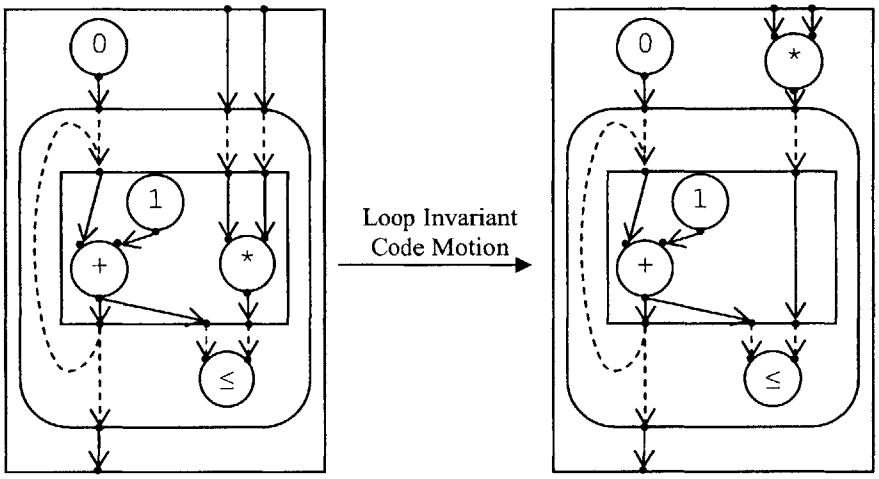

Figure 38: Loop Invariant Code Motion

Loop Invariant Code motion improves loop performance by moving repeated computation out of the loop body. This transformation is simple to implement for Data Flow Graphs, since control flow loops are explicitly represented as loop nodes. We identify any node in a loop's sub-graph which depends only on non-looping source ports as loop invariant, pulling the node out of the loop's sub-graph and passing in the result as an additional input to the loop node. We depend on the Loop Port Elimination to identify original non-looping ports. ***Figure 38*** has an example of such Loop Invariant Code Motion.

Another opportunity for moving computation out of loop bodies is identifying nodes in the sub-graph whose output does not flow into any loop ports. These nodes would be placed after the loop node in the surrounding graph. While this transformation could also improve the performance of some loops, it is not currently performed by the compiler.

4.3.3. Fresh Breeze-Specific Transformations

The transformations presented in this section are specifically designed to make it easier to generate code which takes advantage of the Fresh Breeze memory structure and model of execution. The Fresh Breeze-specific transformations perform analysis on the Data Flow Graph Representation and introduce new kinds of nodes which preserve the results of the analysis. While some new nodes will be seen and used by the code generator, others serve a transient purpose and are removed before the graph flattening phase.

4.3.3.1. Induction Nodes

To take advantage of the parallel threads available to a Fresh Breeze program, we must identify loops whose iterations can be executed in parallel. Before that analysis can proceed, it is useful to know how the values computed in a particular iteration depend on the iteration's number. So, the compiler must identify edges whose values are a simple function of the number of iterations. We restrict ourselves to linear functions upon integer variables. This is a reasonable restriction since array indices and loop termination conditions often take on this form.

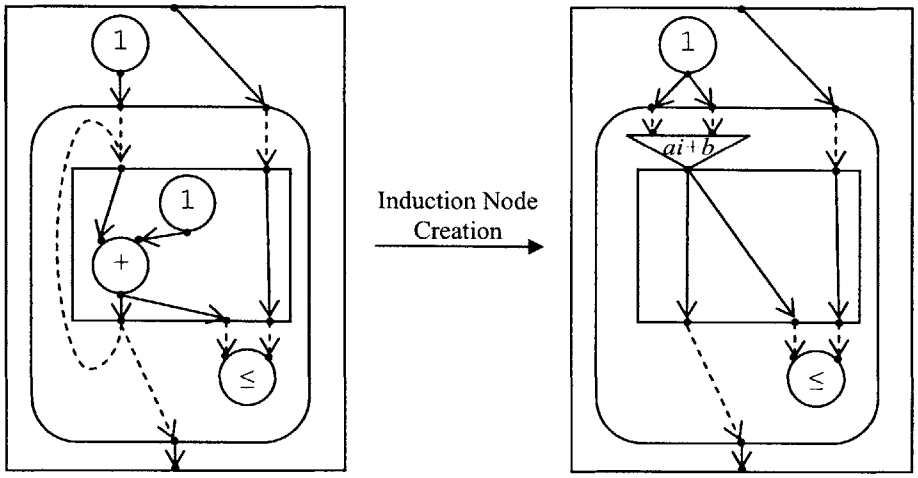

Figure 39: Using Induction Nodes

We use an Induction Node to represent a value which is a linear function of the number of iterations. An induction node has two loop invariant integer inputs, a and b, which can be either constants or non-looping source ports. During the i^{th} iteration of the loop, the node's

output is computed to be $ai + b$, where i is zero-indexed. ***Figure 39*** shows how an induction node, rather than a loop port, can be used to represent the count procedure from ***Figure 30***.

We eliminate a loop port by creating an Induction Node when the loop port's sink edge depends on a series of integer additions which take in loop invariant values and the loop port's source edge. While subtraction nodes can also create Induction Nodes, the transformations in Section 4.3.1.4. remove any subtraction nodes from the representation letting us focus solely on addition nodes. Loop Invariant Code Motion, described in Section 4.3.2.5., makes it easy to identify loop-invariant values.

Some edges that are a linear function of the number of iterations do not correspond directly to an existing loop port. These can be identified by looking at nodes that depend on already identified Induction Nodes. Negating an Induction Node produces another Induction Node: $-(ai+b) = (-a)i + (-b)$. Similarly, adding a constant to an Induction Node or multiplying an Induction Node by a constant also produces an Induction Node: $(ai+b) + c = ai+(b+c)$, $c(ai+b) = (ca)i+(cd)$. Finally, adding two Induction Nodes produces an Induction Node, although multiplying does not: $(ai+b) + (ci+d) = (a+c)i+(b+d)$. The Induction Node Creation transformation identifies and transforms these cases.

4.3.3.2. In-Order Traversals

Walking down the tree-representation of an array to retrieve a particular element can be expensive. Retrieving one element from an n-element array constructed from of k-element chunks takes $\log_k n$ ChunkGet instructions. It would be very expensive to retrieve all the elements of an array if each element took that many instructions. Fortunately, many array traversals are done in order, meaning the array elements are accessed in ascending or descending

order. Identifying these in-order retrievals would enable the compiler to generate code that performs an in-order traversal of the tree, taking $n/(1\text{-}1/k)$ time rather than $n \log_k n$.

```
public static double sum(double[] v) {
  double sum = 0;
  for (int i = 0; i < v.length; i++) {
    sum += v[i];
  }
  return sum;
}
```

Figure 40: Method with an in-order array traversal

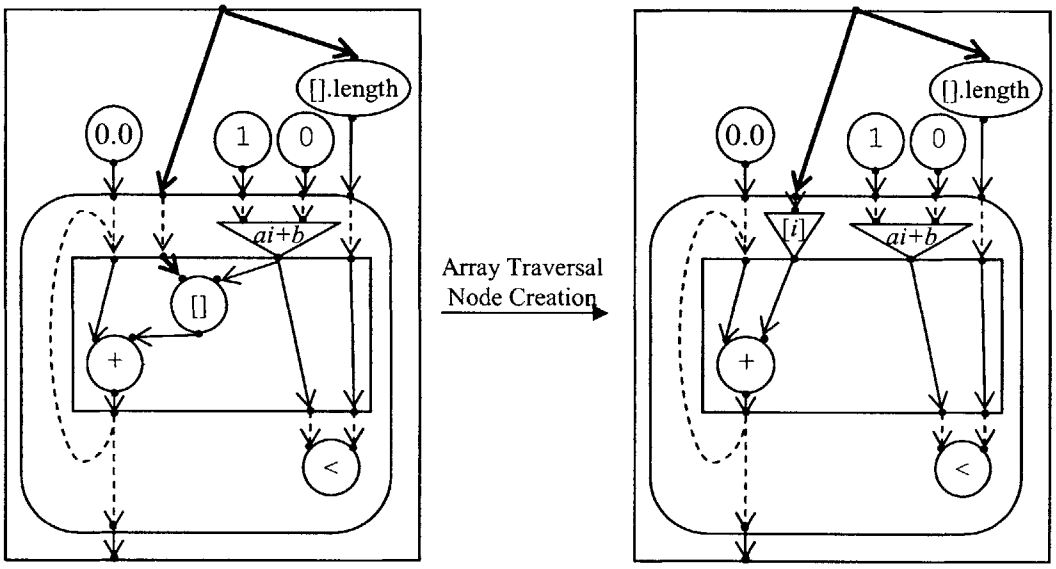

Figure 41: Array Traversal Node Creation

We introduce Array Traversal Nodes to represent in-order array retrievals performed by loops. Having Induction Nodes present makes it very easy to create Array Traversal Nodes. We replace an Array Get Node with an Array Traversal Node when the array is loop-invariant and the index is produced by an Induction Node with a constant factor. ***Figure 40*** contains a method, *sum()*, which traverses its input array in order inside a loop. In ***Figure 41*** we see how the Array Get Node in *sum()* can be converted to an Array Traversal Node.

While identifying array traversals is important, recognizing in-order array construction can lead to even more gains. To construct a new array which differs from an existing array in one element requires creating $\log_k n$ new chunks. When a new chunk is created, almost the entire

contents of the chunk it's replacing needs to be read from memory, and then all k elements of the new chunk need to be written back to the memory. So, changing one element takes $(k \log_k n - 1)$ element reads and $(k \log_k n)$ element writes. But, if the compiler can tell that the entire array is being constructed in order, it can generate code that can just create chunks for the final array, making at most $1/(1-1/k)$ writes per each array element.

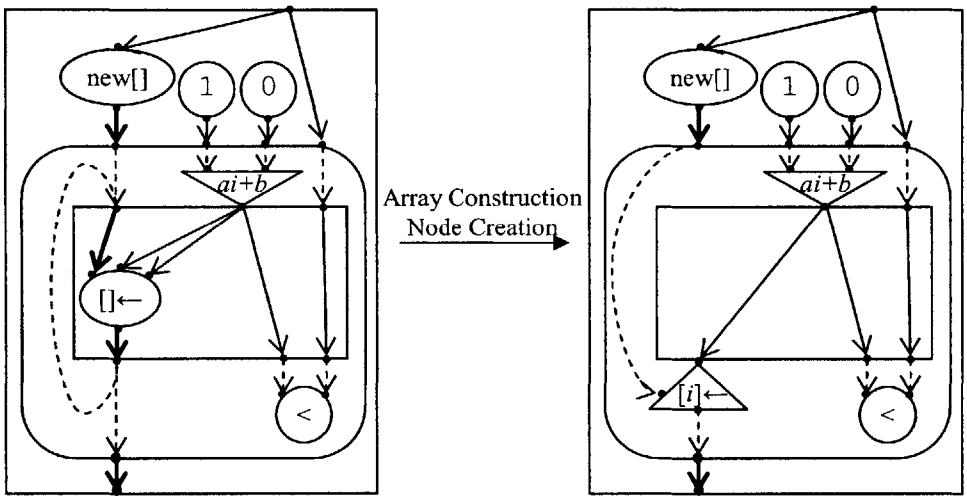

Figure 42: Array Construction Node Creation

In-order array constructions performed inside of loops are represented by Array Construction Nodes. The analysis for creating Array Construction Nodes is similar to the analysis for Array Traversal Nodes. The index should be produced by an Induction Node with a constant factor, but we can't require that the array is loop-invariant, since it's being constructed. What we do require is that the array edge is a loop-port, and that the Array Set Node is the only node that uses that loop-port. ***Figure 42*** shows how an Array Construction Node can be used to represent a simple procedure that creates an integer array with elements equal to their indices.

4.3.3.3. Array Aliasing

While the Array Traversal Node Creation and Array Construction Node Creation transformations work well in general, there is a fairly common case on which they fail to recognize in order traversals. This is the case where a Functional Java method is written as if it

mutates the elements of an array based on their current values. We interpret such a procedure as creating a new array based on an existing array, but the analysis described in Sections 4.3.3.2. and 4.3.3.1. are not sufficiently sophisticated to identify in-order traversals in this case. If we can identify that each iteration of a loop only reads array elements which have not yet been modified, we can separate the array into two, a loop-invariant array port which is only accessed by Array Get and Array Length nodes, and a loop-port whose value is changed by Array Set statements.

```
public static float[] negate(float[] a) {
  for (int i = 0; i < a.length; i++) {
    a[i] = -a[i];
  }
  return a;
}
```

Figure 43: Method with input and output array having the same name

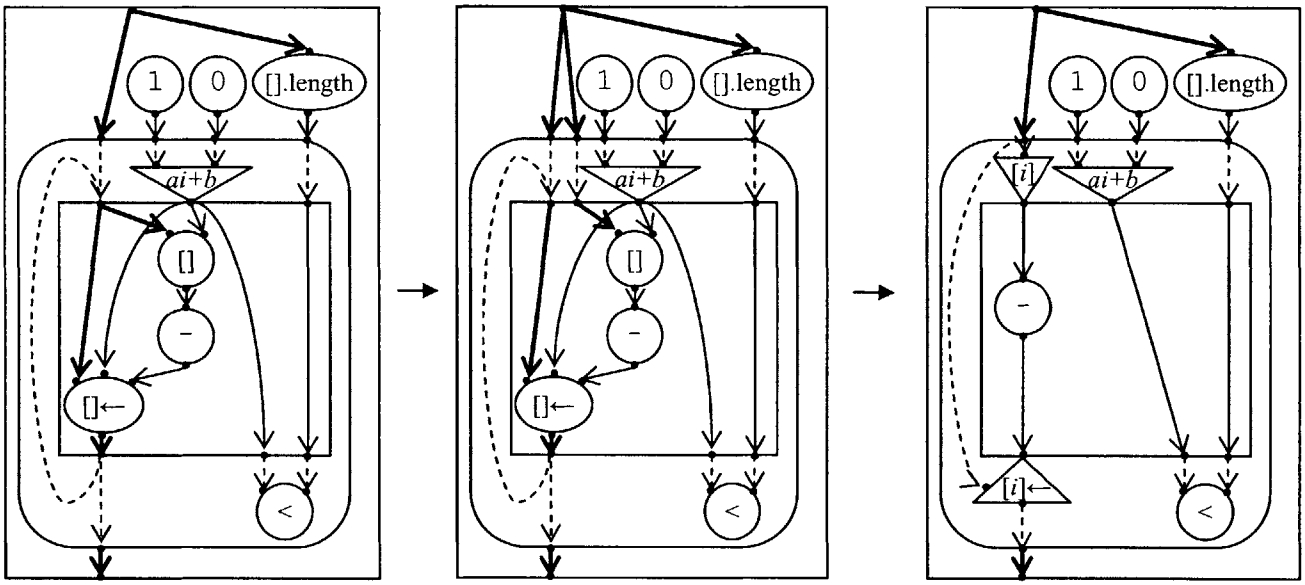

Figure 44: Array Get/Set Separation enables the creation of Array Construction Nodes and Array Traversal Nodes

The Array Get/Set Separation Transformation is conservative. It requires that there is one Array Set which uses the source loop-port, that its index is determined by an Induction Node, and that its output flows solely to the corresponding sink loop port. Additionally, it

requires that all Array Gets which depend on the loop port use the same index as the Array Set and that no other sink edges depend on the retrieved elements. These requirements guarantee correctness of the transformation and are satisfied for simple cases, like the *negate()* procedure in ***Figure 43***. ***Figure 44*** shows how performing this Array Get/Set Separation for *negate()*, allows Array Traversal Node Creation and Array Construction Node Creation to proceed.

4.3.3.4. Parallelization

While identifying in-order array traversals and constructions makes it easy to generate code which makes better use of the Fresh Breeze memory model, we also need the compiler to take advantage of Fresh Breeze's model of execution. In particular, we want to take advantage of Fresh Breeze's multi-threading ability by identifying opportunities for parallelization. One common means of parallelization is identifying loops whose iterations can be spread over multiple threads (see Section 2.4).

For loop to be parallelizable the values computed by a given iteration cannot depend on previous iterations. Additionally we must be able to predict the total number of iterations prior to executing the loop. These conditions are simple to express using our Data Flow Graph representation. The iterations of a Loop Node do not depend on previous iterations when it has no loop-ports. The number of iterations can be predicted when the inputs to the termination condition are either loop invariant or depend on Induction Nodes.

We use the For All Node to represent a parallelizable loop. The node has one special integer input which determines the total number of iterations. The node has one sub-graph, with a special source port which specifies the number of the current iteration. The remainder of the inputs are loop-invariant. The For All nodes outputs must either also be loop invariant, or can flow from an Array Construction Node.

The number of iterations of the For All loop often needs to be calculated dynamically. Without loss of generality we can assume that both inputs being compared come from Induction Nodes, ($ai+b$ and $ci+d$), and that the comparison operator is either $<$, $\leq$, or $\neq$. We can reduce the $<$ case to $\leq$ by taking $b' = b + 1$. In the $\leq$ case we calculate the number of iterations by solving for the rational i where the two lines intersect and taking the ceiling: $i = \left\lceil \frac{b-d}{a-c} \right\rceil$. We handle the edge cases appropriately and interpret a negative input to signify an infinite loop. We can determine where and if a $\neq$ comparison will be satisfied by calculated and comparing both the floor and the ceiling. The analysis is correct only if the original code did not depend on overflows for its correctness, which is a valid assumption for out test cases, but may not be sufficient for a robust compiler implementation. Finally, we rely on constant folding and propagation to simplify or eliminate runtime computation of the number of iterations, when some of the parameters are known at compile time.

4.3.3.5. Associative Accumulations

There are parallelizable computations which compute scalar values rather than arrays. In order to parallelize a loop which computes a scalar value we must have a way of combining values computed independently by different iterations of the loop into a final result. Any operator which performs such a combination must be commutative and associative, giving us the freedom to choose how the iterations of the loop are partitioned.

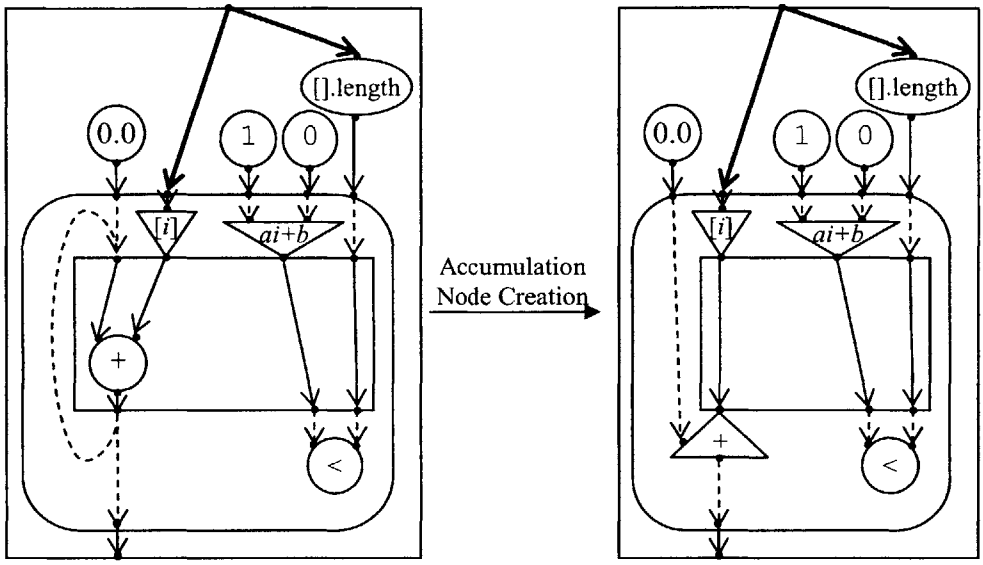

Figure 45: Accumulation Node Creation

We use an Accumulation Node to represent such associative accumulations, including addition, multiplication, bitwise AND, bitwise OR, and bitwise XOR. The absence of subtract nodes simplifies the analysis. For a loop port to be replaced with an Accumulation Node, for any control path, the source port's descendents must have out-degree of one, compute the same commutative operation, and flow into the corresponding sink port. The source port is then replaced with the operator's identity operand, and the sink port is replaced with an Accumulation Node. The procedure *sum()*, as represented in ***Figure 40***, contains a loop port used to sum up the input array's elements. In ***Figure 45*** we see how introducing an Accumulation Node eliminates the final loop-port from *sum()*'s Loop Node, allowing us to replace it with a For All Node.

4.3.4. Transformation Framework

The transformations above are applied in several phases. First, two-operand subtractions are eliminated in one simple pass, in order to simplify the representation, as described in Section 4.3.1.5. Then a set of transformations is applied in sequence until a fixed point is reached. These include Normalizing Transformations (Sections 4.3.1.1 – 4.3.1.4), Optimizing

Transformations (Section 4.3.2.), Induction Node Creation (Section 4.3.3.1.), Array Get/Set Separation (Section 4.3.3.3), Array Traversal Node Creation (Section 4.3.3.2), Array Construction Node Creation (Section 4.3.3.2), and Accumulation Node Creation (Section 4.3.3.5). When a fixed point is reached, we run For All Node Creation (Section 4.3.3.4) to identify the opportunities for parallelization that have been exposed. Since no more loop analysis needs to be performed, we convert Induction Nodes back to integer addition and multiplication nodes, and can reintroduce subtraction. Now, these last few transformations might have introduced more opportunities for standard optimizations, so we rerun the Normalizing and Optimizing Transformations in sequence until the final fixed point is reached. ***Figure 46*** below shows this structure.

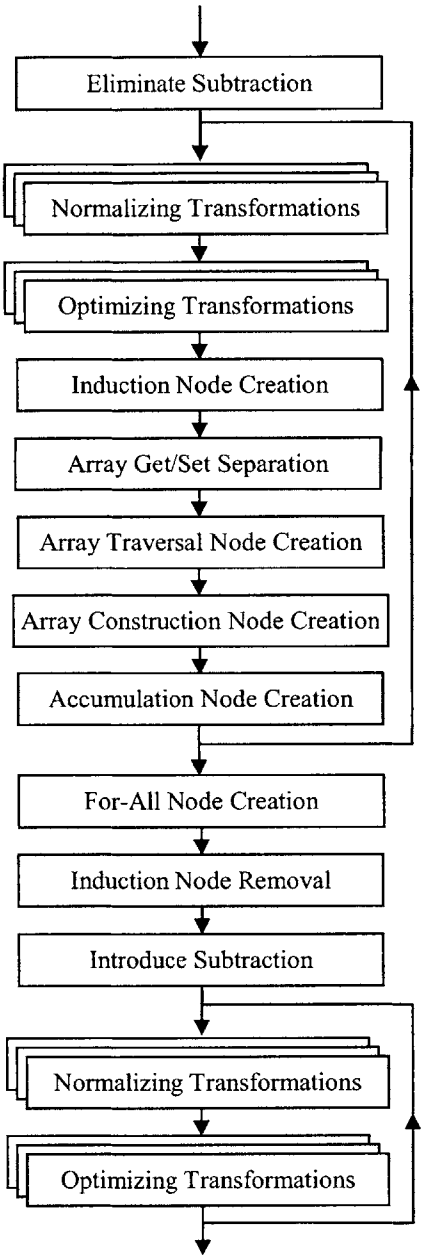

Figure 46: Transformation Framework

5. Analysis

We analyzed the compiler's design and implementation with respect to several array-intensive test cases. We chose several common linear algebra algorithms, implementing them in Functional Java. These algorithms included Matrix Multiplication, in-place Fast Fourier Transform, and Cholesky Decomposition. Implementing these algorithms in Functional Java was fairly straight forward, validating our goal that Functional Java should have simple programmability. The implementations are correct when interpreted as Java, and differ from straightforward Java implementations only in that they observe the Secure Arguments Principle by not mutating their parameters. The compiler's output for these test cases, along with the Functional Java programs themselves, are included and explained in Appendix A.

Although not complete, the Data Flow Graph Construction implementation was sufficient for constructing Data Flow Graphs for the three major test cases. While their effects on final performance are difficult to judge, the normalizing and optimizing transformations were effective at simplifying the representations, making the graphs easier to visualize and analyze. Specifically, Common Sub-Expression Elimination was very effective in reducing the number of nodes in the representation of in-place FFT. When compiling calls to these algorithms that specified the sizes of the input arrays, Inlining combined with Constant Folding and Propagation eliminated several unnecessary computation nodes and conditionals.

The algorithms presented many opportunities for in-order traversal and loop parallelization, validating our assumptions about the types of analysis the compiler should perform. The Fresh Breeze specific transformations were fairly effective at identifying both the in-order traversals and parallelizable loops. Matrix Multiplication was implemented as three

nested loops, and the compiler identified all three loops as being parallelizable, transforming them into For All Nodes. The compiler also identified that both levels of the output two-dimensional array representing the product were constructed in order. Finally, all but one of the Array Gets was transformed to Array Traversal Nodes. In order to eliminate the remaining Array Get, the compiler would need to represent one of the input matrices as an array of columns rather than an array of rows. This would require a high level of cross-procedural analysis.

6. Future Work

There is still some work to be done before the Fresh Breeze compiler can be used to generate Fresh Breeze code. The details of the array representation need to be settled. Once the final representation is specified, the last two phases of the compiler, Graph Flattening and Code Generation, need to be implemented. The Code Generation implementation may benefit from specialized register allocation to improve chunk-related performance. Once the final phases of the compiler are implemented, benchmarks can be performed on the simulator to verify and quantify the conclusions reached in the analysis chapter of this paper (Chapter 5).

6.1. Choosing Array Representation

While the general representation of Fresh Breeze arrays is well determined (See Section 2.3), there are some smaller details which can vary. One such detail is how the array size is stored. The size is necessary both for array bounds checking and for determining the depth of the underlying tree. A simple approach is to store the size as the last 32-bytes of the root chunk, limiting that chunk to 15 children. This is sufficient for both the array-bounds and depth requirements, but will have the affect that a child of a root of an array is no longer an array (since its last 64 bytes are a reference rather than a size.) An alternative approach would be to store both the lower and upper bound in the last 64 bytes of any chunk in the array. While this approach uses more space and makes offset computations more complicated, it maintains additional symmetry between different chunks in the array which may be useful in some circumstances. A third option would be to store array bounds separately from the array, and to pass them as an additional argument when traversing function calls.

Another question to explore is whether all levels of the tree must be full. When representing a sparse array, it may be advantageous to omit chunks with *null* or 0 elements. This can be accomplished by storing an additional bit per reference, or having a special *null* reference value. While this approach saves space and can fit empty arrays of any size on just one chunk, it may require additional operations at each level during an array get or set. An alternative approach would be to make a new array of depth k start of with k chunks, having each entry in the chunk i point to chunk $i+1$, except chunk k which will be initialized to zeros. This approach requires a little more space for a new array, but is still space efficient for sparse arrays, and does not incur the additional lookup costs.

6.2. Implementing Code Generation

Once the concrete array representation is chosen, the compiler can be extended to generate Fresh Breeze instructions. This can be implemented in two phases, first going from the Data Flow Graph Representation to a Flat Intermediate Representation with explicit chunk manipulation primitives, and then generating Fresh Breeze instructions from the Flat Intermediate Representation. This would maintain the generality of the compiler, keeping most of the work independent from the details of the Fresh Breeze instruction set. There is also work to be done in taking full advantage of the Fresh Breeze model of execution when compiling For All Nodes. For example, Fresh Breeze may provide primitives for performing associative accumulation, which could be used to improve performance.

6.3. Register Allocation

While register allocation is an important part of most modern compilers, there are several aspects of the Fresh Breeze architecture which suggest that designing a register allocator for the Fresh Breeze compiler may be an attractive avenue for future work. First, the ChunkGet and

ChunkSet instructions support bulk moves, which improve performance but require that the values are stored in adjacent registers. Depending on final architecture design choices, UIDs of unsealed chunks may be required to stay in registers, or might be allowed to be placed in a Function Activation Record. The existence of both 32bit and 64bit data types along with alignment constraints on 64bit values places additional constraints on the register allocation. Finally, the register file itself has interesting internal structure. Designing a register allocator which satisfies these different constraints may be an interesting project.

7. Conclusion

This work has presented a design and described the implementation of a compiler for the Fresh Breeze parallel architecture. Our main goal was to examine whether functional programs written within the constraints of a popular syntax, which assumes the standard linear, mutable memory model, can be efficiently compiled for Fresh Breeze.

Our design focused on compiling array-intensive linear-algebra algorithms. We designed a suitable input language, Functional Java, and an intermediate representation for representing Functional Java programs as data flow graphs. As part of the representation, we designed special nodes for representing parallelizable loops, in-order traversals and constructions of arrays, and associative accumulations of computed values. We implemented a collection of analyses and transformations which introduce these special nodes into the intermediate representation, making it possible to generate code which takes advantage of the parallelism available in Fresh Breeze.

We analyzed the compiler's output on several representative linear-algebra algorithms. Overall, the compiler implementation worked well at identifying the "low-hanging fruit", or simple opportunities for both parallelization and in-order traversal. We hope future projects will build on this work, first to complete the last phases of the compiler, but also to extend the compiler to handle other data structures and exploit additional types of parallelism.

Appendix A: Test Cases

A.1 Matrix Multiplication

In this Appendix we present the matrix multiplication test case. We provide the source Functional Java program, a textual representation of the transformed Data Flow Graphs generated by the compiler, and a graphical view of the graphs.

```
package linearalgebra;

public class MatrixMultiply {
      public static float[][] multiply(float[][] a, float[][] b) {
            int m_a = a.length;
            int n_a = a[0].length;

            int m_b = b.length;
            int n_b = b[0].length;

            int m = m_a;
            int k = m_b;
            int n = n_b;

            float[][] c = new float[m][n];

            for (int x = 0; x < m ; x++) {
                  for (int y = 0; y < n; y++) {
                        float sum = 0;
                        for (int i = 0; i < k; i++) {
                              sum += a[x][i] * b[i][y];
                        }
                        c[x][y] = sum;
                  }
            }
            return c;
      }
}
```

Figure 47: Functional Java Source Code for Matrix Multiplication

The Functional Java implementation of Matrix Multiplication, found in ***Figure 47***, is a method which takes in two rectangular arrays of floating point values, *a* and *b*, representing the two input matrices, and returns a new rectangular array *c* representing the product matrix . The implementation is straight forward, containing three nested loops.

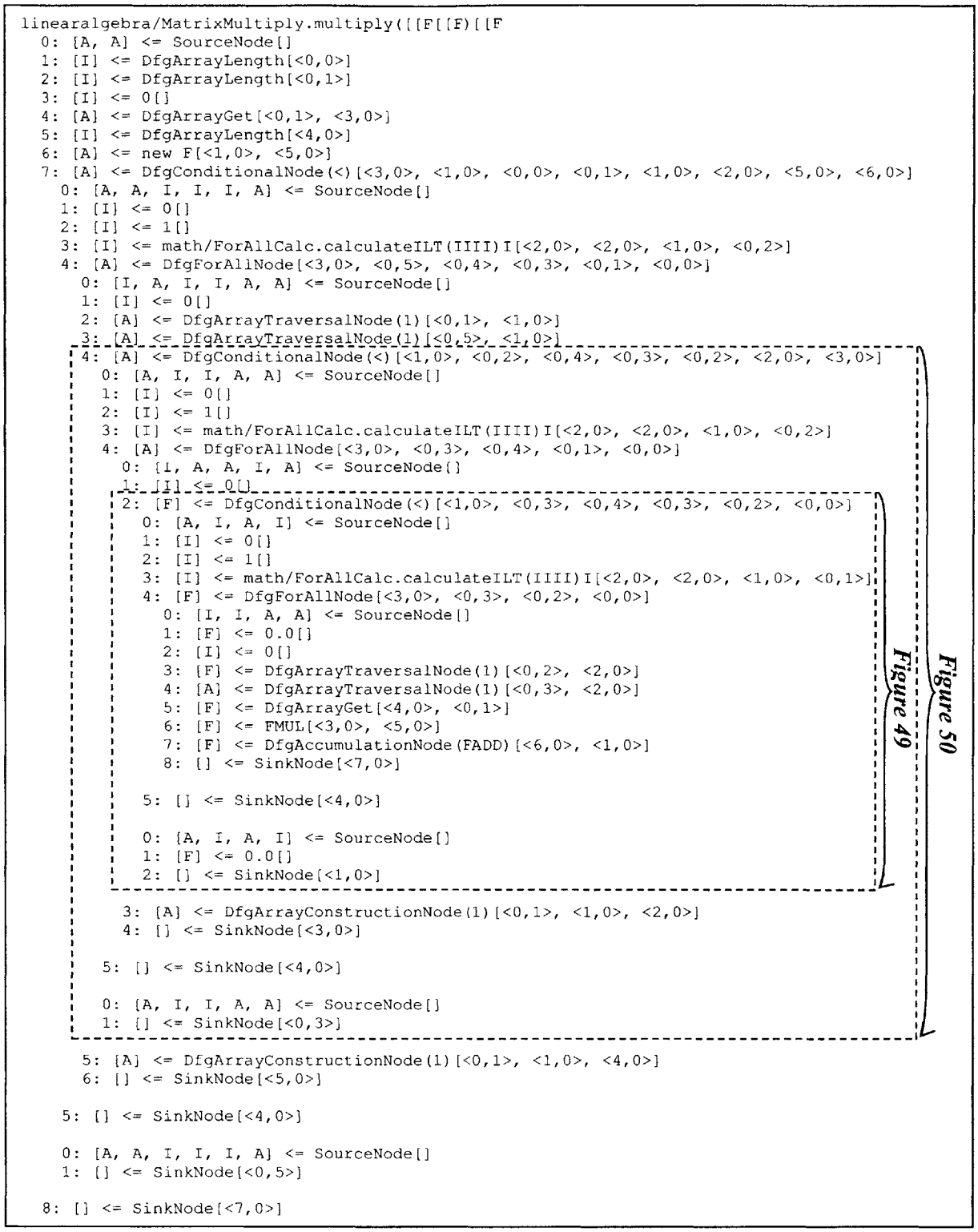

Figure 48: Compiler Output for Matrix Multiplication

Figure 48 contains a textual representation of the final Data Flow Graphs generated by the Analysis and Transformation phase of the compiler, as outputted by the compiler. A line in the printout corresponds to a node in the graph. A node is specified by its index in the graph's node list, followed by a list of output types in brackets, then the node's name, then any additional labeling in parenthesis, and finally the list of input edges in brackets. An input edge is shown by a node index and output port index pair. The graphs within a compound node follow the node's line, and are indented. A blank line follows the last node in a graph.

From a cursory examination of ***Figure 48***, we see three nested For All Nodes and no Loop Nodes, meaning that the compiler parallelized all three loops from ***Figure 47***. Each For All Node is wrapped in a Conditional Node which conditions the first iteration. This is an artifact of the Data Flow Graph Construction phase (Section 4.2), where a *for(){}* loop is seen as a *do{}while()* loop within an *if(){}else{}* statement, represented as a Loop Node within a Conditional Node. A redesigned Loop Node along with a different implementation of the Graph Construction phase, or additional analysis during the Transformation Phase, could eliminate these Conditional Nodes.

Below we give a graphical view of the Data Flow Graph shown textually in ***Figure 48***. We separate the representation into three hierarchical figures, one for each of the For All Nodes. The sections of the printout covered by the graphical figures are depicted by dotted rectangles in ***Figure 48***.

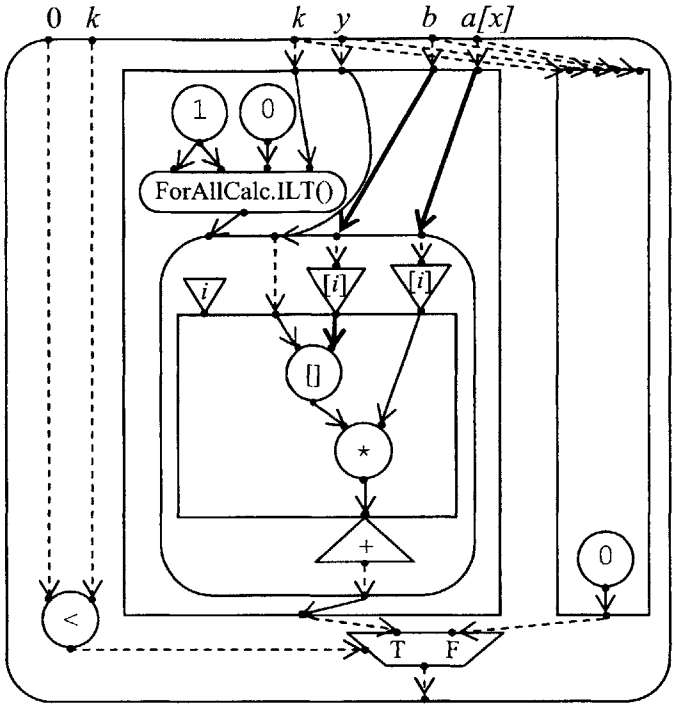

Figure 49: Innermost For All Node

Figure 49 depicts the innermost For All Node and its surrounding Conditional Node. The For All node has four inputs. The first, an integer, determines the number of iterations, and corresponds to variable *k* of the Functional Java code in ***Figure 47***. The second input, also an integer, is passed into the sub-graph and corresponds to variable *y*. The next input is the two dimensional array *b*, and the final input is the one-dimensional array *a[x]*. Loop Invariant Code Motion is responsible for making the last input *a[x]* rather than *a*. Array traversal nodes feed the array's elements, corresponding to *b[i]* and *a[x][i]*, into the sub-graph. An array get node and a multiplication node are used to compute *a[x][i] * b[i][y]*. This value is then summed across all iterations using an Accumulation Node. The special source port, corresponding to the current loop index *i*, is ignored since the Array Traversal Nodes depend on *i* implicitly.

The surrounding Conditional Node compares its two special inputs, which will correspond to the constant 0 and variable *k*, using the less than operator. Its other inputs are the

same as the For All Node's. If 0 < k, it passes its inputs to the For All Node and returns its output. Otherwise it outputs 0. Rather than passing *k* directly to the For All Node it issues a library call to determine the number of iterations. In the general case, this is just *k*, but for $k < 1$, the library call would return 1, as was appropriate for the Loop Node which was replaced by the For All Node. When the library's source code is included in the compilation (which was not the case for ***Figure 48***) Inlining and Constant Folding will replace the method call with a simple Conditional Node which calculates *max(1,k)*.

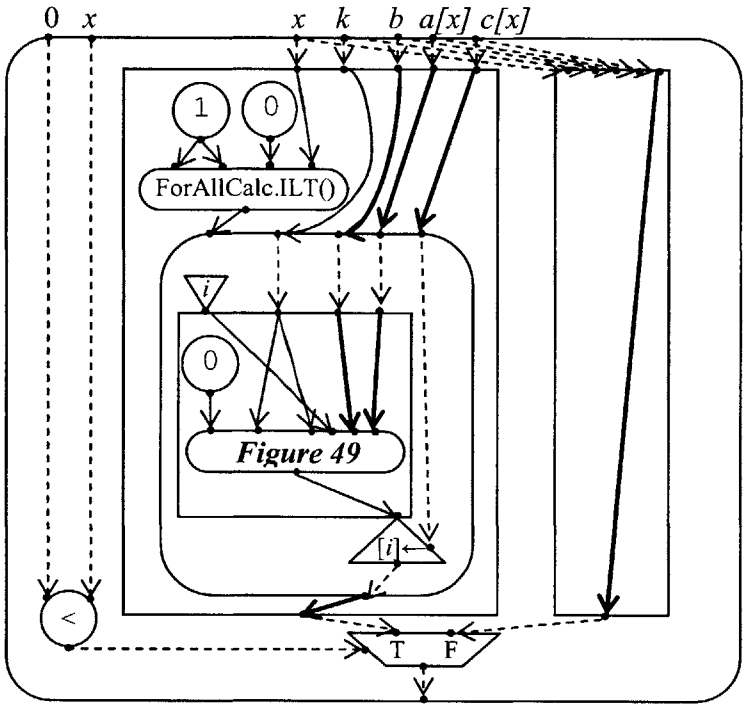

Figure 50: Middle For All Node

Figure 50 shows the middle For All Node. This node takes in 5 inputs. The number of iterations is determined by the first input, an integer which corresponds to variable *x* in the source code. The second input is an integer corresponding to *k*. The final three inputs are arrays, corresponding to *a[x]*, *b*, and *c[x]*. The special source port providing the current iteration number corresponds to the variable *y*. The sub-graph contains the Conditional Node shown in ***Figure 49***. The edges corresponding to the constant 0, variables *k*, *y*, *a[x]*, and *b* are passed into

this Conditional Node. The output flows to an Array Construction Node which fills the array corresponding to *c[x]*. The surrounding Conditional Node serves the same purpose as the one in ***Figure 49***, and bypasses the For All Node outputting *c[x]* directly for $n \leq 0$.

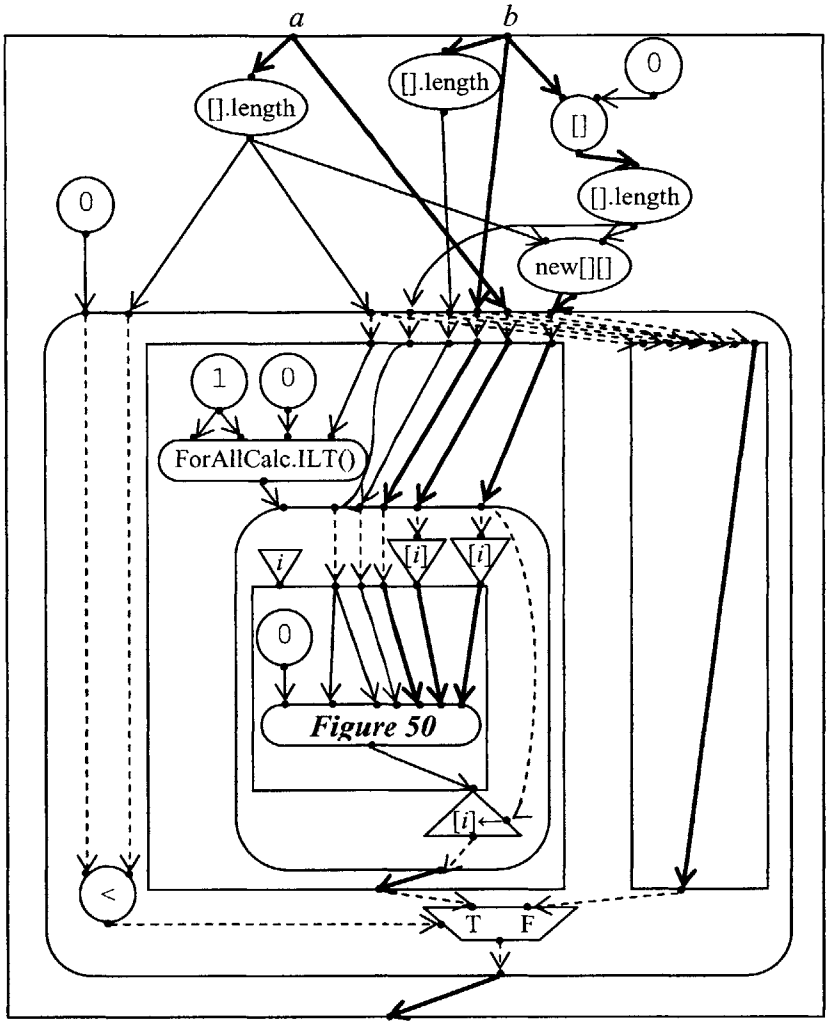

Figure 51: Data Flow Graph for Matrix Multiply Method

Figure 51 contains the outer For All Node, showing the main graph of the *multiply()* method. The method takes in the two arrays, *a* and *b*, calculates their dimensions, constructs the two-dimensional array *c* using those lengths, and passes them to a Conditional Node which wraps the For All Node. Unlike in the source code, *n_a* is never calculated, since the Array Length node is removed by Dead Code Elimination. The For All Node's number of iterations is

determined by an edge corresponding to *m*. Its other five input edges correspond to *n*, *k*, *b*, *a*, and *c*. The last two array edges, *a* and *c*, are passed through Array Traversal Nodes, which pass their x^{th} element into each iteration of the sub-graph. The sub-graph passes its inputs directly to the Conditional Node shown in Figure Y. The Conditional Node returns a one dimensional array, corresponding to *c[x]*. That output is fed into an Array Construction Node which generates the two dimensional array *c*, which is then returned by the For All Node, then by the Conditional Node, and finally by the method's graph.

Overall, the compiler was very successful on the Matrix Multiplication test case. First, the generated Data Flow Graphs preserved the correctness of the code. Second, the compiler extracted a large amount of parallelism from the code, identifying all the loops in program as parallelizable. The efficiency of the code was also improved by moving loop invariant array gets into the outer loop, and then transforming array gets to array transversals whenever possible.

A.2 Cholesky Decomposition

Cholesky Decomposition is a method for decomposing a symmetric positive-definite matrix A, into the product of a lower triangular matrix L, and its transpose L^T, so: $A = L L^T$. In this section, we provide the source Functional Java program, and a textual representation of the transformed Data Flow Graphs generated by the compiler for the Cholesky Decomposition test case.

```
package linearalgebra;

import math.Math;

public class CholeskyDecomposition {
   public static double[][] decompose (double[][] A) {
      int n = A.length;
      double[][] L = new double[n][];

      for (int j = 0; j < n; j++) {
         double d = 0.0;

         L[j] = new double[j + 1];

         for (int k = 0; k < j; k++) {
            double s = 0.0;
            for (int i = 0; i < k; i++) {
               s += L[k][i] * L[j][i];
            }
            L[j][k] = s = (A[j][k] - s) / L[k][k];
            d = d + s*s;
         }

         d = A[j][j] - d;

         L[j][j] = Math.sqrt(Math.max(d, 0.0));
      }

      return L;
   }
}
```

Figure 52: Functional Java Source Code for Cholesky Decomposition

In ***Figure 52*** is the Functional Java implementation of the Cholesky Decomposition algorithm. The method *decompose()* takes in a two-dimensional square array of doubles *A*, and outputs a two-dimensional triangular array L. The method is implemented with three nested *for(){}* loops, with a division for each element in L, and a square root for each pivot (diagonal element) in L.

```
linearalgebra/CholeskyDecomposition.decompose([[D)[[D
  0: [A] <= SourceNode[]
  1: [I] <= DfgArrayLength[<0,0>]
  2: [A] <= new A[<1,0>]
  3: [I] <= 0[]
  4: [A] <= DfgConditionalNode(<)[<3,0>, <1,0>, <0,0>, <1,0>, <2,0>]
    0: [A, I, A] <= SourceNode[]
    1: [I] <= 0[]
    2: [I, A] <= DfgLoopNode(<,2)[<1,0>, <0,2>, <0,1>, <0,0>]
      0: [I, A, I, A] <= SourceNode[]
      1: [I] <= 0[]
      2: [I] <= 1[]
```

```
    3: [I] <= +[<0,0>, <2,0>]
    4: [A] <= new D[<3,0>]
    5: [A] <= DfgArraySet[<0,1>, <4,0>, <0,0>]
    6: [A, D] <= DfgConditionalNode(<)[<1,0>, <0,0>, <0,3>, <5,0>, <0,0>]
      0: [A, A, I] <= SourceNode[]
      1: [A] <= DfgArrayGet[<0,0>, <0,2>]
      2: [I] <= 0[]
      3: [I, A, D] <= DfgLoopNode(<,2)[<2,0>, <0,1>, <0,2>, <1,0>]
        0: [I, A, I, A] <= SourceNode[]
        1: [D] <= 0.0[]
        2: [I] <= 0[]
        3: [I] <= 1[]
        4: [D] <= DfgConditionalNode(<)[<2,0>, <0,0>, <0,1>, <0,2>, <0,0>]
          0: [A, I, I] <= SourceNode[]
          1: [A] <= DfgArrayGet[<0,0>, <0,2>]
          2: [A] <= DfgArrayGet[<0,0>, <0,1>]
          3: [I] <= 1[]
          4: [I] <= DfgConditionalNode(>=)[<0,2>, <3,0>, <0,2>]
            0: [I] <= SourceNode[]
            1: [] <= SinkNode[<0,0>]

            0: [I] <= SourceNode[]
            1: [I] <= 1[]
            2: [] <= SinkNode[<1,0>]

          5: [D] <= DfgForAllNode[<4,0>, <1,0>, <2,0>]
            0: [I, A, A] <= SourceNode[]
            1: [D] <= 0.0[]
            2: [I] <= 0[]
            3: [D] <= DfgArrayTraversalNode(1)[<0,1>, <2,0>]
            4: [D] <= DfgArrayTraversalNode(1)[<0,2>, <2,0>]
            5: [D] <= DMUL[<3,0>, <4,0>]
            6: [D] <= DfgAccumulationNode(DADD)[<5,0>, <1,0>]
            7: [] <= SinkNode[<6,0>]

          6: [] <= SinkNode[<5,0>]

          0: [A, I, I] <= SourceNode[]
          1: [D] <= 0.0[]
          2: [] <= SinkNode[<1,0>]

        5: [D] <= DfgArrayTraversalNode(1)[<0,3>, <2,0>]
        6: [D] <= DSUB[<5,0>, <4,0>]
        7: [A] <= DfgArrayGet[<0,1>, <0,0>]
        8: [D] <= DfgArrayGet[<7,0>, <0,0>]
        9: [D] <= DDIV[<6,0>, <8,0>]
        10: [A] <= DfgArrayGet[<0,1>, <0,2>]
        11: [A] <= DfgArraySet[<10,0>, <9,0>, <0,0>]
        12: [A] <= DfgArraySet[<0,1>, <11,0>, <0,2>]
        13: [D] <= DMUL[<9,0>, <9,0>]
        14: [I] <= +[<0,0>, <3,0>]
        15: [D] <= DfgAccumulationNode(DADD)[<13,0>, <1,0>]
        16: [] <= SinkNode[<14,0>, <0,2>, <14,0>, <12,0>, <15,0>]

      4: [] <= SinkNode[<3,1>, <3,2>]

      0: [A, A, I] <= SourceNode[]
      1: [D] <= 0.0[]
      2: [] <= SinkNode[<0,1>, <1,0>]

    7: [A] <= DfgArrayTraversalNode(1)[<0,3>, <1,0>]
    8: [D] <= DfgArrayGet[<7,0>, <0,0>]
    9: [D] <= DSUB[<8,0>, <6,1>]
    10: [D] <= DfgConditionalNode(DEQ)[<9,0>, <9,0>, <9,0>]
      0: [D] <= SourceNode[]
      1: [I] <= 0[]
      2: [D] <= 0.0[]
      3: [I] <= DfgConditionalNode(DNE)[<0,0>, <2,0>]
        0: [] <= SourceNode[]
        1: [I] <= 0[]
        2: [] <= SinkNode[<1,0>]
```

```
        0: [] <= SourceNode[]
        1: [I] <= 1[]
        2: [] <= SinkNode[<1,0>]

      4: [D] <= DfgConditionalNode(==)[<3,0>, <1,0>, <0,0>]
        0: [D] <= SourceNode[]
        1: [D] <= 0.0[]
        2: [D] <= DfgConditionalNode(DLE)[<0,0>, <1,0>, <0,0>]
          0: [D] <= SourceNode[]
          1: [D] <= 0.0[]
          2: [] <= SinkNode[<1,0>]

          0: [D] <= SourceNode[]
          1: [] <= SinkNode[<0,0>]

        3: [] <= SinkNode[<2,0>]

        0: [D] <= SourceNode[]
        1: [] <= SinkNode[<0,0>]

      5: [] <= SinkNode[<4,0>]

      0: [D] <= SourceNode[]
      1: [] <= SinkNode[<0,0>]

    11: [D] <= math/StrictMath.sqrt(D)D[<10,0>]
    12: [A] <= DfgArrayGet[<6,0>, <0,0>]
    13: [A] <= DfgArraySet[<12,0>, <11,0>, <0,0>]
    14: [A] <= DfgArraySet[<6,0>, <13,0>, <0,0>]
    15: [] <= SinkNode[<3,0>, <0,2>, <3,0>, <14,0>]

  3: [] <= SinkNode[<2,1>]

  0: [A, I, A] <= SourceNode[]
  1: [] <= SinkNode[<0,2>]

5: [] <= SinkNode[<4,0>]
```

Figure 53: Compiler Output for Cholesky Decomposition

Figure 53 contains the compiler's output for Cholesky Decomposition. The compiler was only able to parallelize the innermost *for(){}* loop, creating one For All Node, and leaving the other two as iterative Loop Nodes. This is reasonable, since in the middle loop the assignments to *L[j][k]* depend on the values of *L[j][i]* for $i < k$, so each iteration of the loop depends on the values produced by previous iterations, and is therefore not easy to parallelize. Similarly, for the outer loop, the array assigned to *L[j]* depends on the values of the arrays at *L[k]* for $k < j$, so the outer loop is also a recurrence and cannot be easily parallelized.

We notice that Loop Invariant Code Motion successfully pulled the array gets corresponding to *L[k]* and *L[j]* out of the innermost loop. We included the sources for *math.Math* and *math.ForAllCalc* in the compilation, so Inlining eliminated the calls to

Math.min() and *ForAllCalc.calculateILT()*. The source for *StrictMath.sqrt()* was not included in the compilation for sake of brevity of the output.

A.3 In-Place FFT

In this section we describe the test case which computes the Discrete Fourier Transform. We use a simple in-place implementation of the Cooley-Tukey Fast Fourier Transform (FFT) algorithm. ***Figure 54*** below contains the Functional Java source code for the test case. The implementation starts with a bit-reversal permutation of the input, and then performs a series of butterfly updates, nested in three *for(){}* loops. The input is two one-dimensional arrays of doubles, *x_r* and *x_i*, containing the real and imaginary components of the input data respectively. In order to output both arrays to the caller, we return a two-element array of arrays of double, making the return type a two-dimensional array of doubles. This complication is necessary because the compiler does not support Object types.

```
package linearalgebra;
import math.Integer;
import math.Math;

public class InplaceFFT {
    public static double[][] fft(double[] x_r, double[] x_i) {
        int N = x_r.length;
        if (N != x_i.length)
            return null;
        if (Integer.highestOneBit(N) != N)
            return null;

        // bit reversal permutation
        int shift = 1 + Integer.numberOfLeadingZeros(N);
        for (int k = 0; k < N; k++) {
            int j = Integer.reverse(k) >>> shift;
            if (j > k) {
                  double temp_r = x_r[j];
                  double temp_i = x_i[j];

                  x_r[j] = x_r[k];
                  x_i[j] = x_i[k];

                  x_r[k] = temp_r;
                  x_i[k] = temp_i;
            }
        }

        // butterfly updates
        for (int L = 2; L <= N; L = L+L) {
            for (int k = 0; k < L/2; k++) {
                double kth = -2 * k * Math.PI / L;
                double w_r = Math.cos(kth);
                double w_i = Math.sin(kth);
                for (int j = 0; j < N/L; j++) {
                    double a_r = w_r;
                    double a_i = w_i;

                    double b_r = x_r[j*L + k + L/2];
                    double b_i = x_i[j*L + k + L/2];

                    double tao_r = a_r * b_r - a_i * b_i;
                    double tao_i = a_r * b_i + a_i * b_r;

                    x_r[j*L + k + L/2] = x_r[j*L + k] - tao_r;
                    x_i[j*L + k + L/2] = x_i[j*L + k] - tao_i;

                    x_r[j*L + k]       = x_r[j*L + k] + tao_r;
                    x_i[j*L + k]       = x_i[j*L + k] + tao_i;
                }
            }
        }
        return new double[][]{x_r, x_i};
    }
```

Figure 54: Functional Java Source Code for In-Place FFT

```
linearalgebra/InplaceFFT.fft([D[D)[[D
  0: [A, A] <= SourceNode[]
  1: [I] <= DfgArrayLength[<0,0>]
  2: [I] <= DfgArrayLength[<0,1>]
  3: [A] <= DfgConditionalNode(==)[<1,0>, <2,0>, <0,0>, <0,1>, <1,0>]
    0: [A, A, I] <= SourceNode[]
    1: [I] <= math/Integer.highestOneBit(I)I[<0,2>]
    2: [A] <= DfgConditionalNode(==)[<1,0>, <0,2>, <0,0>, <0,1>, <0,2>]
      0: [A, A, I] <= SourceNode[]
      1: [I] <= math/Integer.numberOfLeadingZeros(I)I[<0,2>]
      2: [I] <= 1[]
      3: [I] <= +[<2,0>, <1,0>]
      4: [I] <= 0[]
      5: [A, A] <= DfgConditionalNode(<)[<4,0>, <0,2>, <0,0>, <0,1>, <0,2>, <3,0>]
        0: [A, A, I, I] <= SourceNode[]
        1: [I] <= 0[]
        2: [I, A, A] <= DfgLoopNode(<,3)[<1,0>, <0,0>, <0,1>, <0,2>, <0,3>]
          0: [I, A, A, I, I] <= SourceNode[]
          1: [I] <= 1[]
          2: [I] <= math/Integer.reverse(I)I[<0,0>]
          3: [I] <= >>>[<2,0>, <0,4>]
          4: [A, A] <= DfgConditionalNode(<=)[<3,0>, <0,0>, <0,1>, <0,2>, <0,0>, <3,0>]
            0: [A, A, I, I] <= SourceNode[]
            1: [] <= SinkNode[<0,0>, <0,1>]

            0: [A, A, I, I] <= SourceNode[]
            1: [D] <= DfgArrayGet[<0,0>, <0,3>]
            2: [D] <= DfgArrayGet[<0,1>, <0,3>]
            3: [D] <= DfgArrayGet[<0,0>, <0,2>]
            4: [A] <= DfgArraySet[<0,0>, <3,0>, <0,3>]
            5: [D] <= DfgArrayGet[<0,1>, <0,2>]
            6: [A] <= DfgArraySet[<0,1>, <5,0>, <0,3>]
            7: [A] <= DfgArraySet[<4,0>, <1,0>, <0,2>]
            8: [A] <= DfgArraySet[<6,0>, <2,0>, <0,2>]
            9: [] <= SinkNode[<7,0>, <8,0>]

          5: [I] <= +[<0,0>, <1,0>]
          6: [] <= SinkNode[<5,0>, <0,3>, <5,0>, <4,0>, <4,1>]

        3: [] <= SinkNode[<2,1>, <2,2>]

        0: [A, A, I, I] <= SourceNode[]
        1: [] <= SinkNode[<0,0>, <0,1>]

      6: [I] <= 2[]
      7: [A, A] <= DfgConditionalNode(<=)[<6,0>, <0,2>, <5,0>, <5,1>, <0,2>]
        0: [A, A, I] <= SourceNode[]
        1: [I] <= 2[]
        2: [A, A, I] <= DfgLoopNode(<=,3)[<0,0>, <0,1>, <1,0>, <0,2>]
          0: [A, A, I, I] <= SourceNode[]
          1: [I] <= 0[]
          2: [I] <= 2[]
          3: [I] <= /[<0,2>, <2,0>]
          4: [A, A] <= DfgConditionalNode(<)[<1,0>, <3,0>, <0,0>, <0,1>, <0,3>, <0,2>]
            0: [A, A, I, I] <= SourceNode[]
            1: [D] <= ITOD[<0,3>]
            2: [I] <= /[<0,2>, <0,3>]
            3: [I] <= 2[]
            4: [I] <= /[<0,3>, <3,0>]
            5: [I] <= 0[]
            6: [I, A, A] <= DfgLoopNode(<,3)[<5,0>, <0,0>, <0,1>, <0,3>, <0,2>, <1,0>, <2,0>,
                                            <4,0>]
              0: [I, A, A, I, I, D, I, I] <= SourceNode[]
              1: [I] <= 0[]
              2: [I] <= 1[]
              3: [I] <= -2[]
              4: [I] <= *[<0,0>, <3,0>]
              5: [D] <= ITOD[<4,0>]
              6: [D] <= 3.141592653589793[]
              7: [D] <= DMUL[<5,0>, <6,0>]
              8: [D] <= DDIV[<7,0>, <0,5>]
```

```
            9: [D] <= math/StrictMath.cos(D)D[<8,0>]
            10: [D] <= math/StrictMath.sin(D)D[<8,0>]
            11: [A, A] <= DfgConditionalNode(<)[<1,0>, <0,6>, <0,1>, <0,2>, <0,4>, <0,3>,
                                                <0,0>, <9,0>, <10,0>]
              0: [A, A, I, I, I, D, D] <= SourceNode[]
              1: [I] <= 2[]
              2: [I] <= /[<0,3>, <1,0>]
              3: [I] <= /[<0,2>, <0,3>]
              4: [I] <= +[<0,4>, <2,0>]
              5: [I] <= 0[]
              6: [I, A, A] <= DfgLoopNode(<,3)[<5,0>, <0,0>, <0,1>, <0,3>, <0,6>, <0,5>, <0,4>,
                                               <3,0>, <4,0>]
                0: [I, A, A, I, D, D, I, I, I] <= SourceNode[]
                1: [I] <= 1[]
                2: [I] <= *[<0,0>, <0,3>]
                3: [I] <= +[<2,0>, <0,6>]
                4: [I] <= +[<2,0>, <0,8>]
                5: [D] <= DfgArrayGet[<0,1>, <4,0>]
                6: [D] <= DfgArrayGet[<0,2>, <4,0>]
                7: [D] <= DMUL[<0,5>, <5,0>]
                8: [D] <= DMUL[<0,4>, <6,0>]
                9: [D] <= DSUB[<7,0>, <8,0>]
                10: [D] <= DMUL[<0,5>, <6,0>]
                11: [D] <= DMUL[<0,4>, <5,0>]
                12: [D] <= DADD[<10,0>, <11,0>]
                13: [D] <= DfgArrayGet[<0,1>, <3,0>]
                14: [D] <= DSUB[<13,0>, <9,0>]
                15: [A] <= DfgArraySet[<0,1>, <14,0>, <4,0>]
                16: [D] <= DfgArrayGet[<0,2>, <3,0>]
                17: [D] <= DSUB[<16,0>, <12,0>]
                18: [A] <= DfgArraySet[<0,2>, <17,0>, <4,0>]
                19: [D] <= DfgArrayGet[<15,0>, <3,0>]
                20: [D] <= DADD[<19,0>, <9,0>]
                21: [A] <= DfgArraySet[<15,0>, <20,0>, <3,0>]
                22: [D] <= DfgArrayGet[<18,0>, <3,0>]
                23: [D] <= DADD[<22,0>, <12,0>]
                24: [A] <= DfgArraySet[<18,0>, <23,0>, <3,0>]
                25: [I] <= +[<0,0>, <1,0>]
                26: [] <= SinkNode[<25,0>, <0,7>, <25,0>, <21,0>, <24,0>]

              7: [] <= SinkNode[<6,1>, <6,2>]

              0: [A, A, I, I, I, D, D] <= SourceNode[]
              1: [] <= SinkNode[<0,0>, <0,1>]

            12: [I] <= +[<0,0>, <2,0>]
            13: [] <= SinkNode[<12,0>, <0,7>, <12,0>, <11,0>, <11,1>]

          7: [] <= SinkNode[<6,1>, <6,2>]

          0: [A, A, I, I] <= SourceNode[]
          1: [] <= SinkNode[<0,0>, <0,1>]

        5: [I] <= *[<0,2>, <2,0>]
        6: [] <= SinkNode[<5,0>, <0,3>, <4,0>, <4,1>, <5,0>]

      3: [] <= SinkNode[<2,0>, <2,1>]

      0: [A, A, I] <= SourceNode[]
      1: [] <= SinkNode[<0,0>, <0,1>]

    8: [A] <= new A[<6,0>]
    9: [A] <= DfgArraySet[<8,0>, <7,0>, <4,0>]
    10: [A] <= DfgArraySet[<9,0>, <7,1>, <2,0>]
    11: [] <= SinkNode[<10,0>]

    0: [A, A, I] <= SourceNode[]
    1: [A] <= null[]
    2: [] <= SinkNode[<1,0>]

  3: [] <= SinkNode[<2,0>]
```

```
   0: [A, A, I] <= SourceNode[]
   1: [A] <= null[]
   2: [] <= SinkNode[<1,0>]

4: [] <= SinkNode[<3,0>]
```

Figure 55: Compiler Output for Fast Fourier Transform

Figure 55 contains the compiler's output for the In-Place FFT. First we look at the bit-reversal permutation. The loop which performs the permutation of the input array did not receive any Array Traversal Nodes or Array Construction Nodes, making this an expensive computation. If the source code was not "In-Place", but created a new array to hold the permuted values, the compiler would be able to identify an in-order traversal. The Array Get/Set Separation transformation is not sophisticated enough to do this restructuring automatically. Yet, even with the restructuring, the creation of the new array would still be out of order. In practice, the permutation step may be avoided altogether, if the output, after some appropriate frequency domain computations, will be subject to an inverse FFT.

The compiler did not identify any of the loops that perform the butterfly updates as parallelizable, as evident in ***Figure 55*** not containing any For All Nodes. This is unfortunate, since the iterations of innermost loop do not depend on values computed by previous iterations. The problem is two-fold. First, Array Get/Set Separation is again not sophisticated enough to see that the array elements being read are not set by other iterations. Additionally, Array Construction Nodes are only created when the array elements are set at some constant interval (usually 1). Here they are spaced by an interval L, which is loop invariant, but not constant. This appears to be an oversight in the design of Array Construction Nodes, preventing the compiler from identifying a large class of parallelizable loops.

One of the few positive aspects of the compiler's output on this test case are the standard Optimizing Transformations applied to the code within the innermost loop. The calculations of array indexes left a lot of opportunities for Common Sub-Expression Elimination and Loop

Invariant Code Motion, which the compiler took advantage of. This resulted in a greatly reduced number of nodes in the sub-graph of the innermost loop.

Bibliography

[1] Andrew W. Appel. Modern Compiler Implementation in Java. Cambridge University Presses, 2002.

[2] Krste Asanovic, et. al., The Landscape of Parallel Computing Research: A View from Berkeley, Electrical Engineering and Computer Sciences, University of California at Berkeley, Technical Report No. UCB/EECS-2006-183, December 18, 2006. http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html.

[3] Jack B. Dennis. Fresh Breeze: A Multiprocessor Chip Architecture Guided by Modular Programming Principles. ACM Sigarch News, March 2003.

[4] Jack B. Dennis. Unpublished Memo on Fresh Breeze Multithreading and Futures. 2006.

[5] Erich Gamma, et. al., Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, 1995.

[6] J. McGraw, S. Skedzielewski, S. Allan, R. Oldehoeft, J. Glauert, C. Kirkham, B. Noyce, and R. Thomas. SISAL: Streams and Iteration in a Single Assignment Language. Language Reference Manual Version 1.2, Technical Report M-146, Rev. 1, Lawrence Livermore National Laboratory, Mar 1985.

[7] J. McGraw. 1982. The VAL Language: Description and Analysis. *ACM Trans. Program. Lang. Syst.* 4, 1 (Jan. 1982), 44-82. DOI= http://doi.acm.org/10.1145/357153.357157

[8] S. Skedzielewski, and J. Glauert. IF1: An Intermediate Form for Applicative Languages. Technical Report M-170, Version 1.0, Lawrence Livermore National Laboratory, July 1985.