View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by DSpace@MIT

Compiling Methods for the Fresh Breeze Processor

by

Curtis George Fonger, Jr.
B.S. Computer Science and Engineering, Mathematics
Massachusetts Institute of Technology, 2008

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 2009

© 2009 Massachusetts Institute of Technology. All rights reserved.

Author ....
Department of Electrical Engineering and Computer Science
May 22, 2009

Certified By..
Jack Dennis
Professor Emeritus
Thesis Supervisor

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

Compiling Methods for the Fresh Breeze Processor

by

Curtis George Fonger, Jr.

Submitted to the
Department of Electrical Engineering and Computer Science

May 22, 2009

In partial fulfillment of the requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The Fresh Breeze project is designing a new parallel architecture that is designed to serve as a model for high-performance computing while maintaining good software programming principles. One of the unique aspects of this architecture is a write-once memory model with hardware-based garbage collection and a global address space. Due to this memory immutability, new designs are needed before code generation can be developed for the Fresh Breeze compiler. This paper outlines designs for arrays and the method calling protocol. These designs are analyzed with respect to running times, possible alternatives, and algorithm performance.

Thesis Supervisor: Professor Jack Dennis
Title: Professor Emeritus

Acknowledgements

I would like to thank Professor Jack Dennis for his guidance and support while completing this project. I would also like to thank Bill Jacobs for discussing several aspects of this work with me and often helping me clarify my ideas. Thank you to my parents for giving me the opportunity to come to MIT and explore my passions. And of course, I would like to thank my wife Andrea for her support each day in too many ways to count.

……………………………………
Curtis Fonger

Table of Contents

Acknowledgements **4**

1. Introduction **7**

2. Background **10**

3. Array Representation **13**

3.1. Array Design **14**

3.2. Design Analysis **20**

3.3. Array Representation Alternatives **21**

3.3.1. Lopsided Tree 21

3.3.2. Extra Size and Depth Bits 23

3.3.3. Storing Size and Depth in Each Chunk 23

3.4. Implementation Analysis **25**

3.5. Sorting Algorithm Performance **28**

3.5.1. Background 28

3.5.2. Theoretical Analysis 29

3.5.3. Experimental Analysis 34

4. Method Call Protocol **41**

4.1. Implementation Details **45**

4.1.1. Method Call 45

4.1.2. Method Return 48

4.2. Method Call Protocol Alternatives **49**

4.3. Possible Additions **50**

5. Future Work **51**

6. Conclusion 53

Appendix A: Array Model Simulation Code 54

Appendix B: Sorting Algorithm Code 64

Bibliography 67

1. Introduction

Over the past several years, it has become apparent that increasing the speed of processors is becoming more difficult. As the difficulty of increasing processor clock speeds increases, parallelism presents itself as the preferred strategy to advance high-performance computing. Under most parallel architectures, programming efficient and robust algorithms is very difficult. Data conflicts, synchronization, and cache coherency problems hinder the effective use of parallelism and often lead to inefficient, bug-ridden programs. The Fresh Breeze project, led by Professor Jack Dennis in the CSAIL Computation Structures Group, aims to solve these issues by designing a processor that integrates a functional programming model with high-performance parallelism [2].

Six principles identified by Professor Dennis have guided the design of this model in order to ensure good software practices [1]. These six principles are based on software modularity, by which any program may be used as a component in any other program. This concept holds for both sequential and parallel computation. These six principles are listed as follows:

1. Information Hiding Principle: modules should not be required to know about the internal implementation of other modules in order to use them, and each module should hide its implementation details from the outside.
2. Invariant Behavior Principle: a program or subroutine should behave in the same way whenever it is called, regardless of the context from which it was called.
3. Data Generality Principle: any type of data should be able to be passed into and returned from modules in the same way.

4. Secure Arguments Principle: the arguments passed into a function cannot be changed by the function itself.
5. Recursive Construction Principle: any collection of modules forming a program should itself be a usable module for other programs.
6. System Resource Management Principle: resource management, such as memory allocation, should not be handled directly by the programs themselves, but rather by the system.

In order to conform to these principles, functional programming has been chosen as the programming paradigm for Fresh Breeze systems. Java, although not originally a functional programming language, can be restricted to a subset of the language that can be considered functional. Therefore a subset of Java known as Functional Java is used as the primary programming language under Fresh Breeze. This language satisfies the first five above principles, allows for the system to satisfy the sixth, and prohibits poor practices such as modifying method arguments, side effects, and aliasing, thus providing the compiler more optimization opportunities for parallelization.

The Fresh Breeze architecture uses a write-once memory model that supports these six principles. Under this model, all memory is organized into Directed Acyclic Graphs and is considered to be non-linear.

This work presents a design for the representation of arrays in memory, a protocol of method calling that lends itself to software modularization, and analyzes the performances of these designs. In Fresh Breeze, arrays are represented as trees with elements stored in the leaves at equal depth. The method calling protocol is designed

such that it supports the data generality principle, the secure arguments principle, and the recursive construction principle.

This remaining part of this thesis is organized into five chapters. Chapter 2 gives the necessary background on the Fresh Breeze processor and its memory model that are needed for the reader to understand the context of the rest of the paper. Chapter 3 describes the representation chosen for arrays under this memory model, analyzes this design and compares it to some alternatives, and examines how well sorting algorithms perform in a parallel system using this array design. In Chapter 4, the mechanism for calling methods is discussed and analyzed. Chapter 5 suggests possible future work to be done for Functional Java code compilation. Chapter 6 then gives the overall conclusions that may be drawn from this thesis.

2. Background

The Fresh Breeze processor consists of several multithreaded processors per chip that can be linked with other Fresh Breeze processors in parallel. The main purpose of this design is to promote stability while enabling efficiency for high-performance scientific computing [2].

One of the unique aspects of the Fresh Breeze architecture is its memory model. The system's memory is divided into discrete, 128-byte units called "chunks." Each chunk in the system is assigned a 64-bit Unique Identifier (UID) that is used when a program wishes to write to or read from a chunk. It is important to mention that chunks cannot be accessed sequentially, unlike most memory models. This UID is a global address, and any thread or program that wishes to use that chunk can only reference it with its UID.

The behavior of the memory model is structured to uphold the aforementioned software principles and provide data safety across threads. When a chunk is allocated, it is initially mutable. A program may write to any of the 128 bytes of the chunk using the ChunkWrite instruction. When the chunk is shared with another thread, the chunk becomes immutable and cannot be changed. This action is called "sealing" the chunk. Only after all references to the chunk have been removed may the chunk be reclaimed by an efficient hardware garbage collector, thus making it available again for future allocation [3].

This concept of immutability of written memory helps to solve many problems that are associated with high-performance parallel programming. In a parallel system,

sharing data between threads is often problematic. Concurrent access to a data structure is usually undesirable and can lead to serious problems, such as data corruption or race conditions. There are various methods that have been used to fix this problem, such as mutexes and semaphores, but these locking mechanisms are often difficult to implement correctly and can lead to other problems like deadlock. The immutability of shared memory prevents these issues and ensures the ability to safely run multiple threads that share data.

Another issue is cache coherency, keeping data up to date and consistent in each processor's cache. Maintaining cache coherency is arduous when trying to write high-performance parallel algorithms and cache coherency protocols are extremely complicated. With the introduction of hardware-enforced immutability of memory across threads, cache coherency no longer is a problem, since any shared data cannot be modified and thus requires no updating.

The hardware garbage collector responsible for reclaiming memory is based on reference counting. For this to be efficient, a certain rule has been imposed on the memory model: there can exist no pointer cycles. Instead, chunks are organized into Directed Acyclic Graphs (DAGs) and only UIDs of sealed chunks may be written to unsealed chunks [7]. This provides a cycle-free heap and allows the garbage collector to use reference counting for garbage collection. Otherwise, the garbage collector would not garbage collect chunk cycles, which could cause memory leaks. Pointer arithmetic is also forbidden, thus ensuring that unreferenced or unsealed chunks cannot be accessed without a deliberate given pointer. This model also allows for files to be represented as a DAG of chunks of unbounded extent. This, along with the global address space, removes

the need for separate memory and file systems, and therefore there is no distinction between the two in a Fresh Breeze system. This helps satisfy the data generality principle by allowing files to simply be a collection of chunks and thereby giving them the ability to be passed to functions like any other object.

In order to design a compiler that can translate Functional Java into Fresh Breeze machine code, the representation of arrays and the method call protocol that the Fresh Breeze compiler will use for code generation must be designed to work with this memory model.

3. Array Representation

When designing an array representation for the Fresh Breeze architecture, the differences between the Fresh Breeze memory model and the memory model in most computers today must be considered. In a standard memory model, memory addresses are laid out linearly and local memory blocks may be allocated and changed without limit. In the Fresh Breeze model, the memory is organized into 128-byte chunks. A chunk is the smallest amount of memory that can be allocated at one time. Given the chunk UID and the offset, the values of the words in a chunk can be written, but after a pointer to that chunk is sent to another thread, all data in the chunk becomes immutable and that portion of memory cannot be changed until it goes out of scope and is garbage collected. Chunk allocation is not guaranteed to be linear. Memory access is only linear within a chunk, not across chunks. Therefore the array representation design must conform to this behavior of memory allocation and modification.

The goal of this design is to make arrays feasible for high-performance computing and have running times comparable to arrays in a standard memory model. The representation should allow for good performance with common array operations, such as accessing an element, changing an element, and adding or removing an element. From a larger perspective, arrays should yield good performance when performing operations such as dot products, matrix multiplication [3], and common algorithms such as sorting and FFTs.

3.1. Array Design

To store an array in memory, we can use a structure that consists of a tree of chunks. Every element in the array is stored in the lowest level of the tree. The top chunk of the tree reserves the first 64 bits to store the current size of the array and the depth of the tree. The size is the number of 32-bit words in our array, even if the array is being used to store 64-bit data. The depth is the number of levels the tree currently has, starting at a depth of 1. The remaining space is used to store either the elements of the array (if the number of elements fits in 15*64 bits) or pointers to the next level of the tree. Here is a sample array of 40 64-bit doubles:

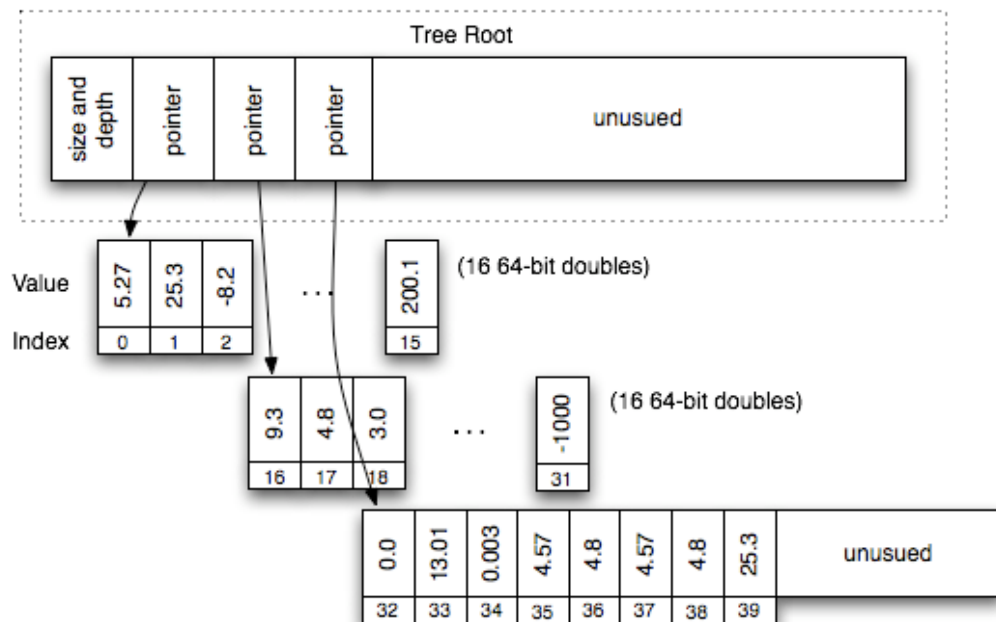

Figure 1

To compute the depth needed to store a given number of elements, it is useful to consider the range of sizes of an array with a certain depth. For convenience, let n be the number of elements in an array with 64-bit elements. Then for a depth d, the range of

sizes for the array is $15 \cdot 16^{d-2} < n \leq 15 \cdot 16^{d-1}$. This is because the root chunk has a branching factor of 15 and all lower chunks have a branching factor of 16. Solving for *d* with this inequality:

$$\begin{array}{ccccc} 15 \cdot 16^{d-2} & < & n & \leq & 15 \cdot 16^{d-1} \\ \log_{16} 15 \cdot 16^{d-2} & < & \log_{16} n & \leq & \log_{16} 15 \cdot 16^{d-1} \\ d-2 & < & \log_{16} n/15 & \leq & d-1 \\ d-1 & < & \log_{16} 16n/15 & \leq & d \\ d-1 & < & \log_{16} s/60 & \leq & d \end{array}$$

The value *s* represents the number of bits needed to store the array. These inequalities are only true when $n \geq 1$ or when $s > 60$, because otherwise the logarithm becomes too small. To fix this, noting that the depth of the tree must always be at least 1, take the maximum of $\lceil \log_{16}(s/60) \rceil$ and 1. Therefore depth of the tree *d* is equal to $\max\{1, \lceil \log_{16}(s/60) \rceil\}$. If the array is being used to store 32-bit integers, this formula can be rewritten as $\max\{1, \lceil \log_{16}(32n/60) \rceil\}$, where *n* is the number of elements in our array.

When accessing an element in the array, an operation we will call AccessElement, one starts at the root node and traverses down the tree until reaching the desired element. This particular path down the tree is known beforehand based on the depth of the tree and the index being accessed. If *d* is the depth of the tree and *s* is the current number of bits in the array, each array access requires *d* memory accesses, or $O(\log(s))$.

Although $O(\log(s))$ seems undesirable, actual performance should be faster when performing operations such as dot product or matrix multiplication. If all the elements in an array are being accessed in order, this array traversal takes constant amortized access time if the pointers to lower nodes in the tree are saved. This prevents the program from having to start at the root node for every AccessElement operation.

To replace an element in the array (ChangeElement), only the nodes of the path in the tree that lead to the desired element need to be copied and modified, not the entire array. For example, if a program wants to change the element at index 18 in the array displayed in Figure 1 and set it to 1234.5, we only need to copy and change the two outlined chunks shown in bold:

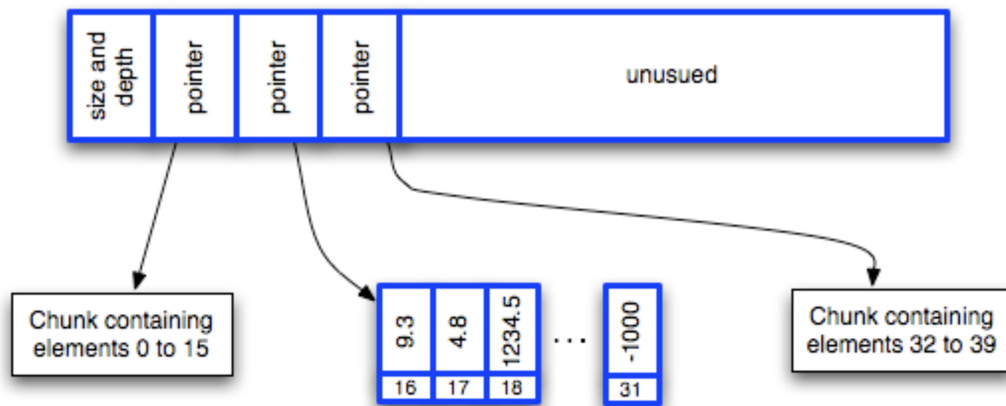

Figure 2

This requires d chunks to be copied and modified, or $O(\log(s))$. The root node needs to be copied and modified because it needs to replace the pointer to the chunk containing elements 16 though 31.

Adding and removing elements from the end of an array are very similar to the operation to change an element, except for the need to dynamically change the number of chunks allocated, change the size label, and if necessary increase or decrease the depth of the tree. To perform the AddElement operation by adding an element to the end of the array, if adding one element does not increase the size of the array past $64*15*16^{d-1}$ bits, we simply employ the strategy of changing the spot after the last element of the array,

adding an extra chunk if needed. However, if the array is already using $64*15*16^{d-1}$ bits, then no more elements can be added to the tree while keeping the same tree depth. Therefore a new root node is created and the depth of the tree is increased by 1. Then the old root node is copied by removing the size and depth labels and shifting the contents to the left. The new 64 bits at the end of this node are then used to either add the new element or to add a pointer to a new chunk which will lead to the element we are adding. Figure 3 demonstrates this transformation:

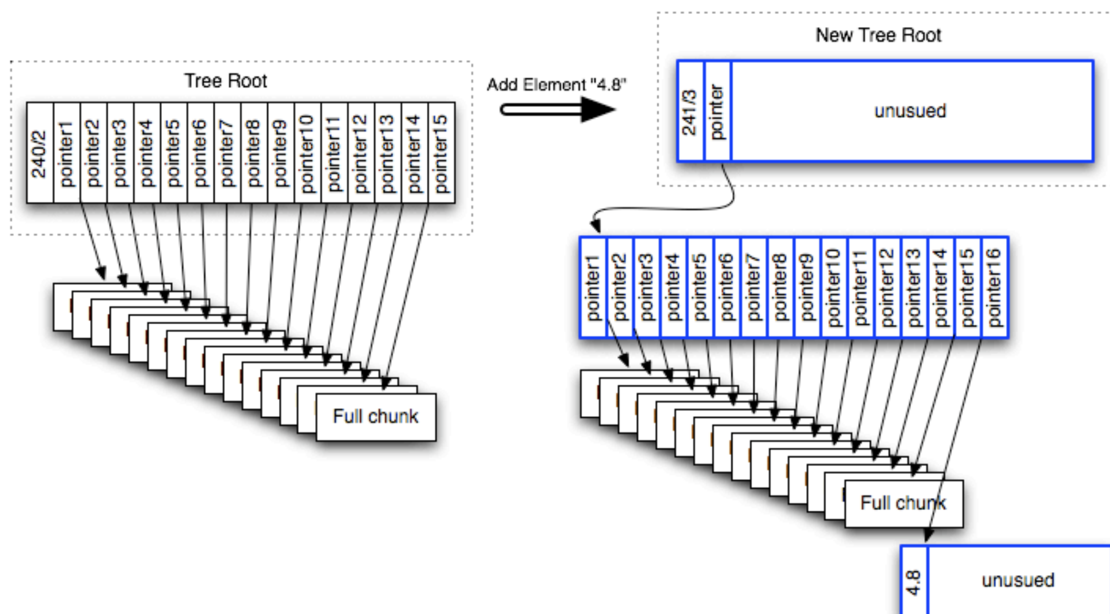

Figure 3

This operation requires d chunks to be copied and modified, where d is the new depth after adding the element.

Removing an element from the end of an array, or RemoveElement, is almost the reverse of adding an element. If removing one element does not decrease the value $\max\{1, \lceil \log_{16}(s/60) \rceil\}$, then only the value of the size of the array needs to be changed,

and thus only the root node is modified. If the removal also frees a chunk, then the reference count of that freed chunk can simply be reduced. However, if the removal decrements the value $\max\{1, \lceil \log_{16}(s/60) \rceil\}$, then the single chunk at depth two of the tree becomes the root node and the appropriate copies and modifications are made, thus decreasing the depth of the tree by one. This is the reverse of the process of increasing the depth of the tree when adding an element. Figure 4 shows an example of this process:

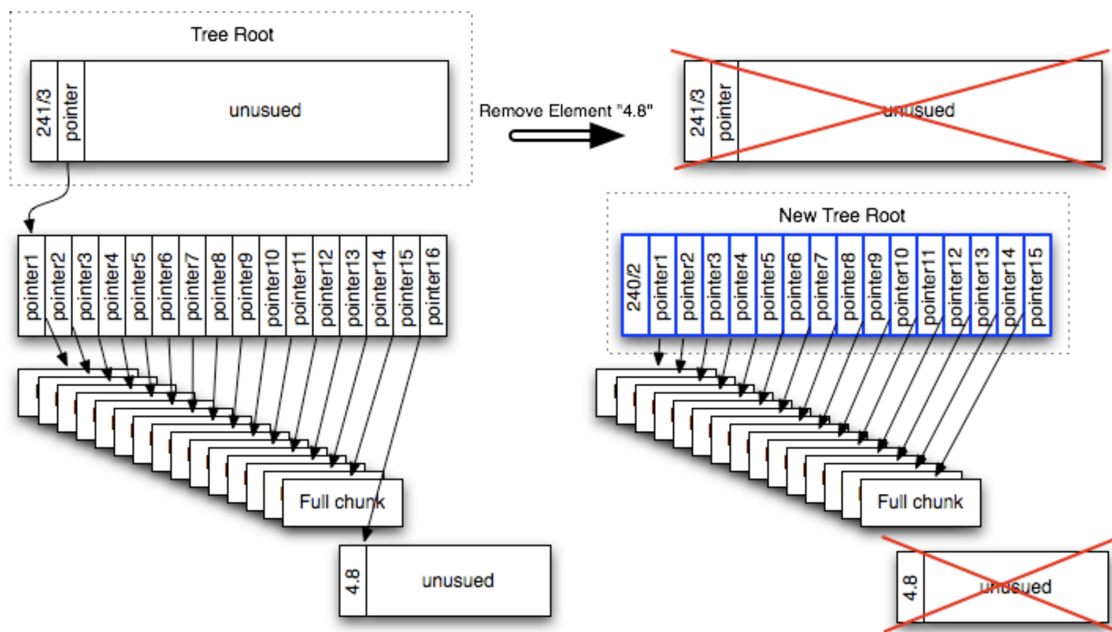

Figure 4

Depending on how the Fresh Breeze Processor is implemented, the RemoveElement operation may be an $O(1)$ or an $O(d)$ operation in the worst case. If there is a way to dereference chunks explicitly, this could be $O(1)$, but otherwise this operation may need to copy and modify d chunks.

Both AddElement and RemoveElement could be $O(\log(s))$ in the worst case, but again considering the frequency with which programs perform array operations in a linear

fashion, these will most likely be constant amortized time when adding or removing a large number of elements at once.

This proposed design allows for a convenient space-saving optimization where the space for a large array for which most of the elements are zero can be dynamically allocated. This is particularly useful if using a data structure such as a sparse matrix, where an array of large size is allocated but only a fraction of that array is actually used. Consider the following example:

```
int[][] sparseMatrix = new int[1000][1000];
sparseMatrix[347][129] = 29;
sparseMatrix[42][241] = -5;
sparseMatrix[271][314] = 1234;
```

Under a traditional memory model, nearly 4 megabytes of RAM would need to be allocated to execute this code if a special data structure for the matrix were not being used. However, with the presented tree structure design, chunks can be allocated as needed when inserting elements into the array. In Fresh Breeze memory chunks, each 64-bit slot for a pointer has an extra bit indicating whether that slot currently stores a valid pointer or not. When accessing an element in an array, a program can check these bits when traversing down the tree. If any of these bits ever indicates that the retrieved pointer is not valid, that means that no value has been inserted at the specified index in the array. In this case, the AccessElement operation can either return zero or a null pointer, depending on the type of array. When using a large array where only small parts of the array are actually used, this optimization can save significantly in both space and average array lookup time.

In traditional Java, allocating an array takes $O(n)$ time, where n is the initial size of the array. This is because depending on the data type of the array, Java needs to initialize all elements of the array to either 0, null, or false. This differs from C/C++, where this initialization does not take place and thus takes constant time for array allocation, but any leftover memory values where the array is allocated become the array's initial values.

With this design optimization for arrays in Functional Java, Fresh Breeze arrays get the advantages of both programming languages. Every element of the array is implicitly initialized to 0, null, or false, but the code allocating the array does not need to fill the array with these values. Fresh Breeze arrays gain the benefits of having a space- and time-efficient design while keeping the same principal form as arrays where all spots are filled.

3.2. Design Analysis

There are several advantages of this design for storing arrays in memory. Keeping the size of the array in the top chunk allows for $O(1)$ performance when a program queries the current length of the array. Although storing the current depth of the tree is not strictly necessary, since it can be calculated from the array size, it is convenient to have it available so that a program does not need to calculate it each time the program accesses an array element.

One disadvantage of this design is the difference between the root node and the other nodes in the tree. Since the first 64 bits of the root node are used for storing the size and depth of the tree, only 960 bits of the chunk remain for array data or pointers, as opposed to 1024 bits in every other chunk in the tree. This makes implementing array

handling in code slightly more troublesome and also dictates that when a program needs to increase the depth of the tree, it needs to copy and modify the old root node, the process shown in Figure 3, instead of simply pointing to the old root node and allocating two new chunks. However, the performance loss associated with this situation should be negligible.

3.3. Array Representation Alternatives

There are several possibilities for the array representation design, a few of which will be presented and compared with the chosen design. A linear array structure is not included in this discussion due to the issues in poor running time when attempting to access an element in the middle or end of the array.

3.3.1. Lopsided Tree

A lopsided tree design would be similar to the chosen representation, but would store the last elements in the array in chunks higher up in the tree structure, rather than all at the same level. There are several possible variations of the details of this design, but the main idea would be to gain a performance boost by utilizing the normally unused space closer to the root node. If one wanted to store the array of elements shown in Figure 1 in this type of array, a data structure of the type shown in Figure 5 may be produced.

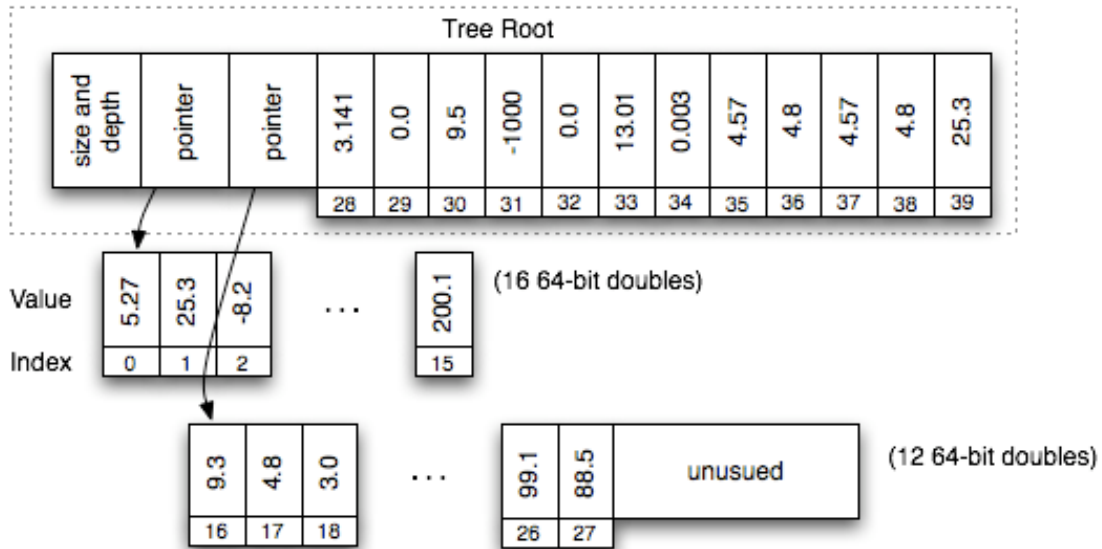

Figure 5

The advantages of this type of design are the reduced number of allocated chunks and the average number of memory accesses to access an element is decreased. The AccessElement function would require more computation, but would require fewer memory accesses, which could lead to increased average performance if accessing memory was much slower than doing a little more computation.

The main disadvantage of this design is the design complexity. The greater number of possible locations for elements significantly complicates the process of determining where the element at a given index is located in the tree, and this complexity increases as the tree's depth is increased. Once a tree reaches 3 or more levels, accessing an element may take several computations based on the size of the tree, and the AddElement and RemoveElement procedures would require extra computation to determine where to redistribute the elements at the end of the array.

3.3.2. Extra Size and Depth Bits

Another design possibility would be to have 34 extra bits for each chunk that would only be used if that chunk were being used in an array. The root node would store the size and depth of the entire tree and every child chunk would store the size and depth of its corresponding subarray. This would eliminate the need to store the size and depth at the beginning of the root node and thus have every node be perfectly divisible by 16 or 32. The disadvantage would be that every chunk would need an extra 34 bits, requiring more hardware for the same amount of space.

One advantage of this design is that it allows for any arbitrary array chunk to be sent to a method or thread and be treated as the root of an array. This would provide slightly better modularity for the passing of subarrays.

This design also reduces the number of allocated chunks on average, but only by a constant amount. For any array of arbitrary size, the greatest number of allocated chunks this design would save would be one. This is because the chosen design uses extra space at the beginning of the root chunk that could be used for one more pointer.

3.3.3. Storing Size and Depth in Each Chunk

One alternative that is similar to the proposed design would be instead of only storing the size and depth of the tree in the root chunk, the size and depth of each subtree could be stored in the root chunk for that subtree. The root chunk would look exactly the same, but the child chunks of the tree would keep track of how many elements were stored in their portion of the subtree. An example of this structure can be seen in Figure 6, which is an array storing 32-bit integers containing 1,347 elements with a tree depth of 3.

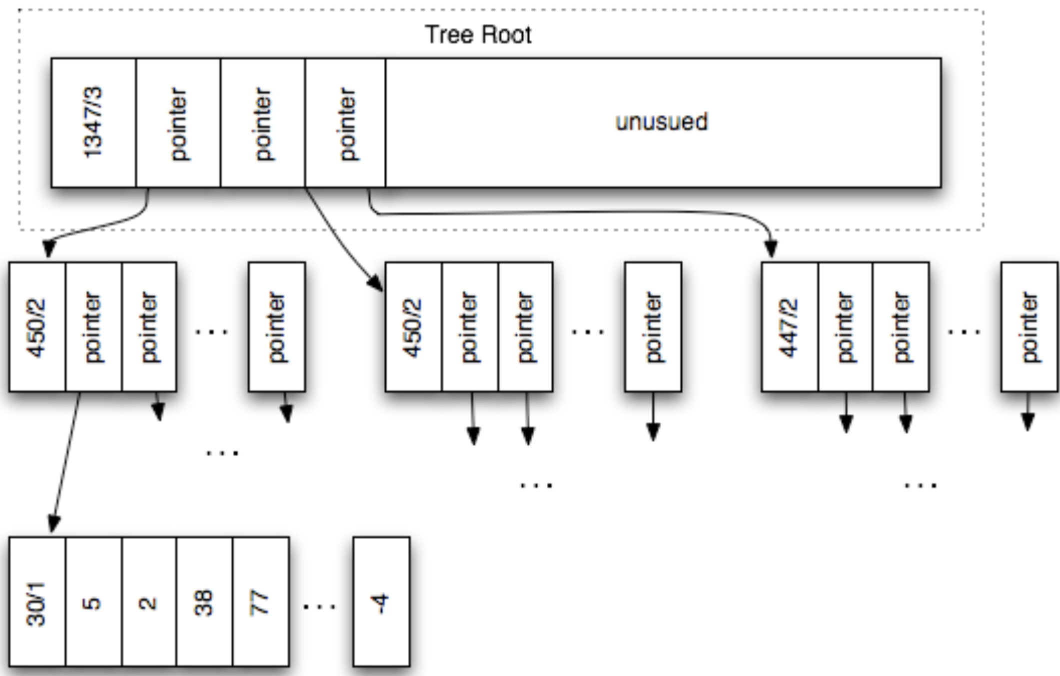

Figure 6

The main advantage of this design is that each subtree is a subarray that fits the array specification in its own right, meaning that since the chunk stores the size and depth of its subarray, any chunk in the array can be viewed as the root of its subarray. This may be useful if a program wanted to send a subarray to a method or multiple subarrays to separate threads for parallel execution.

This design is similar to the design in which each chunk had extra bits that would store the size and depth of an array with that node as the root, except that the first 64 bits of the chunk are used instead. This saves memory chip size at the expense of having to allocate more chunks for an array.

There are two large disadvantages with this design. The main disadvantage is that since each chunk has 15 spots to store pointers to chunks lower in the tree, a program

cannot simply use a bit mask to determine the next chunk as it traverses down the tree during an array lookup. Instead, it would need to divide its current index by powers of 15, a more time consuming operation. However, it is plausible to consider that if a "divide by 15" operation were incorporated into the Fresh Breeze ALU, this operation could be made fast.

The second disadvantage is that this design is slightly less space efficient. For large arrays, more chunks would be needed in this design than in the chosen design. For example, for the array in Figure 6 with 1,347 integers, 49 chunks would be needed to store the array. In the chosen design, only 47 chunks are needed.

3.4. Implementation Analysis

In designing the memory representation of arrays in the Fresh Breeze architecture, how array operations will translate into Fresh Breeze instructions is important to consider for estimating actual runtime performance and implementation complexity for each operation.

For the AccessElement function, a program needs to know the depth of the tree in order to determine which path to take. Once it has the tree depth, it can find which branches of the tree to take by using the appropriate bit masks for that depth. For an array storing 32-bit data elements, a 5-bit mask is used on the last 5 bits of the index, since there are 32 possible positions of the element at the lowest depth, and a 4-bit mask is used on the higher order bits to choose the pointer that points the program to the appropriate chunk in the next level of the tree.

Implementing AccessElement(Array A, int N) for an array containing 32-bit data would look like the code shown in Figure 7. ReadChunk(Pointer p, Index i) reads the 32-

bit word in the chunk at pointer p at position i, where the chunk is divided into 32-bit sections. ReadChunk64(Pointer p, Index i) reads the 64-bit data in the chunk at pointer p at position i, where the chunk is divided into 64-bit sections.

```
AccessElement(Array A, int N) {

  Ptr = A;
  D = GetDepth(A);
  Root = true;
  while (D > 1) {
    Mask = 0b1111 << 4*(D-1)+1;
    I = (N & Mask) >> 4*(D-1)+1;
    if (Root) {
      I++;
      Root = false;
    }
    Ptr = ReadChunk64(Ptr,I);
    D--;
  }
  Elt = ReadChunk(Ptr, (N & 0b11111));
  return Elt;

}
```

Figure 7

ChangeElement is similar to AccessElement in that it traverses down the tree to find the desired element at the specified index. Once it finds the element, it copies the chunk the element is located in and changes the element at the appropriate spot. It then needs to go back up the tree and copy/modify the ancestor chunks to update the pointers to the new children.

One possible implementation for the ChangeElement operation can be seen in Figure 8. The ChangeAndCopyChunk method is a helper function for ChangeElement. The idea is that the operation finds the element it needs to change, copies the chunk, changes the element at the specified index (which it can do because it has not yet shared

the chunk's pointer with another thread), and returns the new chunk. This needs to be recursive because it must update the pointers higher in the tree to point to the new chunks it is creating.

```
ChangeElement(Array A, int N, int Elt) {

  D = GetDepth(A);
  return ChangeAndCopyElement(A,N,true,D,Elt);

}

ChangeAndCopyChunk(Array A, int N, boolean Root, int D,
  int Elt) {

  if (D != 1) {
    Mask = 0b1111 << 4*(D-1)+1;
    I = (N & Mask) >> 4*(D-1)+1;
    if (Root)
      I++;
    Ptr = ReadChunk64(A,I);
    Ptr2 = ChangeAndCopyChunk(Ptr,N,false,D-1,Elt);
    Ptr3 = CopyChunk(A);
    ChangeChunk64(Ptr3,I,Ptr2);
    return Ptr3;
  } else {
    I = N & 0b11111;
    Ptr = CopyChunk(A);
    ChangeChunk(Ptr,I,Elt);
    return Ptr;
  }
}
```

Figure 8

Again, the running time of this operation is $O(\log(s))$, but when changing several elements of an array in order, it can get constant amortized time for each ChangeElement operation for the same reasons that AccessElement often takes constant amortized time. The code shown in Figure 8 is only that of a general ChangeElement operation. With

compiler optimizations, several elements can often be changed at a time if updating an array in a linear fashion.

3.5. Sorting Algorithm Performance

3.5.1. Background

When evaluating the performance of the tree array design for the Fresh Breeze memory model, one should evaluate algorithms that rely on frequent array operations. Sorting algorithms are especially worthy of evaluation due to their complexity and heavy reliance on fast array performance. Several sorting algorithms will be evaluated and contrasted to discuss the advantages and drawbacks of each with under the Fresh Breeze model. An overview of how each sorting algorithm works under the proposed array design will be given, followed by an analysis and comparison of the methods. In this discussion, merge sort and quicksort are evaluated. Heapsort was not considered in this analysis due to the difficulty in parallelizing the algorithm, thus losing one of the most obvious advantages of the Fresh Breeze architecture.

Merge sort is a recursive algorithm that sorts an array by first sorting the left half and the right half of the array, then constructing a new sorted array by using the two sorted array halves. With the proposed array representation, at each step the two halves of the arrays can be sent to the recursive calls unmodified, possibly spawning new threads. After the two recursive calls return, the parent method will have two sorted arrays. Now a new array must be constructed using the two sorted halves, sequentially going through the left and right arrays to construct the new one. This is the only step in which new chunks need to be allocated.

Quicksort is another recursive algorithm that works by randomly choosing an element from the input array as the pivot, then sequentially comparing the other elements to the pivot to determine if they should go to the left or the right of the pivot, often called the partitioning step. After the array is partitioned, the left and right partitions are sent to recursive calls of the method, often being spawned off in new threads. Due to the immutability of Fresh Breeze chunks, the partitioning step cannot be performed in place, so a new array must be constructed at this step. After the recursive calls return, a new array must be constructed with the returned left and right halves of the partition.

3.5.2. Theoretical Analysis

With merge sort, there is only one time during each method invocation when new chunks need to be allocated. This is when a new array is constructed from the left and right sorted halves. This is because the algorithm can reuse the input array without the need for modification when it makes its two recursive calls.

With quicksort, because the memory model forbids in place array manipulation, there are two points during each method invocation where chunks are allocated. These are during the partitioning step before the recursive calls and after the recursive calls when the two returned halves need to be joined. Note that if programmed optimally, the returned left and right halves of the array can be structured such that the parent call can reuse most of the chunks in the left and right halves of the array and only construct new chunks to join together the center of the array. An illustration of this optimization can be seen in Figure 9.

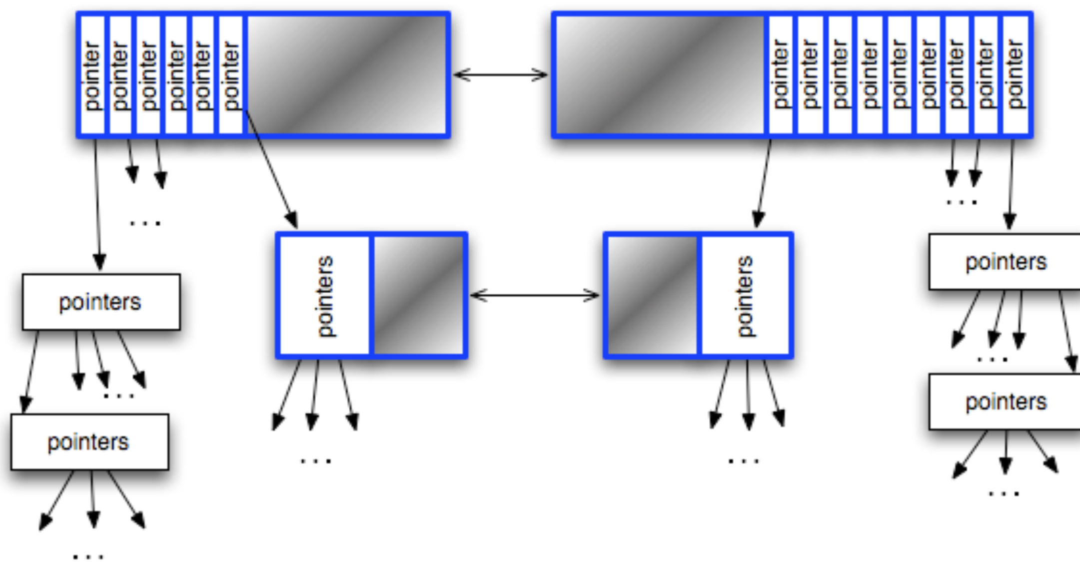

Figure 9

In this figure, the chunks in bold need to be joined together where the unbolded chunks (the chunks on the sides of the figure) can remain unmodified. The gray areas in the bold chunks represent empty space that is to be filled by the data in the corresponding chunk shown by the double arrows when these chunks are recopied. This "join" function saves the algorithm from having to recopy the entire array when both recursive calls return, but rather only needing to recopy the right-most chunks in the left half and the left-most chunks in the right half. It may be beneficial to include this ArrayJoin procedure as special function in the Fresh Breeze standard library for Functional Java. An example of what this procedure may look like can be found in Appendix A as the join32() method in the FBArray class.

With an unoptimized sequential traversal of an array, where each element is accessed by starting at the tree root, the running time of either sorting algorithm would be impacted. However, if a program keeps a trail of UIDs from the root node to the current

leaf node in the registers, then the algorithm can sequentially access the elements of the array by remembering the pointers to the parent nodes. One can see this by observing how many memory accesses it would take to sequentially go through the leaf nodes in a binary tree. If a perfectly balanced binary tree has $n=2^k$ leaf nodes, then there are $2n-1$ nodes in the tree. Consider the tree in Figure 10:

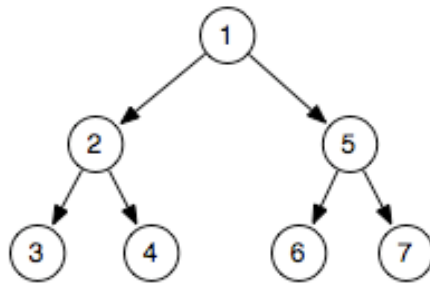

Figure 10

To most efficiently access the leaf nodes, keeping a trail of pointers for the nodes in the registers, the most efficient pattern of accesses is $1 \rightarrow 2 \rightarrow 3 \rightarrow 2 \rightarrow 4 \rightarrow 1 \rightarrow 5 \rightarrow 6 \rightarrow 5 \rightarrow 7$. In general, it takes $n + 2(n-1)$ accesses, which is linear with respect to n. Therefore the array representation of Fresh Breeze arrays does not affect the asymptotic running times of the sorting algorithms with respect to sequential access.

Both merge sort and quicksort perform in $O(n \log n)$ expected running time on a single processor and in $O(n+(n/p) \log n)$ expected running time on multiple processors, where p is the number of processors available. This running time over multiple processors can be derived by considering how the work is divided over p processors. To make the analysis simpler, let $p = 2^{k_1}$ and $n = 2^{k_2}$. Then because the $O(n)$-time parts of the algorithms can be performed in parallel, the top level will take n time, the second

level will take $n/2$ time, the third level will take $n/4$ time, and so on. This will continue until after k_1 levels where we reach $n/2^{k_1}$ time. All levels below that will take $n/2^{k_1}$ time, and there are $k_1 - k_2$ such levels. Overall this leads to a running time of $O(n + (k_1 - k_2)(n/2^{k}))$ = $O(n + (\log n - \log p)(n/p))$. However, this model breaks down as p approaches n and no longer makes sense if p is greater than n, so the $\log p$ should be eliminated in the equation. Therefore the running time is
$O(n+(n/p) \log n)$. If given enough processors to perfectly distribute the recursive calls, the Fresh Breeze parallel architecture achieves $O(n)$ time.

When comparing merge sort and quicksort under the presented array representation and the Fresh Breeze memory model, there are two significant differences between the Fresh Breeze model and the standard architecture of most of today's computers that must be noted. First of all, the tree structure of Fresh Breeze arrays favor sequential operations rather than random access operations. Secondly, the immutability of chunks once they are written eliminates most "in-place" benefits that a sorting algorithm might have.

One of the major benefits of merge sort over quicksort in a parallel architecture is that the input data can be evenly split amongst the available processors at each recursive step, whereas with quicksort there is no guarantee with each recursive call that the input data will be evenly divided. In the worst case, the partition step always chooses a pivot that is either less than or greater than most of the elements, thereby achieving $O(n^2)$ runtime. When this happens, parallelizing the algorithm does not help the runtime performance.

Quicksort is often faster on many architectures due to the ability to sort an array in-place. However, in Fresh Breeze this is not possible because once an array is stored in memory, it cannot be modified and thus any portion of the array that is to be modified must be copied into newly allocated chunks. Thus the Fresh Breeze architecture cannot make use of this principal benefit of quicksort. In merge sort, at each step only one new array needs to be constructed after the recursive calls return. In quicksort, however, a new array needs to be constructed during the partition step and several extra chunks need to be allocated in the optimized array join step after the recursive calls return.

Although not strictly a runtime concern, there exists the issue of programmability of the algorithms. With merge sort, it is possible to program an optimized version of the algorithm in functional Java. This is because the original input array can be sent to each recursive call, along with starting and ending indices to signal which section of the array this particular call should consider. When the two recursive calls return, the active call needs to create an entirely new array in any optimized version, and this can be done easily in Java. However, with quicksort, in order for the algorithm to be optimal, the returned arrays from the recursive calls should be of the form shown in Figure 9, where the data is structured in the chunks such that joining the two arrays only requires joining the middle chunks in the diagram. This cannot be achieved in Functional Java without the special ArrayJoin operation being added to the Fresh Breeze standard library for Functional Java, since direct manipulation of the underlying chunks of the tree structure would be required.

Despite the above disadvantages of quicksort, many advantages of the algorithm still apply with Fresh Breeze. With merge sort, a naïve implementation will result in $2n$ –

1 method calls in the worst case, whereas in quicksort there are always *n* recursive calls because there is one call per pivot element. This may lead to better utilization of the available parallel processors than merge sort will achieve. A smart implementation of quicksort may yield better cache performance if the algorithm is restricted to the registers once the input array is below a certain size. However this may not be an actual negative for merge sort in practice, since it is common to switch to a different sorting algorithm once the array size is sufficiently small and may be sorted in place in a single processor's registers.

3.5.3. Experimental Analysis

Although examining merge sort and quicksort on paper is useful, it is beneficial to see how the two algorithms actually perform in practice. To do this, we have written a program in Java that simulates a simplified version of the Fresh Breeze memory model, Fresh Breeze arrays, and implementations of merge sort and quicksort using these arrays. The full code for this simulation can be found in Appendix A and B.

The objectives of this simulation were to see how these algorithms may actually be programmed optimally using these arrays and to observe how well they perform. Three principal measurements were taken for each algorithm: the number of memory (or chunk) accesses, the number of allocated chunks, and the number of method invocations.

Memory accesses were measured with several optimizations in mind. First, sequential accesses were measured with the pointer trail optimization discussed in the previous section associated with Figure 10. Second, several reads or writes of the same chunk were considered to be only one access, due to likely cache performance and bulk memory transfers in a real processor.

Counting allocated chunks was relatively simple. All that was needed was to count the number of chunk objects instantiated in each algorithm. Determining the number of method calls was equally as simple. The only methods considered to be actual method calls were the recursive calls, since all other methods invocations were simulation methods.

When writing the algorithms, several of the previously discussed optimizations were used. Due to the speed of sorting small arrays of numbers in the registers, when an input array was less than or equal to 16 elements (thus ensuring there would be enough space in the registers), no recursive calls were made and no chunks were allocated until the array was sorted. Instead, both algorithms switched to selection sort and used the registers in place of memory.

In merge sort, the entire input array was passed to each method call, along with starting and ending indices. Quicksort used the ArrayJoin optimization by creating sparse arrays, each with a size equal to the input array, and adding the sorted elements to them with a specified offset so that ArrayJoin could be executed. The two algorithms are shown in Figures 11 and 12:

```
FBArray mergeSort(FBArray a, int start, int end) {
	if (a.size() <= 16) {
		// selection sort details hidden
		return newArray;
	}

	FBArray left = mergeSort(a, start, (start+end)/2);
	FBArray right = mergeSort(a, (start+end)/2, end);
	FBArray newArray = new FBArray();

	int i = 0, j = 0;
	while (i+j < left.size()+right.size()) {
		if (i == left.size()) {
			newArray.addElement(right.read(j)); j++;
		} else if (j == right.size()) {
			newArray.addElement(left.read(i)); i++;
		} else if (left.read(i) <= right.read(j)) {
			newArray.addElement(left.read(i)); i++;
		} else {
			newArray.addElement(right.read(j)); j++;
		}
	}
	return newArray;
}
```

Figure 11

```
FBArray quickSort(FBArray array, int start, int size) {
	if (array.size() <= 16) {
		FBArray newArray = new FBArray(size);
		// selection sort details hidden
		return newArray;
	}

	int pivotPoint = random.nextInt(array.size());
	int pivotValue = array.read(pivotPoint);
	FBArray left = new FBArray();
	FBArray right = new FBArray();

	for (int i = 0; i < array.size(); i++) {
		if (i == pivotPoint) {
			continue;
		}

		int elt = array.read(i);
		if (elt <= pivotValue) {
			left.addElement(elt);
		} else {
			right.addElement(elt);
		}
	}
	FBArray leftDone = quickSort(left, start, size);
	FBArray rightDone = quickSort(right, start+left.size()+1,
size);
	FBArray done = FBArray.arrayJoin(leftDone, rightDone,
start+left.size());
	done.changeElt(start+left.size(), pivotValue);
	return done;
}
```

Figure 12

These two algorithms were applied to several arrays of various sizes. Integer arrays with random values were generated with sizes between 100 and 1200 elements at increments of 20. For each array size, the same array was given to both algorithms. Because quicksort is a randomized algorithm, it was considered important to try to minimize variance, so the same array was given to quicksort 20 times and each measurement was averaged over those trials.

The results of these tests can be seen in Figures 13, 14, and 15:

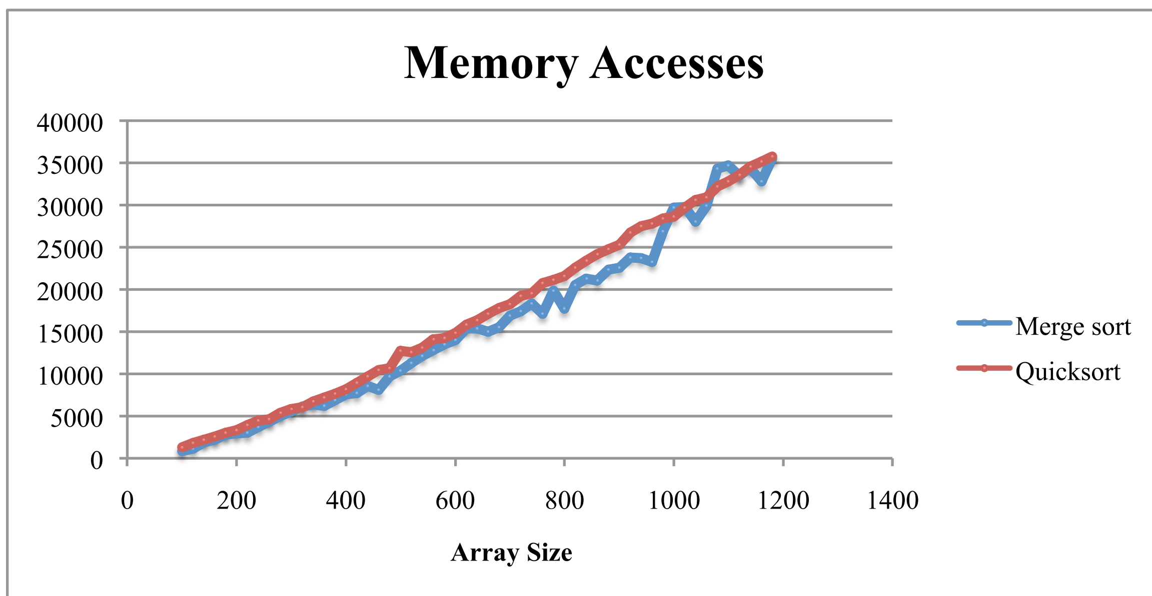

Figure 13

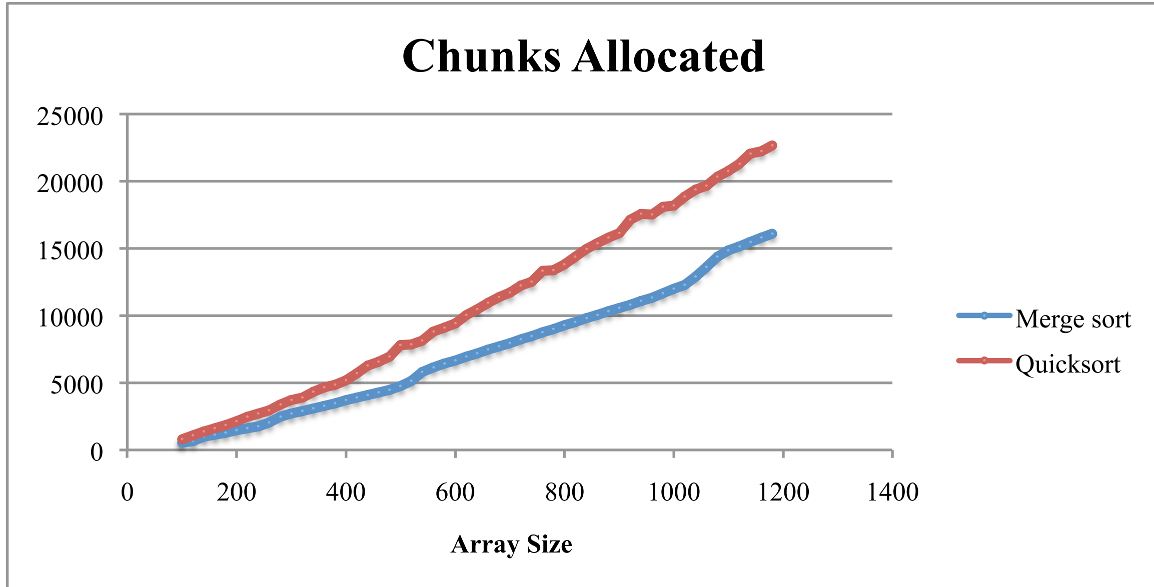

Figure 14

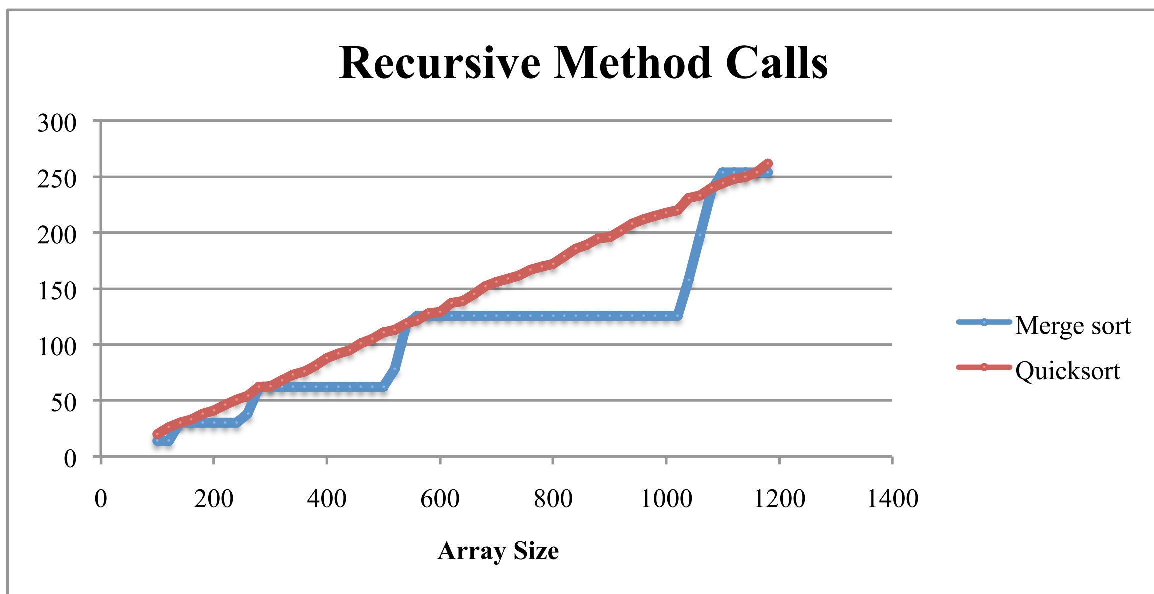

Figure 15

It seems that even though merge sort may a slight advantage on the number of memory accesses, both algorithms have very close results. The graphs depicting chunks allocated and method calls provide a better differentiation between the two.

We predicted that quicksort would require more space, even with the ArrayJoin optimization. On average, it appears that the number of allocations that merge sort makes

is about 68% of the number of allocations that quicksort performs. However, allocating chunks is relatively cheap compared to memory accesses or method spawns. Assuming that the memory management logic always maintains a few free chunks in on-chip memory for use, allocating one chunk should take on average 1 to 2 clock cycles [4].

The results from the method call graph are more surprising. Without the hybrid algorithm approach (in other words, combining merge sort/quicksort with selection sort), quicksort would perform better in this area. However, due to the random nature of quicksort, merge sort actually makes fewer method calls on average due to the even split of the data on each recursion. Due to quicksort's uneven average splitting, it again performs worse than merge sort. This is a major impact on the two algorithms' performance, since a spawn operation takes about 16 to 20 clock cycles to execute, and the join operation's cost is similar [4].

There are several limitations to these results. First, it is unclear if the hybrid strategy of combining these algorithms with selection sort would be worthwhile on actual Fresh Breeze hardware. When the selection sort strategy is removed, the method call results are reversed: quicksort becomes the clear winner, although this data is not shown here.

Second, it is difficult to estimate real cache performance. Although the best estimates were attempted, these estimates could be quite different depending on the specific details of the system.

Third, the number of available processors was not taken into account when performing calculations. For a small number of processors, it is most likely better to

have an even split of the array at each recursion as in merge sort, rather than a random split one can see in quicksort.

In general, whether quicksort or merge sort is faster using the proposed array representation design in the Fresh Breeze architecture will depend on the efficiency of method calls, the number of processors available, and the expense of allocating new chunks, though both the theoretical analysis and the experimental data tend to favor merge sort.

4. Method Call Protocol

In most architectures, such as Intel or PowerPC, method calling involves pushing method arguments that are stored in the registers or in memory onto a stack, a section of RAM reserved for argument passing and local variable storage. In Fresh Breeze, the system is prevented from using a stack for two reasons. First, the immutability of written chunks and the non-linearity of memory hinder the design and time efficiency of using a stack. Second, requiring a stack to pass arguments to and return values from program methods does not lend itself well to a parallel system, as the slave processors might finish their method calls in a different order from which they were called, and thus writing return values to the master processor's stack would be overly complicated.

When designing a method calling protocol for Functional Java, there are several objectives to keep in mind. Method calling is a common operation and therefore needs to be efficient. Minimum usage of memory is preferable and the number of required memory accesses should be low. The design also ought to preserve a program's modularity, thus using only one standard for both single- and multi-threaded behavior. Any program should be able to call a method of any other program simply by observing its interface and passing along the required arguments. Due to the functional programming standard of Fresh Breeze, there also needs to be an efficient and modular way of returning the return values of a method call. As with method calling protocols in other architectures, when a method is invoked, the processor responsible for running the method will need a clean slate of registers with which to work.

The protocol for calling a method in Fresh Breeze should take advantage of the architecture's unique strengths. The universal memory-addressing scheme allows data objects to be accessed by any thread, as opposed to thread-dependent virtual memory that exists with typical processors. Memory immutability also ensures that no data conflicts will arise by sharing data among multiple threads. Reference count based garbage collection removes the issue of deallocation responsibility when sharing data across threads as well.

Based on these aims, the design for calling a method will be to use a tree of chunks to pass the necessary information, such as arguments and the object the method is being called on, to the called method. The UID of this tree can then be stored in a register for access by the method if called on the same processor, or sent to the hardware scheduler to later be stored in one of the slave thread's registers. Depending on the number of arguments to the function, the data structure will either use just one chunk or a tree of chunks to hold the values. Without loss of generality, this data structure will be considered a tree, similar to the design of the array representation. This tree will contain a pointer to where the Program Counter (PC) should be after the function is finished. This is usually just after where the function was called, but in the case of a spawn instruction where the function was sent to a new thread, this would be simply a null pointer, since the thread should not jump to an instruction location after it is done with its computation. The parent method will need to restore its previous registers before the method call, a pointer to a data structure that stores the old registers for the parent function is required. Lastly, the rest of the tree will be the arguments passed to the

function, those being either literal values or pointers to values or objects. A general notion of what this data structure is can be seen in Figure 16:

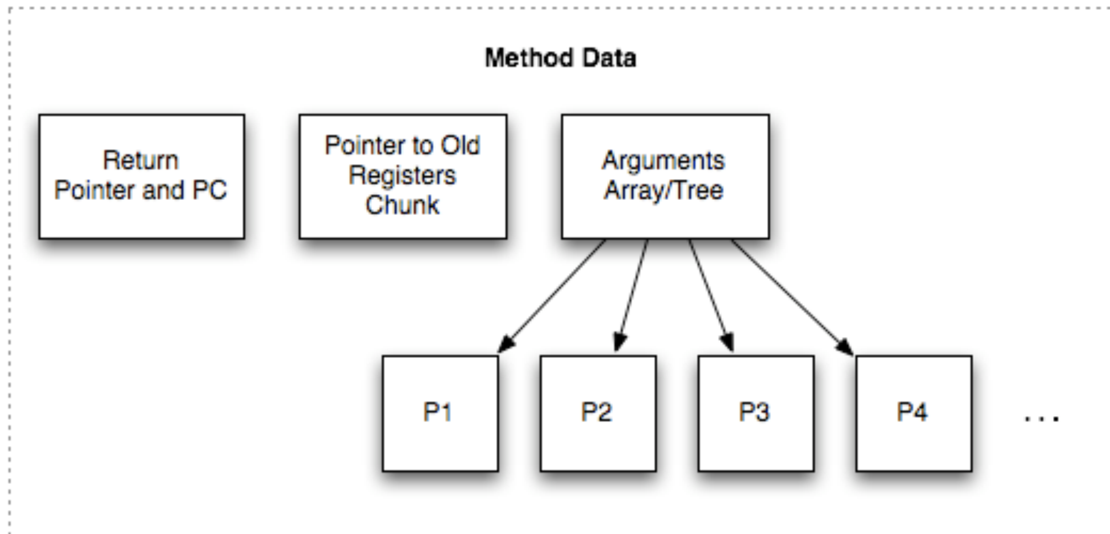

Figure 16

Note that this data structure serves a role equivalent to the role of the stack in other architectures, in that with each method call, all the registers of the previous method are saved, including the register that points to the previous argument data structure. This idea is illustrated in Figure 17.

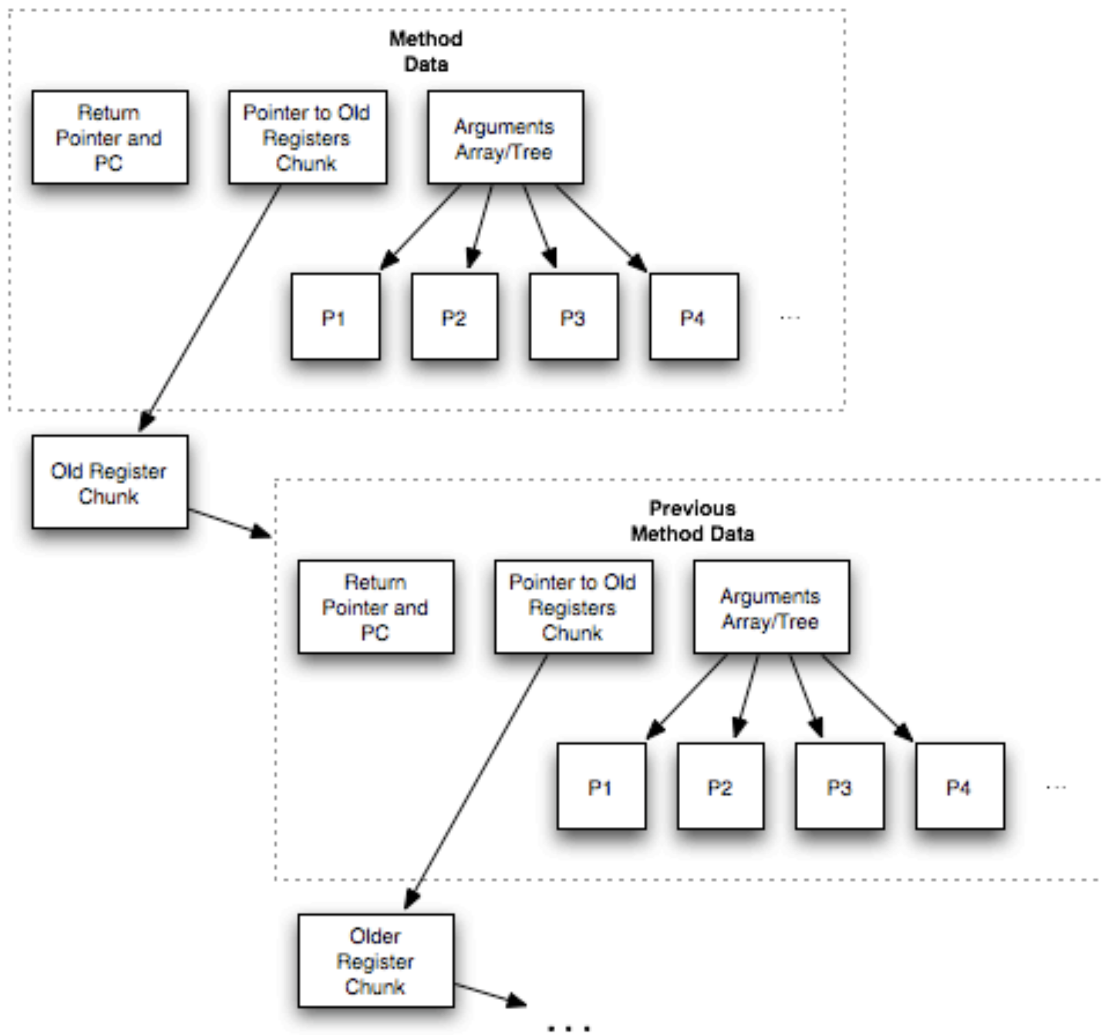

Figure 17

For returning from a method call, one register can be used by convention that will hold a pointer to another tree that will hold the values needed to be returned by the method. This tree will contain the main return value for the method and, in the case of objects and arrays, modified copies of the arguments to the method. Because Functional Java and the memory model dictate that objects cannot be modified once constructed, the data structure also includs a modified copy of the object the method is being called on.

4.1. Implementation Details

4.1.1. Method Call

To call a method, the program begins by allocating a chunk and saving all 32 registers to that chunk. This will be the responsibility of the caller method, thus making this protocol a caller-saved design. This step is only necessary when the called method is not sent to another processor. The program then allocates another chunk and uses it to build a data structure for the parameters that need to be passed to the called method. By convention, a pointer to this data structure will be stored in register 32. The chunk will have its first 8 bytes as the UID of the code chunk that contains the location where the function should return once the method call is finished. Following this, there will be 8 bytes that contain the PC of the return location within the code chunk to which the method will jump to when finished. The next 8 bytes store a pointer to the chunk that is storing the old register values. The remaining 104 bytes of the chunk are devoted to storing the arguments to the method call. If there is not enough room for all the arguments in the 104 bytes, the first 8 bytes will point to another chunk that will contain the necessary arguments. If that is not enough room, then the second 8 bytes will point to another chunk containing arguments, and so on. If there is not enough space for pointers to extra chunks, the first pointer will instead point to a chunk of pointers to chunks of arguments, thus making a tree. From this pattern, a tree can be built that is precisely determined by the number, order, and type of the method's arguments.

The following algorithm precisely describes how this data structure is built:

```
main_chunk[0] = return pointer;
main_chunk[1] = register chunk pointer;
compute how many levels you need;
```

```
store arguments in order such that the fewest arguments
     possible are lower in the tree;

compute how many levels you need:
    if (total size of arguments <= 104 bytes)
        return 0;
    else if (total size of arguments <= (104/8)*128 bytes)
        return 1;
    else if etc...
```

Each argument may have different sizes, so only the bytes that are needed will be used to pass in the argument. For example, an integer would only occupy 4 bytes while a double would occupy 8 bytes. Anything that would require fewer than 4 bytes, such as a boolean, would be stored in 4 bytes. Therefore the arguments section of the chunk will have variable length spots for the arguments. If the method call is an instance method, the first argument will be a pointer to the object the method is being called on. This design is shown in Figures 18 and 19. Figure 18 demonstrates when there is enough space in the chunk to fit all arguments. Figure 19 shows how we start to build a tree if there are too many arguments to fit into one chunk.

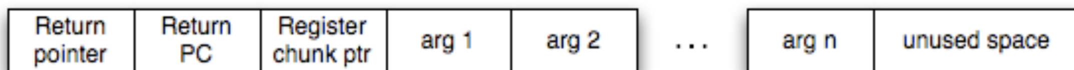

Figure 18

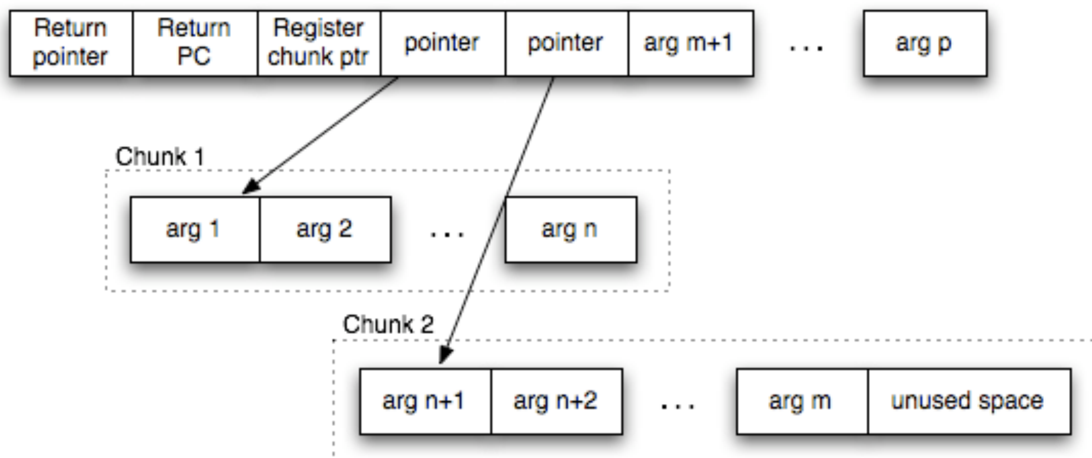

Figure 19

With this design, for any given method signature, there is exactly one data structure that can be built to pass in the arguments to that method. When a tree of chunks is needed to store the arguments, the maximum number of arguments are put higher up in the tree. In Figure 19, chunk 2 is not completely filled with arguments because there is still space in the main chunk where the remaining arguments can fit. Keeping more arguments in the main chunk is more efficient because fewer memory accesses are required than if more arguments were stored in chunk 2 and there was unused space at the end of the main chunk.

If the method call is invoked by a spawn instruction, then the program does the same as shown above except it does not include a register chunk pointer. Instead, it uses the space for a pointer to the "join ticket," a special chunk that contains the UID to the local data segment of the master thread, and the "join point" index in the master thread. The join point is the location in the master thread where the thread will wait for the slave to finish, also providing space for the master thread's status and where the master thread can find the returned data from the slave [5].

In this case, it is not necessary to save the registers for a spawn. Thus only two pointers need to be sent to the processor that is taking over the spawned method call. A pointer to the above data structure is sent along with a pointer to the code chunk of the called method to the hardware scheduler, which determines which processor should handle the spawned method.

4.1.2. Method Return

When a called method wishes to return its return values, it needs to be able to send the return values back to the caller method, including modified copies of the arguments. When a method call is ready to return, it first allocates a chunk and creates a data structure exactly like that in Figures 18 and 19, except it starts the argument list/tree at the beginning of the chunk, eliminating the spots for the return pointer, the return PC, and the register chunk pointer. It fills this data structure with the main return value of the method, then modified copies of the arguments, starting with the object the method was called on if this was an instance method. It then saves a pointer to this chunk in register 31. Finally, it returns to the point denoted by the return pointer and load all the registers stored in the register chunk back to the registers, except for register 31, since that contains the pointer to the return chunk.

If this is a return from a spawned method call, the program includes a pointer to this data structure in the join ticket. The join ticket is sent to the hardware scheduler to wait for the master thread to finish. When the master thread arrives at the join point for the slave thread, it waits in a suspended state, similar to the scheme used in the Cilk-5 multithreaded programming language [6]. It continues to check the hardware scheduler until it sees the join ticket from the slave thread, indicating it has completed. The master

thread then reads the join ticket and loads any return values it needs from the data structure pointed to in the join ticket.

4.2. Method Call Protocol Alternatives

One possible alternative to the above design is instead of using a tree structure for the method arguments, the design could use a linked list of chunks instead. If a method signature had too many arguments to fit into one chunk, the last 8 bytes could be a UID to another chunk containing a list of arguments. Once that chunk was filled, the last 8 bytes would be another UID to another chunk with a list of arguments, and so on. Unfortunately, the problem with this design is if there were too many arguments, the arguments at the end of the list would take several memory lookups to access. Although this would not be a problem with most methods, it is possible to imagine a method that contained a very long list of arguments, in which accessing an element at the end of the list would take $O(n)$ time instead of $O(\log n)$ time, where n represents the number of arguments.

It would also be possible to use a standard Fresh Breeze array, as discussed in previous sections, to hold the necessary information. The main difference between this design and the chosen design would be that the data would be stored only in the leaf nodes rather than on several levels of the tree.

This alternative has both advantages and disadvantages. The main advantage would be that method parameter handling could reuse the same code as array handling. Because the types and number method arguments are always known at compile time, this would be a convenience for the programmer writing the code generator. Depending on the details of this variation, this also may reduce the amount of code generated. If type

data were included in this array, a single section of code could be reused to handle any parameter array, since the size of the tree would be stored with the array.

The main disadvantages of this method is that it would require more memory accesses on average and take slightly more space than the chosen design. Although the tree structure of the chosen design is slightly more complicated, it is not a major problem since the work required to determine the necessary tree structure would be done at compile time as opposed to at runtime.

4.3. Possible Additions

One possible idea would be to add an extra bit to each register in the hardware that would signify whether that register was being used by the caller method. Then when the program saves the registers, it would only need to save those that are being used and could disregard the others. There could be an extra instruction in the Fresh Breeze ISA that would save all registers to a chunk that were marked as being used.

Another topic under consideration is whether the argument tree should be right-heavy rather than left-heavy. This would place the first arguments in the head chunk and the later arguments in the lower chunks, thus building the tree from right to left as opposed to building the tree from left to right, contrasting what the current design is. If more frequently used arguments are placed first, a right-heavy design would be more beneficial to average runtime performance than the current design. However, if more frequently accessed arguments are often at the end of the argument list, the current left-heavy design would yield the best performance.

5. Future Work

Before the compiler for Functional Java to Fresh Breeze machine code can be completed, there is still significant work to be done. The compiler currently is able to translate Java bytecode into optimized data flow graphs (DFGs). The ability to translate DFGs into machine code has not yet been implemented. Along with code generation, future work needs to be completed on topics such as register allocation and method inlining.

A code generator still needs to be implemented for the Fresh Breeze compiler that is able to translate data flow graphs into Fresh Breeze machine code. This could be completed by translating DFGs into machine code directly, or by converting DFGs to an intermediate representation upon which further optimizations could be made.

Register allocation is an area in which there could be interesting future work. Since the Fresh Breeze memory model is significantly different from traditional memory models, there are several new factors that do not exist in most systems that will need to be considered when allocating registers. The division of memory into immutable chunks may influence compiler choices for when data should stay in registers and when it should be moved to main memory. Whereas most architectures have a stack where any amount of space can be allocated for local variables, the fixed-size memory allocation model could require more thought. There may also be instances where addresses of unsealed chunks will need to remain in the registers before they can be stored in memory, and these situations must be identified and handled efficiently.

Due to the functional nature of Fresh Breeze programs and the usage of data flow graphs instead of control flow graphs in the compiler, the problem of determining when methods should be inlined should be explored. As method calling may prove to be expensive, there are possible merits to inlining methods more than in traditional architectures.

6. Conclusion

The work presented in this paper describes the designs for Fresh Breeze arrays and the protocol for calling methods. These are needed before the Functional Java compiler can be completed. We have also evaluated these designs and shown their advantages, disadvantages, and possible future modifications.

As the Fresh Breeze memory model dictates that memory sections (or chunks) be immutable once written, we needed a non-traditional design for arrays. The design was chosen to be a tree of chunks with the size and depth of the tree stored in the root chunk and the array's data stored in the leaf nodes. This provides for $O(\log n)$ running times for array element retrievals and writes. Although this may seem poor, further analysis shows that when using algorithms with mainly sequential element access, this reduces to $O(1)$ amortized time if a pointer trail from the root to the current leaf node is kept in the registers or the local cache.

We also discussed the design for method calling, both in the same thread and with thread spawning. The design was chosen to be efficient with both small and large numbers of arguments. By using a tree structure similar to the one for arrays, we were able to achieve this goal while maintaining simplicity.

In conclusion, these two designs and analyses will help provide a basis for code generation and support the efficiency of the produced code. We expect future work to use these ideas to complete the compiler by implementing code generation and further optimizations.

Appendix A: Array Model Simulation Code

```
package memorymodel;

public class Chunk {

}
```

```
package memorymodel;

public class DataChunk extends Chunk {

	int[] data = new int[32];
	boolean[] canWrite = new boolean[32];

	/**
	 * Constructs a new writable DataChunk with zeros for initial values.
	 * A DataChunk serves as a leaf node in a Fresh Breeze array and thus
	 * stores the actual data values for the array.  The chunk can be
	 * thought of as "unsealed" and a program is able to write to any location
	 * in the chunk.
	 */
	public DataChunk() {
		for (int i = 0; i < canWrite.length; i++) {
			canWrite[i] = true;
		}
	}

	/**
	 * Constructs a new DataChunk that is a shallow copy of c.
	 * This copy is initially completely mutable.
	 * @param c the DataChunk to be copied
	 */
	public DataChunk(DataChunk c) {
		for (int i = 0; i < data.length; i++) {
			data[i] = c.data[i];
		}
		for (int i = 0; i < canWrite.length; i++) {
			canWrite[i] = true;
		}
	}

	/**
	 * Read a 32-bit piece of data from this chunk, returned as a 32-bit integer.
	 * The read location is then "sealed" and can no longer be written to.
	 * @param i the index of the desired element.  For example, if you desire bits
	 * 0 to 31, the index would be 0; 32 to 63, the index would be 1; etc.
	 */
	public int read32(int i) {
		if (i < 0 || i >= 32) {
			throw new RuntimeException("Invalid read32 chunk access " +
						"for index "+i);
		}
		canWrite[i] = false;
```

```java
		return data[i];
	}

	/**
	 * Read a 64-bit piece of data from this chunk, returned as a 64-bit integer.
	 * The read location is then "sealed" and can no longer be written to.
	 * @param i the index of the desired element.  For example, if you desire bits
	 * 0 to 63, the index would be 0; 64 to 127, the index would be 1; etc.
	 */
	public long read64(int i) {
		if (i < 0 || i >= 16) {
			throw new RuntimeException("Invalid read64 chunk access " +
					"for index"+i);
		}
		canWrite[2*i] = false;
		canWrite[2*i+1] = false;
		long temp = data[i*2];
		temp = temp << 32;
		return temp+data[i*2+1];
	}

	/**
	 * Write a 32-bit piece of data to this chunk.  If the location is sealed,
	 * a RuntimeException is thrown.
	 * @param i the index of the element that will be changed, where
	 *   i = (number of data bits before the desired location)/32
	 * @param d the data to be written to that location
	 */
	public void write32(int i, int d) {
		if (i < 0 || i >= 32) {
			throw new RuntimeException("Invalid write32 chunk access " +
					"for index "+i);
		}
		if (!canWrite[i]) {
			throw new RuntimeException("Invalid write32 attempt to a locked "
+"word at index "+i);
		}
		data[i] = d;
	}

	/**
	 * Write a 64-bit piece of data to this chunk.  If the location is sealed,
	 * a RuntimeException is thrown.
	 * @param i the index of the element that will be changed, where
	 *   i = (number of data bits before the desired location)/64
	 * @param d the data to be written to that location
	 */
	public void write64(int i, long d) {
		if (i < 0 || i >= 16) {
			throw new RuntimeException("Invalid write64 chunk access " +
					"for index "+i);
		}
		if (!canWrite[2*i] || !canWrite[2*i+1]) {
			throw new RuntimeException("Invalid write64 attempt to a locked "
+"word at index "+i);
		}
		data[2*i+1] = (int)(0xFFFFFFFF & d);
		long temp = d >> 32;
		data[2*i] = (int)temp;
	}
```

```
}
```

```
package memorymodel;

public class PointerChunk extends Chunk {

	Chunk[] chunks = new Chunk[16];
	boolean[] canWrite = new boolean[16];

	/**
	 * Constructs a new writable PointerChunk with null pointers for
	 * initial values.  A PointerChunk serves as a non-leaf node in a
	 * Fresh Breeze array and thus stores pointers to child nodes.
	 * The chunk can be thought of as "unsealed" once constructed
	 * and a program is able to write to any location in the chunk.
	 */
	public PointerChunk() {
		for (int i = 0; i < 16; i++) {
			canWrite[i] = true;
		}
	}

	/**
	 * Constructs a new PointerChunk that is a shallow copy of c.
	 * This copy is initially completely mutable.
	 * @param c the PointerChunk to be copied
	 */
	public PointerChunk(PointerChunk c) {
		for (int i = 0; i < chunks.length; i++) {
			chunks[i] = c.chunks[i];
		}
		for (int i = 0; i < canWrite.length; i++) {
			canWrite[i] = true;
		}
	}

	/**
	 * Returns a pointer to the Chunk, either another PointerChunk or
	 * a DataChunk, that is stored at the given index.  This location
	 * in the chunk becomes "sealed" and can no longer be written to.
	 * @param i the index of the desired Chunk pointer
	 * @return a pointer to a child Chunk found at index i
	 */
	public Chunk getChunk(int i) {
		if (i < 0 || i >= 16) {
			throw new RuntimeException("Invalid getChunk access " +
						"for index "+i);
		}
		canWrite[i] = false;
		return chunks[i];
	}

	/**
```

```
	 * Write a pointer to Chunk c at index i in this chunk.
	 * If the location is sealed, a RuntimeException is thrown.
	 * @param i the index at which the Chunk pointer will be written
	 * @param c the Chunk pointer to be written
	 */
	public void writeChunk(int i, Chunk c) {
		if (i < 0 || i >= 16) {
			throw new RuntimeException("Invalid writeChunk access " +
					"for index "+i);
		}
		if (!canWrite[i]) {
			throw new RuntimeException("Locked writeChunk slot " +
					"for index "+i);
		}
		chunks[i] = c;
	}

}
```

```
package memorymodel;

public class FBArray {

	private int size;
	private int depth;

	// In this simulation, the root technically has 16 locations
	// for chunk pointers, or 32 32-bit data locations for a data
	// chunk, but in the methods in FBArray we assume that there
	// are only 15 locations for pointers (or 30 locations for
	// integers)
	private Chunk root;

	/**
	 * Constructs an empty Fresh Breeze array with initial size
	 * of 0.
	 */
	public FBArray() {
		root = new DataChunk();
		size = 0;
		depth = 1;
	}

	/**
	 * Constructs a new Fresh Breeze array with the specified size.
	 * The array is initially a sparse array, that is it is not filled
	 * in with initial values.
	 * @param size the desired size of the array, where
	 * bits = size*32.
	 */
	public FBArray(int size) {
		if (size <= 30) {
			root = new DataChunk();
		} else {
			root = new PointerChunk();
		}
```

```
		this.size = size;
		this.depth = 1;
		while (size > 30*(1<<((depth-1)*4))) {
			depth++;
		}
	}

	/**
	 * Reads the element at index "i" in this array and returns
	 * the data as an integer
	 * @param i the index of the desired element
	 * @return the data located at element i as an integer
	 */
	public int read32(int i) {
		if (i < 0 || i >= size) {
			throw new RuntimeException("Invalid array " +
					"access: index="+i+", array size="+size);
		}
		int d = depth;
		Chunk pointer = root;
		while (d > 1) {
			int mask = 0xF << 4*(d-1)+1;
			int n = (i & mask) >> 4*(d-1)+1;
			PointerChunk p = (PointerChunk)pointer;
			pointer = p.getChunk(n);
			if (pointer == null) {
				return 0;
			}
			d--;
		}
		int elt = ((DataChunk)pointer).read32(0x1F & i);
		return elt;
	}

	/**
	 * Reads the element at index "i" in this array and returns
	 * the data as a long
	 * @param i the index of the desired element
	 * @return the data located at element i as a long
	 */
	public long read64(int i) {
		if (i < 0 || i >= size) {
			throw new RuntimeException("Invalid array " +
					"access: index="+i+", array size="+size);
		}
		int d = depth;
		Chunk pointer = root;
		while (d > 1) {
			int mask = 0xF << 4*(d-1);
			int n = (i & mask) >> 4*(d-1);
			PointerChunk p = (PointerChunk)pointer;
			pointer = p.getChunk(n);
			if (pointer == null) {
				return 0;
			}
			d--;
		}
		long elt = ((DataChunk)pointer).read64(0xF & i);
		return elt;
	}
```

```
/**
 * This method changes the 32-bit element located at index
 * "i" in this array to the integer "elt"
 * @param i the index of the desired element to change
 * @param elt the new value for the element at index i
 */
public void changeElt32(int i, int elt) {
	if (i < 0 || i >= size) {
		throw new RuntimeException("Array out of bounds " +
				"access: index="+i+", array size="+size);
	}
	root = changeElt32Help(root,i,depth,elt);
}

/**
 * This method changes the 64-bit element located at index
 * "i" in this array to the long "elt".
 * @param i the index of the desired element to change
 * @param elt the new value for the element at index i
 */
public void changeElt64(int i, long elt) {
	if (i < 0 || i >= size) {
		throw new RuntimeException("Array out of bounds " +
				"access: index="+i+", array size="+size);
	}
	root = changeElt64Help(root,i,depth,elt);
}

private Chunk changeElt32Help(Chunk a, int i, int d, int elt) {
	if (d != 1) {
		assert a instanceof PointerChunk;
		PointerChunk aTemp = (PointerChunk)a;
		int mask = 0xF << 4*(d-1)+1;
		int n = (i & mask) >> 4*(d-1)+1;
		Chunk p1 = aTemp.getChunk(n);
		if (p1 == null) {
			if (d == 2) {
				p1 = new DataChunk();
			} else {
				p1 = new PointerChunk();
			}
		}
		Chunk p2 = changeElt32Help(p1, i, d-1, elt);
		PointerChunk p3 = new PointerChunk(aTemp);
		p3.writeChunk(n, p2);
		return p3;
	} else {
		assert a instanceof DataChunk;
		DataChunk aTemp = (DataChunk)a;
		int n = i & 0x1F;
		DataChunk ptr = new DataChunk(aTemp);
		ptr.write32(n, elt);
		return ptr;
	}
}

private Chunk changeElt64Help(Chunk a, int i, int d, long elt) {
	if (d != 1) {
		assert a instanceof PointerChunk;
```

```
			PointerChunk aTemp = (PointerChunk)a;
			int mask = 0xF << 4*(d-1);
			int n = (i & mask) >> 4*(d-1);
			Chunk p1 = aTemp.getChunk(n);
			if (p1 == null) {
				if (d == 2) {
					p1 = new DataChunk();
				} else {
					p1 = new PointerChunk();
				}
			}
			Chunk p2 = changeElt64Help(p1,i,d-1,elt);
			PointerChunk p3 = new PointerChunk(aTemp);
			p3.writeChunk(n, p2);
			return p3;
		} else {
			assert a instanceof DataChunk;
			DataChunk aTemp = (DataChunk)a;
			int n = i & 0xF;
			DataChunk ptr = new DataChunk(aTemp);
			ptr.write64(n, elt);
			return ptr;
		}
	}

	/**
	 * This method adds the specified element to this array at
	 * index this.size and then increases the array's size by
	 * 1.  This method also increases the array's depth if needed.
	 * @param elt the integer to be added to the end of this array
	 */
	public void addElement32(int elt) {
		if (size+1 <= 30*(1<<((depth-1)*4))) {
			// we don't need to add a level of depth
		} else {
			// we need to add a level of depth
			Chunk temp = root;
			root = new PointerChunk();
			((PointerChunk)root).writeChunk(0, temp);
			depth++;
		}
		Chunk c = addElementHelper32(root,depth,elt);
		size++;
		root = c;
	}

	private Chunk addElementHelper32(Chunk a, int d, int elt) {
		if (d != 1) {
			int mask = 0xF << 4*(d-1)+1;
			int n = (size & mask) >> 4*(d-1)+1;
			Chunk p2;
			PointerChunk p3;
			if (a == null) {
				p2 = addElementHelper32(null, d-1, elt);
				p3 = new PointerChunk();
			} else {
				PointerChunk aTemp = (PointerChunk)a;
				Chunk p1 = aTemp.getChunk(n);
				p2 = addElementHelper32(p1, d-1, elt);
				p3 = new PointerChunk(aTemp);
```

```
            }
            p3.writeChunk(n, p2);
            return p3;
        } else {
            int n = size & 0x1F;
            DataChunk ptr;
            if (a == null) {
                ptr = new DataChunk();
            } else {
                DataChunk aTemp = (DataChunk)a;
                ptr = new DataChunk(aTemp);
            }
            ptr.write32(n, elt);
            return ptr;
        }
}

/**
 * This method takes two Fresh Breeze arrays of equal size and
 * merges them into one array.  The join index is the index
 * where elements in the first array will start to be replaced
 * by elements in the second array.  For example, if you have
 * two arrays: [0, 1, 2, 3, 4, 5] and [10, 11, 12, 13, 14, 15]
 * and a join index of 3, the resulting array would look like
 * this: [0, 1, 2, 13, 14, 15].  This method is basically an
 * optimization designed for algorithms that need to combine
 * two arrays but would like logarithmic performance rather
 * than linear performance.  If the two arrays do not have
 * equal size, a RuntimeException is thrown.
 * @param a1 the first array
 * @param a2 the second array
 * @param joinIndex the index where elements in the first array
 * will start to be replaced by elements in the second array
 * @return a new array that is the joining of both arrays
 */
public static FBArray join32(FBArray a1, FBArray a2, int joinIndex) {
        if (a1.size != a2.size) {
            throw new RuntimeException("Attempting to join two arrays" +
                            " of unequal size: array1.size="+a1.size+", " +
                            "array2.size="+a2.size);
        }
        FBArray joinedArray = new FBArray(a1.size);
        joinedArray.root = joinHelper32(a1.root, a2.root, a1.depth, joinIndex);
        return joinedArray;
}

private static Chunk joinHelper32(Chunk c1, Chunk c2, int d, int joinIndex) {
        if (c1 == null && c2 == null) {
            // there's really nothing to copy, so there's no need to
            // construct empty chunks
            return null;
        }
        if (d != 1) {
            int mask = 0xF << 4*(d-1)+1;
            int n = (joinIndex & mask) >> 4*(d-1)+1;
            PointerChunk curr1 = (PointerChunk)c1;
            PointerChunk curr2 = (PointerChunk)c2;
            Chunk cc1 = null, cc2 = null;
            if (curr1 != null) {
                cc1 = curr1.getChunk(n);
```

```
			}
			if (curr2 != null) {
				cc2 = curr2.getChunk(n);
			}
			Chunk joinedChild = joinHelper32(cc1,cc2,d-1,joinIndex);
			PointerChunk joined = new PointerChunk();
			for (int i = 0; i < 16; i++) {
				if (i == n) {
					joined.writeChunk(n, joinedChild);
				} else if (i < n) {
					if (curr1 == null) {
						joined.writeChunk(i, null);
					} else {
						joined.writeChunk(i, curr1.getChunk(i));
					}
				} else {
					if (curr2 == null) {
						joined.writeChunk(i, null);
					} else {
						joined.writeChunk(i, curr2.getChunk(i));
					}
				}
			}
			return joined;
		} else {
			int n = joinIndex & 0x1F;
			DataChunk curr1 = (DataChunk)c1;
			DataChunk curr2 = (DataChunk)c2;
			DataChunk joined = new DataChunk();
			for (int i = 0; i < 32; i++) {
				if (i < n) {
					if (curr1 == null) {
						joined.write32(i, 0);
					} else {
						joined.write32(i, curr1.read32(i));
					}
				} else {
					if (curr2 == null) {
						joined.write32(i, 0);
					} else {
						joined.write32(i, curr2.read32(i));
					}
				}
			}
			return joined;
		}
	}

	public int size() {
		return size;
	}

	public int depth() {
		return depth;
	}

	public String toString() {
		String s = "[";
		for (int i = 0; i < size; i++) {
			s += read32(i);
```

```
				if (i != size-1){
					s +=",";
				}
			}
			s += "]";
			return s;
		}

	}
```

Appendix B: Sorting Algorithm Code

```
package tests;

import memorymodel.FBArray;

public class MergeSort {

	public static FBArray mergeSort(FBArray a) {
		return mergeSortHelper(a,0,a.size());
	}

	private static FBArray mergeSortHelper(FBArray a, int start, int end) {
		if (end-start <= 16) {
			FBArray tinyArray = new FBArray();
			int[] registers = new int[end-start];
			for (int i = start, j = 0; i < end; i++,j++) {
				registers[j] = a.read32(i);
			}
			int i = 0;
			int j = 0;
			int min = Integer.MAX_VALUE;
			int minIndex = 0;
			while (i < registers.length) {
				while (j < registers.length) {
					if (registers[j] < min) {
						min = registers[j];
						minIndex = j;
					}
					j++;
				}
				int temp = registers[i];
				registers[i] = registers[minIndex];
				registers[minIndex] = temp;
				i++;
				j = i;
				min = Integer.MAX_VALUE;
				minIndex = i;
			}
			for (int k = 0; k < registers.length; k++) {
				tinyArray.addElement32(registers[k]);
			}
			return tinyArray;
		}

		FBArray left = mergeSortHelper(a, start, (start+end)/2);
		FBArray right = mergeSortHelper(a, (start+end)/2, end);
		FBArray newArray = new FBArray();

		int i = 0, j = 0;
		while (i+j < left.size()+right.size()) {
			if (i == left.size()) {
				newArray.addElement32(right.read32(j));
				j++;
			} else if (j == right.size()) {
				newArray.addElement32(left.read32(i));
				i++;
			} else if (left.read32(i) <= right.read32(j)) {
```

```
				newArray.addElement32(left.read32(i));
				i++;
			} else {
				newArray.addElement32(right.read32(j));
				j++;
			}
		}

		return newArray;
	}


}
```

```
package tests;

import java.util.Random;

import memorymodel.FBArray;

public class QuickSort {

	static Random r = new Random();

	public static FBArray quickSort(FBArray a) {
		return quickSortHelper(a, 0, a.size());
	}

	private static FBArray quickSortHelper(FBArray array, int start, int size) {
		if (array.size() <= 16) {
			FBArray newArray = new FBArray(size);
			int[] registers = new int[array.size()];
			for (int i = 0; i < array.size(); i++) {
				registers[i] = array.read32(i);
			}
			int i = 0;
			int j = 0;
			int min = Integer.MAX_VALUE;
			int minIndex = 0;
			while (i < registers.length) {
				while (j < registers.length) {
					if (registers[j] < min) {
						min = registers[j];
						minIndex = j;
					}
					j++;
				}
				int temp = registers[i];
				registers[i] = registers[minIndex];
				registers[minIndex] = temp;
				i++;
				j = i;
				min = Integer.MAX_VALUE;
				minIndex = i;
			}
			for (int l = 0,k = start; l < registers.length; l++,k++) {
				newArray.changeElt32(k, registers[l]);
```

```
            }
            return newArray;
        }

        int pivotPoint = r.nextInt(array.size());
        int pivotValue = array.read32(pivotPoint);
        FBArray left = new FBArray();
        FBArray right = new FBArray();

        for (int i = 0; i < array.size(); i++) {
            if (i == pivotPoint) {
                continue;
            }

            int elt = array.read32(i);
            if (elt <= pivotValue) {
                left.addElement32(elt);
            } else {
                right.addElement32(elt);
            }
        }

        FBArray leftDone = quickSortHelper(left, start, size);
        FBArray rightDone = quickSortHelper(right, start+left.size()+1, size);

        FBArray done = FBArray.join32(leftDone, rightDone, start+left.size());
        done.changeElt32(start+left.size(), pivotValue);

        return done;
    }


}
```

Bibliography

[1] Dennis, J. B. A parallel program execution model supporting modular software construction. In *Third Working Conference on Massively Parallel Programming Models*. IEEE Computer Society, 1998, pp 50-60.

[2] Dennis, J. B. Fresh Breeze: A multiprocessor chip architecture guided by modular programming principles. ACM Sigarch News, March 2003.

[3] Dennis, J. B. Linear Algebra on the Fresh Breeze Chip. Unpublished memo, 2008.

[4] Dennis, J. B. Personal communication.

[5] Dennis, J. B. The Fresh Breeze Model of Thread Execution. *Workshop on Programming Models for Ubiquitous Parallelism*, with PACT-2006, Seattle, September 16, 2006.

[6] Frigo, M., Leiserson, C., and Randall, K. H. The implementation of the Cilk-5 multithreaded language. In Proceedings of the ACM SIGPLAN '98 Conference on Programming language design and Implementation, ACM SIGPLAN Notices 33, 5, May 1998.

[7] Ginzburg, I. *Compiling Array Computations for the Fresh Breeze Parallel Processor*. Thesis for the Master of Engineering degree, MIT Department of Electrical Engineering and Computer Science, May 2007.