Fresh Breeze: A Multiprocessor Chip Architecture Guided by Modular Programming Principles

Jack B. Dennis
MIT Laboratory for Computer Science
Cambridge, MA 02139

Abstract

It is well-known that multiprocessor systems are vastly more difficult to program than systems that support sequential programming models. In a 1998 paper[11] this author argued that six important principles for supporting modular software construction are often violated by the architectures proposed for multiprocessor computer systems. The Fresh Breeze project concerns the architecture and design of a multiprocessor chip that can achieve superior performance while honoring these six principles.

The envisioned multiprocessor chip will incorporate three ideas that are significant departures from mainstream thinking about multiprocessor architecture: (1) Simultaneous multithreading has been shown to have performance advantages relative to contemporary superscalar designs. This advantage can be exploited through use of a programming model that exposes parallelism in the form of multiple threads of computation. (2) The value of a shared address space is widely appreciated. Through the use of 64-bit pointers, the conventional distinction between "memory" and the file system can be abolished. This can provide a superior execution environment in support of program modularity and software reuse, as well as supporting multi-user data protection and security that is consistent with modular software structure. (3) No memory update; cycle-free heap. Data items are created, used, and released, but never modified once created. The allocation, release, and garbage collection of fixed-size chunks of memory will be implemented by efficient hardware mechanisms. A major benefit of this choice is that the multiprocessor cache coherence problem vanishes: any object retrieved from the memory system is immutable. In addition, it is easy to prevent the formation of pointer cycles, simplifying the design of memory management support.

1. Introduction

It is well-known that multiprocessor systems are vastly more difficult to program than systems that support sequential programming models. In a 1998 paper[11] this author argued that six important principles for supporting modular software construction are often violated by the architectures proposed for multiprocessor computer systems. The six principles are:

1. Information Hiding Principle: The user of a module must not need to know anything about the internal mechanism of the module to make effective use of it.
2. Invariant Behavior Principle: The functional behavior of a module must be independent of the site or context from which it is invoked.
3. Data Generality Principle: The interface to a module must be capable of passing any data object an application may require.
4. Secure Arguments Principle: The interface to a module must not allow side-effects on arguments supplied to the interface.
5. Recursive Construction Principle: A program constructed from modules must be usable as a component in building larger programs or modules.
6. System Resource Management Principle: Resource management for program modules must be performed by the computer system and not by individual program modules.

The first two principles have been generally appreciated for many years. For example, the Information Hiding Principle was advocated by David Parnas[26]. The Data Generality Principle is implicit in the goals of the object-oriented programming methodology where any type of object a user wishes to define through a set of methods can be freely passed among program modules. The Recursive Construction Principle simply asks that program modules, once constructed, can be used as components in building higher-level modules — an obviously desirable property but one that is curiously lacking in most programming systems. The importance of the System Resource Management Principle is evident from the ubiquitous presence of virtual memory and paging in most systems intended to support large-scale programs. However, the extension of these ideas to more general global addressing features as in Multics[2] and the IBM AS/400[28] has not been generally adopted.

The continued practice of partitioning data into separate categories of ‘‘memory’’ and ‘‘files’’ seriously impedes the practice of modular programming. It makes the System Resource Management Principle impossible to honor, and constitutes violation of the Data Generality principle because files (as opposed to file names) cannot be used as arguments or results of a program module.

The Secure Arguments Principle[11] is important in computer systems that support concurrent operation of program modules. If two program modules share input data then any change made to the value of the shared argument made by one module will be potentially visible to the other, leading to possible nondeterminate operation and incorrect behavior that could be very difficult to track down. The Secure Arguments Principle requires that this situation not be allowed to happen. In particular, the language in which modules are written should not permit modules to have “side effects” through their input data.

The Fresh Breeze project concerns the architecture and design of a multiprocessor chip that can achieve superior performance while honoring the six principles. As early as 1968[3] it was recognized that properties of a computer or programming system can affect one’s ability to practice modular software construction. One must be able to identify a unit of software that provides well-defined interfaces with other modules and eliminates dependences on the context of use. To satisfy the Resource Management Principle, the system must implement resource management rather than permitting modules to make private resource allocation decisions. To satisfy the Data Generality Principle, there can be no separate file system, and data structures represented in heap storage must have a data type and be able to be passed between program modules. This implies the necessity of a global address space.

In this paper we report on the Fresh Breeze project the author has begun at MIT. Its goal is to demonstrate a realization of this set of principles in the design of a multiprocessor chip for general-purpose computing. To reach this goal, the Fresh Breeze multiprocessor chip incorporates three ideas that are significant departures from conventional thinking about multiprocessor architecture:

Simultaneous multithreading. It has been argued that simultaneous multithreading[29] can yield performance advantages relative to contemporary superscalar designs. This advantage can be exploited through use of a programming model that exposes parallelism in the form of multiple threads of computation. Also, compact ways of encoding dependences among instructions in several closely related threads have been devised in recently reported work[20]. By arranging for a compiler to encode data dependencies between instructions in the machine code, ILP may be exploited without the complex predictive mechanisms of contemporary superscalar microprocessors. The result is that fine-grain parallel execution of several activities can achieve high utilization of processing resources (function units).

Global shared address space. The value of a shared address space is widely appreciated. Through the use of 64-bit pointers, the conventional distinction between ‘‘memory’’ and the file system can be abolished. This will provide a superior execution environment in support of program modularity and software reuse. Some of the benefits of employing shared address spaces have been demonstrated by the Project MAC Multics system[2] and in the success of IBM AS/400 system products[28].

No memory update; cycle-free heap. In a Fresh Breeze computer system data items will be created, used, and released, but never modified once created. The allocation, release, and garbage collection of fixed-size chunks of memory will be implemented by efficient hardware mechanisms. A major benefit of this choice is that the multiprocessor cache coherence problem vanishes: any object retrieved from the memory system is immutable. In addition, it is easy to prevent the formation of pointer cycles, simplifying the design of memory management support.

The following sections introduce our current vision of a Fresh Breeze computer system. After describing the structure of the multiprocessor chip and how it is intended to be used within a larger system, we focus on the memory hierarchy, the nature of the heap implementation, and how it supports scientific computation, using a classical linear system solver as an example. Then the support for stream-based computation is introduced, leading to a discussion of support for transactions on shared data.

2. Architecture

A Fresh Breeze System consists of several Fresh Breeze Processing chips and a shared memory system (**SMS**), as shown in Figure 1. There is also provision for interprocessor communication and message exchange with input-output facilities. Each processing chip includes several (16, say) multithread processors and memory to hold fresh and active data and programs. The architecture of a processing chip is discussed further below.

Memory in a Fresh Breeze system is organized as a collection of *chunks*, each of fixed size. In this paper we assume that a chunk contains 1024 bits and can hold either thirty-two 32-bit words or sixteen 64-bit double words, for example. (A different size may prove a better choice, but

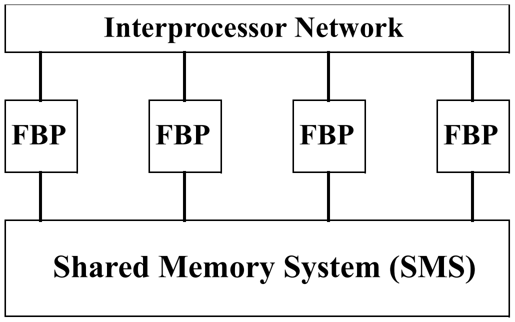

Figure 1. A Fresh Breeze system

we won't know without some testing of the design.) A chunk also includes auxiliary information that indicates the type and format of its contents. Each chunk held in the **SMS** has a unique 64-bit identifier that serves to locate the chunk within the **SMS**, and is a globally valid reference to the data or program object represented by the chunk's contents.

A chunk may contain 64-bit pointers that refer to related chunks, for example, components of a data structure. The rules of operation of a Fresh Breeze system are such that the contents of a chunk are never altered once the chuck has been stored in the **SMS**. (There is one exception to this in the case of a *guard* chunk which is provided in support of transactions processing, as discussed briefly toward the end of this paper.) Moreover, a reference pointer may only be incorporated into a chunk at the time the chunk is first allocated and filled — before its reference count becomes greater than one. It follows that there is no way that a path can be formed that leads from a reference contained in a chunk to the reference identifier of the chunk itself. This property guarantees that the directed graph consisting of the collection of chunks and the references between pairs of chunk will never contain directed cycles.

The **SMS** supports the operations of: obtaining unique identifiers for new chunks; storing a chunk; accessing a chunk from its pointer; and incrementing or decrementing the reference count of a chunk. The **SMS** automatically performs incremental garbage collection of chunks when the reference count of any chunk it holds becomes zero. Note that chunks are always moved between the **SMS** and on-chip memory of the processing chips as complete units, just as cache lines are stored and retrieved in conventional computers.

The Fresh Breeze Processing Chip, shown in Figure 2, includes sixteen multithread processors (**MTP**s) (only four are shown) and additional units that hold on-chip copies of instructions and data. The **MTP**s communicate with eight Instruction Access Units (**IAU**) for access to memory chunks containing instructions, and with eight Data Access

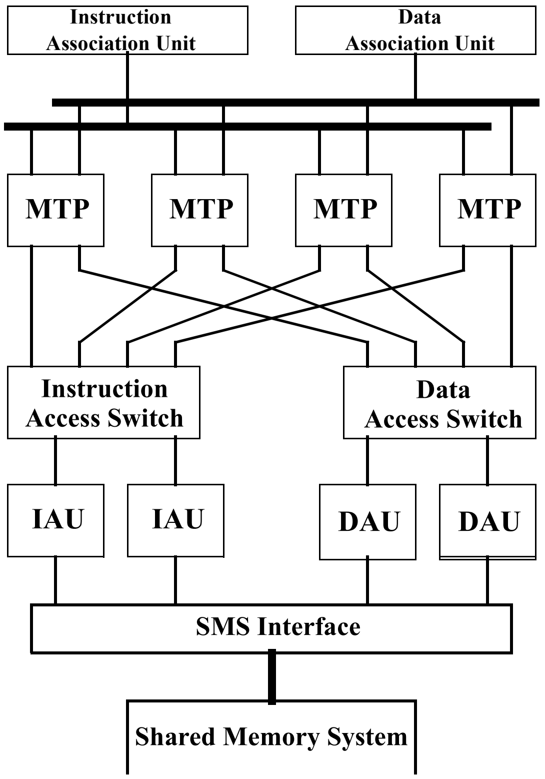

Figure 2. The Fresh Breeze Multiprocessor Chip.

Units (**DAU**) for access to memory chunks containing data. Both kinds of access units have many *slots* that can each hold one chunk of program or data. The *location* of an access unit slot is the combination of the unit number and the slot index within the unit.

Certain registers of the **MTP** can hold pointers used to access data and program chunks. These registers also have an extra field that contains the slot location of the chunk once access to it has been effected and the chunk is present in one of the access units. The Instruction Association Unit (**IAU**) and the Data Association Unit (**DAU**) are augmented associative memories that map 64-bit unique references into chip locations when a chunk is retrieved from the **SMS** by means of its reference identifier. The two association units are related to the TLB (translation lookaside buffer) of a conventional processor, but the contents of a TLB is valid only for the currently executing process, whereas the association units of the Fresh Breeze Processor Chip hold globally valid references and never need to be flushed. The association units also maintain chunk usage data so the "least recently used" criterion may be used to select chunks for purging to the **SMS** when on-chip space is needed for more active programs or data.

The **MTP**s are scalar processors that embody a modified principle of simultaneous multithreading. In brief, each **MTP** supports simultaneous execution of up to four independent *activities*. Each activity presents integer and floating point instructions in a way that allows exploitation of a limited amount of instruction-level parallelism within the activity. Each activity has a private register set for integer, floating point and pointer values, and special registers that hold references to the chunks that hold instructions of the function body being executed by the activity. The integer and floating point function units of the **MTP** are shared by all four activities. An activity is suspended when it encounters a data or program access request that cannot be met immediately from chunks held in the on-chip access units, and also if the activity is waiting for a response message for an input-output transaction. Status information for suspended activities is held in data chunks in the **DAU**s. The Data Association Unit maintains these activity status records in a priority queue as completion signals arrive and assigns them to **MTP**s as activity slots become free.

Execution of a function call initiates a new activity in the body code of the function, creating a *function activation* with a dedicated *function activation record* made up of one or more data chunks. Each function activation executes completely on the processing chip where its initial activity began execution. This execution may spread to several activities running on the same or different **MTP**s on the chip, and may include invocations of functions whose activations are (through remote function call) initiated on other processing chips.

3. Array Data Structures

All data structures used by application programs in a Fresh Breeze system are represented by assemblages of (fixed-size) chunks. Since cycles are not allowed, each such representation must take the form of a DAG (directed acyclic graph). The treatment of arrays will serve to illustrate principles that may be applied to a variety of data structures.

In general, an array is a data structure that represents a mapping of an index set into a set of element values. Often the set of element values is a basic data type such as integers or floating point values represented in a 32 or 64 bit computer word. In a Fresh Breeze system, array elements may also be pointers (references) to objects of any chosen data type. For the Fresh Breeze machine the index set is restricted to non-negative integers less than 2^{32}.

Suppose the elements of an array are 64-bit floating point values. Then 16 element values can be held in one 1024-bit chunk of memory. These must correspond to a contiguous range of index values starting at a multiple of 16. For this format of data in a chunk, auxiliary information consisting of 16 flag bits indicates which of the sixteen value positions hold defined elements of the array.

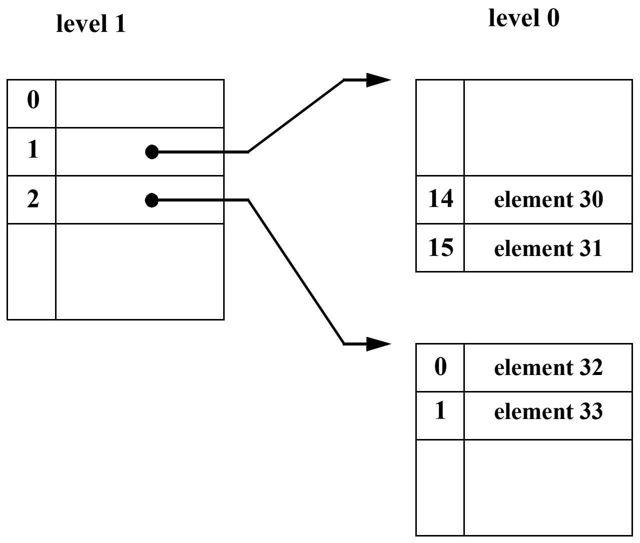

Figure 3. An array with four defined elements.

A large array is represented by a tree of chunks having a depth of as many as eight levels. The number of levels is determined by the largest index value for which the array has a defined element. For example, the tree for an array of 64-bit values with an index set of {0, .. , 255} would have only two levels. Each chunk of the tree is marked with its level (0..7), the leaf nodes that contain element values being at level 0. Each chunk at a higher level in the tree contains up to 16 64-bit pointers to chunks at the next lower level.

Given an array represented by a tree of depth *k*, if there is a chain of pointers leading to a leaf chunk, where at each level *h* the pointer occupies slot $s[h]$, then the element in position $s[0]$ of the leaf chunk corresponds to the index

$$x = \sum_{h=0}^{k-1} s[h] \times 16^h$$

At each level of the tree, flags indicate whether each slot contains a valid pointer. Where there is no valid reference, the array is undefined for index values corresponding to any chain that runs through the invalid slot.

Figure 3 illustrates how three chunks are used to represent an array containing defined 64-bit elements for index values in the set {30,31,32,33}.

To access an array element for a given index value x: Divide the 32-bit binary representation of x into 4-bit sections, and number the eight sections (nibbles) from right to left. This numbering corresponds to the levels of the tree. If the leftmost non-zero section is at level k and the tree has fewer than k levels, the given index x is out of bounds. Suppose the tree has depth k. Then the 4-bit section at position (k - 1) gives the slot containing the pointer to the next chunk on the path to the desired leaf chunk (the leaf chunk itself if k = 1). This procedure is effected in the Fresh Breeze machine by a collaboration between the memory interface unit of the **MTP** executing the activity and the (one or more) **DAU**s that hold chunks that need to be accessed.

Note that large, sparse arrays are represented with reasonable efficiency, at the expense of requiring several accesses to chunks to access any element starting from the root pointer of the array. This latency of access to array elements is accepted in a Fresh Breeze system for the benefit of being able to perform the allocation of new arrays with minimal cost. It is expected that most of this latency will be tolerated through intra- and inter- activity parallelism exploited through simultaneous multithreading.

Arrays are constructed using three basic operations: Create, Augment, and Combine. The `ArrayCreate` operation allocates a chunk (in a **DAU**) and initializes it with up to 16 array elements. This may be done as a single operation in which element values from a contiguous group of processor registers are transferred. If the index range of the array elements is such that a pointer chunk is required, then a second chunk is automatically allocated and initialized with a pointer to the leaf chunk. This is continued to yield a tree (chain) of chunks of the necessary depth.

The `ArrayAugment` operation may be used to fill in undefined elements of a leaf chunk. This is permitted as an "update-in-place" if the reference count of the chunk is one and the chunk has not been stored in the **SMS**. Otherwise, either the array has already been made available to other activities or it has been saved in the **SMS**, and the chunk must not be altered. In this case a new chunk is allocated, the elements of the given array are copied, and the update-in-place is performed on the copy. Note that a chunk will be sent to the **SMS** only if space is needed because on-chip memory is exhausted, or there is a request for access to the data from an activity in another chip.

Arrays represented by more than one leaf chunk are constructed by creating several partial arrays with non-overlapping index ranges, and combining them using the `ArrayCombine` operation. The `ArrayCombine` operation is given two arrays each represented by a tree of chunks. Assume first they have non-overlapping index ranges and every chunk in both trees has a reference count of one. The result of the operation is a pointer to the root chunk of an array representation that incorporates all elements of both given arrays. No new chunk needs to be allocated unless the combined index range demands an additional level and a new root chunk in the tree. One of the given arrays is chosen as the *basis* for forming the combined array. Any subtree of this tree that has an index range not intersecting with the index set of the second array is left intact. For a subtree where there is a non-empty intersection of index sets, the chunks of the basis tree must be updated with elements and/or pointers from the second tree. Note that the chunks of the second array need not have a reference count of one for this procedure to work correctly. If such a chunk has a reference count of one, and its contents are copied into a corresponding basis chunk, the reference count will be decremented making the chunk free. Otherwise the chunk will be retained because access to it is retained by another data structure or activity.

When each chunk is allocated during construction of an array, it is assigned a 64-bit reference identifier supplied by the **SMS** from its pool of free references.

For accessing array elements, the `ArrayRead` operation takes a pointer to an array (tree of chunks) and an index range and transfers the element values to specified processor registers. The chunks along the path to the required leaf chunk are accessed, including retrieval from the **SMS** to on-chip **DAU**s as necessary.

The array Create, Augment, Combine, and Read operations will be implemented by processor instructions that direct action by the **DAU**s. Because these operations may require multiple steps, they will likely be supported in a way that uses several machine instructions, perhaps including a repeat mechanism. The details of these operations are under development.

The processing of a large array may spread over many processing chips of a Fresh Breeze system. Construction of an array may be initiated by an activity that invokes function instantiations on remote processing chips corresponding to some partitioning of the work, either determined automatically by a compiler or through detailed coding by a programmer. An array may be passed as an argument of a function call executed on a remote processing chip. When an activity executing the function code attempts to access parts of the array, these may not yet have been sent to the **SMS**. In this case, the **SMS** will act on the access request by commanding the processor that created each requested part of the array to send the corresponding chunk to the **SMS** so it may be forwarded to the processing chip that needs it.

4. Example: A Linear System Solver

One popular way of solving systems of linear algebraic equations is the method based on lower-upper decomposition as implemented in the Linpack BLAS[16]. This algorithm has long been used as a benchmark for comparing the performance of high-performance computer systems. Here we discuss implementation of this solver on a Fresh Breeze system for high performance.

For our purposes it is convenient to discuss the algorithm as written in the Java programming language[18].

This is shown in Figure 5 where, for simplicity, we have omitted the code that implements "partial pivoting". The class `LinearSystemSolver` contains four methods, one of which is public and is the entry point for users. The arguments of `Solve` are `n` the dimension of the equation system and `A` a n by n+1 matrix that contains the coefficient array in columns 0 through n-1, and the right hand side vector as column n. The algorithm consists of two parts: the "forward elimination" phase coded in method `Forward`, and the "back substitution" phase coded in method `Backward`. Because the principal computation effort occurs in the construction of the new array `An` at each of n steps (this is what gives the algorithm its n^3 complexity), we focus our attention on this part of the program. The elimination steps are performed by successive recursive calls of the method `ForwardStep`. In the body of this method the following lines capture the major computational effort, where k is the step index starting from 0:

```
for (int i = k+1; i < n; i++) {
    temp = A[i][k] / A[k][k];
    for (int j = k+1; j < n+1; j++) {
        An[i][j] = An[i][j] -
            temp * A[i][j];
    }
}
```

The body statement of the nested loop

```
An[i][j] = An[i][j] - temp * A[i][j];
```

is to be executed (n-k) * (n-k+1) times, and all of these can be done concurrently, offering a high degree of parallelism. Figure 4 illustrates the forward elimination step. A compiler for a Fresh Breeze computer can easily recognize this and generate code that creates a separate function instance for each tile of the (n-k) by (n-k+1) index space, where the size of the tiles is chosen as a judicious trade-off governed by the size of the machine and the expected sizes of linear systems to be solved. In this case, how execution of these function instances is distributed over the processing chips of a Fresh Breeze system is not crucially important because the effort required to move data around becomes asymptotically small in comparison with the floating point operations required, as larger matrix sizes are handled. During execution, chunks for each new array `An` will be allocated almost instantly, and chunks containing pieces of earlier `An` instances automatically become free and immediately reusable once all accesses to them have been completed.

5. Streams and Transactions

Arrays and records are the basic data structures provided by many programming languages. A record type is a class of objects, usually of fixed size, that have several fields that may be objects of different types, including references to arrays, other records, or arbitrary objects. Just as records are often implemented in a sequence of computer words in conventional computers, records can be implemented as an array of 32-bit or 64-bit elements in a Fresh Breeze system, so they will not be discussed further here.

A kind of structured data that is increasingly recognized as important in the construction of modular software is a *stream*. A stream is an unending series of values of uniform type that may be passed from one software module to another. Using stream data types, many signal processing and other applications have elegant structure as a straightforward composition of program modules[10][21].

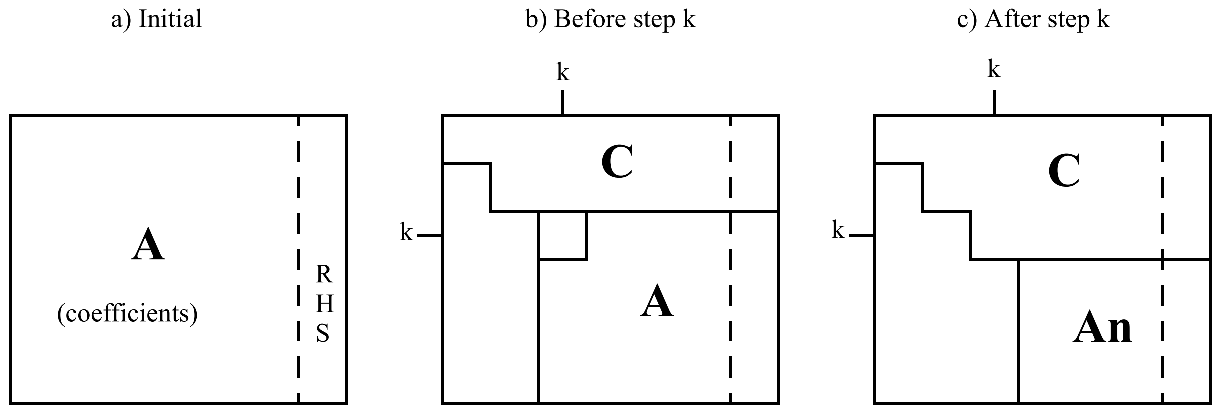

Figure 4. Forward elimination step of the liner system solver.

```
class LinearSystemSolver {
  int n;
  class Row {
    double[] elmnts;
  }

  Row[] C;

  public double[] Solve (int n, double[][] A) {
    double[] X;
    this.n = n;
    C = Forward (A);
    X = Backward (C);
    return X;
  }

  private Row[] Forward (double[][] A) {
    C = new Row[n];
    return ForwardStep (0, A, C);
  }

  private Row[] ForwardStep (
      int k,
      double[][] A,
      Row[] C) {
    double temp, factor;
     double[][] An = new double[n-k][n-k+1];
    C[k].elmnts = new double[n+1];
    // Pivot row
    factor = 1.0 / A[k][k];
    for (int j = k+1; j < n+1; j++) {
      C[k].elmnts[j] = factor * A[k][j];
    }
    // Non pivot rows
    for (int i = k+1; i < n; i++) {
      temp = A[i][k] / A[k][k];
      for (int j = k+1; j < n+1; j++) {
        An[i][j] = An[i][j] - temp * A[i][j];
      }
    }
    // Terminate or Continue
    if (k == n-1) {
      return C;
    } else {
      return ForwardStep (k+1, An, C);
    }
  }

  private double[] Backward (Row[] C) {
    double[] X = new double[n];
    for (int k = n-1; k >= 0; k--) {
      double S = 0.0;
      for (int j = k+1; j <= n-1; j++) {
        S = S + C[k].elmnts[j] * X[j];
      }
      X[k] = C[k].elmnts[n] - S / C[k].elmnts[k];
    }
    return X;
  }

}
```

Figure 5. The linear system solver written in Java.

In a Fresh Breeze system, a stream is naturally represented by a sequence of chunks, each chunk containing a group of stream elements. Either each such chunk may include a reference to the chunk holding the next group of stream elements, or auxiliary chunks containing “current” and “next” references may be used to organize the chunks that hold stream elements. The operations `StreamEnter` and `StreamRemove` are supported, where, in particular, the `StreamRemove` operation causes the executing activity to suspend if the stream contains no elements, and to resume just when elements are entered in the stream.

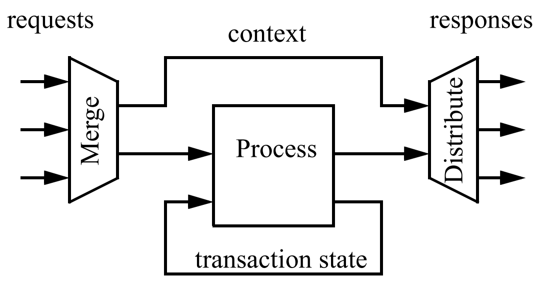

Figure 6. A view of transaction processing.

For the `StreamEnter` operation to work properly with this choice of representation for streams, it must be possible to fill in a slot (the “next” reference) perhaps long after the chunk was allocated and filled with stream elements. To guarantee that a cycle cannot be created, the stream chunk is marked as being of a special *stream* type, and can only be referenced by the “next” reference in another chunk of stream type. This sort of mechanism is similar to that of “incremental arrays” (“I-structures”)[1][7].

In earlier work [13] it was noted that performing asynchronous transactions on an object such as a “shared file” may be viewed as happening in two phases as illustrated in Figure [6]. In the first phase, transaction requests arriving asynchronously are merged into a single (ordered) stream of requests. (Each request includes context information needed to reply to the initiating activity.) This stream of requests is then applied to the subject entity to produce a stream of responses which are distributed to the requesting activities. The processing of the ordered stream of requests is a purely functional computation that manipulates the subject entity as an internal value. (We have noted that lots of potential concurrency may be exploited within this functional computation, including the overlapped execution of multiple requests.) Thus the novel aspect introduced to perform transactions is supporting the (nondeterminate) merge of asynchronous requests.

In[12] it was shown how this may be accomplished by using a special kind of memory object (chunk) that we have called a *guard*. In essence, the guard holds a reference to the tail element of the (ordered) stream of transaction requests. An activity wishing to enter a transaction request forms a request stream element with an empty “next” element field. It then performs a swap operation that substitutes the pointer to its stream element (the new tail element) for the current pointer held by the guard. Finally it installs the pointer of the (old) tail in the “next” field of the new stream element.

In the Fresh Breeze machine, a guard is a special kind of memory element (chunk) that exists only in the **SMS**, thereby avoiding problems associated with cached copies. The instruction `CreateGaurd` is passed directly to the SMS and returns a reference to a unique guard chunk. The `GuardSwap` instruction is also passed directly to the **SMS** where it is forwarded to the component of the **SMS** where the guard was allocated. With this arrangement, `GuardSwap` instructions executed by independent activities on any processing chips can be processed by the **SMS** component in immediate succession; the rate at which transactions can be processed may be as fast as the **SMS** unit can process successive swap instructions, and is not limited by the speed of any of the **MTP**s.

6. Conclusion

Some of the principles and architectural concepts guiding the Fresh Breeze Project have been presented. The use of a cycle-free heap of fixed-size chunks to support general array data structures has been explained and illustrated using a linear system solver.

Some interesting additional features are envisioned for a Fresh Breeze computer system: A *job* is a collection (tree) of function activations that are operating on behalf of one user of a Fresh Breeze system. Each job may have a monitoring activity (part of a superior job) that is normally dormant, but is activated upon job termination or upon the occurrence of exceptional conditions during job execution.

Note that an activity can only access data that it creates, or data that it is given access to through a reference argument of its function instance. Many jobs may be running on a Fresh Breeze System at the same time with no danger that activities of one job can affect data of any other job except where several jobs share access to objects by means of transactions executed through a guard. Input/Output transactions in a Fresh Breeze System are performed by messages sent by activities using unique identifiers for peripheral controllers or devices (capabilities[15][17]) provided to the job by its superior job. These features will permit a high degree of security to be realized.

The Fresh Breeze project is a continuation of work begun many years ago inspired by the potential benefits of recognizing program structure in study of computer architecture[4][15][5][6][8]. This work also builds on the efforts of several other projects[19][24][23].

The Fresh Breeze Project is in an early stage. The first steps are to complete the architectural specification and develop a detailed instruction set. Concurrently, a cycle-accurate simulator of a Fresh Breeze System is being developed using the Java programming language. An important next step is enabling the preparation and evaluation of application programs to demonstrate the merit of architectural choices. For this purpose it is planned to use a restricted dialect of Java and a special back-end processor that converts Java class (bytecode) files into Fresh Breeze machine code.

References

[1] Arvind and R. S. Nikhil and Keshav Pingali. I-structures: data structures for parallel computing. *ACM Transactions on Programming Languages and Systems 11*, 4:598-632, October 1989.

[2] A. Bensoussan, C. T. Clingen, and R. C. Daley. The Multics virtual memory. In *Proceedings of the Second Symposium on Operating System Principles*. ACM, October 1969, pp 30-42.

[3] William E. Boebert. Toward a modular programming system. In *Proceedings of a National Symposium: Modular Programming*. Cambridge, MA: Information Systems Press, 1968.

[4] Jack B. Dennis. Segmentation and the design of multiprogrammed computer systems. *Journal of the ACM 12*, 4:589-602, October 1965.

[5] Jack B. Dennis. Programming generality, parallelism, and computer architecture. In *Information Processing 68*. Amsterdam: North-Holland, 1969, pp 484-492.

[6] Jack B. Dennis. First version of a data flow procedure language. In *Lecture Notes in Computer Science, Volume 19: Programming Symposium*, B. Robinet, Ed. Springer-Verlag, 1974, pp 362-376.

[7] Jack B. Dennis. An operational semantics for a language with early completion data structures. In *Lecture Notes in Computer Science, Volume 107: Formal Description of Programming Concepts*. Berlin: Springer-Verlag, 1981, pp 260-267.

[8] Jack B. Dennis. The evolution of "static" data-flow architecture. In *Advanced Topics in Data-Flow Computing*, Jean-Luc Gaudiot and Lubomir Bic, eds., chapter 2, Prentice-Hall, 1991.

[9] Jack B. Dennis. Machines and models for parallel computing. *International Journal of Parallel Programming 22*, 1:47-77, February 1994.

[10] Jack B. Dennis. Stream data types for signal processing. In *Advances in Dataflow Architecture and Multithreading*, J.-L. Gaudiot and L. Bic, eds. IEEE Computer Society, 1995.

[11] Jack B. Dennis. 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.

[12] Jack B. Dennis. General parallel computation can be performed with a cycle-free heap. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compiling Techniques*. IEEE Computer Society, 1998, pp 96-103.

[13] Jack B. Dennis. A language design for structured concurrency. In *Lecture Notes in Computer Science, Volume 54: Design and Implementation of Programming Languages*. Berlin: Springer-Verlag, 1977, pages 231-242.

[14] Jack B. Dennis and Guang R. Gao. Multithreaded architectures: principles, projects, and issues. In Robert A. Ianucci, editor, *Advances in Multithreaded Computer Architecture*. Kluwer, 1994.

[15] Jack B. Dennis and Earl C. van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM 9*, 3:143-155, March 1966.

[16] J. Dongarra, J. Bunch, C. Moler and G. W. Stewart. *LINPACK Users Guide*, SIAM, Philadelphia, PA, 1979.

[17] Robert S. Fabry. Capability-based addressing. *Communications of the ACM 17*, 7:403-412, July 1974.

[18] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[19] Kei Hiraki, Satoshi Sekiguchi, and Toshio Shimada. Status report of SIGMA-1: A data-flow supercomputer. In *Advanced Topics in Data-FlowComputing*, Jean-Luc Gaudiot and Lubomir Bic, eds., chapter~7. Prentice-Hall, 1991.

[20] Ho-Seop Kim and James E. Smith. An Instruction Set and microarchitecture for instruction level distributed processing. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*. IEEE Computer Society, 2002, pp 71-82.

[21] Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE 75*, 9, September 1987.

[22] Barbara Liskov. Abstraction mechanisms in CLU. *Communications of the ACM 20*, 8:564-576, August 1977.

[23] 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: Reference Manual Version 1.2*. Technical Report M-146, Rev. 1. Lawrence Livermore National Laboratory, 1985.

[24] Gregory M. Papadopoulos and David E. Culler. Monsoon: an explicit token-store architecture. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, IEEE Computer Society, 1990, pp 82-91.

[25] Gregory M. Papadopoulos and Kenneth R. Traub. Multithreading: A revisionist view of dataflow architectures. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*. IEEE Computer Society, 1991, pp 342-351.

[26] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM 15*, 12:1053--1058, December 1972.

[27] Gurindar S. Sohi, Scott Breach, and T. N. Vijaykumar. Multiscalar Processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*. IEEE Computer Society, 1995, pp 414-425.

[28] Frank G. Soltis. *Inside the AS/400*. Loveland, Colorado: Duke Press, 1996.

[29] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous Multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, IEEE Computer Society, 1995, pp 392-403.