The Fresh Breeze Model of Thread Execution

Jack B. Dennis
MIT Computer Science and Artificial
Intelligence Laboratory
Cambridge, MA
617-253-6956

dennis@csail.mit.edu

ABSTRACT

We present the program execution model developed for the Fresh Breeze Project, which has the goal of developing a multi-core chip architecture that supports a better programming model for parallel computing. The model combines the spawn/sync ideas of Cilk with a restricted memory model based on chunks of memory that can be written only while not shared. The result is a multi-thread program execution model in which determinate behavior may be guaranteed while general forms of parallel computation are supported.

Categories and Subject Descriptors

C.0 **[Computer Systems Organization]:** Hardware/software interfaces; D.1.1 [**Programming Languages**]: Applicative (Functional) Programming; D.1.3 [**Programming Languages**]: Concurrent Programming; D.3.3 [**Programming Languages**]: Language Constructs and Features; E.1 [**Data**]: Data Structures.

General Terms

Design, Languages.

Keywords

Program execution model, concurrency, multithreading, determinacy, streams, transactions.

1. INTRODUCTION

With multiprocessing becoming the usual mode of computing, it is more pressing than ever to adopt a model of program execution that supports general parallel programming while avoiding the programing difficulties attending the coordination of concurrent computations in contemporary systems.

This presentation concerns the program execution model developed for the Fresh Breeze Project[8][6], which has the goal of developing a multi-core chip architecture that supports a better programming model for parallel computing, achieving high performance through parallelism rather than increased clock speed.

The Fresh Breeze architecture explores a set of ideas that depart from conventional computer architecture:

Simultaneous multithreading can improve the utilization of function units and allow more effective latency tolerance of memory transactions[22].

A global shared address space permits abolition of the conventional distinction between "memory" and the file system, and supports a superior execution environment meeting requirements of program modularity and software reuse[3][5]

No memory update of shared data in a Fresh Breeze computer system eliminates the multiprocessor cache coherence problem: any object retrieved from the memory system is immutable.

Cycle-free heap. The no-update rule In a Fresh Breeze computer system makes it easy to prevent the formation of pointer cycles[7]. Efficient reference-count garbage collection can be implemented in the hardware.

We will show how a program execution model based on these ideas can provide a platform for robust, general-purpose, parallel computation. In particular, much of the difficulty of writing correct parallel programs stems from problems in getting synchronization correct. The commonly used methods of thread coordination are prone to exhibiting nondeterminate behavior, making errors difficult to track down. Moreover, getting control synchronization correct is no guarantee that accesses to data will be effected without hazard, especially when memory access has significant latency and memory is shared among concurrently executing threads.

Our hypothesis is that handling control and data coordination separately is a bad idea. Here we explore an approach in which data and control synchronization are always performed together, as a single atomic operation. This is offered as an alternative to transactional memory approaches that add complexity, often including redundant computation and retry mechanisms.

Principle: Combine control and data synchronization in the design of mechanisms for coordinating concurrent activities.

We begin with an introduction to the Fresh Breeze memory model. The program execution structure of threads of control and method activations is described. Then we present the basic concurrent programming structure of this paper, an adaptation of the classic fork/join control primitives for concurrent processes. We argue that use of this program structure yields multi-thread programs

that are guaranteed to be determinate. Subsequent sections deal with generalization and extension of the basic concept to programs that operate on data structures and data streams. How a Fresh Breeze system can be used to run inherently nondeterminate computations such as general transaction processing is discussed in the final section.

2. FRESH BREEZE MEMORY MODEL

The Fresh Breeze architecture views memory as a collection of fixed-size chunks, each holding 1024 bits of code or data. A chunk may hold 32 words of 32 bits each, 16 double length words, or 32 instructions of a program method. Each chunk has a unique 64-bit identifier (UID) that serves as a pointer for accessing the chunk. A chunk may also hold up to 16 UIDs to represent structured information. For example, a three-level tree of chunks could represent an array of 16 * 16 * 32 elements.

The collection of chunks containing pointers to other chunks forms a *heap* which may be regarded as a graph. The Fresh Breeze program execution model is designed so that cycles in the heap cannot be constructed. This is ensured by the rule that updates to the contents of chunks are disallowed once they are shared, and marking pointers to chunks in process of definition (creation) so that a thread may not store them in any chunk that is under construction.

In a Fresh Breeze system all on-line data, including files and databases, are held in the chunk memory as data structures in the heap.

In the following, we will show how this memory model can support a variety of program constructions for parallel computation.

3. PROGRAM EXECUTION

Computation in a Fresh Breeze system is carried out by method activations on behalf of a system user. Computation in a method activation is performed by exactly one thread of instruction execution. The thread has access to the code segment of the method, a local data segment, and a set of registers that hold intermediate results. The code segment is read-only and may be shared by many threads for different method activations. The registers and local data segment are private data of the thread and are only accessible to it.

Note that in this model of computation, a thread (method activation) can only access information in the heap that it creates or that is given to it when it begins execution.

4. FORMS OF PARALLELISM

Functional programming languages are known to offer better programmer productivity due to their freedom from side-effects and the mathematical properties of expressions. Another benefit of functional programming languages is the ease of identifying parts of computations that may be executed concurrently. In particular, several important forms of concurrency are naturally represented in functional programs:

Expressions. Subexpressions may always be evaluated concurrently with other subexpressions.

Functions. Function applications may be evaluated concurrently if none uses the results of others.

Data Parallel. The general expression for an array element may be evaluated concurrently over all elements of the array.

Producer/Consumer. If one program module passes a series of values to a second module, then both modules may run concurrently.

Cascaded Data Structure Construction. This is a kind of generalization of producer/consumer concurrency and will be treated in a later section.

We will show how our thread coordination model supports each of these forms of parallelism.

5. THE BASIC SCHEME

Our basic program structure for concurrency is an adaptation of the fork/join model and is similar to the concurrency scheme of Cilk[14]. A *master thread* spawns a *slave thread* (executing a new function activation) that performs an independent computation and terminates by joining with the master thread. The problem to be solved is that of performing the join in a safe manner -- in a way that does not introduce the possibility of hazards. For the present we assume that the slave computation is contributing a single value (one computer word) to the computation being performed by the master thread.

Because, in our model, there is no shared data between the master and slave threads, updating shared data cannot be used to communicate the slave result to the master. (This is good because a shared data mechanism would likely introduce possible hazards.)

Our scheme is the following: When the master spawns the slave thread, it initializes a *join point* and provides the slave thread with a *join ticket* that permits it to exercise (use) the join point. The join point is a special entry in the local data segment of the master thread that has (1) space for a record of master thread status; and (2) space for the result value that will be computed by the slave thread. The join point includes a flag that indicates whether a value has been entered by the slave thread, and a flag that indicates that a join ticket has been issued.

A join ticket is similar to the return address of a method, and is held at a protected location in the local data segment of the slave method activation. It consists of (1) the unique identifier (pointer) of the local data segment of the master thread; and (2) the index of the join point within the master thread local data segment.

Special instructions are provided to access a join point. The instruction. The **Spawn** instruction sets a flag in the join point and starts execution by the slave thread after storing a joint ticket in its local data segment. The **EnterJoinResult** instruction is used by the slave thread to enter its result in the joint point identified by its join ticket. Execution of the **EnterJoinResult** instruction cause the slave thread to quit. The master thread may only read the join value by using the **ReadJoinValue** instruction which returns the join value if it is available, or suspends the master thread if the join value has not yet been entered. Note that the **EnterJoinResult**

instruction must resume the master thread if it finds that the master thread has been suspended.

All of these instructions are "architected"; that is, they are implemented by the Fresh Breeze hardware so that their rules of use cannot be violated. In particular, information in the join point or join ticket cannot be read or written by ordinary instructions.

6. DETERMINACY

Determinacy is the property of a system in which observable results are independent of permitted orders of internal events[17]. In a Fresh Breeze system, what is observable is the sequences of output values produced by threads. In a Fresh Breeze system each thread is sequential and operates on private data and fixed input data. Any heap operations performed by a thread either read fixed data or create new private data. Hence, by itself, operation of each single thread is determinate. Operations at a join point have been defined so that the result is independent of the order of thread arrival. Therefore, a combination of master and slave threads operating as we have specified has determinate behavior. Note that an error in program construction, such as an attempt by the master thread to issue a join ticket twice for the same join point, will cause a thread to hang.

It should be noted that this guarantee of determinacy requires that no more than one slave thread be given a join ticket to the same join point. After all, if two asynchronous threads can arrive at the join point in either order, and the join point accepts the first to arrive, then a hazard will exist. This is avoided by having the spawn instruction set the join point flag when it issues a join ticket, and making it an error to attempt use of a marked join point.

7. GENERALIZATION

The basic scheme we have described can be the basis for constructing highly concurrent programs. The slave thread may act as master for another slave, etc. yielding a team of threads working together to produce a single result. The original master thread may initiate several slave subcomputations each with its own join point. However, the basic scheme does not provide for concurrent computations where the ultimate result is more than a single value. This limitation is overcome by the extensions discussed next.

8. EXTENSIONS

Two extensions to the basic scheme are combining operations and array construction. In a combining operation the combining operator (the *join operator*) must be associative and commutative. Such an operator may be used to combine a finite collection of result values yielding a final result that is independent of the order in which pairs of values are combined using the operator. To implement this using a collection of *n* slave threads, the join point is initialized with two counters set to the same integer *n*. The *issue counter* limits the number of slave threads that may be spawned. the *join counter* tallies the slave threads as each contributes its result value to an accumulator in the join point using the join operator. When exactly *n* threads have used their join tickets to contribute to the join point, the master thread is permitted to continue, using the final accumulated value in its further computation. It should be clear that this arrangement of threads also performs a determinate computation, by the same reasoning used for the basic scheme.

The second extension uses a fixed number of slave threads to generate components of a data structure. In the Fresh Breeze machine, it is natural to associate a memory chunk (the *join chunk*) with the join point, and employ one slave thread to compute each of its components, either 32 scalar values or 16 pointers to arbitrary data structures. In this case we must be certain that each thread contributes the component at a specified index in the join chunk. This is done by giving each slave thread a join ticket that carries the index of its assigned component. Each spawn operation gives a slave thread a join ticket containing an index from the issue counter of the join point, and then increments the counter. As before, the number of slave threads is limited to the preset value of the join counter. When each slave thread exercises the join point, the value or pointer to a data structure it has created is installed in the join chunk at the index position given by its join ticket. This arrangement supports a general form of data parallel computation and includes the guarantee of determinacy.

9. CASCADED DATA STRUCTURES

A *future* is a special kind of value that can be a component of an array or data structure[2][16]. A future is an indication that a genuine data value will appear in its place at some future time. A future is similar to a join point in that it will receive a value when some thread provides it. It is different from a join point in that it is an element of a data structure instead of being a special entry in the local data segment of a thread. What is neat about futures is that they permit a thread to pass a data structure it is generating to a second thread that uses the data structure before the data structure is completely generated.

A future has three states: **undefined**, **defined**, and **waiting**. Two operations are used with futures: **Write** and **Read**. The **Write** operation creates a future (as part of a data structure the thread is building). The future begins life **undefined** and empty. The *producer thread* that executes the **Write** operation may pass the structure containing the future to a *consumer thread* of the data structure. Either of two events may follow. (1) The *producer thread* finishes computing a value and places it in the future, changing its state to **defined**; if the state was **waiting** it also resumes the *consumer thread* which can then make use of the value; or (2) the *consumer thread* performs a Read operation on the future. If the future is **undefined**, the consumer thread is suspended and a pointer to its status record is placed in the future, which changes to **waiting** status; if the future is **defined** the consumer thread continues with the value.

This mechanism supports the concurrent processing of a cascade of data structure transformations, as might occur in a programming language compiler. It can also be used in the

functional processing of database transactions as described in a later section.

10. STREAMS

Concurrent computations in the form of modules that communicate by passing streams of messages from one to another are important in the expression of signal processing and other computations[10][9][18]. These are naturally expressible in functional programming languages that include stream data types[20]. In the Fresh Breeze memory model a stream can have a representation that is a chain of chunks where the chunk at the beginning of the chain holds the next elements to be read from the stream, and the end of the chain is a future waiting to receive further elements yet to be computed. This arrangement of using futures for the implementation of streams again has the property of determinacy. Any assembly of Fresh Breeze threads producing and consuming streams of data using this implementation is guaranteed to be determinate.

11. TRANSACTIONS

So far we have only dealt with computations that may be readily expressed using determinate expressions in a functional programming language. The world of computing is larger than this: Some applications are inherently nondeterminate, for example when two agents compete asynchronously for the last seat on an airline flight. Such computations are said to deal with "state" and appear to require the use of program constructions that have "side effects".

Some computations appear to have "state" but are nevertheless determinate because the data comprising the sequence of states may be regarded as a stream of values. Such computations may be programmed in a purely functional language using stream data types. Here we deal with computations that are truly non-determinate.

A *transaction* is an action that is thought of as altering the values of one or more stored data records. Execution of a transaction by a program may be viewed as a communication (request and response) by the program with an external agent. In this view, the program, without the external agent, may be determinate while the overall operations of the system containing the program and external agent is nondeterminate.

It is helpful to use a programming discipline that conforms to this view. In the Fresh Breeze execution model, transactions are supported by means of an object called a *guard* that occupies a memory chunk owned by the user responsible for the correct maintenance of the shared data, perhaps a large database[12][11][13]. The guard holds a pointer to the head element of a stream (initially empty) that will hold transaction requests. The instruction **EnterRequest** takes the guard and a data structure representing a transaction request as arguments, and installs the request as a new element of the request stream. Any thread that has been given a pointer to the guard object may execute the EnterRequest instruction, hence the guard is an implementation of the nondeterminate *stream merge* operator. It is the only source of nondeterminate behavior in a Fresh Breeze system.

The stream of transaction requests can be processed by a *transaction function* that accepts the stream of requests and a data structure representing the starting state of the shared data or database. The transaction function reads the first transaction request from the request stream and uses it to construct a new version of the shared data. It then calls itself recursively with the remaining elements of the request stream and a pointer to the new version of the data.

This arrangement permits several forms of parallelism in transacton processing: Firstly, the transaction function may expose parallelism within the performance of a single transaction. In addition, execution of several transactions from a single request stream may overlap if the transact function produces a future for the new version before completing its computation, just as with cascaded data structure construction discussed earlier. Of course one database may be partitioned into several sections for which separate streams of requests are processed concurrently; and many separate databases may be resident on the system.

12. CONCLUSION

The program execution model provided by a computer system has an enormous impact on the ability of programmers to efficiently build (create) quality software. A good program execution model can make the expression of algorithms straightforward and easy; the lack of a good program execution model can cause programs to be convoluted and difficult to develop, as is the case with parallel programming today. A good program execution model can support sound principles of program organization, such as the ability to build large programs using concepts of modular software.

The most familiar program execution model is a process executing instructions that perform computations and read and write accesses to a linear array of memory cells. Computer scientists have made this simple model into a powerful tool for programming by developing means for composing programs out of subroutines and library modules. McCarthy's introduction of Lisp[19] showed how general classes of dynamic data structures can be supported within this simple model by adding dynamic memory allocation and garbage collection, providing a sophisticated level of generality within the simple model.

Things became more complicated when we started running multiple programs on computers and began using operating systems that multiplexed use of the machine's resources. It became necessary to deal with concurrent processes that accessed shared data. More challenges were added with the introduction of multiple processor computer systems. For many years it has been customary for the user to duck the problems by writing sequential code that does not exploit the multiprocessing capability of such systems. This is no longer a tenable strategy.

The common wisdom has been to ask: what is the right memory model to use when several processes or threads have concurrent access to a shared memory? In my view, this is a flawed approach.

What is needed is not a memory model but rather a satisfactory program execution model. We build computers to perform computations that are specified by programs, and it is program execution that we desire to be correct. The need is for a program execution model that encompasses concurrency and honors the principles of modular software construction.

There are several program execution models for parallel computing: Operating system support for concurrent threads; language support through thread libraries as in Java or C; the Message Passing Interface (MPI)[21], the Threaded Abstract Machine (TAM)[4], Cilk and JCilk[14]; and now various versions of hardware or software implementations of Transactional Memory[15]. However, none of these schemes have gotten us away from the nemesis of dealing with shared memory. We must move away from the idea that a process can modify any data any time it chooses. We should recognize that the primary intent of a program is to create information, not destroy it. In most cases the intention of a store operation is to place new information in memory for future reference, not to overwrite information generated earlier. A failure of most program execution models for concurrent computation is the lack of a distinction between determinate and nondeterminate computation. Today there is little excuse for expressing a determinate computation in a model or language that fails to guarantee determinate execution. Of the cited models, only MPI can provide such a guarantee, but only for its use in distibuted memory systems, and provided the programmer avoids sending messages from multiple sources to the same receiver. (Similar remarks hold with respect to the Erlang language[1].)

In this presentation we have offered a program execution model that has some appealing qualities: (1) It satisfies the requirements for supporting modular construction of a general class of programs; (2) It provides a guarantee of determinate execution if the features supporting transactions are not used; (3) I believe it is amenable to efficient implementation, for example, using simultaneous multi-threading technology.

One might ask whether the Fresh Breeze model is sufficiently general? With respect to determinate computations, the model supports all of the important paradigms of parallel computing I am familiar with. With respect to nondeterminate computations, I have explored the generality of functionally processing merged streams of transaction requests. I have convinced myself that the prospects are very good. I hope to convince others through further work and publications.

13. ACKNOWLEDGMENT

This work is supported by National Science Foundation research grant 9527253.

14. REFERENCES

[1] Armstrong, J., Virding, R., Wikström, C., and Williams, M. *Concurrent Programming in Erlang*, Second Ed., Prentice Hall, 1996.

[2] Baker, H., and Hewitt, C. *Specifying and Proving Properties of Guardians for Distributed Systems*, Artificial Intelligence Memo 505, MIT Artificial Intelligence Laboratory, Cambridge MA, June 1979.

[3] Bensoussan, A. Clingen, C. T., and Daley, R. C. The Multics virtual memory. In *Proceedings of the Second Symposium on Operating System Principles*, ACM, 1969, 484-492.

[4] Culler, D. E., Goldstein, S. C., Schauser, K. E., and Voneicken, T. TAM – a compiler controlled threaded abstract machine, *Journal of Parallel and Distributed Computing 18*, 3 (July 1993), 347-370.

[5] Daley, R. C., and Dennis, J. B. Virtual memory, processes and sharing in Multics. *Communications of the ACM 11*, 5 (May 1968), 306-312.

[6] Dennis, J. B. Fresh Breeze: A multiprocessor chip architecture guided by modular programming principles. *ACM SIGARCH Computer Architecture News 31*, 1 (March 2003) 7-15.

[7] Dennis, J. B. 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, 96-103.

[8] 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, 50-60.

[9] Dennis, J. B. Static mapping of functional programs: an example in signal processing. In *Proceedings of the conference on High Performance Functional Computing*. A. P. Vim and John Feo, eds., Lawrence Livermore National Laboratory, Livermore, CA, 1995.

[10] Dennis, J. B. 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] Dennis, J. B. *Data Should not Change: A Model for a Computer System*. Computation Structures Group Memo 209, MIT Laboratory for Computer Science, Cambridge MA, July 1981.

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

[13] Dennis, J. B., and Wu, H. *Practicing the Object Modeling Technology in a Functional Programming Framework*.

Computation Structures Group Memo 379, MIT Laboratory for Computer Science, Cambridge, MA, 1996.

[14] 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 (May1998).

[15] Herlihy, M., and Moss, J. E. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th International Symposium on Computer Architecture*, IEEE Computer Society, 1993, 289-300.

[16] Hewitt, C. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence 8*, 3 (1977), 323-364.

[17] Karp, R. M., and Miller, R. E. Properties of a model for parallel computations: determinacy, termination, queueing. *SIAM Journal of Applied Mathematics 14*, 6 (Nov. 1966), 1390-1411.

[18] Lamb, A. A., Thies, W., and Amerasinghe, A. Linear Analysis and optimization of stream programs. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, June 2003.

[19] McCarthy, J. History of LISP. In *History of Programming Languages: The First ACM SIGPLAN Conference on History of Programming Languages*, New York: ACM Press, 1978, 217-223.

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

[21] Snir, M., Otto, S., Huss-Lederman, S., Walker, D., and Dongara, J. *The MPI Core*, second ed., The MIT Press, Cambridge, MA, 1998.

[22] Tullsen, D. M., Eggers, S. J., and Levy, H. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, IEEE Computer Society, 1995, 392-403.