A Parallel Program Execution Model Supporting Modular Software Construction

Jack B. Dennis
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
U.S.A.
dennis@aj.lcs.mit.edu

Abstract

A watershed is near in the architecture of computer systems. There is overwhelming demand for systems that support a universal format for computer programs and software components so users may benefit from their use on a wide variety of computing platforms. At present this demand is being met by commodity microprocessors together with standard operating system interfaces. However, current systems do not offer a standard API (application program interface) for parallel programming, and the popular interfaces for parallel computing violate essential principles of modular or component-based software construction. Moreover, microprocessor architecture is reaching the limit of what can be done usefully within the framework of superscalar and VLIW processor models. The next step is to put several processors (or the equivalent) on a single chip.

This paper presents a set of principles for modular software construction, and descibes a program execution model based on functional programming that satisfies the set of principles. The implications of the pinciples for computer system architecture are discussed together with a sketch of the architecture of a multithread processing chip which promises to provide efficient execution of parallel computations while providing a sound base for modular software construction.

1. Introduction

The ideas put forth here are mostly old; they have been developed by the author and others over the past thirty years.[1] What is new in this presentation is showing how the nature of Model X—the program execution model advocated here as a guide for computer system design—follows from basic requirements for supporting modular software construction.

The fundamental theme of this paper is:

> The architecture of computer systems should reflect the requirements of the structure of programs. The programming interface provided should address software engineering issues, in particular, the ability to practice the modular construction of software.

The positions taken in this presentation are contrary to much conventional wisdom, so I have included a question/answer dialog at appropriate places to highlight points of debate. We start with a discussion of the nature and purpose of a *program execution model*. Our *Parallelism Hypothesis* is introduced in Section 3. In Section 4 we present a set of six principles a programming environment should honor to support true modular software construction. In Section 5 these principles and the parallelism hypothesis are used to derive an ideal program execution model, Model X, step by step. Section 6 comprises a description of a computer system architecture inspired by Model X and the state-of-the-art in computer technology. We conclude with some remarks on the suitability of other models of computation for supporting modular (or component-based) software construction.

2. Program execution models

Models of computation have several important uses relating to the design of computer systems. A model may serve as a pattern for structuring and analyzing programs. The data parallel model and the "message-passing interface" (MPI) are examples. Some models are intended as a basis for assessing performance of systems or applications. Here

[1] One inspiring event for this work was the conference on Modular Programming put together in 1968 by Larry Constantine [5].

we are concerned with models that specify the behavior of a computer system to the extent that it is relevant to correct execution of application programs. We call these *program execution models* because they explicitly describe the actions involved in the execution of a program by the specified computer system. In a sense a program execution model is a formal specification of the *application program interface* (API) of the computer system. it defines the semantics of the target representation for the compilers of high-level languages; and it provides a clear design goal for the computer system architect.

3. Parallelism

Parallelism is important for several reasons:

- Distributed computing involves parallelism.
- Single thread processor architecture is running out of steam.
- Parallelism is an essential aspect of certain computations.

Nevertheless, parallelism introduces the possibility of *nondeterminate* behavior: a software component can produce different results on different runs with the same input data when the data arrive asynchronously or there are variations in the timing of internal events. It is widely appreciated that the possibility of nondeterminate behavior makes complex software difficult to design, develop and debug in a distributed or parallel computing environment. I believe in the programming principle that nondeterminate behavior should be avoided whenever it is not essential to the computation being specified. Furthermore, the semantics of a computation should be the same whether the program is run in a sequential or a parallel/distributed environment.

These ideas are captured in the following two properties of Model X which constitute our Parallelism Hypothesis:

- Program modules in Model X must be determinate unless nondeterminate behavior is desired and explicitly introduced by the programmer.
- Model X must permit the parallel execution of two modules whenever there is no data dependence between them that requires sequential operation.

4. Principles to support modular software construction

A primary concept in modular programming is the interface of a component—the manner in which the component interacts with its users. The most common kind of interface in software construction is the procedure call. Execution of a procedure call supplies a module with a set of input values and requests that the module use those values in constructing result values to be made available to the caller. To attain the full benefits of modular programming the component interface supported by the computer system should meet the following requirements:

- **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.
- **Invariant Behavior Principle:** The functional behavior of a module must be independent of the site or context from which it is invoked.
- **Data Generality Principle:** The interface to a module must be capable of passing any data object an application may require.
- **Secure Arguments Principle:** The interface to a module must not allow side-effects on arguments supplied to the interface.
- **Recursive Construction Principle:** A program constructed from modules must be useable as a component in building larger programs or modules.
- **System Resource Management Principle:** Resource management for program modules must be performed by the computer system and not by individual program modules.

Some of these principles have been generally appreciated for many years: The Information Hiding principle was advocated by David Parnas[29]; it is acknowledged in the encapsulation of objects as data representations together with sets of operations, as in the *abstract data types* of the programming language ML[27].

Appreciation for the Resource Management principle is evident in the success of virtual memory and paging, which relieve the programmer of the (anti-modular) use of memory overlays and the consequent ambiguity of addresses. Nevertheless, the continued practice of partitioning data into “memory” and “files” seriously impedes the practice of modular programming, 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.

On the other hand, violations of some of these principles are acclaimed as important features of popular programming environments. One of these is the Invariant Behavior principle. It is often pointed out that the ability of a module to have different behavior when invoked in differing contexts can be used to provide convenient testing facilities. However, this “feature” also permits disasterous false bindings to

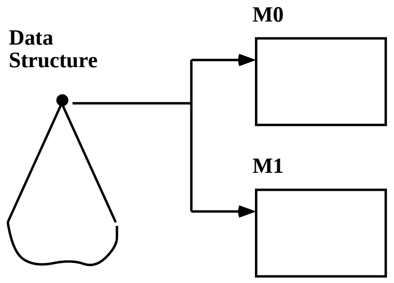


Figure 1. Illustration of the Secure Arguments principle.

occur leading to "code rot". It is better to make all intended behavior of a module selectable by parameter choices and satisfy the principle of Invariant Behavior.

The Secure Arguments principle is particularly important for our derivation of Model X. Figure 1 shows two modules where the arguments of both modules include a shared data structure. According to our Parallelism Hypothesis, it must be permissible to execute both modules concurrently. If either module can make any change to the value of the shared argument that is observable by the other, nondeterminacy will be exhibited whenever the results of executing a module can be affected by the change.

Our principles for modular software construction are violated in one way or another by the popular imperative programming languages. Yet achieving the full benefits of modular programming is not merely a matter of programming language design because large programs and application packages generally depend on operating system services for their correct operation. Using a semantic model that spans all functions and services needed by application programs offers the possibility of making the benefits of modularity in software widely available, and encouraging the use of component-based software in computing practice.

5. Derivation of Model X

In the following sections we develop a program execution model guided by our six principles for modular software construction. Then in Section 6 we discuss a system architecture based on a multithread chip that could implement the model.

5.1. Module representation

Model X must include a representation of the program to be executed—a target for a compiler of a high-level user programming language. First we need an object that will serve as the representation of a program module. For Model X we use a representation of a function from a vector of input types to a vector of result types. We use this abstraction instead of the conventional procedure abstraction because allowing side effects would be in violation of the Secure Arguments principle.

> **Q:** If you disallow side-effects then how do you accomodate programs that manipulate "state"?
>
> **A:** I divide programs that use "state" into two categories:
>
> - (a) Those programs in which the present output of a program module may depend on the entire history of its inputs rather than just the present input. These programs use "history-sensitive" modules but are still functional if viewed as operating on streams of data. They are *determinate*. The use of stream data types to support history-sensitive computations is discussed in Section 5.7.
> - (b) Programs in which a nondeterminate choice is made between alternative courses of action: for example, a reservation system where several agents compete for common resources. Support for nondeterminate computations is discussed in Section 5.8.

In Model X a function module is represented by an object we call a *text* ; just as a function is a fixed value, a text is not modified during function evaluation.

5.2. Invocation of component modules

The execution of (or evaluation of) a function generally calls for the computation of intermediate values which must be held temporarily between steps of evaluation. In Model X these intermediate values are held in a data structure we call an *activation frame* or just a *frame*. A frame exists for each activation of a function from the moment argument values are supplied to the time all results have been produced.

A function may invoke other modules (functions) as required by the Recursive Structure principle. By the Parallelism Hypothesis, the order in which several independent function invocations are made must not be restricted. Therefore Model X provides for the concurrent evaluation of functions invoked by a given function. This leads to the parallel hierarchical structure of texts and frames illustrated in Figure 2.

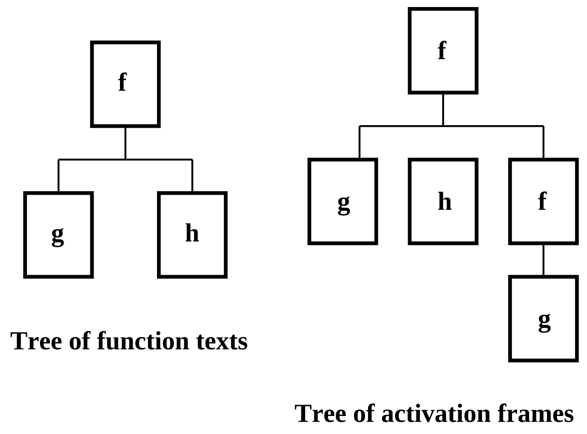


Figure 2. Parallel trees of texts and frames.

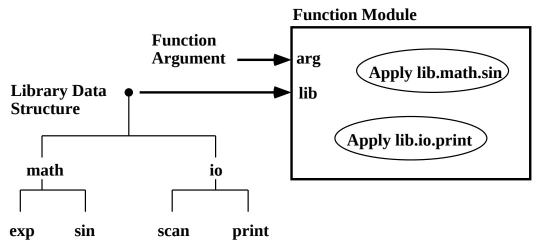


Figure 3. Use of a library of function modules.

5.3. Software component libraries and sharing

In Model X a library of modules is a data structure having the modules as structure components. The structure is passed as an extra argument to each function module that may make use of library elements, as shown in Figure 3. In this way the binding to library elements is not subject to the fortuitous false bindings that may be created by use of user-specified directory paths and search rules, which violate the Invariant Behavior principle.

Two modules may make use of the same component module in their construction. Hence the "tree" of texts is not really a tree, but a DAG (directed acyclic graph).

5.4. Recursion

The principle that any module should be usable in the construction of higher level modules implies the ability of a module to make use of itself as a component. Such recursive calls are supported in Model X by passing a function as an argument to itself. As shown in Figure 4 this scheme generalizes to groups of mutually recursive functions.

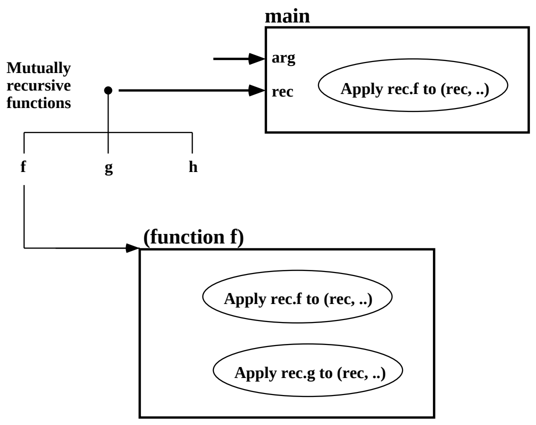


Figure 4. Use of a text structure to access a group of mutually recursive function modules.

Q: But recursion is so simple: just let the backward reference link to the function text, forming a cycle in the graph of texts.

A: In Model X the existence of cycles in data structures leads to problems we wish to avoid. Specifically, building a cyclic structure requires either that the cycle be created in a single step, or that a node in the graph of texts be modified after it has been created. Because handling cycles adds complexity to the model and leads to inefficient memory management, we prefer the chosen scheme.

Recursion is the natural way of representing many algorithms calling for repeated application of a group of steps. Moreover, tree recursions expose the parallelism of "divide-and-conquer" algorithms, as required by the Parallelism Hypothesis. As the following section shows, there are also good reasons to choose simple cases of recursion to model and express conventional iteration schemes .

5.5. Iteration

Iteration occurs where a sequence of results are computed, each obtained from the previous by the same computation. Iterations are naturally expressed as tail recursions, so Model X does not include any specific means for representing conventional iteration schemes.

Q: Why represent iterations by recursion? Iteration is, of course, more efficient, in general.

A: For the purposes of this paper it is beneficial to use the simplest model that provides sufficient generality. If necessary, a smart compiler can convert tail recursions into iterations. With a hardware design (see Section 6) that makes memory allocation a trivial step, run-time allocation for each iteration cycle (recursive call) may actually be the more efficient choice because it permits easier exploitation of parallelism.

With recursion the tree of activations may grow indefinitely.

Q: Isn't a tree much more difficult to allocate memory for than a stack that can grow in a segment of address space?

A: For general purpose parallel computation with generality equal to the best sequential programming environments, support for an unbounded tree of activations is essential. System architectures must provide efficient support for trees of activations, as has been done in several experimental systems [28, 33].

5.6. Data structures

The Data Generality principle requires that any data object of a computation may be passed as an argument or result of a function. For Model X we must choose a general representation for computation data. To have any chance of permitting an efficient implementation our choice must acknowledge the sharing of data. (A data object may be part of the input data of several functions.) Of course, we insist that components of a data object may be processed in parallel becuase this is a major source of parallelism in computation.

The most important feature shared by data structure types is their description of a data object as having components: *records* and *arrays* are familiar examples. Modern programming languages permit data structures to be declared as nests of record and array structures having arbitrary depth. These data structures are trees. To permit sharing, we enlarge the class of trees to the class of DAGs. To eliminate any possibility of violating the Secure Arguments principle, Model X has no update operation on data structures: structures can be constructed from their components, accessed, and released (by deleting an arc of the heap graph). With only these operations it is impossible to construct a cycle in the heap. We exclude cycles because creating a cycle involves updating a node, and opens the door to Secure Argument violations.

Q: But list structures are very important data structure for objects such as queues, and they typically involve cyclic data structures.

A: Queues are a low-level mechanism that can be supported using stream data types and I-structure mechanisms (Section 5.7). Other uses of cyclic structures in lists serve to implement various forms of tree structures which are the basic data structures of Model X.

Q: But what about programs that operate on graphs? Surely graphs in their full generality need cyclic structures for their representation.

A: Graphs have several representations that do not use cyclic data structure. One is an array of elements each representing an arc of the graph by a pair of integers interpreted as node identifiers. Such representations are often used because they are more efficient than representing a graph by an isomorphic data structure.

5.7. Streams and incremental data structures

A kind of data object having increasing importance is an unending sequence of values such as the successive time samples of an audio signal [14]. The designers of signal processing equipment have used the "stream of" abstraction for many years,[2] and there are several specialized programming tools based on streams.

The Sisal functional programming language [26] includes support for programming with stream data types. Figure 5 shows how a typical stream processing module may be written in Sisal. This function produces a result stream containing four elements for each group of three elements in the input stream. Each value in a group of four is defined by interpolation between appropriate values in the input stream. The notation `D[i]` denotes the ith element of stream `D`; the symbol "`||`" denotes concatenation of streams; the clause `stream integer [ ... ]` denotes a stream constant having the integer elements given within the brackets; the special function `Tail (k, D)` denotes the stream formed by removing the first `k` elements of the stream `D`.

Stream processing modules such as `FourForThree` can be cascaded by function composition to form networks of modules in producer/consumer relationships, all of which can be operating concurrently in pipeline fashion, as required by the Parallelism Hypothesis.

To support pipelined operation of stream-processing modules, Model X represents a stream by an incomplete list structure (or chain of records), as Illustrated in Figure 6. A stream element is represented by a "hole" until its value is filled in by the *producer*, a function module that defines a data stream. A *consumer* function is delayed when it

[2]The BLODI block diagram language was in use at Bell Laboratories in the early 1960s [23].

```
type Signal = stream [integer];

function FourForThree (
       D: Signal
       returns Signal )

   let
      n1 := D[1];
      n2 := ( D[1] + 3 * D[2] ) / 4;
      n3 := ( D[2] + D[3] ) / 2;
      n4 := ( 3 * D[3] + D[4] ) / 4;
   in
      stream integer [ n1, n2, n3, n4 ]
          ||  FourForThree ( Tail (3, D) )
   end let

end function
```

Figure 5. A rate changer function written in Sisal.

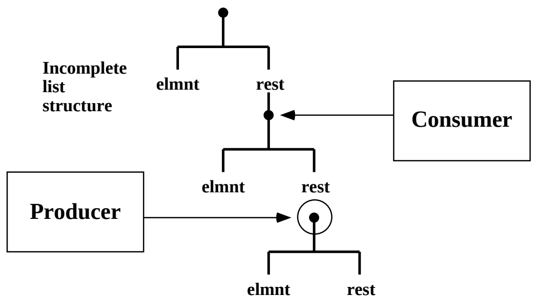


Figure 6. Producer and consumer functions and a data stream.

attempts to access a stream element and finds a hole. It continues once the hole has been filled [17, 12]. Semantically this model is similar to the I-structures of Arvind, Nikhil, and Pingali[3][3].

5.8. Nondeterminate computation

The need for nondeterminate computation arises when events can occur asynchronously that require use of a shared resource. In some cases the competing events arise in an ongoing computation within the computer system, for example, when independent jobs enter print requests in the queue for a printing device. In other cases, the events occur outside the computer system, as when airline agents independently request bookings for the same flight.[4]

> **Q:** But concurrent object-oriented languages [1, 21, 31] make this simple. The nondeterminate selection among competing input messages is built into the object model.
>
> **A:** The problem is that in concurrent O-O languages you have to use the mechanism whether you need to or not. Nondeterminacy is introduced as a consequence of concurrency, whether or not it is desired by the programmer. Object-oriented languages are generally unsatisfactory because of violation of the Secure Arguments principle, and the difficulty of practicing object encapsulation at a depth greater than one due to inadequate support of the Recursive Structure principle.

Generally, each case of nondeterminacy in application code may be viewed as requests arising from function modules and entering a queue to obtain service. The nondeterminacy exists only in entering the requests in the queue. In Model X each source of requests generates a sequence of request messages which are formed into a stream and merged nondeterminately with other request streams. The basic operation is the nondeterminate merge (ND-Merge) [10], which arbitrarily interleaves the elements of two input streams to yield its output stream.

> **Q:** But once you have introduced a mechanism for nondeterminacy, haven't you opened Pandora's box and permitted cycles to be created?
>
> **A:** As the ND-Merge is used in Model X, it merely takes elements of given streams and appends them to a single output stream. The interleaving of stream elements does not introduce any new mechanism that could create cycles in the heap.

6. Architectural considerations

Here we consider the architecture of a highly parallel computer system for executing computations expressed in Model X. First we note an important distinction between two classes of computations based on their need for resource

[3]We can preclude the formation of cycles that could arise from filling a hole as follows: Whenever a shell containing a hole is created we require that (1) the new shell is placed in a hole of a pre-existing shell node (or a root shell node of the heap); and (2) the new shell does not have any structure components except for holes.

[4]It is also true that asynchronously proceeding function activations in Model X compete for allocation of system memory. This involves nondeterminate choice, but this mechanism is part of the system itself, and not explicit in the application program.

management during execution. Then we discuss some important architectural features that allow the achievement of superior performance as well as supporting modular software construction. A brief discussion of the organization of a high performance single chip multithread processor for Model X computations concludes this section on architecture.

6.1. Static and dynamic computations

It is helpful to distinguish two classes of computations according to how processing and memory resources are managed during program execution:

- **Class A (Static).** These computations require no resource (processor/memory) allocation decisions during program execution.
- **Class B (Dynamic).** These computations require dynamic management of resources (load balancing, memory management) during execution.

For static computations, the function activation tree in Model X is effectively fixed and each of its nodes can be statically mapped to the processing resources of the target system. The heap can be implemented as a single segment of address space. Each recursion must be analyzed and shown to be an iteration (tail recursion); each such iteration may produce a series of data structure values which can be assigned to a fixed number of (reused) memory regions. Producer/consumer relationships can be identified and modules assigned to processors so as to achieve a balanced load on processing elements and communication links. This class of computations is a generalization of the *data parallel* class which has been successfully used (in the context of HPF [24] and Fortran 90) as the basis for programming many scientific computations for massively parallel computers [30]. The program analysis required to identify data parallel opportunities is much easier within the framework of a functional programming language such as Sisal, and an extension of compile-time analysis to stream-based computations in signal processing has been developed [15]. Although the class of static computations can be mapped readily to conventional multiprocessor systems, the benefits of multithread processor architecture will yield significantly better performance for these applications.

In the class of dynamic computations (Class B) the control structures involved in program execution are highly data-dependent. It is very difficult to code such problems for effective parallel execution using current systems: The coding is intricate and error-prone, and mechanisms must be used that violate the principles of modular programming. This class includes such programs as compilers for high-level programming languages, symbolic information processing, graphical interface code, as well as much of commercial data processing and many "irregular" scientific applications. The class of dynamic computations can benefit immensely from the support of Model X for modular software and parallel execution.

6.2. Features to support Model X

An attractive computing environment can be created by designing a system so that the principles of modular software construction are honored by an architecture in which resource allocation is flexible and cheap. We suggest that some powerful ingredients for achieving this are:

- Use multithreaded processing architecture [16].
- Support dynamic use of DAG data structures by a hierarchical memory system with a fixed size allocation unit.
- Associate a permanent unique identifier with every data object.

A multithreaded architecture in which successive instructions executed may be from different threads allows several important benefits:[5]

- High utilization of functional units, for example, floating point units.
- Latency tolerance for memory read operations.
- Reduction of context-switching overhead.

Q: Why is it necessary to use permanent unique identifiers for all data objects? Can't the same effect be accomplished through emulation or simulation?

A: Dynamic resource management is required when data objects of various sizes are created, accessed, and released, or their intensity of use varies widely during computation. For efficient memory management, it must be possible to use any available location for an object without having to change the identifier used to reference it. Maintaining access to an object that is susceptible to being moved is complex and inefficient, especially in highly concurrent systems.

Q: Why is it important to use a fixed size unit of allocation?

[5]Some of the early dataflow proposals, for example [11] may be viewed as multithreaded processors in which each instruction is a thread by itself. Several workers soon realized that utilizing some register context (and a notion of short instruction sequences) would yield a more efficient processing architecture, for example [20, 13].

A: When a fixed size unit it used the allocation of memory for a new data object can be made a trivial operation. Otherwise, complex data structures and search algorithms must be used to keep records of storage state. Use of a fixed size of memory allocation unit has several important benefits, especially in the frameworks of a highly parallel system: Storage management operations can be implemented in hardware yielding very inexpensive and efficient allocation of memory. It is never necessary to move a data unit for "logical" reasons. (For performance reasons data units may be duplicated at higher memory levels (caching).) It is never necessary to change the address at which a data unit may be found. It supports universal addressing

Q: What memory model would you implement to support Model X? How would you ensure memory consistency?

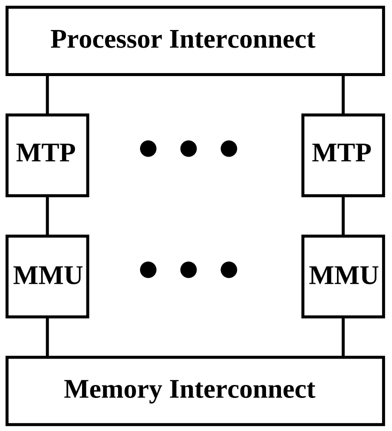
A: The purpose and function of the hardware is entirely specified by the program execution model. The purpose of the hardware is to execute programs, not to implement a memory model. Regarding memory consistency, it is not necessary for a distributed memory supporting Model X to be consistent.[6]

6.3. System architecture

These arguments suggest the system architecture in Figure 7. The overall organization is similar to that of conventional shared-memory multiprocessors. A group of n Multithread Processors (MTP) communicate with each other over the Processor Interconnect. The MTPs have associated Main Memory Units (MMU) which intercommunicate through the Memory Interconnect.

A program to be executed on the proposed system consists of many separate functions. We assume that execution of each instantiation of a function is carried out on a single multithread processor. This is the same policy that has been found to be effective on the Monsoon multiprocessor[28]. In the case of tail-recursive stream-processing functions, it is likely that successive instantiations of a function will be performed by the same processor. We assume further that all data objects are constructed in the local memory of the

[6]Memory consistency models appear to be necessary in conventional multiprocessor systems to ensure that the synchronization primitives used for conventional process synchronization work correctly and are ordered properly with respect to reads and writes of data. In Model X, "synchronization" is combined with data transfer as in the I-structure mechanism for passing stream elements, or the implementation of the ND-Merge.



MTP: Multithread Processor

MU: Memory Unit

Figure 7. Highly-parallel computer system to support Model X.

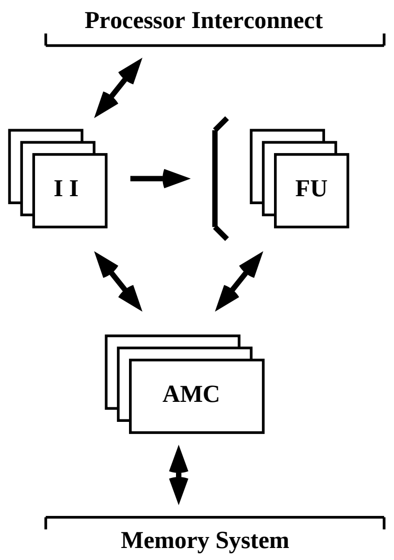
processor where the function is being executed. There are no remote create operations, only remote data structure access requests. Under these assumptions, four message types are used for interprocessor communication in direct support of program execution:

1. Initiate a function instance at a specified node with a given argument (scalar or structure).
2. Return the result (scalar or structure) of a function application to the the site of invocation.
3. Request access to a data structure component at a remote node.
4. Return a structure component value to the requestor.

6.4. The multithread processor chip

The Multithread Processor (MTP) is a single chip that contains Instruction Interpreters (II), Functional Units (FU), and Associative Memory Cache modules (AMC), as shown in Figure 8. The chip is organized so the instruction interpreters share use of the FUs and AMCs and high levels of utilization are possible.

This processor organization is a radical departure from conventional design in that each instruction is independently scheduled for execution. Yet the architecture reflects trends evident in such designs as the "all-cache" architectures, and features of the Monsoon multiprocessor[28]. In particular,



I I: Instruction Interpreter
FU: Functional Unit
AMC: Associative Memory Cache

Figure 8. Organization of the multithread processor.

each MTP uses a fast Associative Memory Cache (AMC) organized in *chunks* of about 32 bytes each, each chunk being identified by a key and accessed using hardware associative search. The program execution scheme is designed to take advantage of the fast and efficient memory allocation and deallocation that may be implemented in hardware with this organization.

The main memory will also be organized in chunk-sized blocks. Each data object has its home in a chunk of main memory, and is brought into an AMC unit automatically when called for by computing activity. A chunk will never hold parts of unrelated semantic objects, so there will be no problems of false sharing. The AMCs serve to reduce the time needed to access heavily referenced objects.

Q: Why not build the proposed system from commercial off-the-shelf microprocessors?

A: Conventional microprocessors (CISC, RISC, Superscalar, VLIW) are designed for single processor workstation/PC use, and are not optimal for multiprocessor use. They are not latency tolerant, and cannot exploit the sharing of functional units. Their context switching cost is too high, and they do not support efficient memory allocation.

Q: But conventional microprocessors have the advantage of high-volume, low-cost production, and massive engineering effort.

A: I believe contemporary ASIC fabrication technology is capable of achieving performance levels close to what can be achieved using massively engineered full-custom design—sufficiently close that the difference will be small compared to the architectural advantages and general benefits from supporting Model X. The unit cost will be higher initially, but keep in mind that the processor is just one part of system cost and the benefits can include improving the effective utilization of other components such as memory devices and interconnection structures.

Q: But the industry is not willing to move to a new programming model. There is too great an investment in software that only runs on conventional APIs.

A: The industry has reached a turning point. Soon, moving to parallel computing and multiprocessing on a chip will be the only possibility for advancing system performance. Available models for parallel computing are lacking in every dimension: performance, programmability, generality. Hopefully it will be possible to successfully introduce a new and powerful programming model in a domain, signal processing and real-time systems, where the importance of legacy software is relatively low, and the benefits of modular software construction will be high.

7. Conclusion

Model X is essentially the Graph/Heap model described by the author in 1974 [9], which evolved from [7, 8]. The “unravelling interpreter” model of Arvind [2] has similar characteristics. The benefits of using unique identifiers for data objects and a fixed unit of allocation were argued by the author in 1968 [7]. The same reasoning was used to justify use of segmentation together with paging in the architecture of the Multics time-sharing system [4, 6]. The incorporation of streams and incomplete data structures in the model stems from ideas of Ken Weng [17], and our recent proposals for supporting nondeterminate computations [18] have evolved from [10]. We hope to be able to construct an experimental system that implements the model described in this paper in the near future. We expect that its application programming language will be ObjectSisal [18].

Among the early proposals for modeling formally the complete semantics of a programming language or computer system are the Church *lambda calculus* and Johnston's *contour model*. The lambda calculus has major significance as an inspiration to programming language designers through the work of John McCarthy and Peter Landin, and efforts of the functional programming community. In contrast to Model X, the lambda calculus provides no direct way of modeling data structures and sharing, and offers little guidance regarding the expression of concurrency or nondeterminate computation. The contour model[22] focussed on explaining the operational semantics of concurrent processes in an Algol-like setting using conventional primitives for creating and synchronizing processes. The *Vienna Definition Language* (VDL)[25] used tree-structures in a formal model for the PL/1 programming language (with numerous violations of our principles for modular software). VDL evolved to the *Vienna Definition Method* (VDM) which incorporates some of the basic principles and notation of "Scottery" [32], and could be used to formally express the operational semantics of Model X. *Linda*[19] is a very different kind of computing model in which independent agents (processes) communicate through their shared access to a global "tuple space". In its original form, Linda is in violation of all six principles of modular software construction.

There are other important computational models, mostly intended as paradigms for analyzing performance issues, and their formulation does not address the program structure issues considered here.

Q: There has been a lot of work on semantic models for computer programs? Has it had any influence on the architecture of practical computer systems? If not, then why?

A: The potential benefits of modular software construction have not been recognized. Complaints about the complexity and cost of software are heard frequently. The fact is that the availability of large memories, paging (virtual memory), and improved programming languages has made programming easy enough that the genuine practice of modular software has not become an important issue. The necessity of introducing concurrency in application software is changing this picture.

References

[1] G. Agha. Concurrent object-oriented programming. *Communications of the ACM 33*, 9:125–141, September 1990.

[2] Arvind and K. Gostelow and W. Plouffe. The (preliminary) Id report: An asynchronous programming language and computing machine. Technical Report 114, Department of Information and Computer Science, University of California, Irvine, September 1978.

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

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

[5] Larry Constantine, Editor. *Modular Programming: Proceedings of a National Symposium.* Cambridge, MA: Information and Systems Press, 1968.

[6] R. C. Daley and Jack B. Dennis. Virtual memory, processes and sharing in Multics. *Communications of the ACM 11*, 5:306–312, May 1968.

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

[8] Jack B. Dennis. On the design and specification of a common base language. In *Proceedings of the Symposium on Computers and Automata*, Brooklyn, NY, 1971, pages 47–74.

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

[10] Jack B. Dennis. A language design for structured concurrency. In *Design and Implementation of Programming Languages*. In *Lecture Notes in Computer Science*, No. 54. Berlin: Springer-Verlag, 1977, pages 231–242.

[11] Jack B. Dennis. Dataflow supercomputers. *IEEE Computer 13*, 11:48–56, November 1980.

[12] 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, pages 260–267.

[13] Jack B. Dennis. The evolution of 'static' data-flow architecture. In J.-L. Gaudiot and L. Bic, editors, *Advanced Topics in Data-Flow Computing*, chapter 2. Prentice-Hall, 1991.

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

[15] Jack B. Dennis. Static mapping of functional programs: an example in signal processing. In *Proceedings of the Conference on High Performance Functional Computing*, A. P. Vim Böhm and John Feo, editors. Livermore, CA: Lawrence Livermore National Laboratory, April 1995, pages 149–163.

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

[17] J. B. Dennis and K. S. Weng. An abstract implementation for concurrent computations with streams. In *Proceedings of the 1979 International Conference on Parallel Processing*. IEEE Computer Society, 1979, pages 35–45.

[18] Jack B. Dennis and Handong Wu. Practicing the Object Modeling Technology in a Functional Programming Framework. Computation Structures Group Memo 379. MIT, Cambridge, 1996.

[19] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems 7*, 1:80–112, January 1985.

[20] Robert A. Iannucci. Toward a dataflow/von Neumann hybrid architecture. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, IEEE Computer Society, 1988.

[21] Yasuhiko Ishikawa and Mario Tokoro. Concurrent programming in ConcurrentSmalltalk. In *Object-Oriented Concurrent Programming*, Yonezawa and Tokoro, editors. Cambridge: The MIT Press, 1987, pages 129–158.

[22] John B. Johnston. The contour model of block structured processes. In *Proceedings of a Symposium on Data Structures in Programming Languages*, ACM, February 1971, pages 55–82.

[23] John Kelly, C. Lochbaum, and Victor Vyssotsky. A block diagram compiler. *Bell System Technical Journal 40*, 3, May 1961.

[24] D. B. Loveman. High performance Fortran. *IEEE Trans. on Parallel and Distributed Technology 1*, 1:25–42, February 1993.

[25] Peter Lucas and P. Lauer and H. Stigleitner. Method and Notation for the Formal Definition of Programming Languages. Technical Report TR 25.087. Vienna: IBM Research Laboratory, June 1968.

[26] 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. Livermore, CA: Lawrence Livermore National Laboratory, 1985.

[27] Robin Milner and Mads Tofte and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[28] 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, pages 82–91.

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

[30] Gary Sabot. A compiler for a massively parallel distributed memory MIMD computer. In *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation*. ACM and IEEE, 1992, pages 4–11.

[31] John Sargeant. Uniting functional and object-oriented programming. In *Object Technologies for Advanced Software*. *Lecture Notes in Computer Science*, No. 742. Berlin: Springer-Verlag, 1993, pages 1–26.

[32] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. Cambridge: The MIT Press, 1977.

[33] T. Yuba and T. Shimada and K. Hiraki and H. Kashiwagi. Sigma-1: a dataflow computer for scientific computation. Electrotechnical Laboratory, 1-1-4 Umesono, Sakuramura, Niiharigun, Ibaraki 305, Japan, 1984.