The EPSILON-2 Hybrid Dataflow Architecture[*]

SAND--89-2622C

DE90 003152

V.G. Grafe J.E. Hoch

November 8, 1989

Abstract

EPSILON-2 is a general parallel computer architecture that combines the fine grain parallelism of dataflow computing with the sequential efficiency common to von Neumann computing. Instruction level synchronization, single cycle context switches, and RISC-like sequential efficiency are all supported in EPSILON-2. The general parallel computing model of EPSILON-2 is described, followed by a description of the processing element architecture. A sample code is presented in detail, and the progress of the physical implementation discussed.

1 Introduction

The attractive properties of a dataflow model of computation have been studied for some time [1,2]. Transparent exploitation of parallelism, efficient synchronization, insensitivity to latency [3], and scalability are among the desirable characteristics. A handful of prototypes have been built [4], but most have suffered from low performance relative to commercially available computers. Recently, several efforts have produced dataflow computers that show promise of overcoming the historical performance limitation. The Sigma-1 [5] relies on a high speed, hash table based associative matching store. The Monsoon architecture [6] continues MIT's dataflow research with an *explicit token store* dataflow processor architecture.

Sandia's EPSILON processor [7,8] demonstrated sustained uniprocessor performance comparable to commercial mini-supercomputers. The EPSILON-2 system builds on the *direct match* (single cycle, instruction level synchronization) proven in the EPSILON processor. The static memory model of the EPSILON processor has been extended to a fully dynamic model, allowing single cycle context switches and dynamic parallelization. The pure dataflow scheduling mechanism has been superseded by a more general model that is a superset of both pure dataflow and strictly sequential execution. More emphasis has been placed on the software system for EPSILON-2, including compilers for both Id [9] and FORTRAN.

The general parallel computation model embodied in EPSILON-2 will first be described. The overall architecture will then be outlined, followed by a more complete description of the EPSILON-2 processor. A simple example code will be presented in detail to illustrate the actual operation of the machine. Some new problems posed by such an inherently parallel system will then be presented, along with indications of the direction being taken to address these issues.

2 The EPSILON-2 Parallel Computation Model

EPSILON-2 is based on an intrinsically parallel computation model. The instruction scheduling model of EPSILON-2 is a generalization of both the von Neumann and dataflow models. The storage model of EPSILON-2 is a parallel generalization of a traditional stack based storage model.

The scheduling and storage models are first presented without regard to the actual implementation.

[*]This work supported by the U.S. Department of Energy at Sandia National Laboratories under Contract DE-AC04-76DP00789.

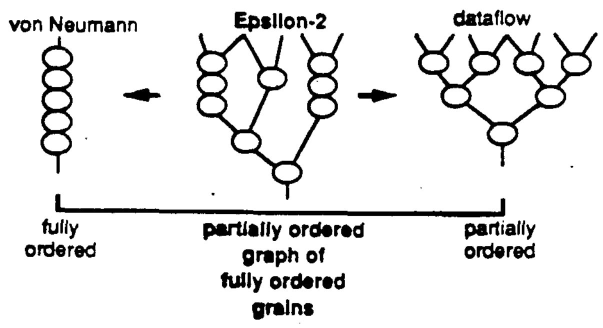

Figure 1: Spectrum of instruction scheduling models.

The description of the EPSILON-2 system that follows then discusses the implementation of the general models.

2.1 The Scheduling Model

The spectrum of instruction scheduling models is shown in Figure 1. At one extreme is the traditional von Neumann scheduling model, where the instructions are fully ordered. The von Neumann scheduling model is thus inherently sequential. This model does not cleanly extend to parallel processing, but it does allow locality to be exploited to great advantage. The other extreme is operation level dataflow, where the instructions are partially ordered by the dependencies in the computation. This model can exploit any form of parallelism, although it does not benefit from locality.

The scheduling model of EPSILON-2 is a generalization of the entire spectrum. There is still a partially ordered graph, as in operation level dataflow, so the model is applicable for any level of parallelism. Each node in the graph is a fully ordered sequence of instructions — a *grain* — rather than a single instruction as in the pure dataflow model. Graphs where all grains are length one correspond to the pure dataflow extreme. A graph composed of a single grain corresponds to the von Neumann extreme. This general scheduling model allows each code to strike a balance between parallelism and sequential efficiency.

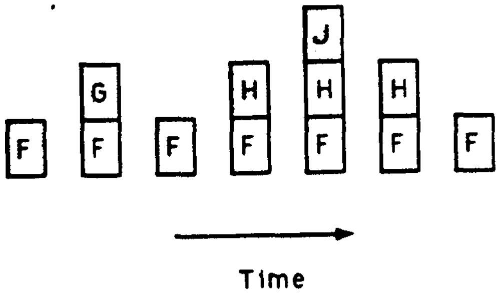

Figure 2: Snapshots of a traditional activation stack. Only the top procedure is active.

```
procedure F
{ ...
  call G
  ...
  call H
  ... }

procedure G
{ ... }

procedure H
{ ...
  call J
  ... }
```

Figure 3: Sample program for storage model comparisons.

2.2 The Storage Model

Traditional storage models are often based on a stack of activation frames. Activation frames are pushed on and popped off the stack as procedures are entered and exited. At any given time, only the procedure working in the topmost activation frame is active. As an example, Figure 2 shows snapshots of a traditional activation stack for the program shown in Figure 3.

EPSILON-2 generalizes the traditional stack model to a tree of activation frames. In EPSILON-2, procedures are invoked (and activation frames allocated) in a concurrent fashion. An invocation represents a

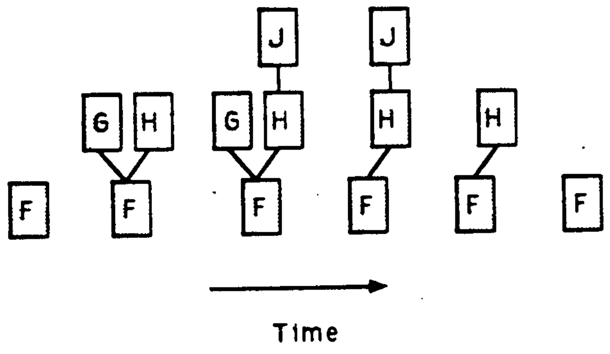

Figure 4: Snapshots in time of one possible unfolding of an EPSILON-2 tree of activation frames. All the activations in the tree may be active concurrently. Returns from independent activations may happen in any order.

fork of a new procedure rather than simply a transfer of control. Figure 4 shows snapshots of the EPSILON-2 activation tree for the program shown in Figure 3. Procedure F invokes procedure G and procedure H in parallel, and H invokes J concurrently. At any given time, the procedures working in any, or all, of the activation frames can be active (not just those working in the leaf activation frames). In fact, even in a single EPSILON-2 processor, any number of procedures can execute concurrently (in reality, their execution will be interleaved). To invoke a procedure, the caller requests an activation frame from any processor within the system. Once the activation frame is allocated, the procedure is bound to the processor which satisfied the request[1].

Together the EPSILON-2 scheduling and storage models form a comprehensive parallel execution model which allows the system to efficiently exploit all forms of parallelism (e.g., instruction-level, vector, loop-level, function-level). The storage model supports the dynamic spawning of concurrent tasks. The scheduling model allows each processor to use parallelism within a task (or parallelism between concurrent tasks on the same processor) to mask the latency associated with remote memory accesses and pipeline delays.

[1]Load balance across processors is achieved through the distribution of activation frame requests among the processors in the system.

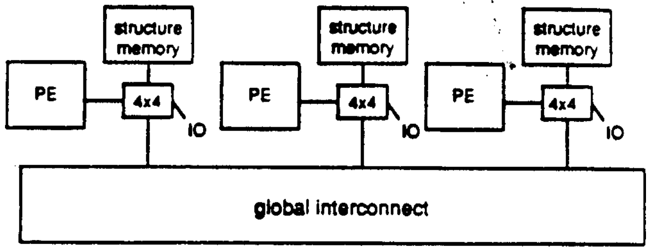

Figure 5: EPSILON-2 execution system architecture.

3 The EPSILON-2 System

The EPSILON-2 system consists of many components — the computer itself, programming tools, debugging and performance analysis tools, simulation capabilities, etc. We will first briefly describe the major components and their functions, then focus on the EPSILON-2 processor and its operation.

The EPSILON-2 architecture is shown in Figure 5. The system is built around a module consisting of a processor board and structure memory board, connected via a four by four crossbar to each other, an I/O port, and the global interconnect. In this way, each additional unit of processing brings with it a unit of structure memory (size and bandwidth), a unit of I/O bandwidth, and a unit of global interconnect bandwidth.

The unit of transaction for all communication in the system is a *token*. Tokens are fixed length entities, composed of a *target portion* (similar to an address) and a *data portion* as shown in Figure 6. The *type* field of both the target and data portions identifies the type of the information contained in the rest of the token. Targets are always typed as some sort of pointer, while data portions may be pointers, floating point numbers, integers, or logicals. The *value* of the data portion is interpreted as indicated by the type. On the processor, the instruction pointer (*IP*) is used to reference an instruction word in the instruction memory. On the structure unit, the instruction pointer is used directly to control the structure unit's operation. On the processor, the frame pointer (*FP*) selects a particular activation frame that the instruction will reference. On the structure unit, the frame pointer is used to address a particular location in the memory array.

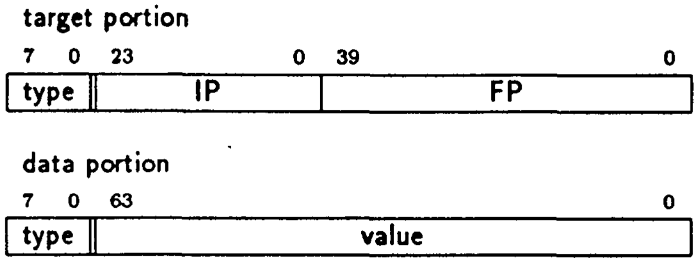

Figure 6: EPSILON-2 token definition. The *IP* (instruction pointer) and *FP* (frame pointer) specify the instruction and activation frame this token is destined for.

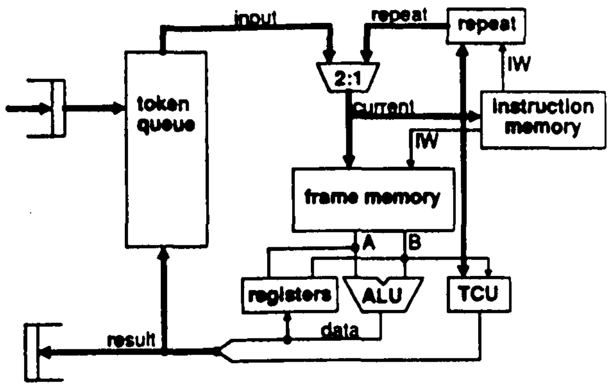

Figure 7: EPSILON-2 processor architecture. The heavy lines represent token paths.

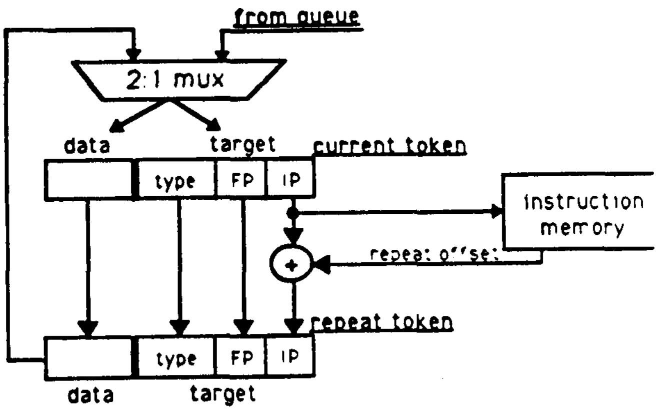

Figure 8: Generation of a repeat token.

The structure units are interleaved in the system address space, thus supporting the abstraction of a shared global memory. They support a variety of data structures via split-phase, token based transactions. A read, for example, is initiated by a processor sending an appropriate token to the structure unit. The data item is read from the memory, and returned to the processor as part of a new token. Traditional arrays, lists, and I-structures [10] are directly supported by the structure units.

The I/O ports are also mapped into the address space of the system. The I/O token interface resembles that of the structure memory units, although it of course deals with I/O devices rather than memory. The global interconnect is a packet switched multistage network. Its design is simplified by the fixed length, token-based communication.

3.1 EPSILON-2 Processor Architecture

The basic structure of the EPSILON-2 processor is shown in Figure 7. Tokens arrive from the local 4x4 switch and are buffered in the token queue. Tokens are read from the token queue and the instruction pointer used to access the instruction memory. The instruction referenced is used to generate a *repeat token* and to generate the addresses of operands and a synchronization point. These addresses are relative to the frame pointer of the current token, so the current activation frame can change with each incoming token (changing activation frames changes contexts). The operands are read from the frame memory and combined by the ALU to generate a result. The result is written into the registers and may be used to generate an output token. The target portion of the output token is generated by the TCU (target calculation unit).

Repeat tokens are generated as shown in Figure 8. A token is read from the token queue, and its instruction pointer used to read the instruction word from memory. The repeat offset in the instruction word is added to the current token's instruction pointer to generate a new instruction pointer. The current token's frame pointer and data portion are used unmodified in the repeat token. The repeat token is then used as the next cycle's current token rather than reading a new token from the token queue. In effect, the repeat offsets in the instruction word are

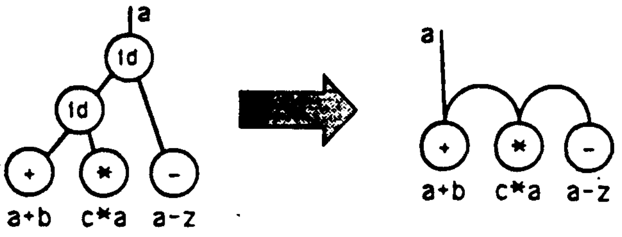

Figure 9: Using repeat to fanout data. The fanout required is shown implemented with a tree of identities on the left, with repeat on the right.

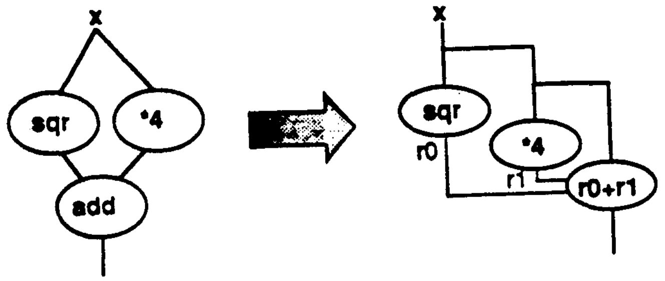

Figure 10: Repeat used for scheduling a grain of computation. The synchronization at the addition has been eliminated in the grain implementation and the latency has been reduced to a single pipeline transit.

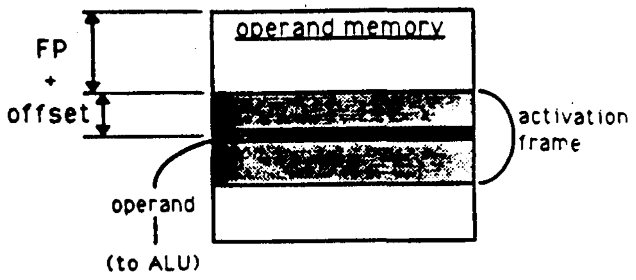

Figure 11: Accessing operands from the current activation frame. The activation frame is selected by the current token. The location within the activation frame is selected by an operand offset in the instruction word.

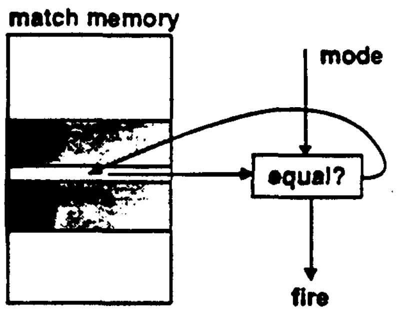

Figure 12: EPSILON-2 direct match synchronization.

used to build a linked list of instructions, where each instruction receives a token on the cycle after its predecessor in the list does. This is used both to fan out tokens for data fanout (multiple uses of the same data item) as shown in Figure 9, and for scheduling fanout (multiple instructions triggered from the same token, i.e., a grain), as shown in Figure 10.

The instruction also contains two offsets for accessing operands. These offsets may select a particular location in the current activation frame as shown in Figure 11, a location relative to the base of the frame memory (used for constants), or a register. In addition, the data portion of the current token may be stored in either of the locations specified by the operand offsets. In a pure dataflow graph, the data portion of the input token is always required by the instruction. In the more general model of EPSILON-2 some tokens carry only scheduling information in which case the data portion is ignored.

An additional offset in the instruction is used to select a location in the match memory. The match memory is organized into activation frames, just as the operand memory. Any number of instructions may synchronize at a given match location. The synchronization is done with a variant of direct match described in [7]. The operation of the direct match is shown in Figure 12. The selected location in the match memory is read. If the value matches the match mode of the instruction, the instruction *fires* (is allowed to execute) and the location is written with zero. If the value does not match the mode, the instruction does not fire and the value is incremented before being written back into the location. The synchronization therefore requires only a single memory read and a single memory write.

The ALU combines the two operands as directed by the opcode in the instruction word. The normal suite of floating point, integer, and arithmetic operations are supported, as well as a variety of special instructions added for greater efficiency.

The TCU is responsible for generating the target portion of any output token generated by the instruction. Targets local to the current activation are generated in much the same manner as the target portion of repeat tokens, retaining the current frame pointer and generating a new instruction pointer by adding an offset (part of the instruction word) to the current instruction pointer. Targets may also be formed using parts of the current operands, allowing tokens to be routed to computed targets (e.g., between contexts).

Each instruction writes its result into a register. These registers may be referenced by any succeeding instruction. It is important to note that register contents are *not* necessarily preserved across grain boundaries (other instructions in independent threads may overwrite the values stored in the registers). Registers are therefore used only for intermediate values within a grain.

The EPSILON-2 processor allows clock level synchronization due to the direct match mechanism. Since each token completely defines the processor's execution context, the processor is capable of switching contexts on each cycle. The ability to have multiple active contexts on each processor combined with the clock level context switch allows tasks which initiate unpredictable latency operations (e.g., data structure reads) to be automatically switched out in favor of other ready tasks.

4 Sample Program

To illustrate the capabilities of the EPSILON-2 architecture, this section steps through a simple example program implemented on EPSILON-2. The code for the example, expressed in the declarative language Id, is shown in Figure 13. Function F takes two arguments — an array descriptor A and an integer x. It computes $x*(A[x]-(x^2+4x))/G(x))$. Function G accepts a single argument, y, and returns $\sqrt{y}$. Figure 14 shows compiled machine graphs for the two functions of Figure 13. Token arcs entering from the left represent arguments passed into the function. Token arcs exiting from the right of function F represent arguments passed to an invocation of G.

```
def F A x =
   x * (A[x] - (x^2 + 4*x)) / G x);

def G y = SQRT y;
```

Figure 13: Id code for sample program.

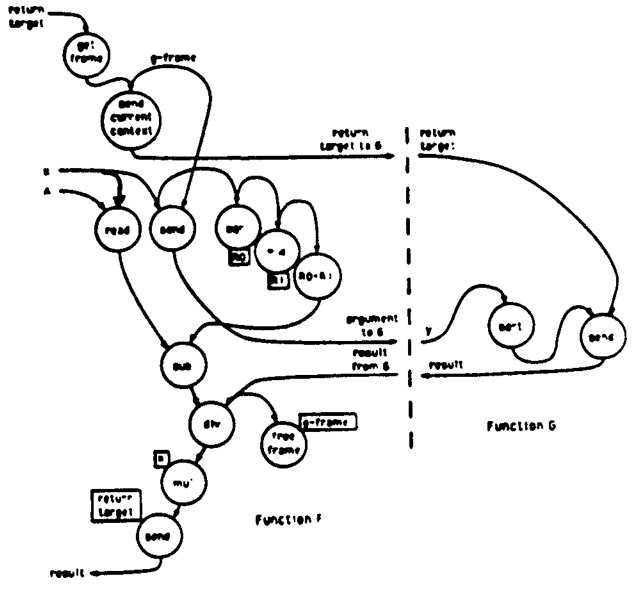

Figure 14: Machine graphs for the sample code.

In addition to its arguments, each function accepts a return target identifying the caller for a particular invocation. The functions route their results back to the return target.

The return target passed into F is used to trigger a GET-FRAME instruction, requesting an activation frame in which to execute G (preferably from the least loaded processor in the system). The result of the frame request is first directed at a SEND-CURRENT-CONTEXT instruction, which sends a return target to the new invocation of G, and is then repeated to a SEND instruction which sends x as an argument to G. The return target identifies the DIV instruction within the current invocation of F. Note that the request for the activation frame may be issued before either of the arguments to F arrive, allowing the latency associated with the request to be overlapped with the computation of the arguments for F or any computation within F.

The first argument to F, the array descriptor A, is directed at a READ instruction[2]. The second argument, x, is initially directed at the READ instruction and then repeated to the SEND instruction described above as well as three arithmetic operations. The READ instruction issues a read request token. The network directs the read request token to the appropriate structure unit which in turn sends a reply token containing the requested data item back to the SUB instruction. The three arithmetic instructions form a sequential grain. The SQR instruction squares x and stores the result in register 0. The MUL instruction multiplies x by 4 and stores the result in register 1. Finally the ADD instruction adds the contents of registers 0 and 1 and generates a token destined for the SUB instruction. The latency associated with the READ instruction and the latency associated with the invocation of G can be masked by these three arithmetic operations. The invocation of G can proceed in parallel with any other computation within F that is not dependent on the results returned by G.

Function G directs its argument, y, at a SQRT instruction and sends the result to its return target — in this case the right operand of the DIV within F. The result from G is then repeated to a FREE-FRAME instruction to release the activation frame in which G executed. The result of the DIV instruction is directed at a MUL instruction which references the value of x stored in the current activation frame for F[3]. Finally the MUL directs its result at a SEND instruction which routes the result of F back to its caller. This SEND references the return target from a known location in the activation frame for F. Note that within F, synchronization (instructions which require two tokens to fire) is only necessary when long latency operations (e.g. READ, GET-FRAME) proceed in parallel with work local to F.

5 Conclusions

The EPSILON-2 architecture provides an ideal foundation for parallel processing. The general parallel scheduling and storage models allow EPSILON-2 to efficiently exploit any form of parallelism. The support for multiple active tasks per processor and the cycle level context switching accomodate the unpredictable latencies associated with physically distributed memory.

There remains much to be done in the development of the EPSILON-2 system. Many of the programming tools taken for granted with traditional computer architectures need to be reformulated to take advantage of the unique capabilities of EPSILON-2. The ability to exploit all forms of parallelism brings with it the possibility that a program will unleash so much parallelism that the machine's resources will be overwhelmed. Strategies for constraining the amount of parallelism unleashed are needed [11]. The general scheduling model allows the tradeoff between parallelism and sequential efficiency to be explored. In fact, the intrinsically parallel nature of EPSILON-2 allows us to turn the traditional problem of parallelizing sequential pro-

[2]The interfaces between functions are defined so that arguments are routed to instructions at predefined offsets from the base of code blocks.

[3]Current pure dataflow machines would require a separate token to transmit the value of x to the MUL instruction.

grams on its head — we now start with inherently parallel programs and search for opportunities to sequentialize portions for greater efficiency. This new concept of *selective sequentialization* also warrants further research.

References

[1] R.M. Karp and R.E. Miller. Properties of a model for parallel conventions: determinacy, termination, queueing. *SIAM journal of applied math*, 1390–1411, November 1966.

[2] J.E. Rodriguez. *A graph model for parallel computations.* Technical Report TR-64, Dept. of Elect. Engr., Project MAC, MIT, September 1967.

[3] Arvind and R. A. Iannucci. *Two Fundamental Issues in Multiprocessing.* Technical Report CSG Memo 226-6, MIT Laboratory for Computer Science, May 1987.

[4] V.P. Srini. An architectural comparison of dataflow systems. *Computer*, March 1986.

[5] T. Shimada, K. Hiraki, K. Nishida, and S. Sekiguchi. Evaluation of a prototype data flow processor of the SIGMA-1 for scientific computations. In *13th Annual International Symposium on Computer Architecture*, June 1986.

[6] G.M. Papadopoulos. *Implementation of a General Purpose Dataflow Multiprocessor.* PhD thesis, Massachusetts Institute of Technology, August 1988.

[7] V.G. Grafe, G.S. Davidson, J.E. Hoch, and V.P. Holmes. The εpsilon dataflow processor. In *Proceedings of the* 16th *International Symposium on Computer Architecture*, 1989.

[8] V.G. Grafe and J.E. Hoch. Implementation of the εpsilon dataflow processor. In *Proceedings of the* 23rd *Hawaii International Conference on System Sciences*, 1990.

[9] Rishiyur S. Nikhil. *ID Reference Manual.* Technical Report CSG Memo 284, MIT Laboratory for Computer Science, March 1988.

[10] Arvind and R.H. Thomas. *I-structures: An Efficient Data Type for Functional Languages.* Technical Report TM-178, MIT Laboratory for Computer Science, September 1980.

[11] D.E. Culler. *Managing Resources in a Parallel Machine.* PhD thesis, Massachusetts Institute of Technology, June 1989.