The εpsilon Dataflow Processor *

V.G. Grafe
G.S. Davidson
J.E. Hoch
V.P. Holmes
Sandia National Laboratories
Albuquerque, New Mexico

Abstract

The εpsilon dataflow architecture is designed for high speed uniprocessor execution as well as for parallel operation in a multiprocessor system. The εpsilon architecture directly matches ready operands, thus eliminating the need for associative matching stores. εpsilon also supports low cost data fan out and critical sections. A 10 MFLOPS CMOS/TTL processor prototype is running and its performance has been measured with several benchmarks. The prototype processor has demonstrated sustained performance exceeding that of comparable control flow processors running at higher clock rates (three times faster than a 20 MHz transputer and 24 times faster than a Sun on a suite of arithmetic tests, for example).

1 Introduction

The dataflow model of computation has been the subject of study for over twenty years. Although much progress has been made, only a handful of dataflow computers have actually been built [1].

In the dataflow model of computation, operations proceed on the availability of data rather than the action of a program counter as in the von Neumann model of computers. Dataflow research began in the late 1960's as a study of models of parallel computation by Karp and Miller [2] and by Rodriguez [3]. As the dataflow model was further explored, researchers began to see that hardware and computer languages could be developed to directly execute computations as specified by the model. The earliest machines executed graphs that did not change as the computations developed, ie., they did not dynamically unfold loops or procedure calls. Dennis and Misunas [4] proposed such a static model, together with a dataflow language VAL [5] and the MIT engineering model, an experimental architecture [4]. Two other static machines were developed, one in the U.S.A by Texas Instruments [6] and the LAU in France [7].

Arvind and Gostelow developed the dynamic model and proposed a new language, Id, and the Tagged Token Dataflow Architecture for executing dynamic dataflow graphs [8]. The dynamic model extends the concept of data token matching for an instruction by including a portion of the matching tag that dynamically changes for each loop instance. Several dynamic dataflow machines have been built, most notably the Manchester computer in England [9] and more recently the Sigma-1 in Japan [10]. In the United States, the research at MIT continues with the development and construction of the Monsoon computer [11]. While these dynamic machines, and the languages that support them, can potentially uncover more parallel work than the static machines, they have the difficult task of managing their finite collection of tags to avoid resource allocation deadlocks.

Davidson and Pierce used strictly software approaches [12] and special purpose hardware accelerators (DFAM [13]) to apply static dataflow principles to high performance, real-time embedded multiprocessor computing for aerospace applications. This early work utilized the SANDAC multiprocessing computer [14].

These early Sandia research efforts utilized the static dataflow model by coupling it to an existing traditional processor. The knowledge gained from this approach was later incorporated into a much more powerful and general purpose family of pure dataflow supercomputer elements, the εpsilon processors. The first of these processors have continued the DFAM tra-

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

dition of extremely fast firing rules by means of the direct matching approach, while incorporating dynamic binding mechanisms and abandoning the earlier reliance on von Neumann processors.

While the overall research scope of our effort includes processors and languages for parallel computation systems, the focus of this paper will be on one part of that system, the εpsilon processor. The εpsilon architecture is described in Sections 2 and 3. Detailed descriptions of the characteristics and features of the prototype processor are first presented, followed by performance measurements. Section 4 describes some of the current work being done with the εpsilon architecture given the lessons learned from the prototype. The principal advances in εpsilon are then summarized in Section 5.

2 The Prototype Processor

The εpsilon prototype was designed with several principles in mind. Chief among these were scalability and design simplicity. Each processor in a multiprocessor system will have its own tagged memory, sharing only network resources with other processors. The performance required of each block in a processor therefore does not increase as the number of processors increase. The design philosophy followed some RISC-like ideas, such as simple control hardware, single clock instruction execution (where possible), and the availability of ways in which to combine simple functions into more complicated ones. The goal of the development was a high speed dataflow processing element, suitable for use in a parallel processing supercomputer.

The processor architecture couples a fast ALU with a tagged memory. Results are routed either back to the local tagged memory or to an external target. The external target could be the tagged memory of another processor, a peripheral, or the host processor. A block diagram of the prototype processor is shown in Figure 1. The tagged memory contains idle or partially enabled instructions, only one of which may become enabled during a given clock cycle. An instruction may be the recipient of up to two data operands, the A and B fields, whose arrival enables the instruction. The result of performing the operation can then be routed back to the local memory through the local feedback FIFO, or to the external network through the external output FIFO, or both. The addresses for this routing come from the LOCAL and GLOBAL fields of the instruction. The instruction tags serve to indicate the presence of data operands.

There is a single, FIFO buffered port from the host into the processor and another from the processor to the host. Communication with the host (and eventually other processors and peripherals) is accomplished with memory-mapped transfers through these two ports. Another FIFO buffered path is provided for local feedback of intermediate results, allowing the εpsilon processor to take advantage of locality in a computation. Both the feedback and external input data are passed through an input stage and written into εpsilon's tagged memory. The writing of data into the tagged memory causes the matching tags to be checked and updated (in a single clock), and may fire an instruction. The data from the memory is sent to the arithmetic and address calculation units, where it is processed. Results are then written to one, both, or neither of the output ports based on the action of the conditional unit.

The prototype processor is constructed as a five stage, non-blocking pipeline (five clock cycles are required from the arrival of a data value until the result of the instruction it fires is returned to the tagged memory). It requires about 1 square foot of board space and consumes approximately 65 watts. It is built entirely with standard, off-the-shelf TTL and CMOS integrated circuits. The pipeline is guaranteed to be non-blocking by the dataflow model of execution. The pipeline is kept completely full as long as there is at least five-fold parallelism, making εpsilon efficient even with low degrees of parallelism. This is a marked departure from many of the earlier dataflow computers that required hundreds of ready instructions to keep their pipelines full [15].

2.1 Tagged Memory

Each word of the TAGGED MEMORY has several independently addressable fields. They are:

A input parameter data field.

B input parameter data field.

OP operation code, made of various sub-fields that control the operation of the ALU, the ADDRESS calculation, and the CONDitional section.

LOCAL destination address for feedback results, made of sub-fields that select destination word and field, and control the repeat function.

GLOB.ˆL destination address for external results, made of sub-fields that select destination word and field, and control the repeat function.

TAGS monitor the state of the input parameter slots, fires instructions when both have arrived.

The two one bit TAGS associated with each word of the memory track the arrival and presence of data in the two parameter slots. Writing the opcode of a word causes the two tags to be cleared, ie., no data has arrived. Writes to the input data slots modify the tags and can fire instructions, according to the following rule:

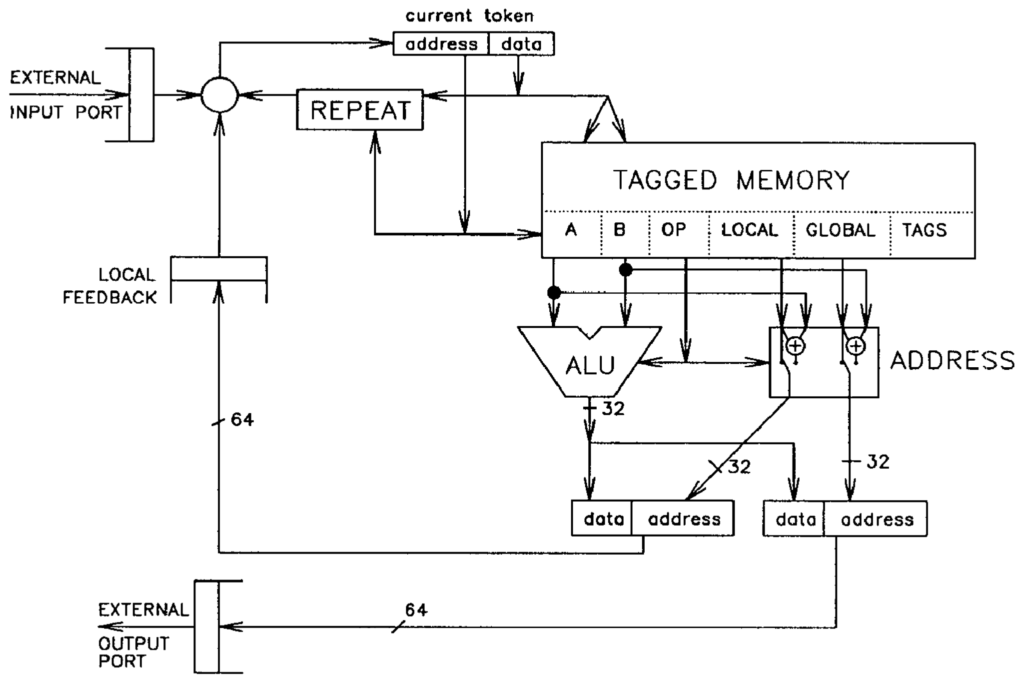

Figure 1. The εpsilon processor prototype.

if (other tag is set)
  then fire op and clear both tags
  else set this tag

In this way, writes to an instruction may fire it, but the instruction need only be checked when one of its operands is written (this is the only time its status is changed). The tag manipulation is performed in a single clock, so the dataflow overhead is no greater than the program counter manipulation of a control flow machine.

Constant values are handled with a slight modification to the scheme described above. Two bits of the opcode are used as *sticky* tags, one for each data field. A sticky data item is defined to be one that, once written, is always available (eg., constants). The tag rule is then modified to replace the tags with the sticky tags rather than clearing them. Sticky tags thus remain set once initialized, and non-constant values behave just as before. Constant values do not have to circulate or be regenerated, another departure from previous dataflow machines.

2.2 Arithmetic operations

The prototype εpsilon processor supports a full complement of arithmetic and logic operations in its ALU section. These include floating point ADD, SUBTRACT, MULTIPLY, DIVIDE, SQUARE ROOT, ABSolute value, NEGATE, MIN/MAX, COMPARE, and SCALEing. Similar arithmetic functions are available for integer data types. Logical operations include NAND, NOR, AND, OR, XOR, XNOR, SET, CLEAR, and a full set of SHIFTs and ROTATEs. Conversions between data types are also supported. Identity operations are also allowed (denoted PASSA), and are used to build many forms of control constructs. The operations supported are determined by the implementation of the arithmetic execution unit, and were chosen to support the needs of scientific computing. Other types of operations could be implemented if needed to support different types of computing.

2.3 Address Calculation

Destination addresses are computed in the ADDRESS calculation section. This operation proceeds in parallel with the arithmetic execution, similar to control flow machines with separate address calculation units. There are two sections to the address calculation unit, one for LOCAL FEEDBACK destinations and one for the EXTERNAL port. Each section is similar in operation with two inputs and three possible modes of address calculation. One mode is for static addresses known at load time and the other two are for run-time calculation of destination addresses. All three modes execute at the same rate. Selection of a particular mode is by a sub-field of the opcode.

One input is the hardcoded target (address) that is loaded with the code. This allows for destinations known at load time, as shown in Figure 2 where $z = (w * x) + y$ is being computed. The arc from the multiplication to the addition is known at load time, so the target destination is loaded with the appropriate address. This is also shown in loader notation on

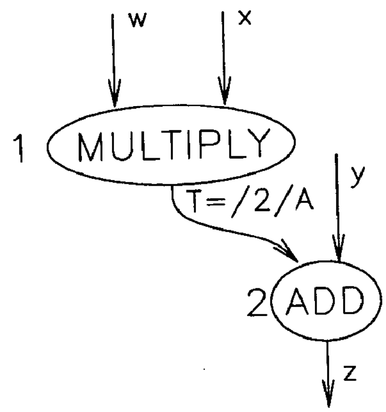

Figure 2. Use of static target.

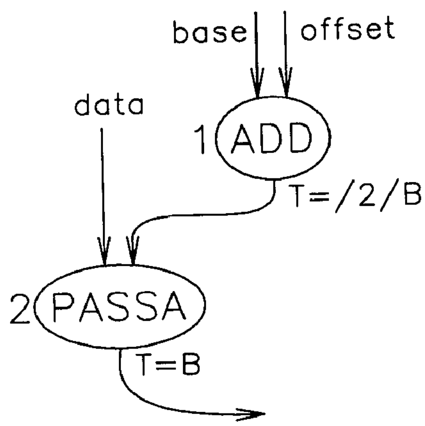

Figure 3. Run-time address computation.

the figure, where the /2 signifies instruction number 2, and the /A signifies the A parameter. The multiply executes when both **w** and **x** have arrived, and writes the product to the first parameter location of the subsequent add instruction.

The second mode allows run-time computation of a destination address. The second input to each side of the address calculation section is a data value from the tagged memory, the A data value for local feedback and the B value for external addresses. This data value can be used as the destination address. An example of this is shown in Figure 3, where the PASSA instruction passes the input data value in the A field to the address written to the input B field. The T=B notation specifies that the target address is taken from the B input field. In this example, a data value, **data**, is to be written to some address computed by adding an **offset** to a **base** address. The result of the addition is written to the B parameter of the PASSA instruction, where it is used as the destination. Thus, this instruction writes **data** to address **base + offset**.

The third mode of address calculation is used when one of the addends to an address is known at load time, but the other is not. An example of this is shown in Figure 4. In this case, **data** is written to the data

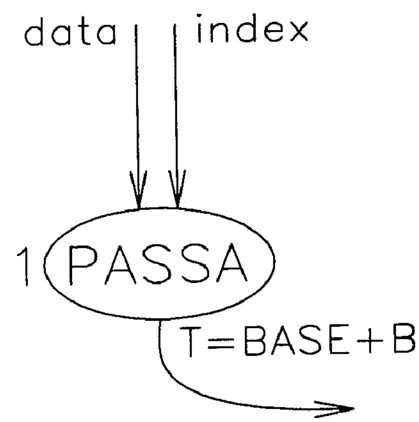

Figure 4. Run-time indexed address computation.

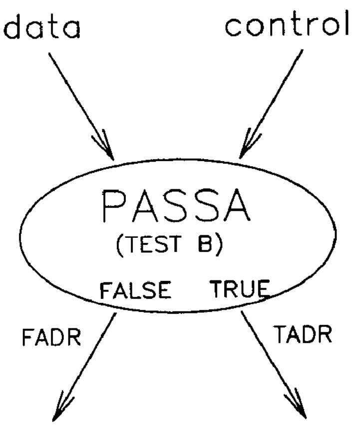

Figure 5. Switch operation in εpsilon

structure element **index** away from the structure start address BASE. BASE is written into the destination field at load time. At run-time, when **data** and **index** have both arrived the instruction will fire and pass **data** to the address formed by adding the B parameter (**index**) to the constant BASE. This mode allows traditional accesses such as arrays to proceed with no address calculation overhead.

2.4 Conditionals

Conditional constructs — **if-then-else**, **while**, etc. — are implemented by controlling the writes to the EXTERNAL and FEEDBACK FIFOs. The status flags from the arithmetic unit and the sign bits (used as boolean values) of the two input parameters, as well as a sub-field of the opcode are used to select which of the two ports to write (LOCAL, GLOBAL, both, or none). Traditional SWITCHes may be built as shown in Figure 5. In this example, a data value, **data**, is to be written to FADR if the control signal, **control**, is false, and to TADR if it is true. This is accomplished in εpsilon by using a PASSA instruction to pass **data** and making the outputs conditional on **control**. When this instruction fires **data** will be written to one of the

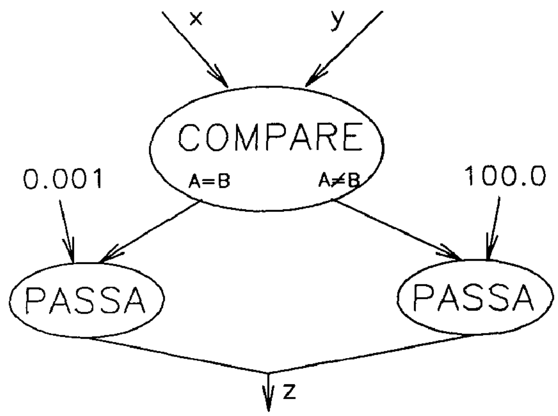

Figure 6. Conditional used as enable to computation graph.

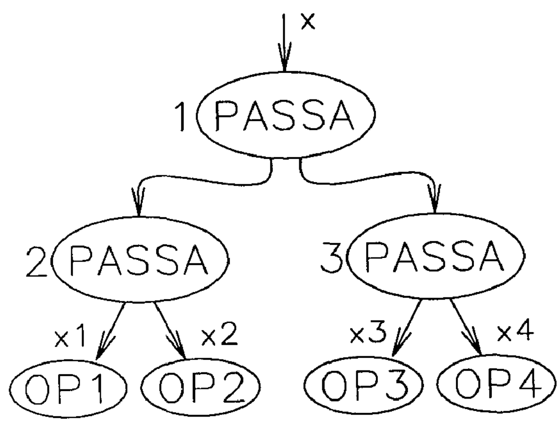

Figure 7. Additional instructions required for data fanout to multiple instructions.

two destinations based on the value of control.

The status flags from the arithmetic unit may be used to implement a different sort of conditional graph as illustrated in Figure 6. In this example the values of two parameters x and y are compared. If they are equal, z will be set to 0.001. If they are not equal, z will be set to 100.0. This implementation of conditionals can result in lower cost conditional graphs than the typical SWITCH-based implementations for case-like constructs.

2.5 Input Handling and Data Fanout

The dataflow scheduling mechanism used in ϵpsilon requires that each instruction have its data written into the tagged memory associated with the opcode. This allows high speed scheduling and execution, but requires that data be duplicated if it is needed by several instructions. The straightforward approach is shown in Figure 7. Here three extra instructions are needed to write the value x to four locations. This duplication requires extra instructions to generate additional copies of the data, and adds additional pipeline transit

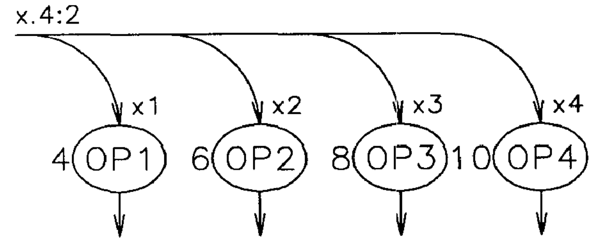

Figure 8. The previous data fanout example using repeats, repeat count equals four and repeat step equals two.

times to the latency of the computation. We have observed this overhead to be as much as 30 to 40 percent of the instructions executed in some codes.

This problem is addressed in ϵpsilon through a ***repeat-on-input*** [16] implemented with the REPEAT section. Address/data pairs are read out of the FIFOs, and written to the location specified by the address. The address contains fields specifying a repeat count and a repeat step, as well as selecting a word and field in the tagged memory. If the count is zero, the next address/data pair is read from the FIFO. If it is nonzero, the step is added to the address, the repeat count is decremented, the same data is written to the new address, and the cycle repeated. The fanout shown in Figure 7 is shown again in Figure 8 using this repeat feature with a repeat step of two words. The .4:2 after the x signifies that x is to be written to four words with a step between words of two. The overhead of data fanout is now reduced to the four clocks required to write the data. No additional instructions are required, and nothing is added to the latency of the computation. The restriction that instructions in a repeat chain must be loaded fixed steps apart is easily satisfied since the dataflow execution model makes no assumptions about instruction location.

The repeat-on-input's exploitation of the locality inherent in parameter duplication gives it advantages over both trees of instructions to duplicate parameters, as required in some dataflow machines [11,15], and destination lists, another proposed approach. With destination lists the execution pipeline must be stopped while the list of destinations is serially traversed, degrading performance. Alternatively, the execution pipeline may be insulated from the list processing with buffers. This incurs extra hardware cost, and adds latency to the computation because of the transfers from the execution pipeline to the list hardware. ϵpsilon's repeat-on-input does not add anything to the computation's latency, and does not force the processor pipeline to idle while data is written to multiple instructions.

Static critical path scheduling information can be exploited with the repeat-on-input. The order of instructions in a repeat chain gives control over the or-

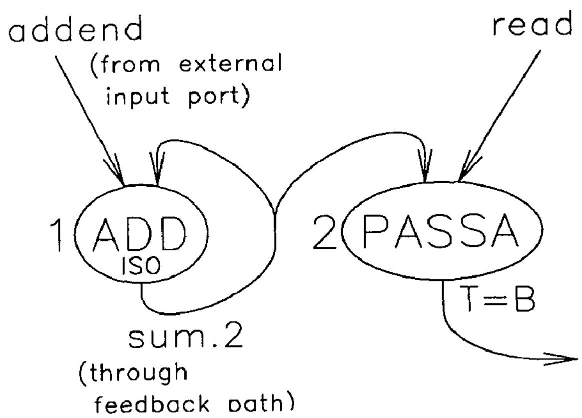

Figure 9. Computing the sum of an arbitrary input stream using isolate and repeat.

der of instruction firing. Operations on critical paths are placed at the front of a repeat chain, ensuring that they will execute before any of the other operations in the chain.

2.6 Critical Sections

Computers' limited resources are often managed through *critical sections*, code that must be executed without interruption from other resource requesters. The synchronization mechanisms required to limit access to these critical sections in control flow computers have received much attention. While dataflow computers have built in synchronization, the problem of uninterruptible instruction streams has not been addressed in previous dataflow designs.

Uninterruptible streams of instructions are supported in εpsilon through a mechanism called *isolate* [17]. Any εpsilon instruction may be declared to be *isolated*. No inputs are read from the EXTERNAL input FIFO as long as the processor is isolated. The processor becomes isolated when it fires an isolated instruction, and remains isolated until the result of that instruction passes through the FEEDBACK FIFO and is written into the tagged memory. If that result immediately fires another isolated operation, the processor will remain isolated, allowing chains of isolated operations to be executed.

An example of the utility of this function is shown in Figure 9, where the sum of an arbitrary input stream is computed. The running sum is initialized to zero. Addends are written to the A input of instruction 1. Each addend fires the add, causing the processor to add the addend to the sum in isolation. The processor remains isolated until the new sum is written back to the B parameter of the add. The sum is also repeated to another memory location for later use (by writing the read parameter). The addition in isolation ensures that no addends are lost or overwritten. Other local feedback data may still fire instructions when the processor is isolated. The isolated operation therefore may not incur any performance penalty. In the worst case, it will incur the single pipeline transit required to feedback the new value of the sum.

The isolation mechanism gives the programmer more explicit control over the execution of a program graph. It can be used for controlling asynchronous access to code segments as in the previous example, and for dictating the relative order of instruction execution. Instructions that enable many other instructions can be isolated, thus guaranteeing that their results are generated before any external inputs are allowed into the instruction stream.

3 Measured Performance of εpsilon

Several benchmark codes have been implemented in εpsilon's native graph representation and run on the prototype processor. The measured performances are compared here to several control flow processors. The codes included simple arithmetic diagnostics, random number generators, and scientific computing benchmarks. The performance measurements provide experimental evidence that a dataflow computer's performance can rival or better that of comparable control flow computers. This demonstration relegates many architectural arguments to second order effects.

Since it is difficult to precisely define what characteristics would make a control flow processor *comparable* to the εpsilon dataflow processor, two approaches were taken here. The first two sets of benchmarks compared εpsilon's performance to that of control flow processors performing the same function. The control flow implementations are comparable to the εpsilon implementation in that single board computers built with these architectures are available and require about the same amount of board space as εpsilon, cost about the same amount, and are built with the same level of technology. This comparison therefore gives a demonstration of the εpsilon dataflow processor's performance relative to control flow processors built with similar resources.

The last set of benchmarks are representative of scientific problems, so comparable processors were chosen to be those with similar performance goals as εpsilon. This set of comparisons gives a demonstration of εpsilon's *absolute* performance compared with control flow processors optimized for scientific computing. The inherent imprecision in defining comparable dataflow and control flow processors makes the performance comparisons less precise than would be the case in comparing control flow vector processors, for example.

There is a long held belief that dataflow computers require more instructions than comparable con-

```
#define MAX 1000000
#define MAXERR 0.1
main()
{ int i;
  float error=0,j,jsqd1,jsqd2,
  oneoverj,shouldbej,shouldbe0;

  for (i=0;i<MAX;i++)
  { j          = (float)i;
    jsqd1      = j * j;
    jsqd2      = j * j;
    oneoverj   = j / jsqd2;
    shouldbej  = jsqd1 * oneoverj;
    shouldbe0  = shouldbej - j;
    if (shouldbe0>MAXERR)
      printf("\nERR,i=%d",i);
    if (shouldbe0>error)
      error = shouldbe0;        }
  printf("max error = %f",error);}
```

Figure 10. Sample arithmetic diagnostic loop.

trol flow computers. Much of this has been shown to be an artifact of parallel processing, rather than dataflow processing [18]. In the benchmarks implemented for the εpsilon uniprocessor prototype, the number of εpsilon instructions required was similar to the number required for the control flow processors. Most of the differences, when present, were due to the CISC nature of the control flow processor being compared. Memory indirection and other multicycle instructions count as only one instruction, but actually cost many clocks of latency. Counting clocks, as the execution timings do, shows that the εpsilon dataflow uniprocessor requires *fewer* primitive (one clock) operations than the control flow uniprocessors. The primitive operations allowed on εpsilon make use of parallel computation and address calculation, and all fit this same model. In this way, the processing hardware is completely utilized with each instruction.

3.1 Arithmetic Diagnostic Benchmark

The first benchmark is a set of simple arithmetic diagnostics originally developed for testing the floating point units of control flow processors. These are tight loops that compute a complicated function of the loop index. The function algebraically reduces to a known value (typically zero or one), so the result of the computation can be checked in each iteration. An example of such a loop is shown in Figure 10. The performance on this type of diagnostic is presented to demonstrate εpsilon's high speed execution on problems with low parallelism, and to show that the εpsilon dataflow pro-

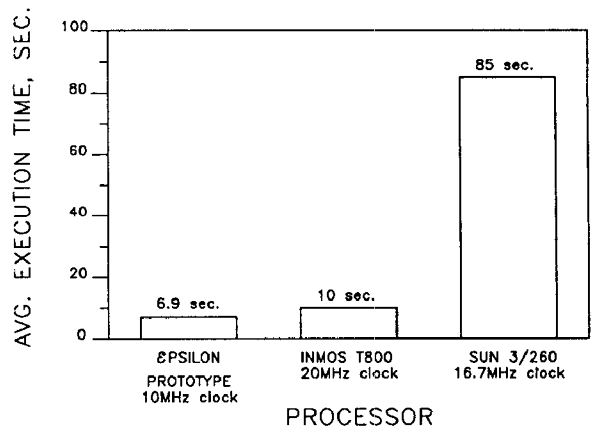

Figure 11. Average execution times on four arithmetic diagnostics.

cessor executes *faster* than comparable control flow machines. The diagnostic also demonstrates the ability of a single εpsilon processor to exploit available parallelism.

Four of these diagnostic codes were run. They emphasized different arithmetic operations: square root, multiply and divide, add and subtract, and a mix of these. They were coded in C for the control flow processors, and directly translated to εpsilon's native graph representation. In fairness to the control flow processors, εpsilon was restrained by the cross-iteration antidependencies [19] to execute only one iteration at a time. As the execution times in Figure 11 show, εpsilon at 10 MHz is faster than the control flow computers. This speed advantage is apparent even on essentially serial codes, even though the control flow processors were running at higher clock rates (the Sun at 16.67 MHz and the T800 at 20 MHz). These results suggest that dataflow uniprocessor computers are not inherently *slower* than comparable control flow computers, especially on problems with low degrees of parallelism.

The dataflow processor's ability to exploit parallelism, even in a uniprocessor configuration, is evident when the four diagnostic loops were run together. The execution times shown in Figure 12 demonstrate that the control flow machines must execute the independent loops in sequence. εpsilon is able to execute them in parallel, exploiting the parallelism to keep its pipelines completely full. εpsilon's speed advantage is now even more apparent. The εpsilon dataflow processor is able to exploit any degree of available parallelism, unlike the control flow processors.

3.2 Bit Manipulation Benchmark

The second benchmark, like the first, was originally developed for control flow processors. It uses various bit manipulations to generate a sequence of random

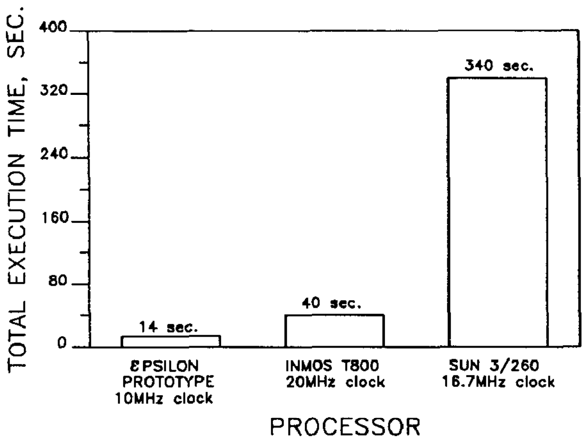

Figure 12. Total execution times for executing all four diagnostics together.

```
float rand()
{#define M 13   /* # of bits to shift */
 #define NmM 18 /* 31 - M = 18 */
 #define MAXrange 2147483647.0
                /* 2**(31)-1*/
 static int a=524287;
 register int b;
  b = a >> M;
  a = a ^ 1;
  b = a << NmM;
  a = abs(a ^ b);
  return (float) a / MAXrange;}
```

Figure 13. Random number generator used as a benchmark.

numbers. The algorithm is shown in Figure 13. The benchmark results are presented in Figure 14 as the time to generate one million random numbers. Again εpsilon is *faster* than the control flow processor, even on a code with a low degree of parallelism. This benchmark demonstrates that εpsilon's performance benefits over comparable control flow processors are present on bit manipulation operations as well as the floating point functions used in the first set of benchmarks.

3.3 Scientific Computing Benchmark

The other set of benchmarks presented are some of the Livermore FORTRAN Kernels [20]. These are a series of FORTRAN kernels taken to be representative of a scientific computing workload. The specific kernels used were chosen for the simplicity of the function performed, with no attempt to either avoid or favor vectorizable codes. In these benchmarks, εpsilon was allowed to execute several iterations in parallel as long as the data dependencies were observed. εpsilon's performance on six of these kernels is shown in Figure 15, along with that of the Convex-C1. The control flow vector computer is significantly faster than εpsilon on the kernels where the algorithm vectorizes well, but its performance falls drastically when vector parallelism is not available. εpsilon's performance is similar on all the kernels since it is determined by the ratio of floating point operations to integer and control operations. The control flow vector computer demonstrates much *more* sensitivity to the type (vector) and amount of parallelism present.

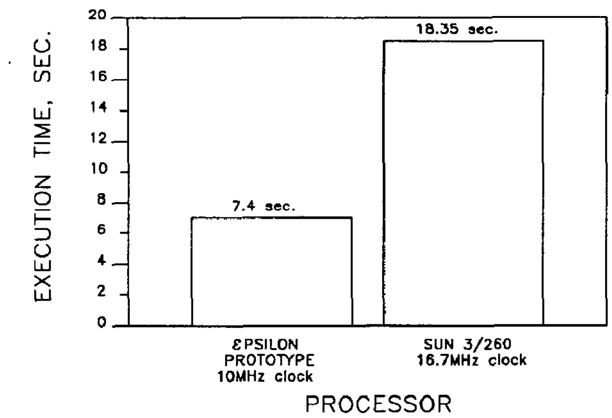

Figure 14. Time to generate one million random numbers.

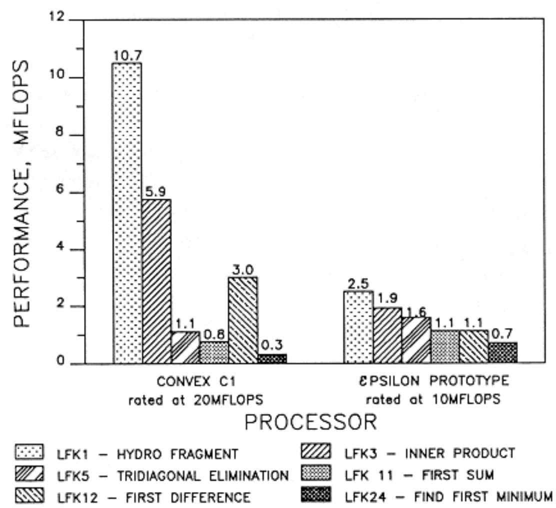

Figure 15. Measured performance on selected Livermore FORTRAN Kernels.

The sustained performance of these two machines on these kernels gives a better indication of what might be expected on a typical workload. Figure 16 shows the harmonic mean of the performances in Figure 15, along with that of the Cray-1S on the same kernels. From these results we would expect that one εpsilon processor would sustain higher through-

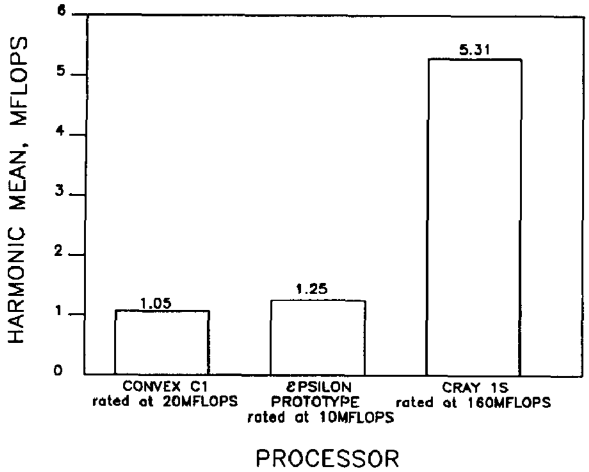

Figure 16. Harmonic mean of performance on FORTRAN Kernels.

put than the Convex-C1 and about one-fourth the throughput of the Cray-1S for a work load accurately represented by these kernels. It is important to note that the εpsilon processor is a single board, wire-wrap, 10 MHz CMOS prototype. The vector machines are multi-board, high speed computers constructed with advanced technology and custom chips. The εpsilon dataflow processor is able to exploit more types of parallelism than the control flow machines. Its performance is therefore determined by the total parallelism in the algorithm rather than how that parallelism is expressed.

4 Current Work

The εpsilon processor prototype described above demonstrates that this architecture points to a promising direction for future supercomputers. Continuing in this direction requires an overall system approach that addresses data structure storage, input/output facilities, network interconnections, and programming tools. All of these issues are currently being addressed.

The static nature of the εpsilon prototype was not conducive to recursive programming, and dynamic parallelization of loops was more cumbersome than desired. A new version of the εpsilon architecture is being developed. This new architecture, currently called εps'88 [21], retains the high performance features of the εpsilon prototype, including direct matching, pipelined processing, and local feedback paths. It abandons the static dataflow model in favor of a dynamic frame-based scheme to better support recursion and looping, and allow a straightforward port of the Id language. The repeat-on-input function has been expanded and generalized in εps'88. Combining the repeat feature with a general frame access capability allows large portions of a dataflow graph to be coalesced into a single node. This combination of features has prompted work on a FORTRAN compiler that has greatly increased the appeal of the εpsilon project to the application programming community.

The design of the εps'88 system is also addressing the network interconnections between the processors, and the associated structure memory units. The design of these elements is nearing completion, and a fully functional multiprocessor system with high level language support should be available within the next year.

5 Summary and Conclusions

The performance measurements suggest that a dataflow computer's performance under even a low degree of parallelism can be competitive with comparable control flow computers. They also show the dataflow computer's ability to exploit parallelism, even in a uniprocessor configuration.

εpsilon's execution pipeline is only five stages. It has the additional benefit of being guaranteed to be non-blocking — once an instruction has fired its required operands are, by definition, ready. Interlocks often required to ensure correct operation of pipelined computer are not required in a dataflow computer such as εpsilon. Because of this, the design of the εpsilon prototype processor is in fact simpler than the design required to build a conventional five-stage, pipelined processor with optimal pipeline control.

Pipelining along the critical path is inherent in εpsilon. The latency between instructions is five clocks. Pipelined computers must have some latency between instructions along a strictly serial thread, but conventional architectures have much greater difficulty finding other ready operations to cover that latency.

The principal result of this work has been the demonstration of a dataflow processor whose sustained performance exceeds that of comparable conventional processors. This comparison of measured performances shows that εpsilon is more efficient than the other processors. The comparison was done in the realm where conventional computers were previously believed to have an architectural advantage over dataflow computers — uniprocessor systems, running codes with low degrees of parallelism.

The εpsilon architecture benchmarks illustrate that a dataflow processor can take advantage of locality in a code, previously thought to be an exclusive property of control flow machines. The prototype processor exploited locality through its local feedback path and the repeat function. Intermediate results may be routed through the FEEDBACK FIFO, decreasing network traffic and latency between instructions. The repeat feature is also used to exploit locality by allowing fanout with strictly local feedback and by allowing multiple uses of the same data to be satisfied in the minimum time.

The performance measurements on the Livermore FORTRAN Kernels demonstrated the dataflow computer's ability to find and exploit any parallelism in the code. This is a distinct difference from traditional computers which require that the parallelism be in a specific form in order to be useful to the processor. Difficult programming practices and time-consuming algorithm changes are made to adapt the parallelism to a particular control flow machine's mold. These practices greatly complicate the task of obtaining acceptable sustained performance from the machine, and are often not portable to the next generation of computers.

References

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

[2] 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.

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

[4] J.B. Dennis and D.P. Misunas. A preliminary architecture for a basic data-flow processor. In *Proceedings of the Second Symposium on Computer Architecture*, December 1974.

[5] W.B. Ackerman and J.B. Dennis. *VAL — a value-oriented algorithmic language: preliminary reference manual.* Technical Report TR-218, MIT Laboratory for Computer Scicnce, June 1979.

[6] M. Cornish, D.W. Hogan, and J.C. Jensen. The Texas Instruments distributed data processor. In *Proceedings of the Louisiana Computer Exposition*, pages 189–193, March 1979.

[7] A. Plas et al. LAU system architecture: a parallel data-driven processor based on single assignment. In *Proceedings of 1976 International Conference on Parallel Processing*, pages 293–302, 1976.

[8] Arvind, K.P. Gostelow, and W. Plouffe. *an asynchronous programming language and computing machine.* Technical Report TR 114a, Dept. of Information and Computer Science, Univ. of California, Irbine, September 1978.

[9] J. Gurd and I. Watson. Data driven system for high speed parallel computing — part 2: hardware design. *Computer Design*, 97–106, July 1980.

[10] 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.

[11] G.M. Papadopoulos. The Monsoon architecture. notes for MIT summer course 6.83s, March 1988.

[12] G.S. Davidson. *A practical paradigm for parallel processing problems.* Technical Report SAND85-2389, Sandia National Laboratories, March 1986.

[13] G.S. Davidson and P.E. Pierce. A multiprocessor data flow accelerator module. In *Military Computing Conference, Conference Proceedings*, 1988.

[14] C.R. Borgman and P.E. Pierce. A hardware/software system for advanced development guidance and control experiments. In *Proceedings AIAA Computers in Aerospace Conference*, pages 377–384, October 1983.

[15] J.R. Gurd, C.C. Kirkham, and I. Watson. The Manchester prototype dataflow computer. *Communications of the ACM*, January 1985.

[16] V.G. Grafe and J.E. Hoch. *Repeat on Input: a New Approach to Data Fanout in Dataflow Computers.* Technical Report SD-4621, Sandia National Laboratories, 1988.

[17] V.G. Grafe and G.S. Davidson. *Uninterruptible Groups of Instructions in Dataflow Computers.* Technical Report SD-4592, Sandia National Laboratories, 1988.

[18] K. Ekanadham, Arvind, and D.E. Culler. *the price of parallelism.* Computation Structures Group Memo 278, MIT Laboratory for Computer Science, 1987.

[19] D.A. Padua and M.J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, December 1986.

[20] F. H. McMahon. *The Livermore Fortran Kernels: A Computer Test of the Numerical Performance Range.* Technical Report, Lawrence Livermore National Laboratory, December 1986.

[21] V.G. Grafe, J.E. Hoch, and G.S. Davidson. *εps'88:Combining the Best Features of von Neumann and Dataflow Computing.* Technical Report SAND88-3128, Sandia National Laboratories, 1988.