IEEE Region 10 Conference, Tencon 92
11th - 13th November, 1992
Melbourne, Australia

A SIMPLE MACHINE

Alex Dickinson
AT&T Bell Laboratories
Room 4E-514
Crawfords Corner Rd
Holmdel NJ 07733
USA

Michael T. Pope
ITD
DSTO
PO Box 1600
Salisbury SA 5018
Australia

ABSTRACT

Functional languages and Turner's SK-combinator reduction mechanism offer the promise of fine grained computation based on simple rules. We describe a simple machine architecture based on an extension to combinator notation. Code travels along a unidirectional *stream* and is executed in parallel by simple finite state machine based cells separated by sections of FIFO buffer. The resulting regularity and structural simplicity ensures a good match to VLSI implementation. We have verified that the extended combinator is a viable execution model, and present preliminary simulation results.

1 Introduction

VLSI technology provides us with a means of implementing complex systems in a very cost efficient manner. As such it has become the medium of preference for implementing von Neumann machines. Although not inherently complex, von Neumann machines require considerable hardware complexity (such as caches and pipelines) in order to overcome their inherent vulnerabilities.

We note however that VLSI is better matched to structures comprised of many iterations of small, identical cells. These structures are simpler to design, and have greater locality of communication leading to faster, denser circuits.

We begin this paper by describing a very simple, regular, and highly parallel computing structure that is well matched to VLSI implementation. We then describe how this simple machine can execute a functional program in parallel using an extension to Turner's SK-combinators [1]. The organization of an actual machine is then described, together with some early simulation results.

2 A Simple Machine

Our model of computation is inspired by a structures found in biological systems [2,3]. Genetic information is encoded as linear strings of base pairs along macromolecules (such as DNA). The information is processed (replicated, transcribed, checked) by enzymes moving along the molecule as represented in Figure 1(a). Naturally the chemical processes are exceedingly complex, involving the 3-dimensional topologies of the molecules concerned. At the conceptual level however, we can see that the structure has a number of advantages for computation including its regularity, one-dimensional parallelism, and locality of communication.

It is not difficult to make the transition from the biological domain to that of the finite state machines (FSM) shown in Figure 1(b). This structure is comprised of Turing machines moving along a fixed tape, their heads reading and writing symbols on the tape. This organization, although retaining the desirable properties described above, is impractical to implement.

In the structure shown in Figure 1(c), the tape is replaced by a shift register (or *stream*) and the machines (or *cells*) remain fixed in place. Symbols move along the stream and pass under the read and write heads of subsequent machines, which recognize symbol sequences and act upon them according to their state transition sequences.

This structure is clearly well matched to VLSI implementation. There are only two fundamental component types, cells and stream elements. Communication between these elements is local. The number of machines and storage elements may be made large, and the stream fed back to form a loop. Symbols may then pass any number of times around this loop until the computation is complete.

The critical issue then becomes, how can we perform practical general computation with such a structure? How can such simple entities as the finite state machines act in concert without central orchestration to execute a complex task? These issues will be addressed in the next section.

0-7803-0849-2/92 $3.00

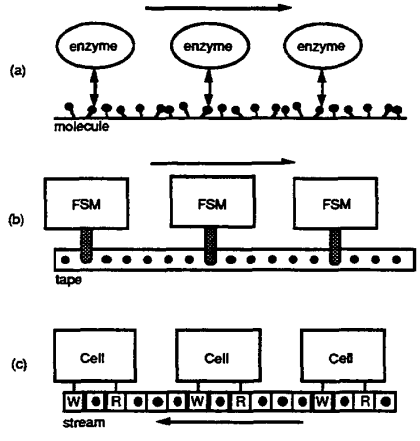

Figure 1: (a) Enzymes traverse a macromolecule performing simple information processing on the symbols represented by base pairs. (b) Finite state machines move along a fixed tape, reading and writing symbols. (c) Symbols are shifted through the read and write heads of fixed finite state machines.

3 Language First

We require a programming paradigm that supports general and parallel computation, and is applicable to our simple structure. Turing machines, though simple and apparently computationally complete [4], are inefficient. Considering a "language first" approach [5] we notice that *functional languages* are both general and ease the extraction of fine grained parallelism [6,7].

It remains to find a functional language implementation technique based on a few very simple rules that can be executed by our finite state machines. Such a technique has been described by Turner [1]: the use of combinators as an execution mechanism for functional languages. The basic combinators denoted $\mathcal{S}$, $\mathcal{K}$ and $\mathcal{I}$ may be regarded as productions that perform simple manipulations of their arguments:

$$\begin{aligned} \mathcal{S}\, f\, g\, x &\rightarrow f\, x\, (g\, x) \\ \mathcal{K}\, c\, x &\rightarrow c \\ \mathcal{I}\, x &\rightarrow x \end{aligned}$$

$\mathcal{S}$ requires a replication and an exchange ($fgx \rightarrow fgxx \rightarrow fxgx$) and the other two only require deletions.

A combinator version of a program may be compiled from the functional language form by translation to the lambda calculus, and the subsequent application of simple recursive algorithmic transformations [8].

In the process of compiling to a combinator form variables are removed from the function body. Instead of naming the variables inside the function, the new combinators that comprise the function serve to replicate and exchange the variables in order that they reach their intended destinations. These destinations are the tests, logic, and arithmetic functions that comprise the heart of the computation. By analogy, we may think of the familiar stack based reverse Polish calculator: a program on such machine does not use named variables. Variables are placed on the stack then replicated and swapped into positions such that operators may be applied. Although it does not involve a stack, the combinator technique may be regarded as a structured way of translating a general function into a form with no variables—only replications, exchanges and deletions that serve to "bubble" data towards destinations.

An additional aspect of the combinator expression is that within the data movement operations provided by the combinators, there is considerable achievable parallelism—in a complex expression there may be a number of combinators ready to execute simultaneously.

These properties of combinator execution suggest that the combinator formulation is appropriate for parallel execution by our simple machine. It does however lack one important feature. The shift register should never have to "back up" so that symbols are only shifted in one direction. As illustrated in Figure 2 the cell can only read the data one symbol at a time in strict sequence. In the following section we describe an extension to combinator notation that allows for unidirectional, parallel execution suitable for use in this simple machir

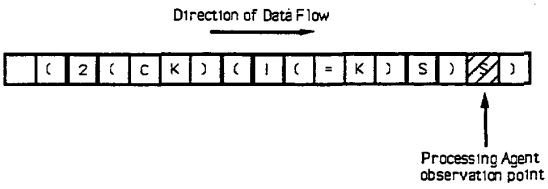

Figure 2: The processing agent can only read a single symbol at a time. At the time the $\mathcal{S}$ is read, it is not possible to determine its number of arguments. If execution begins, but there were too few arguments, the head of the computation would already have proceeded irretrievably out-of-reach to the right.

4 Extended Combinator Notation

In our Extended Combinator Notation (ECN) a count of the number arguments that are associated with the combinator is maintained adjacent to the combinator itself. This figure is initialized at compile time, and continually updated during execution. The essential property of this process is that it may be performed without ever going back to *count* the arguments. Rather, as a particular combinator is executed, a simple set of rules are used to update the argument counts of the combinator phrases that appear as arguments to the currently executing combinator.

In fact there may be any number of arguments associated with a combinator expression: however the executing combinator only touches those it requires. Thus in the general case, where there are a total of n arguments to the $\mathcal{S}$ combinator (as denoted by the superscript):

$$\mathcal{S}^n f g x\ k_1 k_2 \ldots k_{n-3} \rightarrow f x (g x)\ k_1 k_2 \ldots k_{n-3} \tag{1}$$

k_1 to k_{n-3} are unaffected by the $\mathcal{S}$. Internally, both f and g are potential combinator expressions of the form:

$$\mathcal{F}^i a_1 a_2 \ldots a_i \text{ and } \mathcal{G}^j b_1 b_2 \ldots b_j$$

where $\mathcal{F}$ and $\mathcal{G}$ are combinators. We are interested in updating the argument counts i and j to reflect the new positions of $\mathcal{F}$ and $\mathcal{G}$ after the execution of the $\mathcal{S}$ to which they were arguments. By examination of (1) we find the following rules are sufficient for updating argument $\mathcal{S}$ counts: $i \leftarrow i + n - 1$ and $j \leftarrow j + 1$ Thus the $\mathcal{S}$, $\mathcal{K}$, and $\mathcal{I}$ transformations in ECN may then be written:

$$\begin{aligned}
\mathcal{S}^n\ \mathcal{F}^i a_1 a_2 \ldots a_i\ \mathcal{G}^j b_1 b_2 \ldots b_j\ x\ k_4 k_5 \ldots k_n &\rightarrow \mathcal{F}^{i+n-1} a_1 a_2 \ldots a_i\ x\ \mathcal{G}^{j+1} b_1 b_2 \ldots b_j\ x\ k_4 k_5 \ldots k_n \\
\mathcal{K}^n\ \mathcal{C}^i c_1 c_2 \ldots c_i\ x\ k_3 k_4 \ldots k_n &\rightarrow \mathcal{C}^{i+n-2} c_1 c_2 \ldots c_i\ k_3 k_4 \ldots k_n \\
\mathcal{I}^n\ \mathcal{F}^i a_1 a_2 \ldots a_i\ k_2 k_3 \ldots k_n &\rightarrow \mathcal{F}^{i+n-1} a_1 a_2 \ldots a_i\ k_2 k_3 \ldots k_n
\end{aligned}$$

Although these rules appear intricate, execution is in fact very simple. All that is required is a means of replicating, exchanging and deleting the symbols, and a small state machine and adder combination to update the argument counts.

In extended combinator notation we have found a programming paradigm for our simple machine. A functional program can be compiled to an ECN representation, and injected into the stream. Cells execute at least the basic three combinators (or an extended, more efficient set [8]). In the following sections we address some of the issues that arise when we actually define such a machine.

5 Structure: the Stream and Cells

There are two major components of the simple machine: stream elements and cell elements. Symbols (code and data) move along a stream comprised of many shift register-like elements. The stream is split into multiple segments, each connecting two adjacent computational elements, or cells. We shall assume that each stream element is large enough to store a single symbol (say eight bits).

The stream section that connects two cells must have the following properties:

1. The stream must act as a FIFO buffer, accepting symbols from the upstream cell and passing them to the downstream cell. This allows the upstream cell to continue generating symbols if the downstream cell is busy.

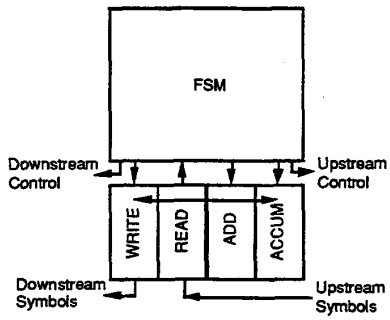

Figure 3: A simplified view of a cell.

2. The stream must be able to replicate and exchange combinator arguments under control of the upstream cell.

3. Allowance must be made for the fact that arguments may exceed the fixed size of a stream section. Therefore upstream sections must be able to *usurp* downstream sections, combining their length.

These functions can be implemented with a pair of simple first-in, first-out (FIFO) buffers. Data items may be *replicated* by copying each symbol to both FIFOs as it emerges from the upstream cell. Data items may be *exchanged* in the stream by storing the leading item in the first FIFO while the other is fed into the second. The FIFOs may then be read out in reverse order, effectively exchanging the order of the data items in the stream.

To allow for arbitrary sized strings to be processed, a FIFO may (when it fills) wait till the next FIFO downstream becomes free. A simple handshaking mechanism then allows the downstream FIFO to become an effective extension of the upstream usurper.

The cell structure is illustrated in Figure 3. The finite state machine controls the recognition of combinators and the sequencing of operations thereon. The adder acts under control of the FSM to calculate new argument counts as described in section 4. The current argument count is required for the parsing of each combinator expression, and is kept in the accumulator. The symbol under the read head is input to the FSM and the adder, and the value under the write head can be overwritten by the FSM or the adder. Only one add operation is required per shift of the stream under our current protocol: the stream contains alternating symbols representing combinators and argument counts.

In addition to the combinator-like operations outlined above, we must be able to perform arithmetic and logical functions on stream data. These operations differ from the combinator-like operations as they are *strict*: they cannot execute until the arguments have been fully reduced (whereas combinator operations may be performed irrespective of the condition of their arguments). Additional operation specific cells with lookahead capability are distributed along the stream to perform these tasks.

6 Simulation

We have developed a simulation of the simple machine. Functions are written in functional Lisp and compiled to ECN. Running on a Lisp Machine, the interpreter performs about 1000 reductions/minute. The program monitors factors such as peak and average parallelism, code size and instruction mix. Table 1 shows results from four functions. The "Cycles" column indicates the total number of evaluation cycles required, which is a measure of run time. Note that levels of parallelism are quite high and code size modest. The second running of the `fact` (factorial) function was performed with an unguarded IF conditional, resulting in fully eager evaluation. This leads to a significant increase in parallelism and a consequent reduction in total cycles of approximately 25%.

Some programs show a very large growth in number of reductions with argument size. A recursive reversal of a four element list for example requires 9972 reductions. The average level of parallelism also grows correspondingly however to 130 (with a peak parallelism of 1215), requiring 77 cycles for completion.

Table 1: Simulation Results

Function	Reductions	Ave Paralellism	Max Paralellism	Cycles	Max Size
`sum`	454	6.4	13	71	767
`member`	292	4.4	9	67	469
`map`	477	8.2	18	58	784
`fact`	128	3.1	6	41	208
`fact`[a]	138	4.8	10	29	269

[a] With eager evaluation.

7 Implementation

As noted earlier, the simplicity and regularity of the architecture suggest that it is well suited to VLSI implementation. An additional advantage is that when partitioned into chips, the architecture only requires a one symbol input from the upstream chip, and a one symbol output to the downstream chip (excluding handshaking, power, clocks, etc). This is an unusually low I/O requirement.

We have recently designed and fabricated a high speed SRAM block for use in the FIFO implementation. The pipelined 64Kbit block performs an atomic read and write operation to a single address every 10ns, promising a throughput of 10^8 symbols per second.

8 Conclusion

The extension to combinator notation that we have described permits parallel execution of code travelling in a unidirectional stream. The simple rules required to implement this model of computation may be executed by finite state machine based cells placed along the data stream.

By moving away from more conventional graph-based reduction [9,10] and adopting a string-based [11] architecture we have constrained communication to be local, and hence fast. In addition, our linear architecture is far more efficiently partitioned than that of tree-based reduction engines [12].

Our simulations have verified that this is a viable execution model, and that it may be implemented on a highly pipelined, simple machine. We are presently continuing with further simulation and test fabrication of VLSI elements.

References

[1] D. A. Turner, "A New Implementation Technique for Applicative Languages," *Software–Practice and Experience*, 9, 31-49, September, 1979.

[2] J. D. Watson, *The Molecular Biology of the Gene*, W. A. Benjamin, Menlo Park, 1975.

[3] J. R. Sampson, *Biological Information Processing: Current Theory and Computer Simulation*, Wiley, 1984.

[4] M. L. Minsky, *Computation: Finite and Infinite Machines*, Prentice Hall, 1967.

[5] J. R. Kennaway and M. R. Sleep, "The 'Language First' Approach," in *Distributed Computing*, Academic Press, 1984.

[6] J. Backus, "Can Programming be Liberated From the Von Neumann Style? A Functional Style and its Algebra of Programs", *Communications of the ACM*, 613–641, August 1978.

[7] W. B. Ackerman, "Data Flow Languages," *IEEE Computer*, February 1982.

[8] S. L. Peyton Jones, *The Implementation of Functional Programming Languages*, Prentice Hall, 1987.

[9] M. Scheevel, "NORMA: A Graph reduction Processor," in *Proc. Conference on Lisp and Functional Programming*, 1986.

[10] M. Crips, T. Field, and M. Reeve, "An Introduction to ALICE: a Multiprocessor Graph Reduction Machine," in *Functional Programming*, S. Eisenbach (ed), John Wiley, 1987.

[11] K. J. Berkling, "Reduction Languages for Reduction Machines," in *Proc. 2nd Annual Symposium on Computer Architecture*, 1975.

[12] G. A. Magó, "A Cellular Computer Architecture for Functional Programming," in *Proc. IEEE COMPCON*, 1980.