The Monadic Processor

Felipe Afonso de Almeida
Instituto Tecnológico de Aeronáutica
Divisão de Ciência de Computação
Departamento de Engenharia de Software
São José dos Campos - São Paulo

Abstract

In a straightforward pipelined implementation of the Explicit Token Store dataflow model there is an imbalance concerning the execution of *dyadic* instructions (instructions with two input operands) . This is due to the fact that two tokens need to be processed for a dyadic instruction before valid operands are available to be processed by the Arithmetical-logical Unit, which at each pipeline cycle can consume two tokens and produce as a result also two tokens.

In this work we investigate an approach to increase the Arithmetical-logical Unit utilization rate in dataflow processors based on the Explicit Token Store model. We propose an abstract realization of this model where there is two tokens queues. One token queue keeps tokens headed for *monadic* instructions (instructions with a single input operand). These tokens are utilized whenever the main path of the processor pipeline (feeded directly by the two token queues) is unable to produce valid operands for the Arithmetical-logical Unit to process. Therefore, increasing the utilization rate of the Arithmetical-logical Unit. The other token queue holds tokens headed for *dyadic* instructions (instructions with two input operands).

1 Introduction

The execution of a dyadic instruction involves the processing of its two input tokens. The Explicit Token Store (ETS) dyadic matching function (refer to [greg90] for a complete discussion of the ETS model) implies that each time during the execution of an instruction the presence-bits (of an ETS store location) is found in the *empty* state, further processing of the instruction is aborted. In the Monsoon realization of the ETS model [Greg88], [Greg90], [Greg91] this is realized by a "bubble" or no-operation that is submited to the ALU stage of the processor element pipeline. Whenever this situation occurs the ALU and subsequent stages of the pipeline are idle; thus no useful work is performed. Therefore the actual rate of utilization is at best 50% if the program consists

mainly of dyadic instructions and on average two output tokens are produced. However due to the presence of monadic instructions, and instructions with a constant input in real programs the utilization rate reaches about 70% [Greg88].

This issue can be viewed simply as a pipeline imbalance. At each pipeline cycle of the ETS processor the execution stages can consume two values and produce two results, while the operand-matching stage takes two cycles to consume the two input tokens for a dyadic instruction. From this point of view an increase in the ALU utilization may be accomplished by increasing the throughput of the slower stages. A more detailed discussion of this issue is found in [Brobst86] and [Greg88].

Our approach is to increase the ALU utilization by providing an alternative source of input operands whenever the main pipeline is unable to generate valid input data values. This implies the use of two token queues. The central idea is to detect as early as possible that further processing of a dyadic instruction will not take place and use any *monadic* tokens (i.e. tokens whose destination are monadic instructions) that are available in the token queues to feed the ALU.

The Monadic processor is an implementation of an ETS pipelined processor. It adopts the two token queues approach of the Monsoon realization of the ETS model [Greg90] and has additional stages to provide the ALU with unprocessed monadic tokens. These tokens are stored into a private token queue (one of the two queues provided).

It should be pointed out that the Monadic processor is an abstraction of a dataflow processor element. It is aimed at providing a platform to investigate how effective the use of unprocessed monadic tokens can be in increasing the ALU utilization. A specific software emulator was built to carry out the investigation. Its implementation, is discussed in [Felipe92].

The article is organized as follows. In section 2 we briefly discuss the ETS model by means of an example. In section 3 and 4 we discuss the arquitecture of the monadic processor . In section 5 the results of the emulation are presented and analized. Finally in section 6 we present the conclusions of the work.

2 The ETS Model

Figure 1 shows an example of a pipelined implementation of an ETS processor. Our discussion is drawn from [greg90]. In this case instruction fetch is done before operand matching since the instruction encodes the offset r of the rendezvous location in the activation frame pointed by the income token c field (the Token Store is augmented with presence-bits to indicate the presence of a partner in the case of dyadic instructions).

Then the operand matching stage queries the presence bits on location $c+r$ of the Token Store. If the slot is empty the token's value is written into the slot and a bubble or no-operation is submitted to the ALU (in this case further processing of the instruction does not take place). On the next stage the bubble

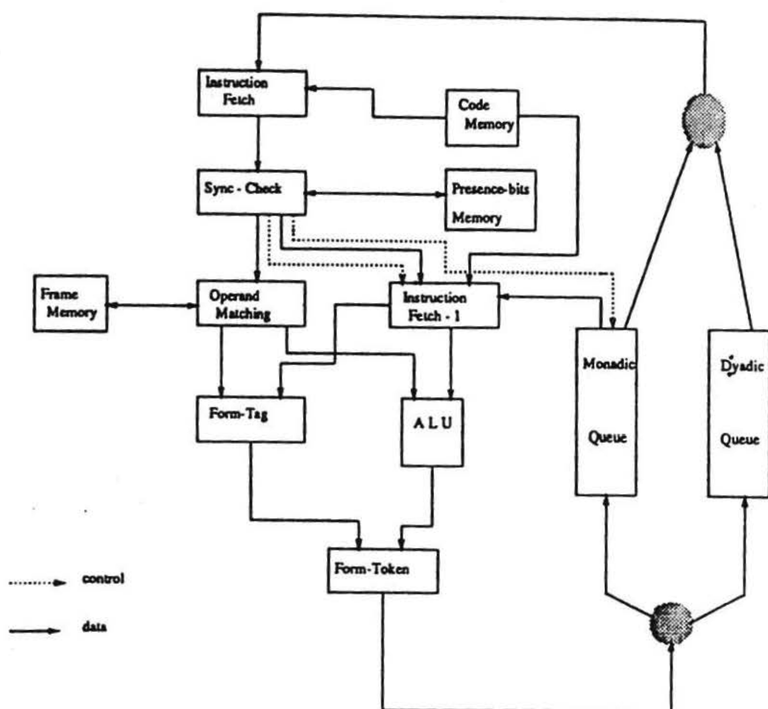

Figure 2: The Monadic Processor

propagates to the form token stage wich in turn does not insert any token into the token queue.

If the slot is full the operand from the slot is extracted, and this operand along with the value on the token being processed are submitted to the ALU. The destination tags are computed in the form tag stage in parallel with the ALU operation. Finally new tokens are constructed in the form token stage which concatenates the destinations from the form tag and the results from the ALU. The new tokens are inserted into the token queue.

3 The Processor Element Architecture

The circular pipeline of the Monadic processor element is depicted in Figure 2. It comprises five stages: instruction-fetch (IF); synchronisation-check (SC); operand-matching (OM) and the extra instruction-fetch (IF1) (operating in parallel); ALU and form-tag stage (FTA) (operating in parallel); and the form-token stage (FT). The three memories are all local. The Token Store of the ETS model is splited in two separated memory: one keeps the presence-bits and the other an operand value (it is called *frame memory*).

It is a synchronous non-blocking ("systolic") pipeline. At each pipeline cycle one token is processed by the IF stage and zero, one or two tokens are produced by the FT stage and stored in one of the token queues. The SC stage has exclusive access to the local presence-bits memory, and the OM stage to the local frame memory.

One of the token queues is defined as a high priority queue and the other as a low priority one. Tokens heading for monadic instructions are always stored in the low priority queue (the *monadic* queue, see Figure 2). Dyadic tokens are always stored in the high priority queue (the *dyadic* queue, see Figure 2). Tokens are processed from the lower priority queue by the IF stage only when the higher priority queue is empty. However a token from the monadic queue may be processed by the IF1 stage. Whenever the execution of a maching function by the SC stage determine that further processing of the instruction is not to take place an additional signal (*sync-non-achieved*) is transmitted to the OM stage and to the monadic queue. At the very next pipeline cyle a token dequeued from the monadic queue is processed by the IF1 stage, producing a valid operand for the ALU. In parallel the OM stage stores the value part of the token received from the SC stage into a frame memory location, but no ALU operand are produced.

One field of the dataflow instruction is expanded to accomodate two new subfields: S1 and S2. They are use by the FT stage to identify from the tokens produced whether they are destined for a monadic instruction (Si=0) or for a dyadic one (Si=1).

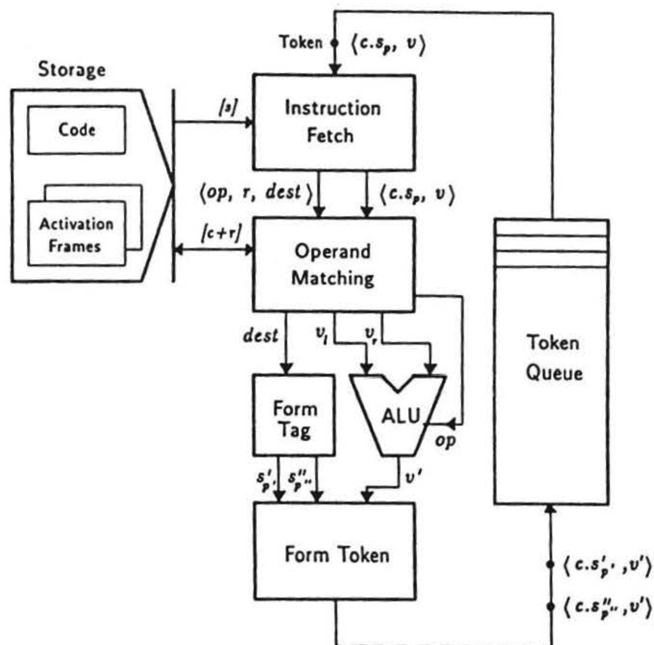

Figure 1: The ETS Pipelined Processor

4 The Pipeline Operation

At each pipeline cycle the IF stage receives one token from the high priority queue - unless it is empty, in which case one token is dequeued from the low priority queue. An instruction is fetched from the instruction memory and both instruction and the income token are dispatched to the next stage.

The SC stage operates on the presence-bits of a location in the presence-bits memory (see Figure 2) as dictated by the matching function specified in the instruction (for more details refer to [Greg90]). If a *monadic* matching function is specified no access is made to this memory (refer to [Greg90]) and the instruction and the income token received from the previous stage are forwarded to the OM stage. In the case of a dyadic instruction a matching-address is computed and used to read a location in the presence-bits memory. The matching function dictates a transition state to be performed on these bits and a result is written back in to same location. If the execution of this function indicates that further processing of the instruction is not to take place a signal (*sync-non-achieved*) is sent to the monadic queue to dequeue a token and transmit it to the IF1 stage. The matching-address, the instruction, the income token and a signal (*sync-non-achieved*) are transmitted to the OM stage. If the execution of the instruction dictates that it is to be proceed with its execution no signal is transmitted to the OM stage and to the monadic queue. The OM stage receives the matching-address, the instruction and the income token.

The OM stage behaves similarly to the ETS pipeline except that it does not access the presence-bits memory. The syncronization role of the matching function is played by the SC stage. The *sync-non-achieved* signal sent by that stage (or its absence) and the operation (determined by the matching function specified in the instruction) to be performed in a location of the frame memory completely determines the action to be executed by this stage.

The IF1 stage on receiving a token from the low priority queue fetches the corresponding instruction and transmit the relevant information to the ALU and the FTA stages.

When the IF1 stage is idle the OM stage is either processing a monadic token or a dyadic one whose partner is already stored in the frame memory. On the other hand whenever the IF1 stage receives a token (from the monadic token queue), the OM stage just operates on a frame memory location. It does not transmit any data to the following stages.

The operation of the FTA, the ALU and the FT stages are as defined for the ETS pipeline implementation as discussed in section 2. However tokens produced by the FT stage are stored into the two tokens queues as discussed early.

5 Results

In this section we analyse the ALU utilization of the Monadic processor A software emulator, designed and built to exercise the monadic and ETS architecture with benchmark programs, were used. The emulator is implemented in OCCAM 2, and runs on an array of transputers on a Meiko Computing Surface [Felipe92]. Eeach stage that comprises the processor pipeline runs on a separate transputer. However the instruction memory is emulated on a dedicated transputer and it is shared by the two instruction-fetch stages. The monadic and the dyadic token queues are both LIFO queues. A single emulated I-memory module* is used to hold the I-structures for the MM, and wave programs.

The benchmark programs are a factorial , a Matrix Multiplication (MM) and a Wave Computation [Nikhil89]. The two first programs are straightforward. We then discuss the third algorithm.

5.1 The Wave Computation

The overall OCCAM 2([Inmos88])code for the wave computation algorithm is depicted below:

```
PROC wave ( [] [] INT c )
  VAL INT n IS SIZE c:
   -- assume: (n = SIZE c [0], i.e. a square matrix)
  SEQ i = 0 FOR n
    SEQ j = 0 FOR n
      IF
        (i = 0) OR (j = 0)
          c[i, j] : 1
        TRUE
          c[i, j] : = c[i-1, j] + c[i, j-1]
:-- end-of-wave-proc
```

In this program both loops can be unravelled, exposing the maximum parallelism available in the algorithm. However there is data dependency of data structure elements among iterations. The dependencies are illustrated in Figure 3 where the computation of the couple-shaded elements depend on the availability of the same element, respectively X, Y, Z. The storage elements which store the result matrix (implemented as an I-structure [Arvind87a]) will receive read requests for elements not yet calculated. These requests are automatically deferred by the I-storage elements [Arvind87b], until the corresponding operation is performed. The program execution involves the use of the synchronization operation between writes and reads performed on I-structure elements. The

*I-memories are data memories (refer to [Arvind 87b]) that holds I-structures. These structures have non-strict access and were defined for the ID language. For further details see [Nikhil90].

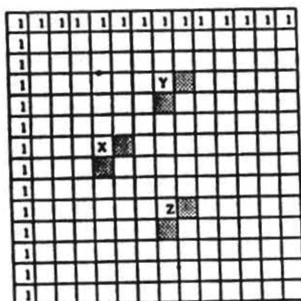

Figure 3: Data Dependence in The Wave Program

deferred read operations will be satisfied later on, at non-deterministic time, generating (possibly) some burst of tokens from the memory element involved.

The parallel behaviour of the program is illustrated in Figure 4. The shared squares represent the elements that can be calculated in parallel at that particular time. It can be seen that the computation advances in parallel along a diagonal wavefront. The parallelism peaks at the main matrix diagonal, decreasing afterwards (compare this with the sequential order of execution effected by the above OCCAM 2 code, that advances one element at a time row-by-row).

5.2 Results and Instructions Profiles

The ALU utilization rate is defined as: ALU Utilization rate: U

$$U = \frac{S}{B}$$

where S is the number of instructions executed by the ALU plus the number (B) of bubbles generated. In this section all the utilization rate values are presented in percentage.

Figure 5 shows the results obtained for the factorial program running on the ETS processor element and on the Monadic processor. For the first architecture the utilization rate is kept around 70% . This is as expected, since about 60% of the total amount of instructions executed are monadic instructions (see the instruction profile displayed in Figure 6).

Figure 4: The Wavefront Computation

Figure 5: Percentage of ALU Utilization against Problem Size(N) for The Factorial Program

Factorial	size N	% Dyadic Instructions Executed	% Monadic Instructions Executed
	2	38.18	60.82
	8	39.84	60.16
	32	39.96	60.03
	128	39.99	60.00
Matrix Multiply	**size NxN**	**% Dyadic Instructions Executed**	**% Monadic Instructions Executed**
	2	31.29	68.70
	4	31.71	68.21
	8	31.62	68.37
	16	31.49	68.5
	23	31.43	68.56
Wave	**Size NxN**	**% Dyadic Instructions Executed**	**% Monadic Instructions Executed**
	2	33.45	66.54
	4	34.38	65.61
	8	34.58	65.41
	16	34.54	65.45
	20	34.54	65.45

Figure 6: Instruction Profiles for The Benchmark Programs

The curve for the Monadic processor shows that the ALU utilization can be effectively increased if monadic tokens (held on a separate token queue) are used as operands, when a matching does not take place in the pipeline operand-matching stage (OM). For problem size N = 8 the utilization rate achieved is over 90% and even higher rates are obtained (over 95%) for larger problem sizes. We should note that after an initial rise the curve levels off when the problem size is larger than N = 8. This behaviour is better understood if we look at the values presented in Figure 7. We note that for a four fold increase in the problem size there is a corresponding increase (approximately of the same proportion) in the number of operations executed (including the processing of bubbles, which constitutes a null operation), by the ALU. However the number of bubbles generated does not increase at the same proportion and decreases (as much as 33% from N = 8 to N = 128) for problem size larger than N = 8, Therefore the ALU utilization rate increases from about 82% to over 98% for problem sizes in the range of N = 2 to N = 32 and as expected very slowly for larger data sizes.

The results for the MM and wave programs are presented in Figure 8. For the ETS architecture the utilization rate achieved is higher than 71% . In this case, the percentage of monadic instructions is about 68% for the MM and 66% for the wave program, as shown in Figure 6. This explain the slightly better results compared to the rates obtained by the factorial program.

Size N	*Operations Executed*	*Bubbles Generated*
2	89	16
8	401	28
32	1597	24
128	6394	21

Figure 7: Factorial Profile: Operations Executed and Bubbles Generated

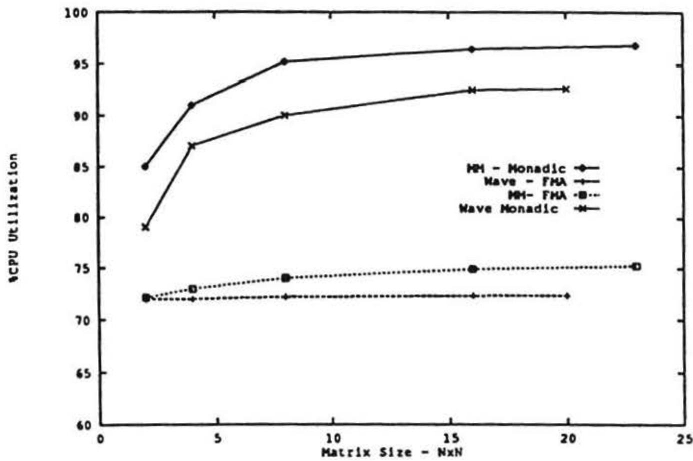

Figure 8: Percentage of ALU Utilization against Matrix Size for The MM and Wave Programs

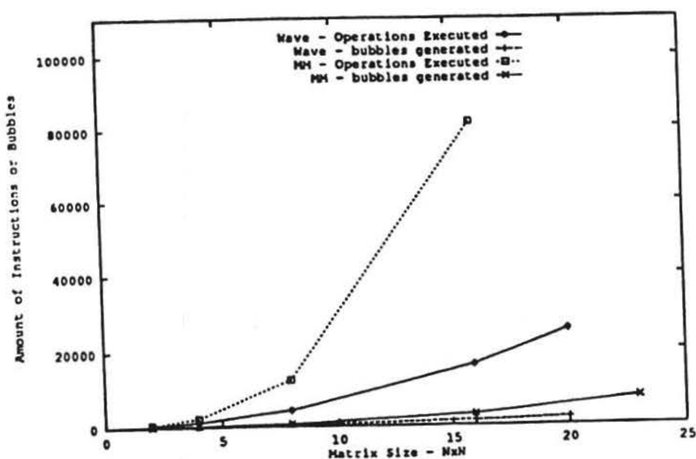

Figure 9: Instruction and Bubles Generated against Problem Size - MM and Wave Programs

The results obtained for the Monadic processor demonstrate that the architecture is also effective in improving the ALU utilization rate for iterative programs (with and without I-structure data dependency among iteractions). Rates higher than 90% are achieved for both benchmark programs, when the problem size is larger than N = 16. We note that the MM program performs better than the wave program throughout the sampled range. The rising shape observed for the curves are due to the fact that the amount of bubbles generated growns at very low rates compared with the rates at which the number of operations (including the processing of bubbles - a null operation) executed by the ALU grows (see Figure 9). From the results above we can conclude that the monadic token queue occupancy is sufficiently large to keep the ALU busy for almost all the occasions when the conventional pipeline data path does not provide an operand. This can be attributed to:

A) The higher proportion of monadic instructions executed. What means that just a small amount from the total of tokens produced needs a partner.

B) That the number and length of sequences of partially executed instructions (i.e. the processing of the instruction stops in the operand-matching stage of the processor pipeline) produced are not too large. This means that fewer monadic tokens must be present in the monadic token queue to keep the ALU busy. We should note that three factors contribute to the reduction in the

number and size of these sequences:

- Both token queues are implemented as LIFO queues. This policy favours a depth-first order of execution of a code block graph and of the process execution tree of the programs because priority is given to the last produced token. For recursive programs these tokens are the most important, which reinforces locality, thus decreasing the number of partially executed instruction produced. For iterative programs this phenomenon is not always observed . However a separate dyadic token queue reduces the chances of an important token staying at the bottom of the queue for long time.

- The processing of unmatched tokens does not preclude the concurrent execution of monadic instructions. Therefore any important dyadic token produced by such instructions are readily available to be processed.

- In the programs a large amount of instructions, that produce two results tokens consists of monadic identity instructions (instructions that just copy as its output token its input token). However the execution of a sequence of these instructions (i.e. there is a sequence of tokens stored in the token queue whose destination is an identity instruction) is interrupted as soon as a dyadic instruction is ready to be further processed. This provides some control on the exposition of parallelism during the execution of a program; thus decreasing the amount of unmatched tokens produced, and consequently the amount of partially executed instructions produced.

6 Conclusion

In a straightforward implementation of the ETS model, the ALU is bubbled each time the first-arriving operand of a dyadic instruction is written into an activation frame slot and further processsing of the instruction does not take place. Therefore for a program consisting only of dyadic operators, with two output arcs, the ALU is at best 50% utilized. We have found that due to monadic operators, and operators with a constant input, the rate achieved is around 70% for the factorial program, and slightly higher for the other two benchmark programs The results obtained for the Monadic processor shows that it is possible to increase the ALU utilization rate to over 90% (for larger problem sizes) by using a monadic token queue, which holds only monadic tokens, as an alternative source of operands to the ALU when it would otherwise be bubbled.

BIBIOGRAPHY

[Arvind87a] Arvind and R.S. Nikhil. Executing a Program on The MIT Tagged-Token Dataflow Architecture. In *Proceedings of the PARLE Conference,*

Eindhoven, The Netherlands,(Lecture Notes In Computer Science,volume 259), pages 1-29. Spring_Verlag, June 1987.

[Arvind87b] Arvind, R.S.Nikhil, and K.K. Pingale. I-structures: Data Structures for Parallel Computing. Technical Report - CSG Memo 269, Laboratory For Computer Science, Computer Structure Group, MIT, February 1987.

[Brobst86] S.A.Brobst. Instruction Schedulling and Token Storage Requirements in a Dataflow Supercomputer. Master's Thesis, Department of Electrical Engineering and Computer Science, MIT 1986.

[Felipe92] F.A.Almeida. *Parallel Software Emulation of Multi-processor Dataflow Machines on Transputer Networks*. PhD Thesis. Computing Laboratory. University of Kent at Canterbury, 1992.

[Greg88] G.M.Papadopoulos. *Implementation of a General Purpose Dataflow Multiprocessor*. PhD thesis, Dept of Electrical Engineering and Computer Science, MIT 1988.

[Greg90] G.M. Papadopoulos and D.E.Culler. The Explicit Token Store. *Journal of Parallel and Distributed Computing*, pages 189-308, December 1990.

[Greg91] G.M.Papadopoulos. *Implementation of a General-Purpose Dataflow Multiprocessor*. Research Monographs in Parallel and Distributed Computing. The MIT Press, Cambridge, Massachusetts 1991.

[Inmos88] Inmos Limited. *OCCAM 2 Reference Manual*. Inmos Limited, 1988.

[Nikhil89] R.S.Nikhil and Arvind. A Dataflow Approach to General Purpose Parallel Computing. Technical Report - CSG Memo 302, Laboratory for Computer Science, Computer Structure Group, MIT, June 1989.