A pipeline bubbles reduction technique for the Monsoon dataflow architecture

Feipei Lai AND Fong-chou Tsai

Department of Electrical Engineering &
Department of Computer Science and Information Engineering
National Taiwan University, Taipei, Taiwan, R.O.C.
E-mail:flai@cad.ee.ntu.edu.tw

Abstract

This paper proposes two types of new auxiliary matching stores (AMS) that can be applied to dynamic dataflow machines with a frame-structured memory. The first type of the auxiliary matching store achieves high matching rate by combining the LRU replacement algorithm and a simple match structure together. The second one achieves high match rate by controlling strictly the number of active processes with an explicit hardware. Owing to some special characteristics of dynamic dataflow computing, this temporary storage must be designed based on the following principles: (a) The number of the active processes must be limited. (b) The related instructions must be arranged nearby. (c) The replacement of the match slot must be deterministic.

1 Introduction

The most recent generation of dataflow machines (*e.g.*, MIT's Monsoon, ETL's EM-4[2], and Sandia's Epsilon-2 [3]) have shown how operands match can be accomplished with a simple hardware structure in two machine cycles. In Monsoon, the explicit storage for operands match is decided by the complier. It eliminates the associative matching and implicit storage allocation which are used by previous work [4]. However, this mechanism still incurs some pipeline bubbles. For each dyadic (two-operands) instruction, the memory slot for matching must be accessed twice. Only the second memory access can get the instruction fired because the first one will do nothing and leave the pipeline a bubble. As shown in [1], the possibility of the pipeline bubbles is 28.75%.

From the preceding discussion, it is quite obvious that reducing the pipeline bubbles needs an additional unit to share the burden of matching operations of the CPU. Here we use a fundamental property of the program behavior to gain the speedup. The fundamental property is the *locality*. We can futher divide the locality into two aspects; spatial and temporal[5]. Thoreson *et al.*[6] proposed models of instruction reference patterns in dataflow programs, and suggested potential spatial and temporal localities in dataflow environments. Tokoro *et al.*[7] presented a definition of the working set for dataflow machines based on simultaneous execution and the principle of (temporal) locality, and evaluated various combinations of fetch policies and replacement policies.

In dynamic dataflow computing, data dependency produces potential spatial localities. Specially, for efficiency, there are now many researches trying to get a sequence of instructions executed in order, even in the dataflow environment[1, 2, 3] which further enhances the potential spatial localities. However, temporal locality is not produced in the frame-structured operands memory. This is because an activity frame is allocated for each iteration of a loop or for a recurrence of a recurrent construct. In other words, a frame corresponds to a code block (process) such as one iteration of a loop or a function call. In the traditional imperative language, temporary locality comes from iterations of a loop or recalls of a function by using the same storages. However, in dynamic dataflow machine like Monsoon, many active processes can exist concurrently, either active or inactive therefore even spatial localities cannot be assured.

The study [10] on the cache memory for data flow machines is quite similar to our work. Although the ideas were applied on the dataflow cache implementation, many ideas can also be applied to the matching store. In our method, we use a less complicated mechanism and a simpler structure to implement. In the first type of the auxiliary matching store, we uses an easy replacement algorithm, a simple structure and several prioritized token queues. In this type of the matching store, we assume that the resource manager is capable of controlling the number of the activity frame dispatched. Our architecture can tolerate a slight variance in the number of the active processes.

The circuits of the second type are a little bit complicated. There is an explicit circuit to control the number of active processes, several matching blocks and prioritized token queue. In this type, we assume the instructions for interprocess parameter/return-

1060-3425/94 $3.00 © 1994 IEEE

value passing (linking instructions) have a special field to indicate if they are to be pushed into the controlling process queue (CPQ). We will present the rest of paper as follows. In section 2, we will take a look at Monsoon. In section 3, we will describe the design principles of an AMS in the dynamic dataflow environment. In section 4, our architectures will be presented in detail. In section 5, we will make a conclusion.

2 Monsoon

In order to get a general picture, dynamic dataflow execution will be described first, then, the whole architecture of Monsoon will be presented.

2.1 Dynamic Dataflow Execution

Dynamic dataflow execution is characterized by three properties: (1) the program presentation is a partial order of essential dependences, (2) instructions are scheduled based on availability of operands, and (3) iteration and recursion are supported in full generality. Dynamic dataflow execution is formalized as rules for propagating tagged data tokens through dataflow graphs[11]. A node fires when tokens with identical tags are present on the input arcs and produces result tokens on the output arcs. The tagged-token instruction scheduling paradigm supports a nonblocking processor pipeline that can overlap instructions with closely related or completely unrelated computations. Thus, parallelism can be exploited at all levels. It also provides a graceful means of integrating asynchronous and potentially out-of-order memory response and synchronization events into the normal flow of execution, allowing communication latency to be masked by excess parallelism[15]. Tagged-token dataflow architectures, developed at MIT[4], Manchester University[12], and the Electrotechnical Laboratory[13, 14], approximate this model quite closely. In these machines, the dataflow firing rule is realized by a sophisticated hardware *matching store*, essentially a large associative memory. When a token arrives at a processor, the tag it carries is compared against the tags of tokens in the matching store. If no match is found, the incoming token is added to the store. If one is found, the matching token is extracted and the corresponding instruction is enabled for execution, eventually producing new tagged tokens. The tag serves as a name for the synchronization point of two values destined for the same instance of the same instruction. Each loop iteration or function invocation must be provided with a new set of synchronization names. However, the matching operation places considerable complexity on the critical path of instruction processing.

Owing to the matching store problem in the tagged-token architecture, it inspires the development of the ETS (Explicit Token Store). The ETS approach shifts much of the low-level storage management burden associated with dataflow execution to the complier in order to simplify the hardware system. The arcs in the code block, (i.e., the local variables of the function) are statically mapped onto slots in the frame by coloring the graph[17]. Each instruction specifies the location of its operands, as a simple effective address calculation, so no matching is required. When a code block is invoked, the caller dynamically allocates an *activation frame*, thereby providing local storage for the activation.

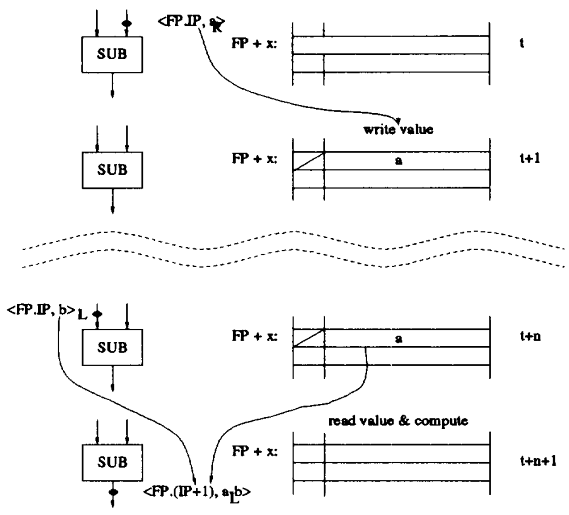


Figure 1: Explicit dyadic matching operation.

Token			
TYPE 8	TAG 64	TYPE 8	VALUE 64

TAG			
X 8	IP 24	PE 8	FP 24

Figure 2: The format of the token.

Each frame slot has associated presence bits specifying the disposition of the slot. The dynamic dataflow firing rule is realized by a simple state transition on these presence bits, as illustrated in Figure 1. At time *t*, the first token with data *a* arrives at the address FP+x. The slot is found empty, so the value on the token is deposited in the slot (making it full) and no further processing of the instruction takes place. At time *t+n*, the second token with data *b* arrives and the slot is full, so the value is extracted (leaving the slot empty) and the corresponding instruction, *sub*, executes and produce one or more new tokens. In general, the order of arrival of tokens is indeterminate, so the first token may be destined for either port. Initially, all slots in a frame are empty and upon completion of the activation they will return to that state.

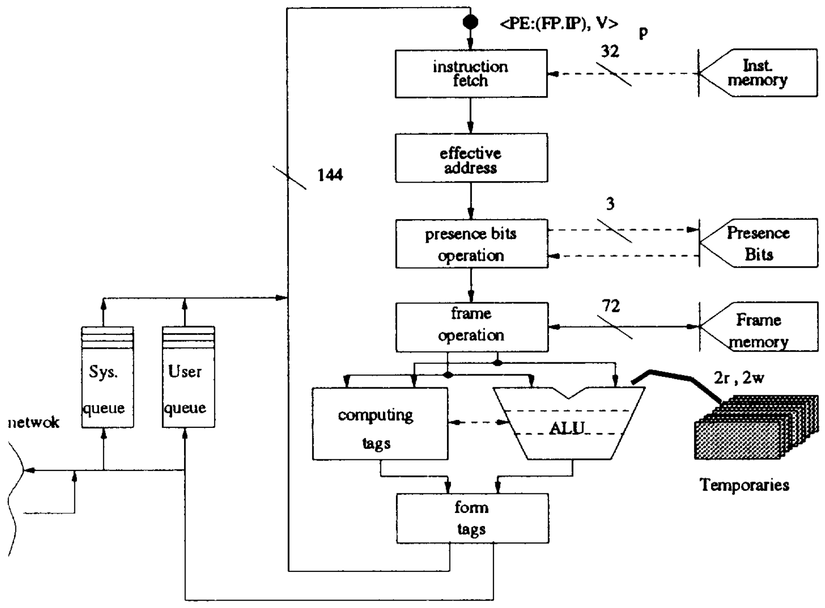


Figure 3: The Monsoon pipeline.

2.2 Architecture

Monsoon is a general purpose multiprocessors system which incorporates an explicit token store. A Monsoon machine comprises a collection of highly pipeline processing elements (PE's) connected via a multistage packet switch network to each other and to a set of interleaved memory modules (IS's) that support I-structure storage[16] as well as imperative storage. Messages in the interprocessor network are tokens and request tokens –precisely the same format used within the PE and IS. Thus, the hardware makes no distinction between interprocessor and intraprocessor communication.

2.3 Token

The Monsoon tags and values are 72-bit quantities comprising 8 bits of hardware type information and 64 bits of data, as shown in Figure 2. A token is a tag-value pair, 144 bits in size. A value can be a 64-bit signed integer, an IEEE double precision floating-point number, a bit field or a boolean, a data memory pointer, of course, a tag. A tag encodes two pointers: a pointer to the next instruction to execute, IP, and a pointer to the activation frame, FP, that provides the context in which the next instruction is executed. On Monsoon, a given activation frame resides entirely on a single processing element.

2.4 Pipeline Operation

Figure 3 is a detailed view of the eight-processor pipeline stages. A token can be inserted into the pipeline every cycle. After a delay of eight cycles, zero, one, or two tokens emerge from the bottom.

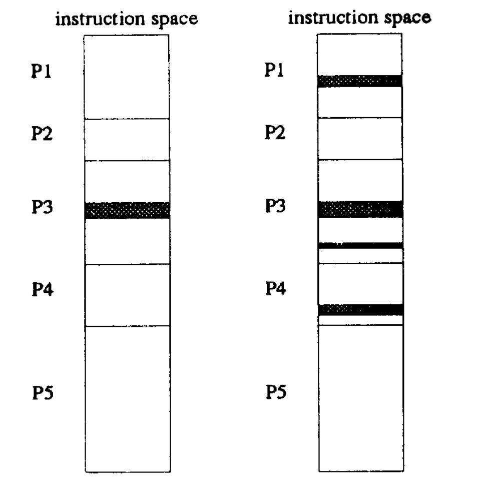
3 Design Principle

In a dynamic machine like Monsoon, many code blocks can exist concurrently, either active or inactive therefore even spatial localities cannot be assured. To cope with this problem, there are several design principles to be followed. Prior to discussing the details of design principles, several basic ideas and possible implementations of the auxiliary matching stores are presented.

The first idea is instruction space. Here, we build a control flow model which constructs a traditional program or a dataflow graph which constructs a dataflow program as a continuous space. Each instruction will occupy one piece of the space called a slot. A program can comprise many subprograms (subroutines). For convenience, a subprogram can be abbreviated as P, see Figure 4.

In this paper, we consider a process as a code block and vice versa, where a process could be a subroutine or an iteration of a loop.

The primary difference of the working space, gray areas shown in Figure 4, between traditional programs and dataflow programs is that the working space of traditional programs is continuous while in dataflow programs is fragmentary as shown in Figure 4. If the loop is considered, the situation of the working space of dataflow programs will become more complicated, as shown in Figure 5. In Figure 5, P3 corresponds to



(a) The traditional program.(b) The dataflow program

Figure 4: The instruction space.

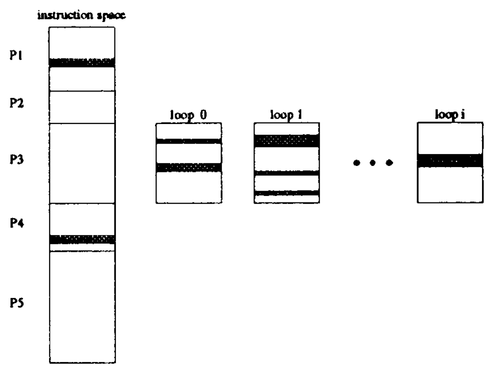


Figure 5: The loop mapping in dynamic dataflow program.

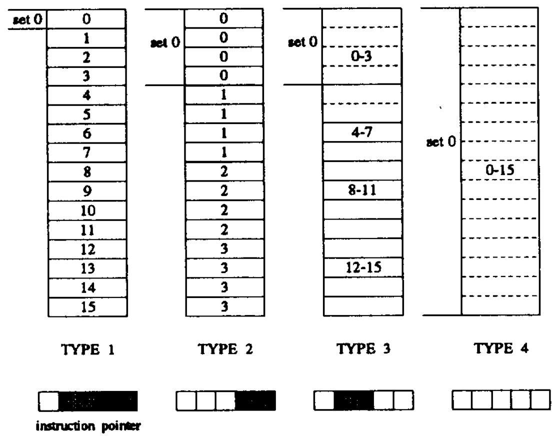


Figure 6: Several implementations of auxiliary matching store.

a loop consisting of i iterations. Each iteration can have different working space.

Figure 6 shows four types of the auxiliary matching store (AMS). An access into a slot of the AMS is addressed by the instruction pointer.

In TYPE 1, the lower part of an instruction pointer is used to access the AMS, in which each slot is mapped into an address and considered as a set.
In TYPE 2, the lower part of an instruction pointer is used to access the AMS but one set comprises n slots. For example, in Figure 6, n is equal to 4. All tokens with the same low part of an instruction pointer will be mapped to the same set.
In TYPE 3, the middle part of an instruction pointer is used to access a set and all tokens with the same range of the low part of an instruction pointer will be mapped to the same set. In Figure 6, the lower addresses 0 to 3 are mapped to set 0.
In TYPE 4, the AMS is implemented as a fully associative memory. That is, an instruction can be stored in any slot of the AMS.

Based on the behavior of dynamic programs and the limitation of an AMS space, several design principles must be followed. So, the efficiency of an AMS can be achieved. Some design principles are listed as follows:

- The number of active code blocks must be limited.
- The related instructions must be arranged nearby.
- The replacement of the matching slot must be deterministic.

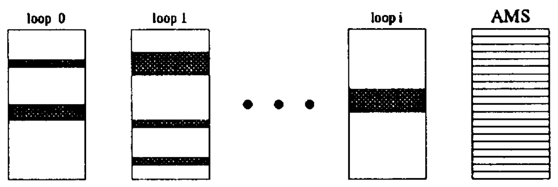


Figure 7: The relation between iterations and the space of the AMS.

3.1 The number of active code blocks must be limited.

A program spends most of the executing time in loops and function calls. When a loop is processed, many iterations can exist concurrently. All these iterations have the same IP (instruction pointer).

In TYPE 1, a set consists of one slot and a slot can only hold a token. For a loop, all of the iterations will be mapped to the same area. If too many iterations are active, spillings will happen frequently because tokens belonging to different iterations but to the same IP possibly enter the AMS simultanously. Figure 7 [1] illustrates the relation between iterations and the AMS. In TYPE 2, one set consists of n slots and different IPs are mapped to different sets. Therefore, we can roughly estimate that n iterations can be active concurrently and the spilling rate will not be high. But once the number of active iterations exceeds n, then, the spilling rate will raise up because more instructions compete for the n slots in the AMS. When more than one iteration are active, we can predicate that TYPE 3 has less spilling rate than TYPE 1 does. In TYPE 4, there is not fixed relation between IP and the slot in a set. Therefore, it will have the greatest ability in accumulation of the number of active iterations among the four types.

After investigating the relation between the iterations[2] and the space of the AMS. We conclude that the more code blocks are active, the more missings happen.

To limit the number of active code blocks, some approaches can be applied. The first approach is to limit the number of iterations in a loop to K [18]. Assume that the number of active iterations is approximately equal to the number of active code blocks. The second approach is that the resource manager puts the limitation on the maximum number of the activity frames dispatched. The last approach is to use explicit hardwares controlling the number of active code blocks.

[1]Here, we make the assumption that the instruction space of a loop is equal to the size of the AMS so different instructions are mapped to different slots of the AMS with the exception of the ones with the same destinations.

[2]Each iteration can be considered as a code block or a process.

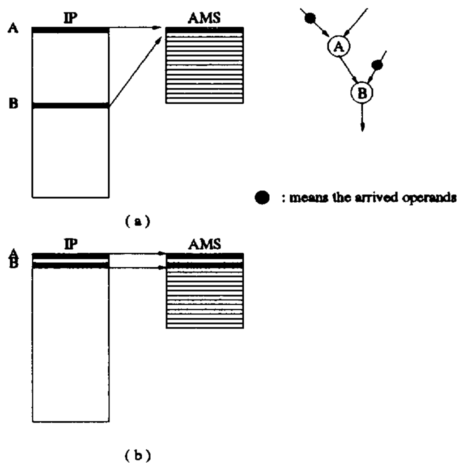


Figure 8: The illustration of nearby arrangement.

3.2 The related instructions must be arranged nearby.

Addresses of instructions in dataflow program can be assigned randomly without affecting the result of executions, because instructions are fired by the data synchronization principle. This property should be exploited to enhance localities and to reduce missing rate.

In TYPE 1 of the AMS, if the related instructions arranged like (a) of Figure 8, it is perfectly possible for a spilling to happen. In TYPE 2, without consideration of other instructions, the (a) arrangement can be tolerated because of n slots in one set. However, the (b) of Figure 8 will map A and B into different sets. In TYPE 3, the (a) and (b) arrangements cause the same result. Both (a) and (b) map A and B into the first set. When processes concurrently executed are considered, we will favour TYPE 2 because it can tolerate more concurrent iterations. In TYPE 4, the (a) and (b) arrangements make no difference.

After examining the relation between instruction sequence and spilling, we are concerned how instructions can be arranged nearby. Two properties are discussed first.

- D_level[7].
- E_level[7].

In fact, a dataflow program is a dataflow graph. An instruction can be considered as a node in the graph. Then, D_level is the minimum number of spanning from the root level. The D_level of an instruction can be referred to as the level (or timing) at which at least one of its inputs becomes available. E_level is the maximum number of spanning from the root level. The E_level of an instruction can be referred to as the level

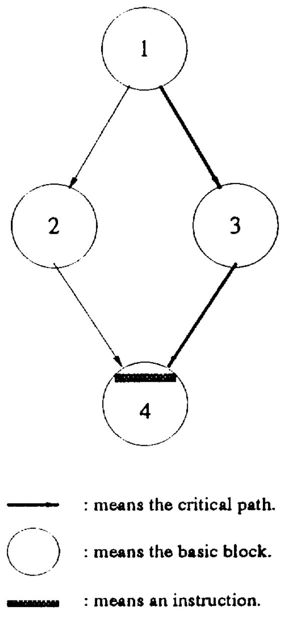


Figure 9: The waiting time on the critical path.

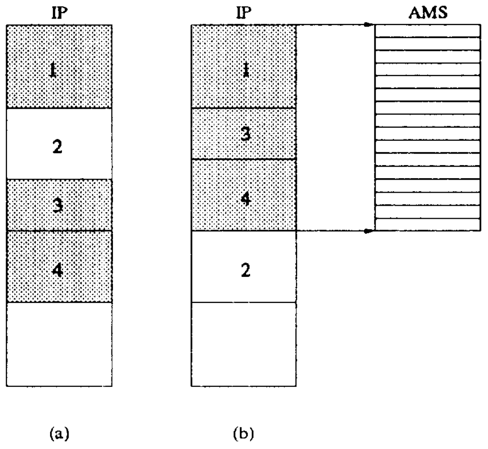


Figure 10: The mapping of the critical blocks.

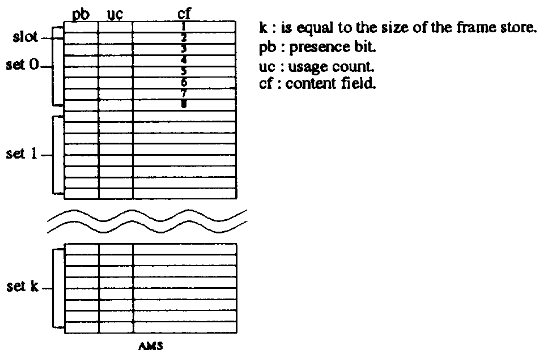


Figure 11: The FOSS structure.

(or timing) at which all of its inputs become available. The D_level and E_level algorithms are shown in appendix A.1 and appendix A.2. $t_{waiting}$ is defined as the difference between E_level and D_level.

$$t_{waiting} = \text{E\_level} - \text{D\_level}.$$

In Figure 9, if an instruction is in basic block 4, its waiting time will have two possible values because of the two possible paths from block 1 to block 4. Here, the one passing through the critical path is our choice.

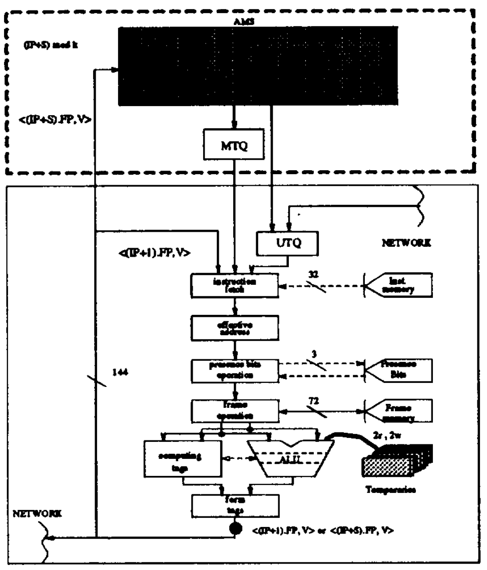
We assume that $t_{waiting}$ is quite small. By setting the node addresses in ascending order of D_level along the critical path, spatial localities can be enhanced. For example, in Figure 10, we list two possible assignments of the data flow graph in Figure 9. We prefer part (b) to part (a) because the path 1-3-4 has higher possibility to be taken than the path 1-2-4. In (b), the blocks which will be executed frequently are assigned together, then, spilling will not happen frequently.

3.3 The replacement of the AMS must be deterministic.

The objective of the replacement is to spill out inactive code blocks and to hold active code blocks. Several approaches are provided as follows:

- LRU (Least Recently Used).
- LIFO (Last–In First–out).
- maximum $t_{waiting}$.

Based on three reasons, LRU is chosen when a token has been staying in a slot for a long time. First, the partner is the result of a remote request. We assume the latency is quite long so that it is not worth waiting for response. Second, the partner has been thrown out of the AMS because of spilling or other reasons so it is impossible to be matched. Third, the code block including this token has been temporarily



MTQ : matching token queue

UTQ : unmatching token queue

Figure 12: Monsoon architecture and FOSS.

inactive. In these three cases, old token should be substituted by a new one. LIFO can be applied in which a token has been waiting for a long time so its partner should arrive soon or in which the number of active code blocks will not increase. Why will LIFO not increase the number of active code blocks? Because in our design a token thrown out of the AMS means that the block including this token will not be active for a while. Therefore, if a last-in token is thrown out first, the corresponding code block [3] will be inactive so that the number of active code blocks will not increase. The token with maximum $t_{waiting}$ should be spilled out because it wastes too much time in the slot and still cannot be matched. To spill out the token with maximum $t_{waiting}$, we need some extra circuits to support this function.

4 Architecture

In this section, there are two kind of AMS (auxiliary matching store) to be discussed. Both of them are used to reduce the number of pipeline bubbles. The first one called (FOSS) has a simple structure whereas the second one called (SOCS) has a complicated structure. Both kinds of AMS can limit the number of active code blocks but the first one needs the support of the resource manager.

[3] If a token is held in the AMS, the code block including that token is active.

4.1 FOSS

In a slot of AMS, there are three fields, PB, UC, and CF, respectively, as shown in Figure 11. Presence bits are used to indicate whether a slot is full or not. Initially, PB (presence bits) is set to *empty* because the token will be spilled out. When a token is put into a slot, its presence bit will be set to *full*. After matching or spilling happens and a token is spilled out, presence bit will be set to *empty*. Initially, UC (usage count) will be set to 0 which means no token is in that slot, while 1 means this token is the newest one. When a token enters, the UC associated with that slot will be set to 1 and the others will be increased by 1 except those ones with UC equal to 0. In fact, UC records how long a token has been staying in a slot. UC is used to support the LRU or FILO algorithm. CF (content field) is employed to hold a token. In our design, there are eight slots in a set because of eight stages in the pipeline of Monsoon. To fill up the eight stages of the pipeline, eight code blocks are allowed to be active. Ideally, each code block can occupy one stage. Therefore, there are eight slots in a set. The size of the AMS is equal to the size of the activity frame, *k*. That is because at any time the number of tokens which needs to match will not exceed the size of the activity frame.

When two result tokens are produced, shown in Figure 12, the one with (IP+S) index will be sent into the set of AMS indexed by IP mod *k* (where *k* is the size of the frame store, and S is the offset between the current instruction and the destination), and the other with (IP+1) index directly passes into the pipeline.

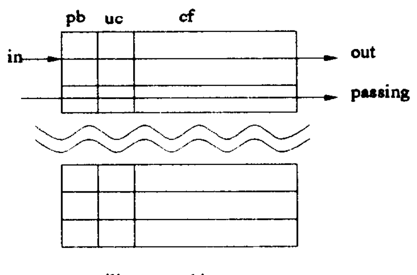


Figure 13: The illustration of **in, out** and **passing**.

The token with (IP+1) index always has higher priority than the one coming from the matching token queue (MTQ) or the unmatching token queue (UTQ). Similarly, the priority of MTQ is higher than UTQ.

After the access address is computed, a token is compared to the slots with presence bit full. If match happens, the result package is put into MTQ. Otherwise, find an empty slot in the set to save this token. If eight slots are all full, then, spilling action become true. When both of MTQ and UTQ are empty, the tokens in AMS will be put into UTQ.

By the assignment of priority and the usage of the LRU algorithm, our AMS can automatically limit the number of active code blocks and keep the code blocks active longer. By assigning the token with IP+1 index to the highest priority, we make sure that the fired instructions are more closely related and the same code block can be active longer. The token coming from MTQ has the second highest priority because we want to assure that there are less pipeline bubbles and less new active code blocks. By assignment of the lowest priority to a token from UTQ, the number of active code blocks can be limited.

When spilling happens, it means that the number of the active code blocks exceeds the limitation. So one inactive code block should be thrown out of the AMS[4] and it would not get active again soon. If many code blocks become active or inactive shortly and frequently, the locality will be dramatically destroyed.

The operations of the AMS are expressed below. We use the grammar of C language to define each formula.

The condition for **in, out** and **passing** are described as following, see Figure 13

[4] In practice, only one token will be thrown out of the matching slot. If we can make sure that this token will not be active in the short time, then, in turn, it will limit the other tokens belonging to the same block to be fired because of the data dependency. In consequence, throwing out a token can be thought of as throwing out a block.

```
in
         if(MTQ&&UTQ!=empty)&&(token.type==
                                          dyadic)
out      if(match|| spill|| fresh)
passing
         if((MTQ&&UTQ==Empty)||(token.type==
                                          monadic)
```

The action of the auxiliary matching store ∈ {**match, spill, fill, fresh**}.

```
match    ∃ j ∈ (set(i).slot(j).PB==full)
         ∋ (set(i).slot(j).CF.FP==token.FP)
         then
         out_to_MTQ(set(i).slot(j).CF, token);
         set(i).slot(j).PB=empty;
         set(i).slot(j).UC=0;

spill    ∀ j∈ (set(i).slot(j).PB==full)
         ∋ (set(i).slot(j).CF.FP!=token.FP)
         then
         if(set(i).slot(j).UC==MAX)
         out_to_UTQ(set(i).slot(j).CF).

fill     ∃ j ∈ set(i).slot(j).PB==empty
         then
         set(i).slot(j).CF=token;
         set(i).slot(j).PB=full;

fresh    ∀ i, j ∋ set(i).slot(j).PB==full
         then
         out_to_UTQ(set(i).slot(j).CF);
         set(i).slot(j).PB=empty;
```

Here, i = IP mod k, k = activity frame size, j∈(1, 2, ⋯, 8).
out_to_MTQ is a function which puts its pair of parameters into MTQ.
out_to_UTQ is a function which puts its parameter into UTQ.
We assume that it is possible to know in advance that a token will be sent to a dyadic or a monodic instruction. Using the combination of the LRU algorithm and the assignment of the priority to the code blocks we can control the number of active code blocks and keep the locality.

4.2 SOCS

As shown in Figure 14, the SOCS architecture is very similar to FOSS. However, the eight slots of a set in FOSS are divided into eight AMS banks. One bank is for a code block. A token that belongs to the same code block will be put in the same AMS bank. Every bank has k sets [5] and k slots where each set comprises one slot. Each slot is same as the one in FOSS with exception of UC. In SOCS, each slot only has two fields, PB and CF. ITB (instruction transfer buffer) has three fields VB (valid bit), FP (frame pointer) and TC (token count). VB specifies the slot of ITB full or empty. A slot of the ITB is allocated to an activity

[5] k is equal to the size of the activity frame size.

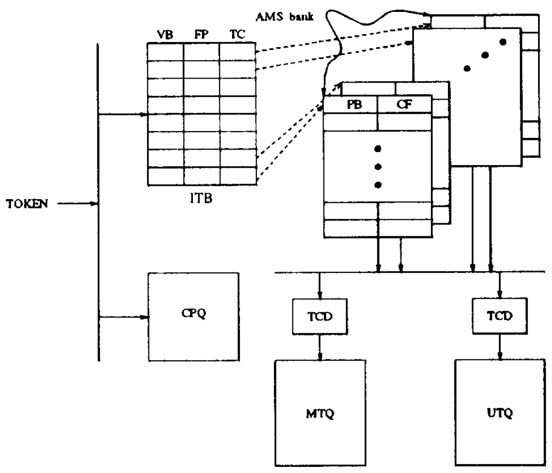


Figure 14: The structure of the SOCS

frame. Initially, VB is available and set to empty. If a slot is assigned to a code block[6], then, VB associated with that slot will be set to full. TC (token count) serves as a counter to count how many instructions in an AMS bank. Whenever a token accesses an AMS bank, then, the TC corresponding to that bank will be increased by 1. After spilling, matching or refreshing happens, TC will be decreased. The mechanism is that every token sent to MTQ or UTQ will pass through TCD (token count decrementor), and then TCD can decrease TC by 1 or 2.

When a token is produced, it[7] needs to be put into AMS banks. First, its FP will be compared against the FPs in the slots of the ITB, which VB are equal to full. If the equivalent one is found, this token will be put into the corresponding AMS bank, and then the corresponding TC increased by 1. Otherwise, find an available slot and then put the FP of the token into its FP field and set its VB field to full and increase the TC field by 1, and then put this token into the corresponding AMS bank. If there is no available slot, this token will be deposited into the CPQ (controlling process queue).

After this token is put into the AMS bank, in turn, the IP of this token is used to access the slot of the AMS bank addressed by IP mod k. If another token has the same IPs, then match occurs and the result package will be sent to the MTQ. TCD will decrease the corresponding TC by 2. If no one is found, spilling will happen and the one originally in the slot will be sent to UTQ. Also, TCD will decrease the corresponding TC by 1. If no other token in the slot, this token will be deposited into the slot and PB (presence bit), originally empty, will be set to full. If a TC becomes 0, then a token will be taken from the CPQ, in turn, the process is alike to a new token put into ITB. When a token is taken from the CPQ, in effect, a new code block becomes active. In the opposite, if a token is put into CPQ, one code block is inactive.

The CPQ is employed to control the number of active code blocks. If there are no available slots in ITB, it means that the number of the active code blocks is exactly eight which is the maximum number of code blocks allowed to be active. Therefore, if a token is going to enter the AMS, it will be excluded into the CPQ. When a token is sent to the CPQ, it denotes that a code block is suspended. Only instructions about interblock (interprocess) parameter/return–value passing could be put into CPQ. If these kinds of instructions are suspended, then, there are no other instructions of the same block can be executed because of data dependency. Using this mechanism, we can strictly control the number of code blocks at eight.

[6]It means that the FP field of the slot is allocated a frame pointer.

[7]It is not the token with IP+1 index.

5 Conclusion

In this paper, we first described the problem of pipeline bubbles. Then, we made an analysis about the behavior of dynamic dataflow programs, and concluded that the number of active code blocks must be limited and the related instructions must be arranged nearby. We also discussed the approaches to limiting active code blocks and arranging related instructions. Finally, we proposed two structures to overcome the problems of dynamic dataflow programs and to achieve a high matching rate.

A Appendix

A.1 D_level

The rules for determining the D_level for each node N_j of an acyclic graph G(N, A) are as follows;

```
D($N_j$): D_level for each node $N_j$ of a graph G(N,A)
  { for all $N_j$ of N { D($N_j$) := infinite };
    V :=empty;
    for all $N_j$ of N where $N_j$ has input arcs
    { D($N_j$) := 0; V = V + $N_j$ };
    i := 0;
    while V $\neq$ empty;
    { W := empty; i:=i+1;
        for all $N_j$ of V
            for all $N_k$ which has an arc $A_{jk}$
                if D($N_k$) > i then
                { D($N_k$) := i; W=W+$N_k$ };
        V := W;
    };
  }
```

A.2 E_level

The rules for determining the E_level for each node N_j of an acyclic graph G(N, A) are as follows;

```
E($N_j$): E_level for each node $N_j$ of a graph G(N,A)
  { for all $N_j$ of N { E($N_j$) := 0 };
```

```
        V :=empty;
        for all $N_j$ of N where $N_j$ has input arcs
        { E($N_j$) := 0; V = V + $N_j$ };
        i := 0;
        while V $\neq$ empty;
        { W := empty; i:=i+1;
            for all $N_j$ of V
                for all $N_k$ which has an arc $A_{jk}$
                    if E($N_k$) < i then
                      { E($N_k$) := i; W=W+$N_k$ };
            V := W;
        };
    }
```

References

[1] G. M. Papadopoulos and D. E. Culler, "Monsoon: an Explicit Token-Store Architecture," In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp.82-91, 1990.

[2] S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama., and T. Yuba. "An Architecture of a Dataflow Single Chip Processor," In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pp.46-53, 1989.

[3] V. G. Grafe and J. E. Hoch, "The Epsilon-2 Hybrid Dataflow Architecture," In *Proceeding of Compcon90*, pp. 88-93, March 1990.

[4] Arvind and Culler, D. E. dataflow architectures. In *Annual Reviews in Computer Science,* Annual Reviews Inc., Palo Alto, CA, 1986, vol. 1, pp. 225-253; Reprinted in Thakker, S. S.(Ed.). *"Dataflow and reduction Architectures,"* IEEE Computer Society Press, 1987.

[5] A. J. Smith, "Cache memories," *Comput. Survey,* vol. 14, no. 3, pp. 473-530, Sept. 1982.

[6] S. A. Thorenson and A. E. Oldehoeft, "Instruction reference patterns in dataflow programs," in Proc. *ACM Annu. Conf.*, pp. 211-217, Oct. 1980.

[7] M. Tokoro, J. R. Jagannathan, and H. Sunahara, "On the working set concept for dataflow machines," In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pp. 90-97, July 1983.

[8] G. M. Papadopoulos and K. R. Traub, "Multithreading: A Revisionist View of Dataflow Architectures," In *Proceedings of the 18th Annual International Symposium on Computer Architecture, Toronto*, pp. 342-351, Mar. 1991.

[9] M. Takesue, "A load control mechanism for dataflow machines," *System and Computers in Japan,* Scripta Technica, Inc. vol. 19, no. 10, p-p. 55-69, Oct. 1988 (This is a translated version of the paper appeared in it Tran. IEICE Japan, vol. J70-D, no. 10, pp. 1878-1889, Oct. 1987.)

[10] Masaru Takesue, "Cache Memories for Data Flow Machines," IEEE Tran. Comput., vol. 41, no. 6, pp. 677-687, 1992.

[11] Arvid, and Gostelow, K. P., "The U-interpreter,", IEEE computer, vol. 15, no. 2, Feb. 1982.

[12] Gurd, J., Kirkam, C. C., and Watson, I., "The Manchester prototype dataflow compute," *Comm. ACM* vol. 28, no. 1, pp. 34-52, Jan. 1985.

[13] Hiraki, K., Sekiguchi, S., and Shimada, T., "System architecture of a dataflow supercomputer," Tech. Rep. Computer System Division, Electrotechnical Labotory, 1-1-4 Umezono, Sakuramura, Niihari-gun, Ibaraki 305, Japan, 1987.

[14] Shimada, T., Hiraki., and Nishida, K., "An architecture of a data flow machine and its evaluation," *Proceedings of CompCon 84*, IEEE, pp. 486-490, 1984.

[15] Arvid, and Iannucci, R. A., "Two fundamental issues in multiprocessing," *Proceedings of DFVLR-Conference on Parallel Processing in Science and Engineering, Bonn-Bad Godesberg, W. Germany, Jun.* 1987.

[16] Arivnd, Nikhil, R. S. and Pingali, K. K., "I-structure: Data structures for parallel computing," Tech. Rep. Computation Structures Group Memo 269 , MIT Laboratory for Computer Science, MIT, Feb. 1987.

[17] Chaitin, G., Auslander, M., Chandra, A., Cocke, J., Hopkins, M., and Markstein, P., "Register allocation via coloring," *Comput. Lang.* 6m pp. 47-57, 1981.

[18] Culler, D. E., "Managing parallelism and resource in scientific dataflow programs," Ph.D. thesis, MIT Department of Electrical Engineering and Computer Science, MIT, Jun. 1989; Tech. Rep. TR446, MIT Laboratory for Computer Science, MIT.

[19] Micah Beck, Richard Johnson, and Keshav Pingali, "From control flow to dataflow," Journal of parallel and distributed computing vol. 12, pp. 118-129 1991.

[20] Cytron, R., Ferrante, J., Rosen, B. k., Wegman, M. N., and Zadeck, F. k., "An efficient method of computing static single assigment form", In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages*, pp. 25-35, Jan. 1989.

[21] Ferrante, J., Ottenstein, K. J., and Warren, J. D., "The program dependency graph and its uses in optimization,"*ACM Trans. Programming Languages Systems* vol. 9, no. 3, page 319-349, Jun. 1987.